Data Mining and Knowledge Discovery
https://doi.org/10.1007/s10618-019-00646-y

Wrangling Messy CSV Files by Detecting Row and Type Patterns

G.J.J. van den Burg · A. Nazábal · C. Sutton

Received: 26 November 2018 / Accepted: 19 July 2019

Abstract Data scientists spend the majority of their time on preparing data for analysis. One of the first steps in this preparation phase is to load the data from the raw storage format. Comma-separated value (CSV) files are a popular format for tabular data due to their simplicity and ostensible ease of use. However, formatting standards for CSV files are not followed consistently, so each file requires manual inspection and potentially repair before the data can be loaded, an enormous waste of human effort for a task that should be one of the simplest parts of data science. The first and most essential step in retrieving data from CSV files is deciding on the dialect of the file, such as the cell delimiter and quote character. Existing dialect detection approaches are few and non-robust. In this paper, we propose a dialect detection method based on a novel measure of data consistency of parsed data files. Our method achieves 97% overall accuracy on a large corpus of real-world CSV files and improves the accuracy on messy CSV files by almost 22% compared to existing approaches, including those in the Python standard library. Our measure of data consistency is not specific to the data parsing problem, and has potential for more general applicability.

Keywords Data Wrangling · Data Parsing · Comma Separated Values

G.J.J. van den Burg
The Alan Turing Institute, London, UK.
E-mail: gvandenburg@turing.ac.uk
ORCID: 0000-0001-5439-6248

A. Nazábal
The Alan Turing Institute, London, UK.
E-mail: anazabal@turing.ac.uk
ORCID: 0000-0002-9414-7139

C. Sutton
Google, Inc. Mountain View, CA, USA.
E-mail: charlessutton@google.com
Other affiliations: The Alan Turing Institute, London, UK,
School of Informatics, The University of Edinburgh, UK.
ORCID: 0000-0002-0041-3820

CSV is a textbook example of how not *to design a textual file format.*

— The Art of Unix Programming, Raymond (2003).

1 Introduction

The goal of data science is to extract valuable knowledge from data through the use of machine learning and statistical analysis. Increasingly however, it has become clear that in reality data scientists spent up to 80% of their time on importing, organizing, cleaning, and wrangling their data in preparation for the analysis (Dasu and Johnson, 2003; Lohr, 2014; Kandel et al., 2011; Crowdflower, 2016; Kaggle, 2017). Collectively this represents an enormous amount of time, money, and talent. As the role of data science is expected to only increase in the future, it is important that the mundane tasks of data wrangling are automated as much as possible. One reason that data scientists spent so much time on data wrangling issues is due to what has been called the *double Anna Karenina principle* of data wrangling: "every messy dataset is messy in its own way, and every clean dataset is also clean in its own way" (Sutton et al., 2018).[1] Because of the wide variety of data quality issues and data formats that exist ("messy in its own way"), it is difficult to re-use data wrangling scripts and tools, perhaps explaining the manual effort required in data wrangling.

This problem can be observed even in the earliest and what might be considered the simplest stages of the data wrangling process, that of loading and parsing the data in the first place. In this work, we focus on comma-separated value (CSV) files, which despite their deceptively simple nature, pose a rich source of formatting variability that frustrates data parsing.[2] CSV files are ubiquitous as a format for sharing tabular data on the web: government data repositories often present their data in CSV format,[3] and based on our data collection we conservatively estimate that GitHub.com alone contains over 19 million CSV files. Advantages of CSV files include their simplicity, portability, and potential to be tracked in version control.

However, despite some standardization effort (RFC 4180; Shafranovich, 2005) a wide variety of subtly incompatible variations of CSV files exist, including the 34 different formats among the CSV files in our data. For example, we observe that values can be separated by commas, semicolons, spaces, tabs, or any other character, and can be surrounded by quotation marks or apostrophes to guard against delimiter collision or for no reason at all. Figure 1 illustrates a few of the problems that real-world CSV files exhibit and is based on files encountered in this study. Figure 1(a) illustrates a normal CSV file that uses comma as the delimiter and has both empty and quoted cells. Figure 1(b)

[1] This is related to the principle of the fragility of good things (Arnold, 2003).

[2] While we use the term *CSV file* throughout, our discussion and our proposed method applies directly to *tab*-separated value (TSV) and *delimiter*-separated value (DSV) files, as well as `.dat` and `.txt` files that are effectively CSV files but use a different file extension.

[3] Mitlöhner et al. (2016) survey 200,000 CSV files from open government data portals.

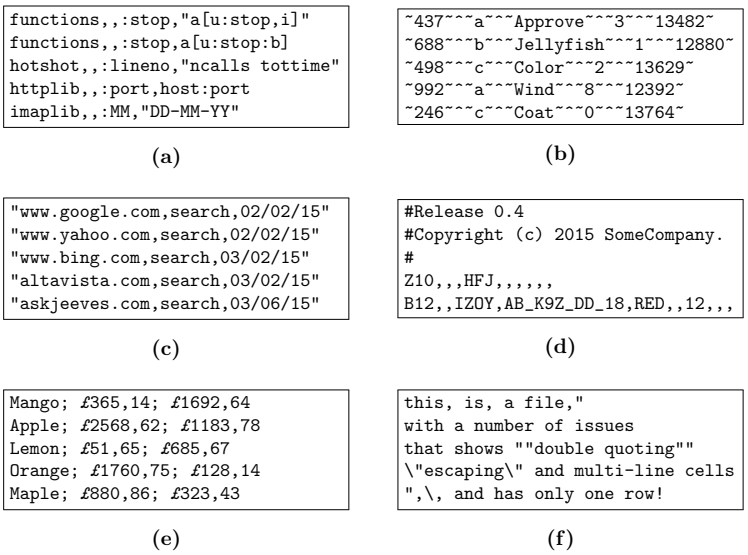

Fig. 1 Illustration of some of the variations of real-world CSV files. See the main text for a description of each of the files.

shows a variation that uses the caret symbol as delimiter and the tilde as quotation mark. Next, Figure 1(c) illustrates an *ambiguous* CSV file: each row is surrounded with quotation marks, implying that the correct interpretation is a single column of strings. However, if the quotation marks are stripped a table appears where values are separated by the comma. Figure 1(d) illustrates a file with comment lines, which are not supported by the CSV standard. Figure 1(e) is adapted from Döhmen et al. (2017) and illustrates that different choices for the delimiter can result in the same number of columns (the semicolon, space, comma, and pound sign all yield three columns). Finally, Figure 1(f) illustrates a number of issues simultaneously: quote escaping with an escape character and using double quotes, delimiter escaping, and multi-line cells.

It may surprise the reader that we seem to claim that CSV parsing is an open problem. However, as Figure 1 illustrates, there is a remakarble diversity of formatting parameters in CSV files, including different delimiters, quoting characters, and so on — we call these parameters the *dialect* of a file. Automatically detecting the dialect of a CSV file is a problem that has received little attention. Indeed, although almost every programming language provides functionality for parsing CSV files, very few are robust against the format variation of real-world files. To the best of our knowledge, Python is the only programming language whose standard library supports automatic dialect detection; however, our experiments show that this method fails to detect the dialect in 36% of non-standard CSV files, and no other methods exist that achieve higher accuracy. This means that in practice almost every file requires manual inspection before the data can be loaded, because it may contain a non-standard format and could therefore be parsed incorrectly.

While dialect detection is generally easy for a human analyst, it is nontrivial to do so automatically because every dialect will yield *some* table (even if it is incorrect) and it is not straightforward to define a function that identifies the correct dialect reliably.

In this paper we present a method for automatically detecting the dialect of a CSV file through a novel *data consistency measure*. With this measure we can search the space of potential dialects for one that yields the most consistent parsed data. By *consistency* here we consider primarily (a) the shape of the parsed data, which we capture using an abstraction called *row patterns*, and (b) the *data type* of the cells, such as integers or strings. This aims to capture how a human analyst might identify the dialect: searching for a character that results in regular row patterns and using knowledge of what real data "looks like". This data consistency measure is independent of the CSV problem, and may be more broadly applicable as a way to quantify whether a data table has a "natural shape" in other data wrangling and data cleaning problems.

The paper is structured as follows. In Section 2 we present an overview of related work on both table detection and CSV parsing. Next, Section 3 gives a formal description of CSV dialect detection. Our proposed data consistency measure is presented in Section 4. Results of a thorough comparison of our method with the few existing alternatives are presented in Section 5. Section 6 concludes the paper.

2 Related Work

Only very few publications have paid any attention to the problem of CSV parsing. Mitlöhner et al. (2016) explore a large collection of CSV files from open data platforms of various governments and find significant variability in the structure and format of CSV files, but do not present a novel CSV parser. In recent work, Döhmen et al. (2017) present a so-called "multi-hypothesis" parser for messy CSV files that constructs a tree of parsing configurations and assigns a score to each based on heuristic metrics. The authors evaluate the parser on 64 files with known ground truth from the UK open government data portal. Our preliminary analysis showed that this parser is not very robust against many variations of CSV files, possibly due to the small evaluation set used in the paper. To address this issue we compare our proposed solution to this method on a corpus of thousands of CSV files with ground truth.

A topic closely related to CSV parsing and dialect detection is extracting tables from free text (e.g. emails, text files, etc.). Work on this topic includes that of Ng et al. (1999), who locate tables based on various surface features. Later work by Pinto et al. (2003) applies a similar strategy but uses conditional random fields and expands the problem by identifying the semantic role of each row in the table (i.e. header, data row, etc.). More recent work includes that of Eberius et al. (2013) and Koci et al. (2016) on the DeExcelerator program for extracting and annotating the semantic role of tables in Excel files. CSV parsing differs from these approaches because CSV files have more structure

than free text tables, but are less well-defined than Excel files. CSV files use a specific character to delimit cells and can employ the quoting mechanism to let cells span multiple lines and guard against delimiter collision. This explains why these methods cannot be readily applied to CSV parsing.

More broadly, there is work on data wrangling that relates to importing data. Fisher et al. (2008) present the PADS system for retrieving structured data from ad-hoc sources such as log files. However, the authors explicitly mention that CSV files are not a source of ad-hoc data. Moreover, the PADS system may not be robust against features of real-world CSV files such as comment sections and line breaks in quoted cells. Well-known work on data wrangling that aims to reduce the time that human analysts spent on this task is that of Kandel et al. (2011) with the Wrangler system. In follow-up work, Guo et al. (2011) provide a method for automatically suggesting data transformations to the analyst and introduce a "table suitability metric" that quantifies how well a table corresponds to the relational format, also known as *tidy data* (Wickham, 2014). Although CSV files are not considered in these works, we evaluate the table suitability metric from Guo et al. (2011) for CSV dialect detection.

Finally, it is worth mentioning efforts that aim to solve the problem of CSV parsing by proposing extensions or variations on the CSV format. A study on the use and future of CSV files on the web was performed by a working group of the World Wide Web Consortium (Tennison, 2016). The group proposed to provide metadata about a CSV file through an accompanying JSON[4] description file (Tennison and Kellogg, 2015; Frictionless Data, 2017). While this recommendation could certainly address some of the issues of CSV parsing, it requires users to specify and maintain a secondary file besides the CSV file itself. Moreover, it does not address the issues of the many existing messy CSV files. Alternatives such as the CSVY format (Rovegno and Fenner, 2015) propose to add a YAML[5] header with metadata. While this does combine the metadata and tabular data in a single file, it requires the user to adopt special tools, which may limit the adoption of these formats.

3 Problem Statement

We start by reiterating commonly used definitions for tabular data. Consider attributes A_ℓ for $\ell = 1, \ldots, L$, each with a corresponding domain $\mathcal{V}_\ell$ (Codd, 1970). The domain $\mathcal{V}_\ell$ is the set of allowed values for the attribute, e.g. the set of all floating point numbers. A tuple $\mathbf{t}_i$ is an ordered sequence of values from the domains, i.e.

$$\mathbf{t}_i \in \mathcal{V}_1 \times \mathcal{V}_2 \times \cdots \times \mathcal{V}_L. \tag{1}$$

Next, we define a *table* as an array of tuples, i.e. $\mathbf{T} = [\mathbf{t}_1, \ldots, \mathbf{t}_n]$. Note that a table is more appropriate to describe the rows in a CSV file than a relation,

[4] JavaScript Object Notation (Crockford, 2006).
[5] YAML Ain't Markup Language (Evans, 2001).

because a relation is a *set* of tuples whereas a table allows for duplicate tuples and captures the order of tuples.

Unfortunately the above definitions are insufficient to capture all real-world uses of CSV files because files can contain multiple tables (e.g. when exported from spreadsheets). Thus a CSV file can contain tables $\mathbf{T}_1, \ldots, \mathbf{T}_S$ where the attributes and data domains of the table $\mathbf{T}_s$ depend on s. This generalization also allows for headers and comment lines by considering these to be separate tables in the CSV file. In the next paragraph we focus on files with a single table to simplify the presentation, but our description generalizes straightforwardly to files with multiple tables.

A table $\mathbf{T}$ is transformed to a CSV file $\mathbf{x}$ by a *formatter*. The formatter f can be decomposed into two components as $f = f_2 \circ f_1$. The first component f_1 takes the table $\mathbf{T} = [\mathbf{t}_1, \ldots, \mathbf{t}_n]$, with data domains $\mathcal{V}_\ell$, and outputs a table $\mathbf{C} = [\mathbf{c}_1, \ldots, \mathbf{c}_n]$ where the domain for all attributes is that of strings.[6] This function thus includes the conversion of numeric values to their string representation and is therefore not perfectly invertible in general. Subsequently, the second component f_2 of the formatter converts the string table $\mathbf{C}$ to a CSV file $\mathbf{x}$. This function takes parameters such as the delimiter and the quote character that affect how the string tuples $\mathbf{c}_i$ in $\mathbf{C}$ are converted to the lines of the CSV file. These parameters are typically referred to as the *dialect* of the CSV file, and we denote these by $\boldsymbol{\theta}$. We assume that $\boldsymbol{\theta}$ contains all of the configuration parameters required by the formatter, so that given $\boldsymbol{\theta}$ and a table $\mathbf{T}$, the output of the formatter is deterministic. Therefore, we denote the formatter by a function $f_2(f_1(\mathbf{T}), \boldsymbol{\theta})$.

In this work we consider a dialect of three components: the delimiter (θ_d), the quote character (θ_q), and the escape character (θ_e). The delimiter is used to separate the elements of the string tuples $\mathbf{c}_i$, the quote character surrounds certain elements, and the escape character can be used to signal that certain delimiter or quote characters should not be interpreted. Each of these components can be absent in a CSV file, in which case we denote them by ε (the empty string). Our approach could easily be extended to include other formatting parameters, such as the presence of a comment character or the use of a specific line terminator. Moreover, some formatters contain additional parameters, such as those that control the rules for when a cell should be quoted or not, but we do not consider these in our dialect detection approach as they are not needed when parsing a file. We can now present dialect detection as the inverse problem of recovering the dialect $\boldsymbol{\theta}$ from an observed CSV file.

Definition 1 (Dialect Detection) Let $\mathbf{x}$ be a CSV file created by a formatter $f = f_2 \circ f_1$ using a dialect $\boldsymbol{\theta} = (\theta_d, \theta_q, \theta_e)$ for f_2, that contains tables $\mathbf{T}_1, \ldots, \mathbf{T}_S$ with $S \geq 1$. Then *dialect detection* is the problem of identifying $\boldsymbol{\theta}$ from $\mathbf{x}$.

[6] Formally, f_1 decomposes to a set of functions for each data domain, $g_\ell : \mathcal{V}_\ell \to \Sigma^*$, where Σ^* is the set of all strings, i.e. the Kleene closure of an alphabet Σ (Kleene, 1956). This process is also known as *type casting*.

In general, this problem is difficult to solve automatically. First, both f_1 and f_2 are typically unknown, which means that $\mathbf{x}$ is the only information available. Second, as Figure 1(e) illustrates, detecting the correct dialect requires knowledge of what the data represents to properly disambiguate potential dialects. This is easy for a human familiar with representations of data, but, informally, it is difficult for a computer since the CSV file is simply a sequence of characters and there is no automatic way to verify that a given dialect correctly parses a CSV file (every dialect yields *some* table). Third, any file with a somewhat regular pattern of symbols can be claimed to be a CSV file, even if its contents are not human-readable. This means that theoretically the search space of potential dialects is of the order $|\mathcal{X}|^3$ where $\mathcal{X}$ is the set of unique characters in the file.

4 A Consistency Measure for Dialect Detection

Given the problem of dialect detection defined above, the goal is to find a way to identify whether a given dialect is correct. We propose that a CSV file is parsed with a correct dialect rather than an incorrect one when the resulting tuples appear more *consistent*. Our notion of consistency is modelled on the approach a human would take to determine the correct dialect, i.e. searching for a dialect that would result in rows of similar length and using knowledge of what real data "looks like". Hence, we develop a consistency measure for dialect detection based on two components: a measure for row length consistency called the *pattern score*, and a measure for type consistency called the *type score*.

In the following, we define both of these components and their combination in the complete data consistency measure. Furthermore, we explain how to construct and prune the search space of potential dialects, address the issue of tie breaking among equally consistent dialects, and present the computational complexity of our method. We refer to Figure 2 as a running example that illustrates the different components of the data consistency measure.

4.1 Pattern Score

The pattern score is the main driver of the data consistency measure. It is based on the observation that because CSV files generally contain data tables, we expect to find consistent row lengths when we parse the file with the correct dialect. The number of cells in each parsed row, and the consistency of that number throughout the file, is therefore indicative of the correctness of the dialect. Recall that the CSV file $\mathbf{x}$ is created from the string table $\mathbf{C}$ using the dialect $\boldsymbol{\theta}$ through $\mathbf{x} = f_2(\mathbf{C}, \boldsymbol{\theta})$. Let $\widehat{\mathbf{C}}$ denote the parsing result when the CSV file is parsed using dialect $\hat{\boldsymbol{\theta}}$ and denote the elements of $\widehat{\mathbf{C}}$ (i.e. the tuples in the table) by $\hat{\mathbf{c}}_i$. Furthermore, let $\hat{c}_{i,j}$ be the elements of tuple $\hat{\mathbf{c}}_i$ for $j = 1, \ldots, L_i$ and $i = 1, \ldots, n$.

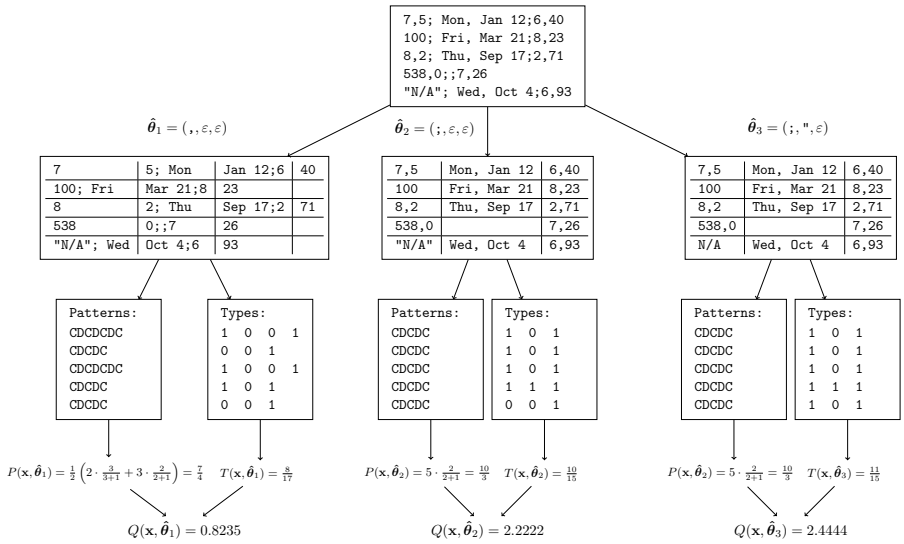

Fig. 2 Illustration of the data consistency measure for different dialects on a constructed example. The figure shows how different dialects can result in different row patterns and pattern scores, as well as different type scores. The difference between $\hat{\boldsymbol{\theta}}_2$ and $\hat{\boldsymbol{\theta}}_3$ is due to the fact that the string `N/A` belongs to a known type, but the string `"N/A"` does not.

We define a *row pattern* as a sequence of characters from the alphabet $\{\texttt{C}, \texttt{D}, \texttt{Q}\}$ where `C` denotes a cell, `D` denotes the delimiter, and `Q` denotes a quote character. Each tuple $\hat{\mathbf{c}}_i$ has a corresponding row pattern that is constructed iteratively by adding the letter `C` to the row pattern for every element $\hat{c}_{i,j}$ in $\hat{\mathbf{c}}_i$ and adding the letter `D` after each cell except the last one (`D` reflects the delimiter).[7] A symmetrically quoted cell (e.g. `"a"`) also adds `C` to the row pattern as this is a regular cell, but for an *asymmetrically* quoted cell (e.g. `a"b`) the sequence `CQC` is added. This can happen when the quote character of a dialect occurs an odd number of times in the file and can be an indication that a potential dialect is incorrect. Thus the row pattern can be seen as a "signature" of the row: it reflects the order of cells and delimiters as well as the presence of spurious quote characters. Refer to Figure 2 for examples of row patterns constructed from parsing results for different dialects.

Note that a CSV file with only one table will have a single unique row pattern if the correct dialect is used for parsing the file. We denote by $\mathcal{P}$ the set of *unique* row patterns, with $K = |\mathcal{P}|$. Let N_k be the number of tuples $\hat{\mathbf{c}}_i$ in $\widehat{\mathbf{C}}$ that yield row pattern $\hat{p}_k \in \mathcal{P}$. Furthermore, for each row pattern $\hat{p}_k \in \mathcal{P}$ we compute its length M_k as the number of delimiters `D` in $\hat{p}_k$. Then, we define the *pattern score* as

$$P(\mathbf{x}, \hat{\boldsymbol{\theta}}) = \frac{1}{K} \sum_{k=1}^{K} N_k \frac{M_k}{M_k + 1}. \tag{2}$$

[7] This is a slight simplification of how the row patterns are constructed. For the full

The pattern score is designed to favor row patterns that occur often and those that are long, while also favoring a smaller number of unique row patterns. When the correct dialect is chosen we expect to observe fewer unique row patterns that occur frequently in the file. Additionally, the ratio of the pattern length favors longer row patterns, which indicate a regular occurrence of delimiters and cells, over shorter patterns, which may indicate an incorrectly chosen delimiter. Notice how in Figure 2 the correct dialect yields a single row pattern throughout the file.

While this function works well for CSV files that contain tables, it gives a value of 0 when the entire file is a single column of data. To handle these files in practice, we replace the numerator by $\max\{\alpha, M_k\}$ where α is a small constant. This constant must be chosen such that single-column CSV files are detected correctly, while avoiding false positive results that assume regular CSV files are a single column of messy data. It was found empirically that $\alpha = 10^{-3}$ achieves this goal well.

4.2 Type Score

While the pattern score is the main component of the data consistency measure, Figure 2 shows that the type score is essential to obtaining state-of-the-art results. The goal of the type score is to act as a proxy for understanding what the cells of the file represent, thus capturing whether a dialect yields cells that "look like real data". To do this, the type score measures the proportion of cells in the parsed file that can be matched to a known data type.

The type score is based on a function h that takes a string as input and returns 1 if the string holds data from a known data type and 0 otherwise. The following types are considered known data types: empty strings, URLs, email addresses, numbers in various formats, times, percentages, currency values, alphanumeric strings, NaN values, dates, and combined date and time (see Appendix A for detailed descriptions). The function h is implemented using regular expression tests for each of these types and the tests are designed to be mutually exclusive. Then, the *type score* is defined as the proportion of cells in the parsing result with a known data type, that is,

$$T(\mathbf{x}, \hat{\boldsymbol{\theta}}) = \frac{1}{Z} \sum_{i=1}^{n} \sum_{j=1}^{L_i} h(\hat{c}_{i,j}), \tag{3}$$

where $Z = \sum_i L_i$ is the total number of cells in $\widehat{\mathbf{C}}$. Note that in Figure 2 the type score allows differentiating between two potential dialects that receive the same pattern score.

description see Appendix B.2.

4.3 Potential Dialects

We optimize the data consistency measure by computing it for each dialect in a set of potential dialects. As mentioned in Section 3 the theoretical number of potential dialects for a CSV file $\mathbf{x}$ with $\mathcal{X}$ unique characters is $|\mathcal{X}|^3$. It is important to prune this search space to reduce the computation time and to increase the accuracy of our method. Let $\mathcal{D}$, $\mathcal{Q}$, and $\mathcal{E}$ denote the set of potential delimiters, quote characters, and escape characters, respectively. The set of potential dialects $\Theta_{\mathbf{x}}$ is the product set of these three sets, pruned in two different ways to reduce its size (described below). As a preprocessing step to constructing $\Theta_{\mathbf{x}}$ all URLs in the file are removed to eliminate characters that only occur in URLs (e.g. `:` and `/`).

It is unnecessary to consider all characters in $\mathcal{X}$ as potential delimiters $\mathcal{D}$, since letters and numbers are not typically used as delimiters. The Unicode category of a character can be used to apply this reasoning in a way that is portable to files in different languages and encodings (The Unicode Consortium, 2018). All characters whose major Unicode category is `Letter` or `Number` are not considered as delimiters. Furthermore, open and close brackets (categories `Ps` and `Pe`) as well as control characters (categories `Cc` and `Co`) are not considered as delimiters. For the latter category we make an exception for the Tab character (`\t`). Finally, we eliminate four characters from $\mathcal{D}$ that are extremely unlikely to be delimiters, i.e. {`.`, `/`, `'`, `"`}, and we add the empty string ε to detect single-column CSV files.

The set of quote characters $\mathcal{Q}$ is

$$\mathcal{Q} = (\{\texttt{'}, \texttt{"}, \texttt{~}\} \cap \mathcal{X}) \cup \{\varepsilon\}. \tag{4}$$

This means that we consider the apostrophe, quotation mark, and tilde as quote character if they occur at least once in the file, and add the empty string ε for files without quote characters. For the set of escape characters, $\mathcal{E}$, we again use the Unicode category and allow characters from the "Punctuation, Other" (`Po`) category, but remove some common characters that fall in this category, and add the empty string, thus

$$\mathcal{E} = (\{x \in \mathcal{X} : \text{CAT}(x) = \texttt{Po}\} \cup \{\varepsilon\}) \setminus \{\texttt{!}, \texttt{?}, \texttt{"}, \texttt{'}, \texttt{.}, \texttt{,}, \texttt{;}, \texttt{:}, \%, \texttt{*}, \texttt{\&}, \texttt{\#}\}, \tag{5}$$

with $\text{CAT}(x)$ returning the Unicode category of a character x. The sets of blocked delimiters, included quote characters, and removed escape characters are considered features of our method and an implementation in software may choose different sets or allow the user to modify them.

To further prune the set of potential dialects $\Theta_{\mathbf{x}} = \mathcal{D} \times \mathcal{Q} \times \mathcal{E}$ we use two strategies. First, dialects where the escape character θ_e never precedes either the delimiter θ_d or the quote character θ_q are removed from consideration. Second, dialects where the delimiter θ_d always falls inside quoted segments do not need to be considered as these are equivalent to the dialect with the same θ_q and θ_e where the delimiter is the empty string. The effect of this careful construction of $\Theta_{\mathbf{x}}$ is that empirically we find that $|\Theta_{\mathbf{x}}| \approx 0.2\,|\mathcal{X}|$ on average, as opposed to the theoretical $|\mathcal{X}|^3$.

4.4 Data Consistency Measure

The complete data consistency measure for the CSV file $\mathbf{x}$ and a potential dialect $\hat{\boldsymbol{\theta}}$ is defined as the product of the pattern score and the type score

$$Q(\mathbf{x}, \hat{\boldsymbol{\theta}}) = P(\mathbf{x}, \hat{\boldsymbol{\theta}}) \cdot T(\mathbf{x}, \hat{\boldsymbol{\theta}}), \tag{6}$$

and the dialect is selected through search over the set of potential dialects $\Theta_{\mathbf{x}}$

$$\boldsymbol{\theta}^* = \underset{\hat{\boldsymbol{\theta}} \in \Theta_{\mathbf{x}}}{\arg\max}\, Q(\mathbf{x}, \hat{\boldsymbol{\theta}}). \tag{7}$$

There are two minor caveats to this search procedure, both caused by the fact that the type score is necessarily imperfect. Because it is not feasible to capture all possible types of data in the type score, the situation can arise where the type score returns a value of 0 for a potential dialect. When this happens for two dialects, it is desirable to use the pattern score to decide between the two. Thus, in our implementation we use $\max\{\beta, T(\mathbf{x}, \hat{\boldsymbol{\theta}})\}$ instead, with $\beta = 10^{-10}$ a small constant.

Additionally, the data consistency measure can be the same for multiple dialects. In some cases these ties can be broken reliably based on the parsing result for each dialect. For example, if the same consistency score is obtained for two dialects that only differ in the quote character and where the quote character is ε for one dialect, then this tie can be broken by checking if the parsing result is the same for both dialects. If it is, then the quote character has no effect and the correct dialect is the one where the quote character is ε. Similar tie-breaking rules can be formulated for the delimiter and the escape character. If it is not possible to break the tie reliably, our method returns no result. In a practical setting, this is preferred over returning an incorrect result.

Finally, the computational complexity of our method can be quantified as follows. Constructing the set of potential dialects requires the construction of $\mathcal{D}$, $\mathcal{Q}$, and $\mathcal{E}$, each of which can be done in $\mathcal{O}(|\mathcal{X}|)$ time. The two pruning strategies for removing unnecessary escape characters and delimiters have $\mathcal{O}(|\mathbf{x}|\,|\mathcal{D}|\,|\mathcal{Q}|)$ and $\mathcal{O}(|\mathbf{x}|\,|\mathcal{D}|\,|\mathcal{Q}|\,|\mathcal{E}|)$ complexity, respectively. This yields an $\mathcal{O}(|\mathcal{X}| + |\mathbf{x}|\,|\mathcal{D}|\,|\mathcal{Q}|\,|\mathcal{E}|) = \mathcal{O}(|\mathbf{x}|\,|\mathcal{X}|^3)$ worst-case complexity for constructing $\Theta_{\mathbf{x}}$. Computing the data consistency measure for each $\hat{\boldsymbol{\theta}} \in \Theta_{\mathbf{x}}$ requires the construction of the row patterns and the computation of the pattern score (both $\mathcal{O}(|\mathbf{x}|)$ operations) as well the computation of the type score, which requires parsing the file and checking the data type for each cell, an $\mathcal{O}(|\mathbf{x}|\,|\mathcal{T}|)$ operation with $\mathcal{T}$ the set of data types considered. Combining the above yields a worst-case complexity of $\mathcal{O}(|\mathbf{x}|\,|\mathcal{X}|^3 + |\mathbf{x}|\,|\mathcal{T}|\,|\Theta_{\mathbf{x}}|) = \mathcal{O}(|\mathbf{x}|\,|\mathcal{T}|\,|\mathcal{X}|^3)$ for our method. However, as mentioned above $|\Theta|_{\mathbf{x}} \approx 0.2\,|\mathcal{X}|$ in practice, so the realistic runtime will be on the order of $|\mathbf{x}|\,|\mathcal{T}|\,|\mathcal{X}|$. Furthermore, we can speed up the search procedure in an implementation of our method by keeping track of the maximum value of the data consistency measure, Q_{max}, and skipping the computation of the type score for a dialect $\hat{\boldsymbol{\theta}}$ with $P(\mathbf{x}, \hat{\boldsymbol{\theta}}) < Q_{max}$, since $T(\mathbf{x}, \hat{\boldsymbol{\theta}}) \in [0, 1]$ and therefore such a dialect will not improve Q_{max}.

5 Experiments

In this section we present the results of an extensive comparison study performed to evaluate our proposed method and existing alternatives. Since variability in CSV files is quite high and the number of potential CSV issues is large, an extensive study is necessary to thoroughly evaluate the robustness of each method. Moreover, since different groups of users apply different formats, it is important to consider more than one source of CSV files. In this section we present the details of two corpora of CSV files, describe how ground truth was obtained, and give brief descriptions of existing methods. Subsequently we present results of the comparison study that evaluates detection accuracy, runtime, and failure modes of the methods.

The method presented above was created using a *development set* of CSV files from two different corpora.[8] The experimental results below are based on an independent *test set* that was unknown to the authors during the development of the method. This split aims to avoid overfitting and provide a proper estimate of the accuracy of our method. To make our work transparent and reproducible, we release the full code and data set annotations through an online repository.[9]

5.1 Data

Data was collected from two sources: the UK government's open data portal (UKdata; `data.gov.uk`) and GitHub (`github.com`). These represent different groups of users (government employees vs. programmers) and we expect to find differences in both the format and the type of content of the CSV files. The CSV files in these corpora are considered to be representative for CSV files encountered in the real-world: data scientists often work with open data sources similar to the UKdata corpus, and often share data on platforms such as Github.com. Moreover, Github is considered a good source of *messy* CSV files, as the service places no restrictions on the format of CSV files that can be uploaded. Data was collected by web scraping in the period of May/June 2018. A development set was randomly sampled, with 3776 files from UKdata and 4536 files from GitHub. These were used to develop and fine-tune the consistency measure presented above, and in particular were used to develop the type detection engine.

An independent test set was sampled with 5000 files from each source. During development we noticed that the GitHub corpus often contained multiple files from the same code repository. Because these files usually have the same structure and dialect, they decrease the variability in the data. Therefore, a

[8] We refer to a "development set" instead of the more commonly used term "training set" because there is no explicit training of parameters in our method.

[9] See: `https://github.com/alan-turing-institute/CSV_Wrangling`. A reference implementation of our method is available at: `https://github.com/alan-turing-institute/CleverCSV`.

limit of one CSV file per GitHub repository was put in place for the test set. Thus we expect that the test set has greater variability and difficulty than the development set. Furthermore, repositories that were used for the development set were not used for the test set, ensuring independence between the two. Unfortunately a similar restriction could not be placed on the files from the UK data portal. It is worth emphasizing that the test set was not used in any way during the development of the method.

5.2 Ground Truth

To evaluate the detection method, ground truth for the dialect of the CSV files is needed. This was created through both automated and manual ways. The automated method is based on very strict functional tests that allow only simple CSV files with elementary cell contents. These automatic tests are sufficient to accurately determine the dialect of about a third of the CSV files, the remaining files were labelled manually. Files that could not reasonably be considered CSV files were removed from the test set (i.e. HTML, XML, or JSON files, or text files without tabular data). The same holds for files for which no objective ground truth could be established, such as files formatted similarly to the example in Figure 1(c). After filtering out these cases the test set contained 4386 files from GitHub.com and 4969 files from the UK government open data portal.

5.3 Alternatives

Since the dialect detection problem has not received much consideration in the literature, there are only a few alternative methods to compare to. We briefly present them here.

5.3.1 Python Sniffer

Python's built-in CSV module contains a so-called "Dialect Sniffer" that automatically detects the dialect of the file.[10] There are two methods used to detect the dialect. The first method is used when quote characters are present in the file and counts adjacent occurrence of a quote character and another character (the potential delimiter) and selects the pair that occurs most frequently. The second method is used when there are no quote characters in the file. In this case a frequency table is constructed that indicates how often a potential delimiter occurs and in how many rows (i.e. comma occurred x times in y rows). The character that most often matches the expected frequency is considered the delimiter, and a fallback list of preferred delimiters is used when a tie occurs. The method also tries to detect whether or not double quoting is used

[10] The dialect sniffer was developed by Clifford Wells for his Python-DSV package (Wells, 2002) and was incorporated into Python version 2.3.

within cells using a regular expression. During our research we found that this regular expression can run into "catastrophic backtracking" for certain CSV files. Therefore we place a timeout of two minutes on this detection method (normal operation never takes this long, so this restriction only captures this specific failure case). This method also detects when whitespace following the delimiter can be stripped. We do not include this in our method because the CSV specification states, "Spaces are considered part of a field and should not be ignored" (Shafranovich, 2005).

5.3.2 HypoParsr

HypoParsr (Döhmen et al., 2017) is the first dedicated CSV parser that takes the problem of dialect detection and messy CSV files into account.[11] The method uses a hierarchy of possible parser configurations and a set of heuristics to try to determine which configuration gives the best result. Unfortunately it is not possible to use the HypoParsr package to detect the dialect without running the full search that also includes header, table, and data type detection. Therefore, we run the complete program and extract the dialect from the outcome. However, this means that both the running time and any potential failure of the method are affected by subsequent parsing steps. This should be kept in mind when reviewing the results. As the method can be quite slow, we add a timeout of 10 minutes per file. Finally, the quote character in the dialect is not always reported faithfully in the final parsing result, since the underlying parser can strip quote characters automatically. We developed our own method to check what quote character was actually used during parsing.

5.3.3 Wrangler Suitability Score

In Guo et al. (2011) a table suitability metric is presented that balances consistency of cell types against the number of empty cells and cells with potential delimiters in them. We call this method "Suitability" in the tables. This can be used to detect the dialect of CSV files by selecting the dialect that does best on this metric. The suitability metric uses the concept of column type homogeneity, i.e. the sum of squares of the proportions of each data type in a column. Since the exact type detection method used in the paper is not available, we use our type detection method instead. The set of potential dialects is constructed in the same way as for our method, with the exception that the list of potential delimiters from Guo et al. (2011) is used, i.e. $\mathcal{D} = \{\texttt{,}, \texttt{:}, \texttt{|}, \texttt{\textbackslash t}\}$, and pruning of the search space is not applied as this is considered a feature of our method.

[11] An R package for HypoParsr exists, but this was removed from the R package repository. We nonetheless include the method in our experiments using the last available version.

				Proposed			
Property	HypoParsr	Sniffer	Suitability	Pattern	Type	No Tie	Full
Delimiter	87.48	86.82	65.41	92.61	88.33	91.38	**94.92**
Quotechar	82.90	92.36	44.60	95.23	90.10	93.80	**97.36**
Escapechar	87.96	94.37	74.85	97.95	96.26	95.44	**99.25**
Overall	80.60	85.45	38.19	90.99	83.61	90.61	**93.75**

(a) GitHub corpus

				Proposed			
Property	HypoParsr	Sniffer	Suitability	Pattern	Type	No Tie	Full
Delimiter	97.97	91.89	80.20	99.70	93.80	99.26	**99.82**
Quotechar	90.56	92.21	26.34	99.46	89.56	99.13	**99.70**
Escapechar	98.05	98.79	82.61	**100.00**	97.67	99.42	99.98
Overall	90.44	90.84	25.32	99.40	87.18	99.11	**99.68**

(b) UKdata corpus

Table 1 Accuracy (in %) of dialect detection for different methods on both corpora. Failure of a detection method is interpreted as an incorrect detection. "Pattern" and "Type" respectively indicate detection using only the pattern score or only the type score. "No Tie" indicates our method without tie-breaking.

5.3.4 Variations

In addition to our complete data consistency measure, we also consider variations to investigate the effect of each component. Thus, we include a method that only uses the pattern score and one that only uses the type score. We also include a variation that does not use tie-breaking.

5.4 Results

The methods are evaluated on the accuracy of the full dialect as well as on the accuracy of each component of the dialect. The performance on non-standard (messy) CSV files is evaluated as well as the runtime of each method. Finally, we investigate the failure cases of the methods.

5.4.1 Detection Accuracy

The accuracy of dialect detection is shown in Tables 1(a) and 1(b) respectively for the GitHub and UKdata corpora. We see that for both corpora and for all properties our full data consistency method outperforms all alternatives, with an exception for detecting the escape character in the UKdata corpus, where the pattern-only score function yields a marginally higher accuracy. It is furthermore apparent that the GitHub corpus of CSV files is more difficult than the UKdata corpus. This is reflected in the number of dialects observed in these corpora: 8 different dialects were found in the UKdata corpus vs. 33 in the GitHub corpus. We postulate that this difference is due to the na-

				Proposed			
	HypoParsr	Sniffer	Suitability	Pattern	Type	No Tie	Full
Standard (3502)	85.75	90.89	44.12	93.15	86.26	93.46	**95.80**
Messy (884)	60.18	63.91	14.71	82.47	73.08	79.30	**85.63**
Total (4386)	80.60	85.45	38.19	90.99	83.61	90.61	**93.75**

(a) GitHub corpus

				Proposed			
	HypoParsr	Sniffer	Suitability	Pattern	Type	No Tie	Full
Standard (4938)	90.46	90.91	25.05	99.43	87.30	99.15	**99.72**
Messy (31)	87.10	80.65	67.74	**93.55**	67.74	**93.55**	**93.55**
Total (4969)	90.44	90.84	25.32	99.40	87.18	99.11	**99.68**

(b) UKdata corpus

Table 2 Accuracy (in %) of dialect detection for different methods on both corpora separated by standard and messy CSV files. The numbers in parentheses represent the number of files in each category. Failure of a detection method is interpreted as an incorrect detection.

ture of the creators of these files. CSV files from the UK government open data portal are often created using spreadsheet applications and are therefore more likely to adhere to the CSV format (Shafranovich, 2005). On the other hand, the creators of files in the GitHub corpus are more likely to use non-standard or custom-made tools for creating CSV files and may therefore use different formatting conventions. Even though the files in the UKdata corpus can be considered more "regular" our method achieves a considerable increase in detection accuracy over standard approaches. The low accuracy of the table suitability metric from (Guo et al., 2011) shows that this is not an appropriate way to detect the dialect of CSV files.

The pattern score is almost as good as the full consistency measure, which confirms our earlier statement that it is the main driver of the method. However it is clear that the type score brings further improvement of the accuracy and that the type score alone does not suffice to accurately detect the dialect. The variant of our method that does not use tie-breaking yields a lower overall accuracy on both corpora, indicating the benefit of tie-breaking in our method.

5.4.2 Messy CSV Files

By separating the files into those that follow the CSV standard and those that do not, we can further illustrate how our method improves over existing methods. Files are considered "standard" when they use the comma as the delimiter, use either no quotes or the quotation mark as the quote character, and do not use an escape character (Shafranovich, 2005). Table 2 shows the accuracy of dialect detection for standard and non-standard (*messy*) files. Our method improves over existing methods on both types of files and achieves an average improvement on messy files of 21.4% over the Python Sniffer.

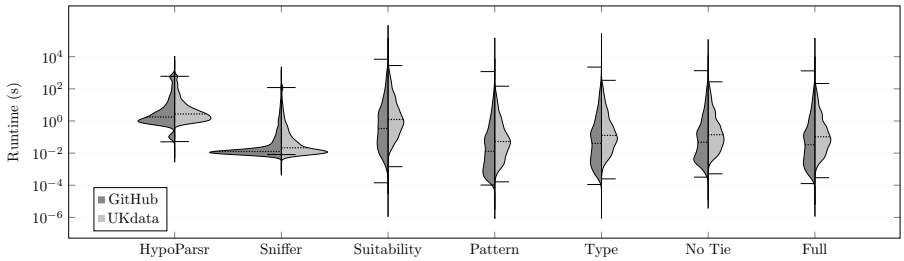

Fig. 3 Runtime violin plots for both corpora. The whiskers show the minimum and maximum values and the dashed lines show the median. See the note on HypoParsr in the main text.

5.4.3 Runtime

Figure 3 shows violin plots of the runtime for each method for both corpora. Although HypoParsr is the slowest detection method, this is not completely accurate because the reported runtime is the time needed for the entire parsing process instead of only the dialect detection. The Python dialect sniffer is the fastest method, which can most likely be attributed to its simplicity in comparison to the other methods. Finally, all variations of our method have similar runtime characteristics and slightly outperform the Wrangler suitability metric. We note that our method has not been explicitly optimised for speed, and there are implementation improvements that can be made in this respect. However, the mean of the computation time for our method lies well below one second, which is acceptable in practice.

5.4.4 Failure

Table 3 investigates the failure cases of the methods and shows the percentage of files where no result was obtained and where an incorrect result was obtained. Note that a method can return no result due to a timeout or an exception in the code (for the Python Sniffer or HypoParsr), or due to a tie in the scoring measure (for the Wrangler suitability metric or our method). When the correct dialect can not be detected, it is more desirable in practice to return no result than to return an incorrect result, as the former gives a signal that user intervention is needed whereas the latter does not. The table shows that compared to existing methods the proposed method has the smallest proportion of files in both failure cases. However, the proportion of files where an incorrect result is returned is nonzero and future work may focus on addressing this issue.

As Table 1(a) shows, an incorrect detection in our method occurs mostly due to an incorrect detection of the delimiter. Analysing these failure cases in more detail reveals that an incorrect detection occurred mostly because our method predicted the space instead of the comma as the delimiter, due to regular patterns of whitespace occurring in text cells. Other files had the

				Proposed			
	HypoParsr	Sniffer	Suitability	Pattern	Type	No Tie	Full
No Result	10.12	4.90	22.96	1.69	1.30	4.24	**0.30**
Incorrect	9.28	9.64	38.85	7.32	15.09	**5.15**	5.95
Correct	80.60	85.45	38.19	90.99	83.61	90.61	**93.75**

(a) GitHub corpus

				Proposed			
	HypoParsr	Sniffer	Suitability	Pattern	Type	No Tie	Full
No Result	1.85	1.21	16.72	**0.00**	0.04	0.56	**0.00**
Incorrect	7.71	7.95	57.96	0.60	12.78	**0.32**	**0.32**
Correct	90.44	90.84	25.32	99.40	87.18	99.11	**99.68**

(b) UKdata corpus

Table 3 Percentage of files where the methods returned no result, an incorrect result, or the correct result for dialect detection. In practice it is preferable to return no result instead of an incorrect result to signal the need for user intervention.

comma as the true delimiter, but had only one column of data. In these cases the true comma delimiter could be deduced by the human annotator from a header or because certain cells that contained the comma were quoted, but this type of reasoning is not captured by the data consistency measure. In other failure cases the pattern score predicted the correct delimiter, but the type score gave a low value, resulting in a low value of the data consistency measure. Some of these failure cases can certainly be addressed by improving the type detection procedure.

6 Discussion

A major challenge for today's data scientists is the inordinate amount of time spent on preparing data for analysis. One of the difficulties they face is importing data from messy CSV files that usually require manual inspection and reformatting before the data can be loaded from the file. In this paper we have presented a method for automatic dialect detection of CSV files that achieves high accuracy on a large corpus of real-world examples, and considerably improves on the state of the art for messy CSV files. This represents an important step toward automatically loading structured tabular data from messy sources. Thus, there is significant potential for implementation of our method in various software packages and programming languages, and it could benefit existing tools for data wrangling such as FlashExtract (Le and Gulwani, 2014), Trifacta/Wrangler (Kandel et al., 2011), and Wrattler (Petricek et al., 2018). This will enable data scientists to spend less time on mundane data wrangling issues and more time on extracting valuable knowledge from their data.

The proposed data consistency measure is inspired by the way a human analyst would solve dialect detection. The consistency measure emphasizes a regular pattern of cells in the rows and favors dialects that yield a greater proportion of cells with identifiable data types. While we apply the consistency measure only to CSV dialect detection in this paper, it is likely to have applications outside this domain. For instance, it could be used for identifying unstructured tables in HTML documents or free text, or for locating the tables within CSV or spreadsheet files. The idea of mimicking the approach that a human takes could be used for other problems in data wrangling. Finally, our framework of a formatter that converts tabular data to a file using a vector of parameters can be applied to other data formats (e.g. binary files).

Although our method achieves high detection accuracy on both datasets, it does not achieve perfect accuracy on *messy* CSV files. On these files we improve the state of the art by 21.4%, but our method fails to accurately detect the dialect on 14% of messy files. This is indicative of the difficulty of the dialect detection task: the many variations of CSV files make it hard to develop a method that works well in general. A weakness of our method is that it can fail when a CSV file contains many cells of an unknown type, or when an incorrect dialect yields a comparable pattern score but a higher type score than the correct dialect. Since the type detection associates a known type with only 91.6% of cells on average, some of the failures of our method can be addressed by expanding the set of data types. For example, fields with a list of numbers surrounded by `[` and `]` can confuse the type detection when the comma is used within this list, as do columns of MAC addresses due to the presence of the `:` character. Moreover, single-column files can cause confusion, especially when the column consists of natural text that includes spaces. This could be addressed by designing a pre-test based only on the type score that specifically identifies single-column files. Finally, single-row files are often misidentified due to the presence of multiple potential delimiters and the lack of sufficient information for the pattern score. These cases are straightforward to detect and refer back to the user for manual inspection. We consider these improvements topics for future research.

Another topic for future research is the further pruning of the set of potential dialects to remove false positives. One possibility would be to use co-occurrence of characters similar to the Python Sniffer method to eliminate incorrect dialects. This was not included in the current work because the co-occurrence method of Sniffer often fails. Several opportunities exist for speeding up our method, including pruning the search space of parameters more agressively, and simplifying the type detection method.

Acknowledgements

The authors would like to acknowledge the funding provided by the UK Government's Defence & Security Programme in support of the Alan Turing Institute. The authors thank Chris Williams for useful discussions.

References

Arnold VI (2003) Catastrophe Theory. Springer Science & Business Media

Codd EF (1970) A relational model of data for large shared data banks. Communications of the ACM 13(6):377–387

Crockford D (2006) The application/json media type for Javascript Object Notation (JSON). Tech. Rep. RFC 4627, Internet Requests for Comments

Crowdflower (2016) Data science report. URL `visit.figure-eight.com/data-science-report.html`, accessed 2018-11-19

Dasu T, Johnson T (2003) Exploratory data mining and data cleaning, vol 479. John Wiley & Sons

Döhmen T, Mühleisen H, Boncz P (2017) Multi-hypothesis CSV parsing. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management, ACM, pp 16:1–16:12

Eberius J, Werner C, Thiele M, Braunschweig K, Dannecker L, Lehner W (2013) DeExcelerator: a framework for extracting relational data from partially structured documents. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, ACM, pp 2477–2480

Evans C (2001) YAML draft 0.1. URL `yaml.org`, accessed 2018-11-19

Fisher K, Walker D, Zhu KQ, White P (2008) From dirt to shovels: fully automatic tool generation from ad hoc data. In: ACM SIGPLAN Notices, ACM, vol 43, pp 421–434

Frictionless Data (2017) CSV dialect specification. URL `frictionlessdata.io/specs/csv-dialect`, accessed 2018-11-19

Guo PJ, Kandel S, Hellerstein JM, Heer J (2011) Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In: Proceedings of the 24th annual ACM Symposium on User Interface Software and Technology, ACM, pp 65–74

Kaggle (2017) The state of data science & machine learning. URL `www.kaggle.com/surveys/2017`, accessed 2018-09-27

Kandel S, Paepcke A, Hellerstein J, Heer J (2011) Wrangler: Interactive visual specification of data transformation scripts. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, pp 3363–3372

Kleene SC (1956) Representation of events in nerve nets and finite automata. In: Shannon CE, McCarthy J (eds) Automata Studies, Princeton University Press

Koci E, Thiele M, Romero Moral Ó, Lehner W (2016) A machine learning approach for layout inference in spreadsheets. In: IC3K 2016: Proceedings of the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management: volume 1: KDIR, pp 77–88

Le V, Gulwani S (2014) FlashExtract: a framework for data extraction by examples. In: ACM SIGPLAN Notices, ACM, vol 49, pp 542–553

Lohr S (2014) For big-data scientists, "janitor work" is key hurdle to insights. The New York Times URL `www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html`, accessed 2018-11-22

Mitlöhner J, Neumaier S, Umbrich J, Polleres A (2016) Characteristics of open data CSV files. In: 2nd International Conference on Open and Big Data (OBD), pp 72–79

Ng HT, Lim CY, Koo JLT (1999) Learning to recognize tables in free text. In: Proceedings of the 37th annual meeting of the Association for Computational Linguistics, ACL, pp 443–450

Petricek T, Geddes J, Sutton C (2018) Wrattler: Reproducible, live and polyglot notebooks. In: 10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)

Pinto D, McCallum A, Wei X, Croft WB (2003) Table extraction using conditional random fields. In: Proceedings of the 26th annual international ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, pp 235–242

Raymond ES (2003) The art of Unix programming. Addison-Wesley Professional

Rovegno J, Fenner M (2015) CSVY: YAML frontmatter for CSV file format. URL `csvy.org`, accessed 2018-11-19

Shafranovich Y (2005) Common format and MIME type for comma-separated values (CSV) files. Tech. Rep. RFC 4180, Internet Requests for Comments
Sutton C, Hobson T, Geddes J, Caruana R (2018) Data Diff: Interpretable, executable summaries of changes in distributions for data wrangling. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, pp 2279–2288
Tennison J (2016) CSV on the web: A primer. Tech. rep., W3C
Tennison J, Kellogg G (2015) Metadata vocabulary for tabular data. Tech. rep., W3C
The Unicode Consortium (2018) The Unicode Standard, Version 11.0.0
Wells C (2002) Python-DSV. URL `python-dsv.sourceforge.net`, accessed 2018-11-08
Wickham H (2014) Tidy data. Journal of Statistical Software 59(10):1–23

A Data Type Detection

As mentioned in the main text, we use a regular expression based type detection engine. Below is a brief overview of the different types that we consider and the detection method we use for that type. The order of the types corresponds to the order in which we evaluate the type tests, and we stop when a matching type is found. The complete code is available in an online repository.[12]

Empty Strings

Empty strings are considered a known type.

URLs and Email Addresses

For this we use two separate regular expressions.

Numbers

We consider two different regular expressions for numbers. First, we consider numbers that use "digit grouping", i.e. numbers that use a period or comma to separate groups of thousands. In this case we allow numbers with a comma or period as thousands separator and allow for using a comma or period as radix point, respectively. Numbers in this form can not have E-notation, but can have a leading sign symbol. The second regular expression captures the numbers that do not use digit grouping. These numbers can have a leading sign symbol (`+` or `-`), use a comma or period as radix point, and can use E-notation (i.e. `123e10`). The exponent in the E-notation can have a sign symbol as well.

Time

Times are allowed in `HH:MM:SS`, `HH:MM`, and `H:MM` format. The AM/PM quantifiers are not included.

Percentage

This corresponds to a number combined with the `%` symbol.

[12] See: `https://github.com/alan-turing-institute/CSV_Wrangling`.

Currency

A currency value is a number preceded by a symbol from the Unicode `Sc` category (The Unicode Consortium, 2018).

Alphanumeric

An alphanumeric string can follow two alternatives. The first alternative consists of first one or more number characters, then one or more letter characters, and then zero or more numbers, letters, or special characters. An example of this is the string `3 degrees`. The second alternative first has one or more letter characters and then allows for zero or more numbers, letters, or special characters. An example of this is the string `NW1 2DB`. In both alternatives the allowed special characters are space, period, exclamation and question mark, and parentheses, including their international variants.

N/A

While `nan` or `NaN` are accepted in the alphanumeric test, we add here a separate test that considers `n/a` and `N/A`.

Dates

Dates are strings that are not numbers and that belong to one of forty different date formats. These date formats allow for the formats `(YY)YYx(M)Mx(D)D`, `(D)Dx(M)Mx(YY)YY`, `(M)Mx(D)Dx(YY)YY` where `x` is a separator (dash, period, or space) and parts within parentheses can optionally be omitted. Additionally, the Chinese/Japanese date format and the Korean date format are included.

Combined date and time

These are formats for joint date and time descriptions. For these formats we consider `<date> <time>` and `<date>T<time>` as well as those with a time zone offset appended.

B Algorithm Details

B.1 Parser

The code we use for our CSV parser borrows heavily from the CSV parser in the Python standard library, but differs in a few small but significant ways. First, our parser only interprets the escape character if it proceeds the delimiter, quote character, or itself. In any other case the escape character serves no purpose and is treated as any other character and is not dropped. Second, our parser only strips quotes from cells if they surround the entire cell, not if they occur within cells. This makes the parser more robust against misspecified quote characters. Finally, when we are in a quoted cell we automatically detect double quoting by looking ahead whenever we detect a quote, and checking if the next character is *also* a quote character. This enables us to drop double quoting from our dialect and only marginally affects the complexity of the code.

B.2 Row Patterns

The full description of how row patterns are constructed is as follows:

1. Create an empty string **s**.
2. Iterate over the characters of the CSV file **x**,
   (a) If the character is a carriage return or newline, append `R` to **s**.
   (b) If the character is the delimiter θ_d, append `D` to **s**.
   (c) If the character is the quote character θ_q, append `Q` to **s**.
   (d) If the character is the escape character θ_e, treat the *next* character as a normal character if it is the delimiter, quote character, or escape character and append `C` to **s**. If is not such a character, append `CC`.
   (e) For any other character, append `C` to **s**.
3. Iterate over the characters of the string **s**,
   (a) If the character is `Q`, mark the current position as the start or end of a quoted segment unless the *next* character is also `Q` (i.e. double quotes).
   (b) Replace all quoted segments by `C`.
4. Fill in empty cells by:
   (a) Replacing all occurrences of `DD` in **s** by `DCD`.
   (b) Replacing all occurrences of `DR` in **s** by `DCR`.
   (c) Replacing all occurrences of `RD` in **s** by `RCD`.
   (d) Inserting `C` at the start of **s** if it begins with `D`.
   (e) Appending `C` at the end of **s** if it ends with `D`.
5. Reduce consecutive occurrences of `C` to a single `C`.
6. Strip a trailing `R` from **s** if present.
7. Split **s** on `R`. The resulting substrings are the row patterns.