Learning constraints in spreadsheets and tabular data

***Abstract*—Spreadsheets, CSV (comma separated value) files and other tabular data representations are in wide use today. However, modeling, maintaining and discovering formulas in tabular data and spreadsheets can be time consuming and error-prone. In this work, we investigate the automatic discovery of constraints (formulas and relations) in raw tabular data. We see multiple promising applications for this technique, e.g. in rediscovering constraints, auto-completion and error checking. Our method takes inspiration from inductive logic programming, constraint learning and constraint satisfaction. Common spreadsheet functions are represented as predicates whose arguments must satisfy a number of constraints. Constraint learning techniques are used to identify the predicates that hold between blocks of rows and columns of the tables. We show that our approach is able to accurately discover constraints in spreadsheets from various sources.**

I. Introduction

Millions of people across the world use spreadsheets every day. The tabular representation of the data is often intuitive, and the programming of functions in individual cells is quickly learned. However, large and complex sheets (possibly with multiple tables and relations between them) can be hard to handle. Many end-users lack the understanding of the underlying structures and dependencies in such sheets and the data they contain. This is especially the case when spreadsheets have been exported from other software such as Enterprise Resource Planning (ERP) systems. In this case, often a comma-separated values (CSV) format is used meaning that all formulas are lost, including inter-sheet formulas and relations. Even in manually created spreadsheets, it can be challenging to be consistent and correct with formulas across big tables. For example the influential Reinhart-Rogoff economical paper "Growth in a Time of Debt" had some of its claims contested [8], after an investigation of the used Excel sheets was shown to contain mistakes in formulae.

In this paper, we investigate whether learning techniques can be used to infer constraints (formulas and other relations) from raw spreadsheet data. This is a new and unconventional machine learning problem. Consider the example in Figure 1a, where header names such as **Total** and **Rank** already suggest the usage of spreadsheet operations such as *SUM* or *RANK*. Looking at the first row in table T_2 and the data in table T_1, it is clear that the values in this row were computed by summing the values in the corresponding columns (**1st Quarter** through **Total**) of T_1. Examining T_1 also shows that computations are not only performed column-wise but row-wise as well: the cells in the column **Total** are obtained by summing the rows of the previous four columns. This provides a flavor of how this problem is different from standard mining and learning settings, where the data is just in rows and variables are in columns. Here everything is mixed. The data is relational on the one hand, since we have multiple tables with relationships between them (e.g. foreign-keys). On the other hand, the data is most often mixed, e.g. textual and numeric. Therefore, understanding and learning constraints in tabular data is a novel and challenging problem for machine learning.

The question that we answer in this paper is: is it possible to discover or reconstruct constraints in flat tabular spreadsheet data? To answer this question we contribute a general-purpose method and system, named *TaCLe* (from: Tabular Constraint Learner, pronounced "tackle"), for discovering row-wise and column-wise constraints. It operates directly on (headerless) tables of a spreadsheet in an unsupervised setting, that is, it reasons on raw tabular data directly with no example constraint instantiations given. We demonstrate the utility of our approach in an experimental evaluation. Moreover, we sketch additional application scenario's such as autocompletion and error detection.

The approach that we introduce borrows techniques from logical and relational learning [4], where the discovery of clausal constraints has been studied in inductive logic programming [13, 9]; and from constraint learning, where several approaches to learning sets of constraints have been developed in a constraint programming setting [2, 3, 1]; and from mining in databases, where inductive techniques have been applied to discover constraints (such as functional and multi-valued dependencies) in relational databases [14]. It also builds on work on program synthesis, in particular, on Flashfill [7], where the definition of a function (over textual cells only) is learned in spreadsheet data from very few examples. Our approach contrasts with these in that it focuses on learning both column- and row-constraints across tables on integer, floating point as well as textual data.

This paper is organized as follows. Section II introduces concepts relevant to our approach and our problem statement. Section III presents our approach. Section IV presents the evaluation of the approach. Section V shows how our system can be used for applications. Section VI presents a more detailed discussion of the related work. Section VII provides conclusions.

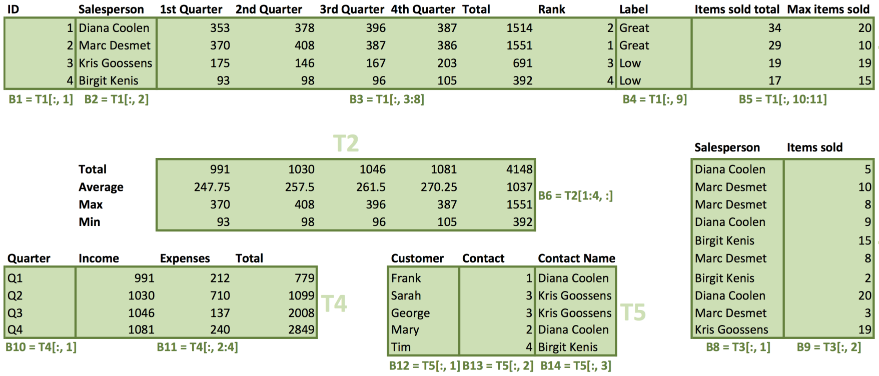
II. Formalization

Our goal is to automatically discover constraints between the rows and the columns of tables in a spreadsheet. This is applicable not just to data from spreadsheets, but to any data in tabular form, hence the name.

We first introduce some terminology and the concept of a *constraint template*, after which we define the problem and make some additional considerations.

A. Terminology

Spreadsheets and tabular data may conceptually consist of multiple tables, such as in Figure 1a. Note that a table can contain a header; however, we wish to reason over entire rows



ID	Salesperson	1st Quarter	2nd Quarter	3rd Quarter	4th Quarter	Total	Rank	Label	Items sold total	Max items sold
1	Diana Coolen	353	378	396	387	1514	2	Great	34	20
2	Marc Desmet	370	408	387	386	1551	1	Great	29	10
3	Kris Goossens	175	146	167	203	691	3	Low	19	19
4	Birgit Kenis	93	98	96	105	392	4	Low	17	15

Total	991	1030	1046	1081	4148
Average	247.75	257.5	261.5	270.25	1037
Max	370	408	396	387	1551
Min	93	98	96	105	392

Salesperson	Items sold
Diana Coolen	5
Marc Desmet	10
Marc Desmet	8
Diana Coolen	9
Birgit Kenis	15
Marc Desmet	8
Birgit Kenis	2
Diana Coolen	20
Marc Desmet	3
Kris Goossens	19

Quarter	Income	Expenses	Total
Q1	991	212	779
Q2	1030	710	1099
Q3	1046	137	2008
Q4	1081	240	2849

Customer	Contact	Contact Name
Frank	1	Diana Coolen
Sarah	3	Kris Goossens
George	3	Kris Goossens
Mary	2	Diana Coolen
Tim	4	Birgit Kenis

(a) Example spreadsheet (black words and numbers only). Green background indicates headerless tables, dark borders indicate maximal type-consistent blocks. Most tables only contain type-consistent columns while table T_2 also contains type-consistent rows. However, due to the location of the header of T_2 we can infer that it consists of row data. This example was created by combining several Excel sheets from several exercise sessions based on [15].

$$\mathit{SERIES}(T_1[:,1])$$
$$T_1[:,1] = \mathit{RANK}(T_1[:,5])^*$$
$$T_1[:,1] = \mathit{RANK}(T_1[:,6])^*$$
$$T_1[:,1] = \mathit{RANK}(T_1[:,10])^*$$
$$T_1[:,8] = \mathit{RANK}(T_1[:,7])$$
$$T_1[:,8] = \mathit{RANK}(T_1[:,3])^*$$
$$T_1[:,8] = \mathit{RANK}(T_1[:,4])^*$$
$$T_1[:,7] = \mathit{SUM}_{row}(T_1[:,3\text{:}6])$$
$$T_1[:,10] = \mathit{SUMIF}(T_3[:,1], T_1[:,2], T_3[:,2])$$
$$T_1[:,11] = \mathit{MAXIF}(T_3[:,1], T_1[:,2], T_3[:,2])$$
$$T_2[1,:] = \mathit{SUM}_{col}(T_1[:,3\text{:}7])$$
$$T_2[2,:] = \mathit{AVERAGE}_{col}(T_1[:,3\text{:}7])$$
$$T_2[3,:] = \mathit{MAX}_{col}(T_1[:,3\text{:}7]),$$
$$T_2[4,:] = \mathit{MIN}_{col}(T_1[:,3\text{:}7])$$
$$T_4[:,2] = \mathit{SUM}_{col}(T_1[:,3\text{:}6])$$
$$T_4[:,4] = PREV(T_4[:,4]) + T_4[:,2] - T_4[:,3]$$
$$T_5[:,2] = \mathit{LOOKUP}(T_5[:,3], T_1[:,2], T_1[:,1])^*$$
$$T_5[:,3] = \mathit{LOOKUP}(T_5[:,2], T_1[:,1], T_1[:,2])$$

(b) Constraints learned for the example (left), except for *ALLDIFFERENT* (18), *PERMUTATION* (2) and *FOREIGNKEY* (5). Constraints marked with $*$ were not present in the original document

Fig. 1: Running example

and columns of data, and hence we will consider **headerless tables** only.

Formally, a (headerless) table is an $n \times m$ matrix. Each entry is called a *cell*. A cell has a **type**, which can be numeric or textual. We further distinguish numeric types in subtypes: integer and float. We also consider *None* as a special type when a cell is empty; *None* is a subtype of all other types.

A row or a column is **type-consistent** if all cells in that row or column are of the same base type, i.e. numeric or textual. We will use notation $T[a,:]$ to refer to the a-th row of table T and similarly $T[:,a]$ for the a-th column. For example, in Figure 1a, $T_1[:,1] = [1,2,3,4]$ and $T_3[1,:] = ['\text{Diana Coolen}', 5]$. $T_3[1,:]$ is not type-consistent while $T_1[:,1]$ is.

The most important concept is that of a **block**.

Definition 1. A **block** has to satisfy three conditions: 1) it contains only entire rows or entire columns of a single headerless table; 2) it is contiguous; and 3) it is type-consistent. The rows or columns have to be contiguous in the original table meaning that they must visually form a block in the table; and each of the rows/columns has to be of the same type. If a block contains only rows we say it has *row-orientation*, if only columns, *column-orientation*.

In line with this definition, we can use the following notation to refer to blocks: $B = T[a\text{:}b,:]$ for a row-oriented block containing rows a to b in table T; and similarly $B = T[:,a\text{:}b]$ for a column-oriented block. We will refer to the *vectors* of a block when we wish to refer to its rows/columns independently of their orientation.

A block has the following properties:

- *type*: a block is type-consistent, so it has one type
- *table*: the table that the block belongs to
- *orientation*: either row-oriented or column-oriented
- *size*: the number of vectors a block contains
- *length*: the length of its vectors; as all vectors are from the same table, they always have the same length.
- *rows*: the number of rows in the block (in row-oriented blocks this is equivalent to the size)
- *columns*: the number of columns in the block (in row-oriented blocks this is equivalent to the length)

Example 1. Consider the (headerless) table T_1 in Figure 1a. Its rows are not type consistent (i.e. they contain both numeric and textual data). However, the table can be partitioned into five column-oriented blocks B_1, B_2, B_3, B_4, B_5, as shown in the figure ($B_1 = T_1[:,1]$, $B_2 = T_1[:,2]$, $B_3 = T_1[:,3\text{:}8]$, $\ldots$).

Definition 2. Block containment $\sqsubseteq$. A block B' is contained in a block B, $B' \sqsubseteq B$, iff both are valid blocks (contiguous, type consistent) with the same orientation and table, and each of the vectors in B' is also in B. For row-oriented blocks: $B' \sqsubseteq B \Leftrightarrow B = T[a\text{:}b,:] \wedge B' = T[a'\text{:}b',:] \wedge a \leq a' \wedge b' \leq b$ and similarly for column-oriented blocks.

We will sometimes write that B' is a *subblock* of B or that B is a *superblock* of B'. An example of block containment is $T_1[:,3\text{:}6] \sqsubseteq T_1[:,3\text{:}8]$, which contains the sales numbers of all employees for the four quarters.

B. Constraint templates

The goal is to learn constraints over blocks in the data. The knowledge needed to learn a constraint is expressed through *constraint templates*. A *constraint template* s is a triple $t =$ *(Syntax, Signature, Definition)*:

TABLE I: Overview of constraint templates implemented in *TaCLe*. Subgroups in normal font have to be vectors, whereas subgroups in **bold** may contain more than one vector; $\mathit{discrete}(x)$ is a shortcut for: $\mathit{textual}(x) \vee \mathit{numeric}(x)$. Templates marked with $*$ are *structural*, i.e. they are not functional and cannot be implemented in most spreadsheet software. Templates marked with † are *aggregate* templates, the table shows *SUM* but *TaCLe* also supports *MAX*, *MIN*, *AVERAGE*, *PRODUCT* and *COUNT*.

Syntax	Signature	Definition
$\mathit{ALLDIFFERENT}(B_x)^*$	$\mathit{discrete}(B_x)$	All values in B_x are different. $i \neq j$: $B_x[i] \neq B_x[j]$
$\mathit{PERMUTATION}(B_x)^*$	$\mathit{numeric}(B_x)$, $\mathit{ALLDIFFERENT}(B_x)$	The values in B_x are a permutation of the numbers 1 through $\mathit{length}(B_x)$.
$\mathit{SERIES}(B_x)$	$\mathit{integer}(B_x)$ and $\mathit{PERMUTATION}(B_x)$	$B_x[1] = 1$ and $B_x[i] = B_x[i-1] + 1$.
$\mathit{FOREIGNKEY}(B_{fk}, B_{pk})^*$	B_{fk} and B_{pk} are *discrete*; they must have different *table*s but the same *type*; and $\mathit{ALLDIFFERENT}(B_{pk})$	Every value in B_{fk} also exist in B_{pk}
$B_r = \mathit{LOOKUP}(B_{fk}, B_{pk}, B_{val})$	B_{fk} and B_{pk} are *discrete*; arguments $\{B_{fk}, B_r\}$ and $\{B_{pk}, B_{val}\}$ within the same set have the same *length*, *table* and *orientation*; B_r and B_{val} have the same type; and $\mathit{FOREIGNKEY}(B_{fk}, B_{pk})$.	$B_r[i] = B_{val}[j]$ where $B_{pk}[j] = B_{fk}[i]$
$B_r = \mathit{LOOKUP}_{\mathit{fuzzy}}(B_{fk}, B_{pk}, B_{val})$	*Same as lookup*	$B_r[i] = B_{val}[j]$ where $B_{pk}[j] \leq B_{fk}[i]$, j maximal
$B_r = B_1 \times \mathit{LOOKUP}(B_{fk}, B_{pk}, B_{val})$	Arguments $\{B_r, B_1, B_{fk}\}$ are *numeric*, arguments $\{B_{pk}, B_{val}\}$ are *discrete* and within both sets all arguments have the same *length*, *table* and *orientation*; also $\mathit{FOREIGNKEY}(B_{fk}, B_{pk})$.	$B_r[i] = B_1[i] \times \mathit{LOOKUP}(B_{fk}, B_{pk}, B_{val})[i]$.
$B_r = B_1 \times B_2$	Arguments $\{B_r, B_1, B_2\}$ are all *numeric* and have the same *length*	$B_r[i] = B_1[i] \times B_2[i]$.
$B_r = B_1 - B_2$	Arguments $\{B_r, B_1, B_2\}$ are all *numeric* and have the same *length* and *orientation*	$B_r[i] = B_1[i] - B_2[i]$.
$B_r = \frac{B_1 - B_2}{B_2}$	*Same as* $B_r = B_1 - B_2$	$B_r[i] = (B_1[i] - B_2[i])/B_2[i]$.
$B_r = \mathit{PROJECT}(\mathbf{B}_x)$	Arguments $\{B_r, \mathbf{B}_x\}$ all have the same *length*, *orientation*, *table* and *type*; $\mathbf{B}_x$ contains at least 2 vectors; and $B_r = \mathit{SUM}_{\mathit{orientation}(\mathbf{B}_x)}(\mathbf{B}_x)$	At every position i in 1 through $\mathit{length}(B_r)$ there is exactly one vector v in $\mathbf{B}_x$ such that $v[i]$ is a non-blank value, then $v[i] = B_r[i]$.
$B_r = \mathit{RANK}(B_x)$	$\mathit{integer}(B_r)$; $\mathit{numeric}(B_x)$; and $\mathit{length}(B_r) = \mathit{length}(B_x)$	The values in B_r represent the rank (from largest to smallest) of the values in B_x (including ties)
$B_r = \mathit{PREV}(B_r) + B_{pos} - B_{neg}$	Arguments $\{B_r, B_{pos}, B_{neg}\}$ are all *numeric* and all have the same *length*, which is at least 2	$B_r[i] = B_r[i-1] + B_{pos}[i] - B_{neg}[i]$.
$B_r = \mathit{SUM}_{\mathit{row}}(\mathbf{B}_x)^\dagger$	B_r and $\mathbf{B}_x$ are *numeric*; $\mathit{columns}(\mathbf{B}_x) \geq 2$; and $\mathit{rows}(\mathbf{B}_x) = \mathit{length}(B_r)$	$B_r[i] = \sum_{j=1}^{\mathit{columns}(\mathbf{B}_x)} \mathit{row}(i, \mathbf{B}_x)[j]$
$B_r = \mathit{SUM}_{\mathit{col}}(\mathbf{B}_x)^\dagger$	B_r and $\mathbf{B}_x$ are *numeric*; $\mathit{rows}(\mathbf{B}_x) \geq 2$; and $\mathit{columns}(\mathbf{B}_x) = \mathit{length}(B_r)$	$B_r[i] = \sum_{j=1}^{\mathit{rows}(\mathbf{B}_x)} \mathit{column}(i, \mathbf{B}_x)[j]$
$B_r = \mathit{SUMIF}(B_{fk}, B_{pk}, B_{val})^\dagger$	B_{fk}, B_{pk} are *discrete*; B_r, B_{val} are *numeric*; within the sets $\{B_{val}, B_{fk}\}$ and $\{B_{pk}, B_r\}$ arguments have the same *length* and *orientation*; B_{fk} and B_{val} have the same *table*; B_{fk} and B_{pk} must have different *table*s but the same *type*; and $\mathit{ALLDIFFERENT}(B_{pk})$	$B_r[i] = \sum_{j=1}^{\mathit{length}(B_{val})} \begin{cases} B_{val}[j] & \text{if } B_{fk}[j] = B_{pk}[i] \\ 0 & \text{otherwise} \end{cases}$
$B_r = \mathit{SUMPRODUCT}(B_1, B_2)$	Arguments $\{B_r, B_1, B_2\}$ are *numeric*; $\mathit{length}(B_1) = \mathit{length}(B_2) \geq 2$; and $\mathit{rows}(B_r) = \mathit{columns}(B_r) = 1$	$B_r[1] = \sum_{j=1}^{\mathit{length}(B_1)} B_1[j] \times B_2[j]$.

- *Syntax* specifies the syntactic form of the constraint $s(B_1, ..., B_n)$, that is, the name of the template together with n abstract arguments B_i. Thus, a constraint is viewed as a relation or predicate of arity n in first order logic. Note that a function $B_r = f(B_1, ..., B_n)$ can be represented with the $(n+1)$-ary predicate $s_f(B_r, B_1, ..., B_n)$. Each argument will have to be instantiated with a block.
- *Signature* defines the requirements that the arguments of the predicate must satisfy. This can concern properties of individual blocks as well as relations between properties of arguments, for example that the corresponding blocks must belong to the same table or have equal length. In terms of logical and relational learning [4], the Signature is known as the *bias* of the learner, it specifies when the arguments of a constraint are well-formed. We can capture this bias for a template s using a predicate Sig_s.
- *Definition* is the actual definition of the constraint that specifies when the constraint holds. Given an assignment of blocks to its arguments, it can be used to verify whether the constraint is satisfied or not by the actual data present in the blocks. In logical and relational learning this is known as the background knowledge. We introduce the

predicate Def_s to capture this background knowledge for a template s.

Example 2. The constraint templates implemented in *TaCLe* are defined in Table I. A non-trivial example is, for example, the constraint template for the row-based sum:

- Syntax: $B_r = \mathit{SUM}_{row}(\mathbf{B}_x)$, for arguments B_r and $\mathbf{B}_x$.
- Signature: B_r has to be a single vector ($\mathit{size} = 1$) while $\mathbf{B}_x$ can be a block ($\mathit{size} >= 1$), which can be derived from the use of a normal or **bold** font. Both blocks have to be numeric. This constraint is orientation-specific, so it requires that the number of rows in $\mathbf{B}_x$ equals the length of B_r. Moreover, we add to the bias that the number of columns to sum over is at least 2.
- Definition: each value in the vector B_r is obtained by summing over the corresponding row in $\mathbf{B}_x$.

SUM is an aggregate constraint, and a similar constraint is available for aggregate operators *MIN*, *MAX*, *AVERAGE*, *PRODUCT* and *COUNT*. The *conditional* aggregate variant *SUMIF* only ranges over those cells that satisfy a condition on a related cell in that table, e.g. $T_1[:, 10]$ and the data in T_5 for matching salesperson names.

It is helpful to see the analogy of constraint templates with first order logic (FOL) and constraint satisfaction. From a FOL perspective, a constraint of the form $B_r = \mathit{RANK}(B_x)$ can be seen as a predicate $\mathit{RANK}(B_r, B_x)$ where *RANK* is the name of the predicate and its arguments B_r and B_x are terms, which can be seen as either uninstantiated variables or as concrete values. This also holds in our setting, where an instantiation of a variable corresponds to a concrete block. For example, for the spreadsheet in Figure 1a, when we write $T_1[:, 8] = \mathit{RANK}(T_1[:, 7])$, then the value of B_r is the 8th vector in T_1: $B_r = T_1[:, 8] = [2, 1, 3, 4]$ and the value of B_x is the 7th vector: $B_x = T_1[:, 7] = [1514, 1551, 691, 392]$.

With this interpretation, we can speak about the signature and definition of a constraint template being *satisfied*. We say that a signature (definition) of a constraint template s with n arguments is satisfied by the blocks $(B_1, ..., B_n)$ if $\mathit{Sig}_s(B_1, ..., B_n)$ (respectively $\mathit{Def}_s(B_1, ..., B_n)$) is satisfied. Likewise, the template is satisfied if both the signature and definition are satisfied; in logic programming, we would define the predicate s using a Prolog like clause: $s(B_1, ..., B_n) \leftarrow \mathit{Sig}_s(B_1, ..., B_n) \wedge \mathit{Def}_s(B_1, ..., B_n)$. Under this interpretation, the term constraint and constraint template can be used interchangeably.

Definition 3. A **valid argument assignment** of a constraint template s is a tuple of blocks $(B_1, ..., B_n)$ such that $s(B_1, ..., B_n)$ is satisfied, that is, both the signature and the definition of the corresponding constraint template are satisfied by the assignment of $(B_1, ..., B_n)$ to the arguments.

C. Problem Definition

The problem of learning constraints from tabular data can be seen as an inverse *constraint satisfaction problem* (CSP). In a CSP one is given a set of constraints over variables that must all be satisfied, and the goal is to find an instantiation of all the variables that satisfies these constraints. In the context of spreadsheets, the variables would be (blocks of) cells, and one would be given the actual constraints and functions with the goal of finding the values in the cells. The inverse problem is, given only an instantiation of the cells, to find the constraints that are satisfied in the spreadsheet.

We define the inverse problem, that is the **Tabular Constraint Learning Problem**, as follows:

Definition 4. *Tabular Constraint Learning.*
Given a set of instantiated blocks $\mathcal{B}$ over tables $\mathcal{T}$ and a set of *constraint templates* $\mathcal{S}$: **find** all constraints $s(B'_1, ..., B'_n)$ where $s \in \mathcal{S}$, $\forall i : B'_i \sqsubseteq B_i \in \mathcal{B}$ and $(B'_1, ..., B'_n)$ is a satisfied argument assignment of the template s.

The input is a set of blocks, and in Section III we will discuss how these can be extracted from a spreadsheet. Figure 1b shows the solution to the tabular constraint learning problem when applied on the blocks of Figure 1a and constraint templates listed in Table I.

D. Other considerations

1) Dependencies: In Table I one can see that for some constraints we used the predicate of another constraint in its signature, e.g. for *PERMUTATION*. This expresses a dependency of the constraint on that other *base* constraint. This can be interpreted as follows: the signature of the constraint consists of its own signature plus the signature of the base constraint, and its definition of its own definition plus the definition of the base constraint. In FOL, we can see that one constraint entails the other, for example if $\mathit{PERMUTATION}(B_x)$ holds for a block B_x, then $\mathit{ALLDIFFERENT}(B_x)$ also holds.

In Section III we will see how, apart from easing the specification of the signature and definition, such dependencies can be used to speed up the search for constraints.

2) Redundancies: Depending on the application, some constraints in the solution to the tabular constraint learning problem may be considered *redundant*. This is because constraints may be logically equivalent or may be implied by other constraints, e.g. if you know $\mathit{PERMUTATION}(B_x)$, $\mathit{ALLDIFFERENT}(B_x)$ is redundant.

Within the same constraint, there can be equivalent argument assignments if the order of some of the arguments does not matter. For example for the product constraint, $B_r = B_1 \times B_2 \equiv B_r = B_2 \times B_1$ so one can be considered redundant to the other. Two different constraints can also be logically equivalent, due to their nature e.g. for addition / subtraction and product / division: $B_r = B_1 \times B_2 \equiv B_1 = B_r / B_2$.

Finally, when the data has rows or columns that contain *exactly* the same values, then any constraint with such a vector in its argument assignment will also have an argument assignment with the other equivalent vectors.

Dealing with redundancy is often application-dependent. Therefore, in Section III we first explain our generic method for finding all constraints, before describing some optimizations that avoid obvious equivalences (Section III-D). Later, we will investigate the impact of redundancy in the experiments.

III. Approach to Tabular Constraint Learning

The aim of our method is to detect constraints between rows and columns of tabular data. Recall that a valid argument assignment for a constraint is an assignment of a block to each of the arguments of the constraint, such that the signature and definition of the constraint template is satisfied.

Our proposed methodology contains the following steps:

1) Extract headerless tables from tabular data
2) Partition the tables into contiguous, type-consistent *superblocks*
3) Generate for each constraint template all valid argument assignments in two steps:
   a) For each constraint template s, generate all assignments $(B_1, \ldots, B_n)$ where each B_i is a superblock and the B_i are *compatible* (defined below) with the signature of s.
   b) For each generated assignment $(B_1, \ldots, B_n)$ in 3a), find all valid constraints $s(B'_1, \ldots, B'_n)$ such that for all i holds $B'_i \sqsubseteq B_i$ and the signature and definition of s are satisfied.

The core of our method is step 3. In principle, one could use a generate-and-test approach by generating all possible blocks from the superblock and testing each combination of blocks for each of the arguments of the constraints. However, each superblock of size m has $m * (m + 1)/2$ contiguous subblocks, meaning that a constraint with n arguments would have to check $O(n^{m^2})$ combinations of blocks.

Instead, we divide this step into two parts: in step 3a, we will not reason over individual (sub)blocks and their data, but rather over the properties of the superblocks. Consider table T_1 in Figure 1a and the *SUM*$_{row}$ constraint template. Instead of considering all possible subblocks of T_1, we first reason over the properties of the superblocks. For example, B_2 and B_4 are not numeric so they can be immediately discarded. Superblocks B_1, B_3 and B_5 are valid candidates for B_r, however, since they all have length 4, $\mathbf{B}_x$ needs at least 4 columns which is only satisfied by B_3. So the assignments generated in step 3a would be $(B_1, B_3), (B_3, B_3), (B_5, B_3)$.

In step 3b, we start from an assignment $(B_1, \ldots, B_n)$ with superblocks B_i, and generate and test all possible (sub)block assignments to arguments using the properties and actual data of the blocks. As we only have to consider the blocks $B'_i \sqsubseteq B_i$ contained in each superblock, this is typically much easier. For the above example and (B_5, B_3), each of the vectors of B_5 will be considered as candidate for the left-hand side, and one can enumerate all subblocks of B_3 for the right-hand side and verify the signature (e.g. at least size 4) and test whether for each row the definition is satisfied.

We now describe in more detail how headerless tables are extracted (step 1, Section III-A) and how superblocks are generated from them (step 2, Section III-B), how the candidate superblock assignments are generated (step 3a, Section III-C1) and how the actual assignments are extracted from that (step 3b, Section III-C2). In Section III-D we describe a number of optimizations designed to improve the algorithm.

A. Table extraction

Many spreadsheets contain headers that provide hints at where the table(s) in the sheet are and what the meaning of the rows and columns is. However, detecting tables and headers is a complex problem on its own [5]. Furthermore, headers may be missing and their use often requires incorporating language-specific vocabulary, e.g. English.

Instead, we assume tables are specified by means of their coordinates within a spreadsheet and optionally a fixed orientation of each table. The orientation can be used to indicate that a table should be interpreted as having only rows or only columns.

We developed two simple prototypes to help with the specification of the tables:

a) Automatic detection: Under the following two assumptions, the table detection task becomes easy enough to be automated; 1) tables are rectangular blocks of data not containing any empty cells; and 2) tables are separated by at least one empty cell from other tables. The sheet is then processed row by row, splitting each row into ranges of non-empty cells and appending them to adjacent ranges in the previous row. Headers can be detected, for example, by checking if the first row or column contains only textual values. If a header row (column) is detected, it is removed from the table and the orientation of the table is fixed to columns (rows), otherwise, we assume there is no header and the orientation is not fixed.

b) Visual: The above assumptions do not hold for many tables. However, since the specification of tables is usually easy for humans, we propose a second approach which allows users to indicate tables using a visual tool (screenshots are available in the accompanying github repository with metadata[1]). Users select tables excluding the header and optionally specify an orientation. The tool then generates the specification of the tables automatically.

B. Block detection

The goal of the block detection step is to partition tables into maximally type-consistent (all-numeric or all-textual) blocks. First, we preprocess the spreadsheet data so that currency values and percentual values are transformed into their corresponding numeric (i.e. integer or floating point) representation (e.g. 2.75\$ as 2.75 or 85% as 0.85). Then, each table is partitioned into maximal type-consistent blocks.

To find row-blocks, each row is treated as a vector and must be type-consistent; similarly for columns and column-blocks. Then, adjacent vectors that are of the same type are merged to obtain the maximally type-consistent superblocks.

C. Algorithm

Our method assumes that constraint templates and maximal blocks are given and solves the Tabular Constraint Learning problem by checking for each template s: what combination of maximal blocks can satisfy the signature (*superblock assignment*) and which specific *subblock assignments* satisfy

[1] https://github.com/for-DSAA2016-review/TaCLe

Algorithm 1 Learn tabular constraints

```
Input: $S$ – constraint templates, $\mathcal{B}$ – maximal blocks
Output: $C$ – learned constraints
procedure LearnConstraints($\mathcal{B}$, $S$)
    $C \leftarrow \emptyset$
    for all $s$ in $S$ do
        $A \leftarrow \mathit{SuperblockAssignments}(s, \mathcal{B})$
        for all $(B_1, \ldots, B_n) \in A$ do
            $A' \leftarrow \mathit{Subassignments}(s, (B_1, \ldots, B_n))$
            for all $(B'_1, \ldots, B'_n) \in A'$ do
                $C \leftarrow C \cup \{c_s(B'_1, \ldots, B'_n)\}$
    return $C$
```

TABLE II: Translation of signature requirements to superblock constraints. The following need not be relaxed: $\mathit{length}(B) = x$, $\mathit{orientation}(B) = x$, $\mathit{table}(B) = x$ and $\mathit{size}(B) \geq x$.

Requirement	Superblock constraint
$\mathit{type}(B) = x$	$\mathit{basetype}(\mathit{type}(B)) = \mathit{basetype}(x)$
$\mathit{size}(B) = x$	$\mathit{size}(B) \geq x$
$\mathit{columns}(B) = x$	$\mathit{if}\ \mathit{orientation}(B) = \mathit{column} : \mathit{columns}(B) \geq x$
	$\mathit{if}\ \mathit{orientation}(B) = \mathit{row} : \mathit{columns}(B) = x$
$\mathit{rows}(B) = x$	$\mathit{if}\ \mathit{orientation}(B) = \mathit{column} : \mathit{rows}(B) = x$
	$\mathit{if}\ \mathit{orientation}(B) = \mathit{row} : \mathit{rows}(B) \geq x$

both the signature and the definition. The pseudo-code of this approach is shown in Algorithm 1.

This separation of checking the *properties* of superblock assignments from checking the *actual data* in the subblock assignment controls the exponential blow-up of combinations to test. Furthermore, we use constraint satisfaction technology in the first step to efficiently enumerate superblock assignments that are compatible with the signature.

1) Generating superblock assignments (Step 3a):

Given a constraint template s and the set of all (maximal) superblocks $\mathcal{B}$, the goal is to find all combinations of superblocks that are compatible with the constraint signature. An argument assignment $(B_1, ...B_n)$ is *compatible* with the signature of template s if for each block B_i there exists at least one subblock $B'_i \sqsubseteq B_i$ that satisfies the signature.

The choice of one argument can influence the possible candidate blocks for the other arguments, for example, if they must have the same length. Instead of writing specialized code to generate and test the superblock assignments, we make use of the built-in reasoning mechanisms of constraint satisfaction solvers.

A Constraint Satisfaction Problem (CSP) is a triple (V, D, C) where V is a set of *variables*, D the domain of possible values each variable can take and C the set of constraints over V that must be satisfied. In our case, we define one variable V_i for each argument of a constraint template. Each variable can be assigned each of the maximal blocks in $\mathcal{B}$, so the domain of the variables consists of $|\{\mathcal{B}\}|$ block identifiers.

To reason over the blocks, we add to the constraint program a set of background facts that contain the properties of each block, namely its type, table, orientation, size, length and number of rows and columns.

The actual constraints must enforce that a superblock assignment is *compatible* with the requirements defined in the signature.

Table II shows how the conversion from requirements of the signature to CSP constraints works: Requirements on the lengths, orientations and tables of blocks can be directly enforced since they are invariant under block containment ($\sqsubseteq$). Typing requirements need to be relaxed to check only for base types (numeric and textual). Minimum sizes can be directly enforced, but exact size requirements are relaxed to minimum sizes, since blocks with too many vectors contain subblocks of the required (smaller) size. Finally, restrictions on the number of rows or columns behave as length or size constraints based on the orientation of the block they are applied to. Subgroups of row-oriented (column-oriented) blocks will always have the same number of columns (rows), however, the number of rows (columns) might decrease.

The *SuperblockAssignments*(s,$\mathcal{B}$) method will use these conversion rules to construct a CSP and query a solver for all solutions. These solutions correspond to the valid superblock assignments for constraint template s.

Example 3. Consider the constraint template $B_r = \mathit{SUM}_{\mathit{row}}(\mathbf{B}_x)$, then the generated CSP will contain two variables V_r, V_x corresponding to arguments B_r and $\mathbf{B}_x$. Given maximal blocks $\mathcal{B}$, the domain of these variables are $D(V_r) = D(V_x) = \{1, \ldots, |\mathcal{B}|\}$. Finally, the signature of $B_r = \mathit{SUM}_{\mathit{row}}(\mathbf{B}_x)$ (see Table I) will be translated into constraints:

$$\mathit{numeric}(V_r) \wedge \mathit{numeric}(V_x) \wedge \mathit{columns}(V_x) \geq 2$$
$$(\mathit{orientation}(V_x) = \mathit{column}) \Rightarrow (\mathit{rows}(V_x) \geq \mathit{length}(V_r))$$
$$(\mathit{orientation}(V_x) = \mathit{row}) \Rightarrow (\mathit{rows}(V_x) = \mathit{length}(V_r))$$

2) Generating subblock assignments (Step 3b): Given a superblock assignment $(B_1, \ldots, B_n)$ from the previous step, the goal is to discover valid *subassignments*, i.e. assignments of subblocks $(B'_1, \ldots, B'_n)$ (for all i, $B'_i \sqsubseteq B_i$) that satisfy both the signature and the definition of template s.

Example 4. Consider the sum-over-rows constraint template $B_r = \mathit{SUM}_{\mathit{row}}(\mathbf{B}_x)$. An example superblock assignment from Figure 1a is $(B_r, \mathbf{B}_x) = (T_1[:, 3{:}8], T_1[:, 3{:}8])$. Note that this assignment does not satisfy the signature yet because B_r contains more than one vector. This step aims to generate subassignments $(B'_r \sqsubseteq B_r, \mathbf{B}_x{}' \sqsubseteq \mathbf{B}_x)$ that do satisfy the signature and test whether they satisfy the definition: $\forall i, B_r[i] = \sum_{j=1}^{length(\mathbf{B}_x)} \mathit{row}(i, \mathbf{B}_x)[j]$. In Figure 1a there is exactly one such subassignment: $(T_1[:, 7], T_1[:, 3{:}6])$.

In this step, we could also formulate the problem as a CSP. However, few CSP solvers support floating point numbers, which are prevalent in spreadsheets. Furthermore, in a CSP approach we would have to *ground* out the definition for each of the corresponding elements in the vectors. This is inefficient, as already such blocks may not satisfy the signature or the first elements in the data may not satisfy the definition.

On the other hand, a simple generate-and-test approach, as illustrated in Algorithm 2, will usually suffice. Given a superblock assignment, all disjoint subassignments are generated taking into account the required size, columns or rows.

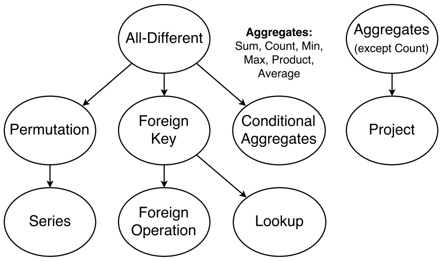


Fig. 2: Dependency graph of Table I; an arrow from s_1 to s_2 indicates that s_2 depends on s_1 (its signature includes s_1).

For every (disjoint) subassignment the exact signature (e.g. subtypes will not have been checked yet) and definition will be tested and all satisfying subassignments are returned.

Algorithm 2 Generate-and-test for *Subassignments*

```
procedure SUBASSIGNMENTS($s$, $(B_1, \ldots, B_n)$)
    $n \leftarrow$ number of arguments of template $s$
    $A_{sub} \leftarrow \emptyset$
    for all $(B'_1, \ldots, B'_n)$ where $\forall i : B'_i \sqsubseteq B_i \wedge \mathit{disjoint}_{j \neq i}(B'_i, B'_j)$ do
        if $\mathit{Sig}_s(B'_1, \ldots, B'_n) \wedge \mathit{Def}_s(B'_1, \ldots, B'_n)$ then
            $A_{sub} \leftarrow A_{sub} \cup \{(B'_1, \ldots, B'_n)\}$
    return $A_{sub}$
```

D. Optimizations

This section discusses various design decisions and optimizations aimed at improving the speed, the extensibility of our system and the elimination of redundant constraints.

1) Template dependencies: As discussed in Section II-D1, some constraint templates depend on others by including templates they depend on in their signature (see Table I).

In Inductive Logic Programming, one often exploits implications between constraints to structure the search space [4]. Our approach uses the dependencies between templates to define a dependency graph. We assume that signatures do not contain equivalences or loops, and hence the resulting graph is a directed acyclic graph (DAG). Figure 2 shows the dependency graph extracted from the signatures in Table I. Constraint templates that have no dependencies are omitted.

Using the dependency graph ($\mathcal{D}$) we can reorder the templates such that a template occurs after any template it depends on. Concretely, in Algorithm 1 line 5 the input templates S are handled following an ordering that agrees with the partial ordering imposed by $\mathcal{D}$ (e.g. *ALLDIFFERENT* before *FOREIGNKEY*). When using this ordering, templates later in the ordering can use the learned constraints of the templates they depend on. To enable this reuse of constraints, the set of learned constraints C is added to the argument of *SuperblockAssignments* on line 6.

The previously learned constraints are used to speed up the *SuperblockAssignments* step. For constraint templates where some arguments are also part of a constraint that it depends on, it is not needed to search for superblocks and subblocks from scratch. Instead, one can start from all and only the actual argument assignments of the base constraint, and only search matching superblocks for the remaining arguments.

Example 5. Consider *FOREIGNKEY*(B_{fk}, B_{pk}), which states that every value in B_{fk} also exists in B_{pk}; its signature includes *ALLDIFFERENT*(B_{pk}). There are 18 *ALLDIFFERENT* constraints to be found in Figure 1a, hence, the *SuperblockAssignments* only needs to check which superblocks for B_{fk} are compatible (different table, same type) with these 18 assignments to B_{pk}.

Instead of generating one CSP to find all assignments, a CSP is generated for every known assignment of the depending constraint (and variables involved), which then searches for all assignments completing this partial assignment. The procedure to generate CSPs remains the same otherwise.

2) Redundancy: We consider two types of redundancy that we aim to eliminate during search. As noted in Section II-D1, for some constraint templates there are *symmetries* over the arguments that lead to trivially equivalent solutions, for example, $B'_R = B'_1 \times B'_2 \Leftrightarrow B'_R = B'_2 \times B'_1$. Such duplicates are avoided by defining a *canonical* form for these constraints. In practice, we define an arbitrary but fixed block ordering and require those blocks that are interchangeable to adhere to this ordering. Moreover, among the semantically equivalent constraints product ($B'_R = B'_1 \times B'_2$) and division ($B'_1 = B'_R / B'_2$) we only added product, as well as only difference ($B_r = B_1 - B_2$) and not sum ($B_1 = B_r + B_2$). However, division and sum could easily be added, as they could be added explicitly in post-processing based on the matching product and difference constraints.

A last type of redundancy we chose to reduce is that there may be multiple *overlapping* subblocks that satisfy the definition of a constraint template. For example, consider the constraint $B_r = \mathit{SUM}_{col}(T[:, 1{:}n])$ where $T[:, k{:}n]$ consists of only zeros, then $B_r = \mathit{SUM}_{col}(T[:, 1{:}j])$ will be true for all $k - 1 \leq j \leq n$. In *TaCLe* we have, therefore, chosen to only consider *maximal* subblocks.

In some cases a maximal subblock might falsely include irrelevant columns. Consider superblock $\mathbf{B}_x = T[:, 1{:}3] = [[200, 300], [200, 150], [1, 2]]$ and $B_r = [200, 300]$, then *TaCLe* will find the constraint $B_r = \mathit{MAX}(T[:, 1{:}3])$. Should the target constraint be $B_r = \mathit{MAX}(T[:, 1{:}2])$, then $T[:, 3]$ must be split off into a separate block.

Alternatively, we can output all valid assignments instead of only maximal subblocks, e.g. $B_r = \mathit{MAX}(T[:, 1])$, $B_r = \mathit{MAX}(T[:, 1{:}2])$ and $B_r = \mathit{MAX}(T[:, 1{:}3])$, however, this might produce many redundant constraints including constraints that accidentally cover too little data.

3) Constraints: The constraints that are currently supported in our system are shown in Table I. We included most formulas that we encountered in tutorial spreadsheets including the popular *SUM* and *LOOKUP* constraints[2]. We also added three structural constraints (*ALLDIFFERENT*, *PERMUTATION* and *FOREIGNKEY*) so that they can be used in the signature of other constraint templates; they are also popular in constraint satisfaction [1].

For *FOREIGNKEY*, *LOOKUP*, aggregates and *PROJECT* we use custom, optimized methods to find subassignments.

[2] https://support.office.com/en-us/article/Excel-functions-by-category-5F91F4E9-7B42-46D2-9BD1-63F26A86C0EB

Instead of the more generic generate-and-test implementation, we use, for example, caching, maximal range-finding and partial tests. While conditional aggregates use generate-and-test, they employ similar optimizations in their test method.

IV. Evaluation

In this section we experimentally validate our approach. We studied three main questions concerning the recall (*Q1*), precision (*Q2*) and speed (*Q3*) of our algorithm.

First, our implementation is illustrated using a case study on the spreadsheet corresponding to the example introduced in Figure 1a. In order to quantify the results and generalize our findings we also evaluate our algorithm on a benchmark of 34 spreadsheets that we assembled from various sources.

A. Method

In order to obtain a benchmark of spreadsheets we looked at three sources: 1) spreadsheets from an exercise session for teaching Excel based on [15]; 2) spreadsheets from tutorials online; and 3) publicly available *data* spreadsheets such as real-world crime statistics or financial reports (available at the same repository[1]). All spreadsheets were converted to CSV and the table definitions are fixed. Group definitions are only specified if the spreadsheet contains ambiguous *None* values.

For every spreadsheet we specified a ground-truth of constraints, i.e. the **intended** constraints that are expected to be (re-)discovered. These were determined using the formulas (if present), context and headers in the original spreadsheets. We encoded all intended constraints using our representation (in canonical form). Not all intended constraints are currently supported by *TaCLe*, therefore, we sometimes consider the subset of **supported** constraints. Constraints that are currently outside of the scope of this system are, for example, nested mathematical or logical formulas.

We focus on *functional* constraints that could be used as formulas in spreadsheets, ignoring the structural constraints *ALLDIFFERENT*, *FOREIGNKEY* and *PERMUTATION*. Moreover, we included detected equalities as intended constraints (in 5 spreadsheets). All experiments were run using Python 3.5.1 on a Macbook Pro, Intel Core i7 2.3 GHz with 16GB RAM.

B. Case study

Let us illustrate our implementation using the example presented in Figure 1a. *TaCLe* takes a few seconds to find the constraints in Figure 1b (excluding structural constraints). These include 5 spurious *RANK* constraints, e.g. $T_1[:,1] = \mathit{RANK}(T_1[:,5])$, that are true by accident. Moreover, there is 1 *LOOKUP* constraint that was not intended in the original spreadsheet (looking up ID based on Salesperson) and which is symmetric to the intended one (Salesperson based on ID).

For this example our primary goal of rediscovering all intended constraints is achieved, the recall (i.e. $\frac{\text{intended discovered}}{\text{all intended}}$) is 1.0. The implementation also returns 6 additional constraints, therefore, our precision (i.e. $\frac{\text{intended discovered}}{\text{all discovered}}$) is $12/18 = 0.67$, which is mostly due to the spurious RANK constraints. We believe this is due to the many short vectors which increases the chance for constraints to be true by accident.

TABLE III: Summary of properties and results per spreadsheet category

	Exercises (9)		**Tutorials** (21)		**Data** (4)	
	Overall	Sheet avg	Overall	Sheet avg	Overall	Sheet avg
Tables	19	2.11	48	2.29	4	1
Cells	1231	137	1889	90	2320	580
Intended Constraints	34	3.78	52	2.48	6	1.50
Recall	0.85	0.83	0.88	0.87	1.00	1.00
Recall Supported	1.00	1.00	1.00	1.00	1.00	1.00
Precision	0.97	0.98	0.70	0.91	1.00	1.00
Speed (s)	1.62	0.18	1.76	0.08	0.81	0.20

C. Benchmark

Table III gives an overview of the spreadsheets in the different categories and summarizes the results of our experiments. Notice that for the data category the average number of cells is higher while the number of intended constraints and tables is lower compared to the other categories.

Q1. How many intended constraints are found by TaCLe?: The overall recall achieved by *TaCLe* is 0.88 $(81/92)$. The overall and average recall per category is also high, as shown in Table III. The intended constraints not discovered correspond to constraints not supported by the system (supported recall is 1).

Q2. How precise is TaCLe?: Across all spreadsheets *TaCLe* achieves a precision of 0.79 $(81/102)$. While the average precision per spreadsheet is over 0.90 for all categories, the overall precision is much higher for the exercises and data categories (0.97 and 1.00) than for the tutorials category (0.70). Examining the tutorial spreadsheets confirms that there are a few spreadsheets that have a high number of additional constraints. A qualitative analysis of these spreadsheets shows that their lower precision is caused by duplicate data or multiple ways to calculate the same vectors. For these spreadsheets the precision could be increased by post-processing the constraints to detect implications or heuristically selecting one among multiple ways to compute the same vector.

Q3. How fast is TaCLe?: The average runtime across all spreadsheets is $0.13s$. Across categories the average running times fluctuate in the range between $0.09s$ to $0.22s$. While most spreadsheets can be analyzed very quickly, running times can go up to around $1s$ for others.

Of the 32 supported constraint templates (including equality), the slowest by far is fuzzy lookup using $\pm 36\%$ of the running time. It is followed by product ($\pm 5\%$), relative difference ($\pm 5\%$) and various row-aggregates (each $3.5\% - 4\%$).

To illustrate that using dependencies to find constraints incrementally can increase the efficiency, we ran the benchmarks using FOREIGNKEY instead of ALLDIFFERENT as

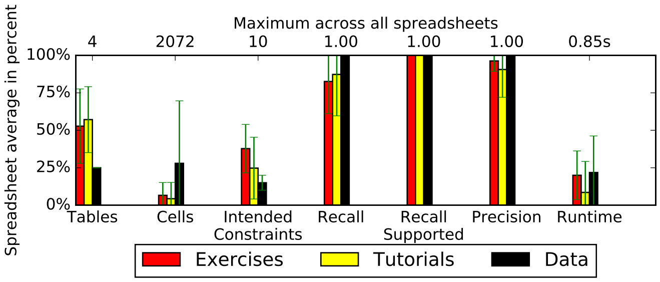


Fig. 3: Overview of the average properties and performance of spreadsheets within every benchmark category relative to the maximum across all spreadsheets. Green bars indicate standard deviation within the category.

base constraint to find conditional aggregates such as SUMIF. This assumption is too strong in practice, not all keys have to be used in the conditions. However, the total running times per category are decreased by 6% (exercises), 30% (tutorials) and 3% (data), showing the strong effect of the use of dependencies.

D. Testing the limits

Our algorithm relies on a correct table and group detection step, and we noticed that exported CSV files can make this more difficult in case of rounding errors due to low-precision formatting as well as misformatting of strings so they are interpreted as numbers. Automatically generated sheets can also contain many duplicate (equal) rows or columns, which are a source of increased runtime and decreased recall for our algorithm. For example, we tested a spreadsheet ("exps") with 6 tables, 531 cells and many duplicate columns across tables. While we obtained a recall of 1.0, the algorithm ran for $14.30s$ and the overall precision was 0.52 and only 0.11 when counting constraints with equal columns separately.

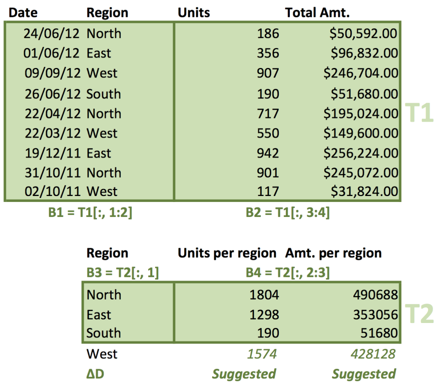
V. Applications

In this section, we illustrate how various motivating applications could be realized using our system.

A. Autocompletion

Autocompletion can be seen as using knowledge derived from the current data to predict new values before they are written by the user. A snapshot (D_0) is used to learn constraints and these are combined with new data (ΔD) that the user has added. Using ΔD we can keep track of which vectors have changed and which vectors are now *incomplete*. *Functional* constraints, such as $B_r = \mathit{SUM}_{col}(\mathbf{B}_x)$, are able to compute a result (B_r) based on its *input* ($\mathbf{B}_x$). If all vectors in the input blocks have changed and the result vector is incomplete, the constraint can compute the resulting value(s).

Consider Figure 4 containing two tables. Learning on a snapshot that does not contain the row below T_2, we find constraints $T_2[:,2] = \mathit{SUMIF}(T_1[:,2], T_2[:,1], T_1[:,3])$ and $T_2[:,3] = \mathit{SUMIF}(T_1[:,2], T_2[:,1], T_1[:,4])$. The new data ΔD consists of the new value *West* in the row below T_2. Therefore, vector $T_2[:,1]$ is changed and vectors $T_2[:,2]$, $T_2[:,3]$ have become incomplete. The *SUMIF* constraints have enough input to fill the incomplete vectors and predict the values in the last row: $T_2[4,2] = 1547$ and $T_2[4,3] = 428128$.



Date	Region	Units	Total Amt.	
24/06/12	North	186	$50,592.00	
01/06/12	East	356	$96,832.00	
09/09/12	West	907	$246,704.00	
26/06/12	South	190	$51,680.00	
22/04/12	North	717	$195,024.00	T1
22/03/12	West	550	$149,600.00	
19/12/11	East	942	$256,224.00	
31/10/11	North	901	$245,072.00	
02/10/11	West	117	$31,824.00	
B1 = T1[:, 1:2]		B2 = T1[:, 3:4]		

Region	Units per region	Amt. per region	
B3 = T2[:, 1]	B4 = T2[:, 2:3]		
North	1804	490688	
East	1298	353056	T2
South	190	51680	
West	*1574*	*428128*	
ΔD	*Suggested*	*Suggested*	

Fig. 4: Autocompletion works by learning constraint on a snapshot and combining those constraints with new data ΔD. The figure shows how, given a new value *West*, we can suggest values for the other cells in that row.

B. Error detection

We consider two cases of error detection. The first setting is the *online* detection of errors and is similar to the autocompletion setting. Using snapshots, we search for *conflict* constraints, which are learned on a previous snapshot but do not agree with newly added data ΔD. Such conflicts can indicate (typing) errors in the data just entered. For example, in case 'Wes' instead of 'West' was added in Figure 4.

The second setting is *offline* error detection in a static spreadsheet. Instead of snapshots, one can repeatedly search for conflict constraints between the original spreadsheet and a sheet with a row or column removed. Many schemes can be devised for choosing which row(s) or column(s) are removed.

VI. Related Work

TaCLe combines ideas from several existing approaches and different fields of research.

First, it borrows techniques from logical and relational learning [4], as it finds constraints in multiple relations (or tables). However, unlike typical logical and relational learning approaches, it focuses on data in spreadsheets and constraints on columns or rows in the sheets. Furthermore, it derives a set of simple atomic constraints rather than a set of rules that each consist of multiple literals as in clausal discovery [13, 9]. Nevertheless, several ideas from logical and relational learning proved to be useful, such as the use of a canonical form, a refinement graph, and pruning for redundancy. Also related to this line of research is the work on deriving constraints in databases such as functional and multi-valued dependencies [14, 11] although that line of research has focused on more specialized techniques for specific types of constraints. Furthermore, the constraints and formulas used in spreadsheets are often different from those in databases.

Second, there exist a number of algorithms in the constraint programming community that induce constraint (programs) from one or more examples and questions. Two well-known approaches include ModelSeeker [1] and Quacq [2]. The former starts from a single example in the form of a vector of values and then exhaustively looks for the most specific constraints from a large constraint library that hold in the vector. To this aim, it cleverly enumerates different de-vectorized tables; for instance, if the initial vector was of dimension 1×20, it would consider rearrangements of size 2×10, 4×5, 5×4, and Modelseeker would then look for both column, row and even diagonal constraints. Key differences are that we assume given spreadsheets with *multiple* tables; and that ModelSeeker's constraints are restricted to integer values and often typical for constraint programming community, e.g. aimed at combinatorial optimization and scheduling. Conacq [3] acquires a conjunction of constraints using techniques inspired on Mitchell's version spaces to process examples; its successor Quacq uses active learning to query the user regarding satisfiability. Each of the examples/queries is processed for each constraint, hence an important issue is how fast the algorithm converges to the entire solution set (e.g. for Sudoku). The focus is again on integer variables and typical combinatorial optimisation constraints.

Third, our work borrows from the seminal work of Gulwani et al. [7] on program synthesis for tabular data, which is incorporated in Microsofts Excel. In FlashFill, the end-user might work on a table containing the surnames (column A) and names (column B) of politicians. The first row might contain the values A1 = "Blair" and B1 = "Tony", and the user might then enter in cell C1 the value "T.B.". At that point FlashFill generates a program that produces the output "T.B." for the inputs "Blair" and "Tony", and also applies it to the other politicians (rows) in the table. While it might need an extra example to deal with names such as "Van Rompuy, Herman" and determine whether it should output "H.V." or "H.V.R.", FlashFill learns correct programs from very few examples. Flashfill has been extended to e.g. FlashExtract [10] to produce a set of tables starting from an unstructured document (a txt or HTML file) with a set of examples marked for extraction. However, unlike our approach, Flashfill requires the user to identify explicitly the desired inputs/outputs from the function as well as to provide explicitly some positive and sometimes also negative examples. In contrast, our approach is unsupervised, although it would be interesting to study whether it could be improved through user interaction and through user examples. Another interesting open issue (for all of these techniques with the exception of logical and relational learning) is how to deal with possibly incomplete or noisy data.

Finally, there are the works on BayesDB [12] and Tabular [6], two recent probabilistic modeling languages that have been specifically designed for dealing with relational data in tabular form and that can, as FlashFill, also automatically fill out missing values in a table. However, unlike the other mentioned approaches, it is based on an underlying probabilistic graphical model that performs probabilistic inference rather than identify "hard" constraints.

VII. Conclusions

Our goal is to automatically identify constraints in a spreadsheet. We have presented and evaluated our approach, implemented as the system *TaCLe*, that is able to learn many different constraints in tabular data. The resulting method has high recall, produces limited number of redundant constraints and is sufficiently efficient for normal and interactive use.

The approach is designed to find constraints that hold over entire columns and rows. In future work we plan to extend this to learn arbitrary nested constraints (e.g. $B_r = (B_1 + B_2)/B_3$), as well as constraints over only a subset of the vectors. This may give rise to more redundant and spurious constraints, which was not problematic up to this point.

Two promising ways to mitigate redundant and spurious constraints are heuristic filtering or post-processing of the constraint set and the integration of our approach in an interactive setting where users can receive and provide feedback. The latter is more similar to an active learning setting.

A final direction is that the current approach assumes noise-free data (but see the part on error correction); in inductive logic programming there has been much work on how to handle noisy data, while in constraint satisfaction a popular alternative is to consider *soft* constraints that can be violated (at a cost). This could extend the approach to new application domains, beyond traditional spreadsheets.

References

[1] Nicolas Beldiceanu and Helmut Simonis. *Principles and Practice of Constraint Programming*, chapter A Model Seeker: Extracting Global Constraint Models from Positive Examples, pages 141–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *IJCAI*, 2013.

[3] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *ECML*.

[4] Luc De Raedt. *Logical and Relational Learning*. Springer-Verlag New York, Inc., 2008.

[5] Jing Fang, Prasenjit Mitra, Zhi Tang, and C. Lee Giles. Table header detection and classification. In *AAAI*, 2012.

[6] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. Tabular: A schema-driven probabilistic programming language. Technical Report MSR-TR-2013-118, December 2013.

[7] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 317–330, New York, NY, USA, 2011. ACM.

[8] Thomas Herndon, Michael Ash, and Robert Pollin. Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics*, 2013.

[9] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. *ICTAI*, 1(section II):45–52, 2010.

[10] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, page 55, 2014.

[11] Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *DKE*, 12(1):83–99, 1994.

[12] Vikash K. Mansinghka, Richard Tibbetts, Jay Baxter, Patrick Shafto, and Baxter Eaves. Bayesdb: A probabilistic programming system for querying the probable implications of data. *CoRR*, abs/1512.05006.

[13] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.

[14] Iztok Savnik and Peter A. Flach. Discovery of multivalued dependencies from relations. *Intell. Data Anal.*, 4(3-4):195–211, 2000.

[15] Eddy Van den Broeck and Erik Cuypers. *MS Excel 2010*. Uitgeverij De Boeck, 2011.