Learning constraints in 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 between spreadsheets can be time consuming and error-prone. In this work, we investigate the automatic discovery of constraints (functions and relations) from 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 predicates (constraints) 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 different 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. Especially 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 formula's are lost, including inter-sheet formulas and relations. Even in manually created spreadsheets, it can be challenging to be consistent and correct with formula's across big spreadsheets. For example the influential Reinhart-Rogoff economical paper "Growth in a Time of Debt" had some of its claims contested [7], after an investigation of the used Excel sheets was shown to contain some mistakes in formulae.

In this paper, we investigate whether learning techniques can be used to infer constraints (formula's and other relations) from raw spreadsheet data. This is a new and unconventional machine learning problem. Consider the example in Figure 1a, where the header's names already suggest the usage of spreadsheet operations such as *average* or *sum*. Looking at the first row in Table 2 and the data in Table 1, it is clear that the values in this row were computed by summing the values in the corresponding column of Table 1. Examining Table 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 taking the row-wise sum of the previous four columns. This provides a flavor of how this problem is different from standard data mining 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. And on the other hand, the data is most often mixed textual and numeric. One can see that understanding and learning constraints in tabular data is hence a new and challenging problem for machine learning.

The approach that we introduce borrows techniques from logical and relational learning [13], where the discovery of clausal constraints has been studied in an inductive logic programming setting [14, 8]; 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 [15]. It also builds on work on program synthesis, in particular, on Flashfill [6], 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 focusses on learning both column- and row-constraints, as does Modelseeker [1] which is restricted to traditional CSPs. It contrasts with Flashfill in that it can learn from numeric data too, as well as general constraints in addition to functions. A more detailed discussion on related work is contained in Section VI.

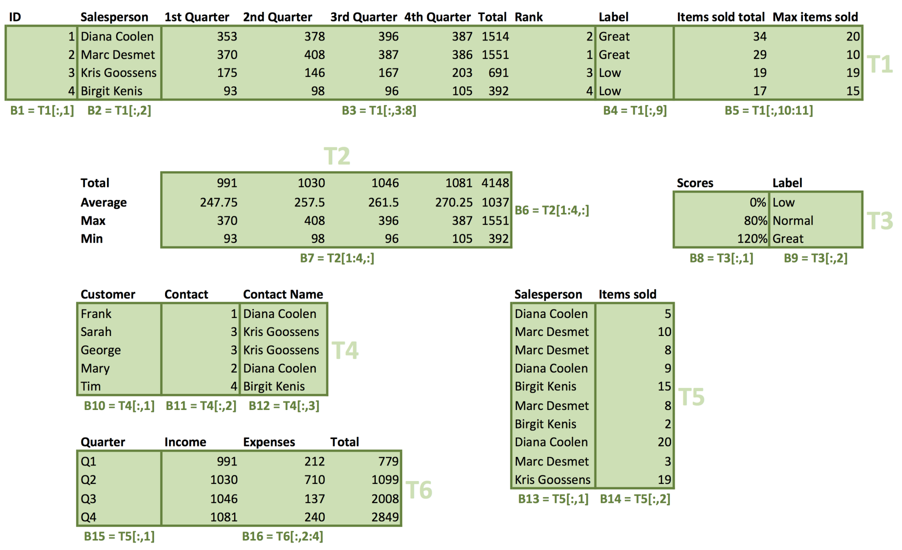
The question that we answer in this paper is: is it possible to discover or reconstruct structural constraints (relations, functions) in flat tabular spreadsheet data? To answer this question we contribute a general-purpose method and system, named *TaCLe* (pronounced as the word "tackle"), for discovering row-wise and column-wise constraints. It operates directly on (headerless) tables of a spreadsheet in an unsupervised setting, as it reasons on that 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.

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

II. Formalization

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

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


T1
B1 = T1[:,1]
B2 = T1[:,2]
B3 = T1[:,3:8]
B4 = T1[:,9]
B5 = T1[:,10:11]
T2
B6 = T2[1:4,:]
B7 = T2[1:4,:]
T3
B8 = T3[:,1]
B9 = T3[:,2]
T4
B10 = T4[:,1]
B11 = T4[:,2]
B12 = T4[:,3]
T5
B13 = T5[:,1]
B14 = T5[:,2]
T6
B15 = T5[:,1]
B16 = T6[:,2:4]

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

Scores	Label
0%	Low
80%	Normal
120%	Great

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

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

(a) Example spreadsheet (black words and numbers only). Green background indicate headerless tables, dark borders indicate maximal type-consistent blocks.

$$\begin{aligned}
&\mathit{SERIES}(T_1[:,1])\\
&T_1[:,8] = \mathit{RANK}(T_1[:,7])\\
&T_1[:,8] = \mathit{RANK}(T_1[:,3])^1, T_1[:,8] = \mathit{RANK}(T_1[:,4])^1\\
&T_1[:,1] = \mathit{RANK}(T_1[:,5])^1, T_1[:,1] = \mathit{RANK}(T_1[:,6])^1\\
&T_1[:,1] = \mathit{RANK}(T_1[:,10])^1\\
&T_1[:,7] = \mathit{SUM}_{row}(T_1[:,3:6])\\
&T_1[:,10] = \mathit{SUMIF}(T_5[:,1], T_1[:,2], T_5[:,2])\\
&T_1[:,11] = \mathit{MAXIF}(T_5[:,1], T_1[:,2], T_5[:,2])\\
&T_2[1,:] = \mathit{SUM}_{col}(T_1[:,3:7])\\
&T_2[2,:] = \mathit{AVERAGE}_{col}(T_1[:,3:7])\\
&T_2[3,:] = \mathit{MAX}_{col}(T_1[:,3:7]),\\
&T_2[4,:] = \mathit{MIN}_{col}(T_1[:,3:7])\\
&T_4[:,2] = \mathit{LOOKUP}(T_4[:,3], T_1[:,2], T_1[:,1])^2\\
&T_4[:,3] = \mathit{LOOKUP}(T_4[:,2], T_1[:,1], T_1[:,2])\\
&T_6[:,2] = \mathit{SUM}_{col}(T_1[:,3:6])\\
&T_6[:,4] = \mathit{PREV}(T_6[:,4]) + T_6[:,2] - T_6[:,3]
\end{aligned}$$

(b) Constraints present in the running example (left), except *ALLDIFFERENT* (16), *PERMUTATION* (2) and *FOREIGNKEY* (5). [1](Spurious constraints) [2](Redundant constraints not originally present)

Fig. 1: Running example

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 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, that is, 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 $T_5[:,1] = [\text{'Diana Coolen'}, 5]$. The latter is not type-consistent while the former 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 it 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: $B = T[a{:}b,:]$ for a row-oriented block containing rows a to b in table T; and similarly $B = T[:,a{:}b]$ for a column-oriented block. We will refer to the *vectors* of a block when we wish to refer to its rows/columns independent~~ly~~ of their orientation.

A block has the following properties:

- *type*: a block is type-consistent, so it has a 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*: this is orientation specific, for row-based blocks this is the size, for column-based blocks the length
- *columns*: also orientation specific, for row-based blocks this is the length, for column-based blocks the size

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 ($B_1 = T_1[:,1]$, $B_2 = T_1[:,2]$, $B_3 = T_1[:,3{:}8]$, $\dots$).

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: $B' \sqsubseteq B \Leftrightarrow B = T[a{:}b,:] \wedge B' = T[a'{:}b',:] \wedge a \leq a' \wedge b' \leq b$ 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 is that $T_1[:,3{:}6] \sqsubseteq T_1[:,3{:}8]$, which contains the sales numbers of all employees for the four quarters.

TABLE I: Overview of constraint templates implemented in *TaCLe*. The templates marked with † are aggregate constraint templates, the table shows the version for sum but the implementation also supports max, min, average, product and count. Subgroups in normal font have to be vectors, while subgroups in **bold** may contain more than one vector. Templates marked with $*$ are *structural*, i.e. they are not functional and cannot be implemented in most spreadsheet software.

Syntax	**Signature**	**Definition**
$\textit{ALLDIFFERENT}(B_x)^*$	$\textit{discrete}(B_x)$	All values in B_x are different: $B_x[i] \neq B_x[j]$ if $i \neq j$
$\textit{PERMUTATION}(B_x)^*$	$\textit{numeric}(B_x)$, $\textit{ALLDIFFERENT}(B_x)$	The values in B_x are a permutation of the numbers 1 through $\textit{length}(B_x)$.
$\textit{SERIES}(B_x)$	$\textit{integer}(B_x)$ and $\textit{PERMUTATION}(B_x)$	$B_x[1] = 1$ and $B_x[i] = B_x[i-1] + 1$.
$\textit{FOREIGNKEY}(B_{fk}, B_{pk})^*$	B_{fk} and B_{pk} are both *discrete*; $\textit{type}(B_{fk}) = \textit{type}(B_{pk})$; $\textit{table}(B_{fk}) \neq \textit{table}(B_{pk})$; and $\textit{ALLDIFFERENT}(B_{pk})$	Every value in B_{fk} also exist in B_{pk}
$B_r = \textit{LOOKUP}(B_{fk}, B_{pk}, B_{val})$	B_{fk} and B_{pk} are both *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 $\textit{FOREIGNKEY}(B_{fk}, B_{pk})$.	$B_r[i]$ is the same value as looking up $B_{fk}[i]$ in B_{pk} and returning the corresponding value in B_{val}.
$B_r = \textit{LOOKUP}_{\textit{fuzzy}}(B_{fk}, B_{pk}, B_{val})$	*Same as lookup*	$B_r[i]$ is the same value as looking up the last item in B_{pk} smaller than $B_{fk}[i]$ and returning the corresponding value in B_{val}.
$B_r = B_1 \times \textit{LOOKUP}(B_{fk}, B_{pk}, B_{val})$	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 $\textit{FOREIGNKEY}(B_{fk}, B_{pk})$.	$B_r[i]$ is the obtained by multiplying $B_1[i]$ with $\textit{LOOKUP}(B_{fk}, B_{pk}, B_{val})[i]$.
$B_r = B_1 \times B_2$	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*	$B_r[i] = B_1[i] - B_2[i]$.
$B_r = \textit{PROJECT}(\mathbf{B_x})$	Arguments $\{B_r, \mathbf{B_x}\}$ all have the same *length*, *orientation*, *table* and *type*; $\mathbf{B_x}$ contains at least 2 vectors; and $B_r = \textit{SUM}(\mathbf{B_x}, \textit{orientation}(\mathbf{B_x}))$	At every position i in 1 through $\textit{length}(B_r)$ there is exactly one vector v in $\mathbf{B_x}$ such that $v[i]$ is a non-blank value, then $v[i] = B_r[i]$.
$B_r = \textit{RANK}(B_x)$	$\textit{integer}(B_r)$; $\textit{numeric}(B_x)$; and $\textit{length}(B_r) = \textit{length}(B_x)$	The values in B_r represent the rank (from largest to smallest) of the values in B_x (including ties)
$B_r = \textit{PREV}(B_r) + B_{pos} - B_{neg}$	Arguments $\{B_r, B_{pos}, B_{neg}\}$ are all *numeric* and all have the same *length*, which is at least 2	The values in B_r are a running total, each value $B_r[i] = B_r[i-1] + B_{pos}[i] - B_{neg}[i]$.
$B_r = \textit{SUM}_{\textit{row}}(\mathbf{B_x})^\dagger$	B_r and $\mathbf{B_x}$ are *numeric*; $\textit{columns}(\mathbf{B_x}) \geq 2$; and $\textit{rows}(\mathbf{B_x}) = \textit{length}(B_r)$	Each value in B_r is obtained by summing over the corresponding row in $\mathbf{B_x}$.
$B_r = \textit{SUM}_{\textit{col}}(\mathbf{B_x})^\dagger$	B_r and $\mathbf{B_x}$ are *numeric*; $\textit{rows}(\mathbf{B_x}) \geq 2$; and $\textit{columns}(\mathbf{B_x}) = \textit{length}(B_r)$	Each value in B_r is obtained by summing over the corresponding column in $\mathbf{B_x}$.
$B_r = \textit{SUMIF}(B_{fk}, B_{pk}, B_{val})^\dagger$	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 $\textit{ALLDIFFERENT}(B_{pk})$	The value for $B_r[i]$ is obtained by summing all values $B_{val}[j]$ where $B_{fk}[j] = B_{pk}[i]$
$B_r = \textit{SUMPRODUCT}(B_1, B_2)$	Arguments $\{B_r, B_1, B_2\}$ are all *numeric*; $\textit{length}(B_1) = \textit{length}(B_2) \geq 2$; and $\textit{rows}(B_r) = \textit{columns}(B_r) = 1$	$B_r[i] = \sum_{i=1}^{\textit{length}(B_1)} B_1[i] \times B_2[i]$.

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 template*s. A *constraint template* s is a triple $s =$ *(Syntax, Signature, Definition)*:

- *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.
- ~~the~~ *Signature* defines the requirements that the arguments of the predicate must satisfy. This can concern any property 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 [13], the Signature is known as the *bias* of the learner, it specifies when the constraint its arguments are well-formed.

- *Definition* is the actual definition of the constraint that specifies when the constraint holds. Given an assignments 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.

Example 2. Several constraint templates are illustrated in Table I. A non-trivial example is the constraint template for the row-based sum:

- Syntax: $B_r = SUM_{row}(\mathbf{B_x})$, where B_r and $\mathbf{B_x}$ are the arguments;
- Signature: B_r has to be a single vector (*size* $= 1$) while $\mathbf{B_x}$ can be a block (*size* $>= 1$), which can be derived from the use of a normal or **bold** font. The two 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. While not strictly needed, we also add to the bias of this template that the number of columns to sum over is larger than 2 as a block with two columns will also be captured by the $B_r = B_1 + B_2$ constraint;
- Definition: each value in the vector B_r is obtained by summing over the corresponding row in $\mathbf{B_x}$.

It is helpful to see the analogy of constraint templates with first order logic (FOL) and constraint satisfaction. From a FOL perspective, the name of the constraint (e.g. *SUM*) is just the predicate name, and the arguments B_r and B_x are the 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 $B_r = RANK(B_x)$ and the spreadsheet in Figure 1a, when we write $T_1[:, 8] = RANK(T_1[:, 7])$, 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 $Sig_s(B_1, ..., B_n)$ (respectively $Def_s(B_1, ..., B_n)$) is satisfied. Likewise, the template is satisfied if both the signature and definition are satisfied; in logic programming, we would write a Prolog-like clause: $s(B_1, ..., B_n) \leftarrow Sig_s(B_1, ..., B_n) \wedge Def_s(B_1, ..., B_n)$. Under this interpretation, the term constraint and constraint template can be used interchangeably.

Definition 3. A **valid argument assignment** of a constraint s is a tuple $(B_1, ..., B_n)$ 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 **Tabular Constraint Learning Problem** as follows:

Definition 4. *Tabular Constraint Learning.*
Given a set of instantiated blocks $\mathcal{B}$ and a set of *constraint templates* $\mathcal{S}$: **find** all constraints $s(B'_1, ..., B'_n)$ where $s \in \mathcal{S}$ and $(B'_1, ..., B'_n)$ is a valid argument assignment of the template s.

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 constraint. This can be interpreted as follows: the signature of the constraint consists of its own signature plus the signature of the depending constraint, and its definition of its own definition plus the definition of the depending constraint. In FOL, we can see that one constraint entails the other, for example if $PERMUTATION(B_x)$ holds for a block B_x, then $ALLDIFFERENT(B_x)$ also holds.

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

a) 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.

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 may be logically equivalent due to their nature, e.g. for addition/subtraction and product/division: $B_r = B_1 \times B_2 \equiv B_1 = B_r / B_2$.

When the data has rows or columns containing exactly the same data, then any constraint with such a vector in its argument assignment will also have an argument assignment with the other equivalent vectors.

Because dealing with redundancy is often application-dependent, we explain in the next section our generic method for finding all constraints. We describe some optimisations that avoid obvious equivalences in Section III-D and 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 maximally contiguous, type-consistent *superblocks*
3) Generate for each constraint template all valid argument assignments in two steps:
   a) For each constraint c, generate all atoms $c(B_1, \ldots, B_n)$ where each B_i is a maximal *superblock* and the B_i are compatible with the signature.
   b) For each generated atom $c(B_1, \ldots, B_n)$, find all valid $c(B'_1, \ldots, B'_n)$ such that for all i holds $B'_i \sqsubseteq B_i$ and the signature and definition of $c(B'_1, \ldots, B'_n)$ 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.

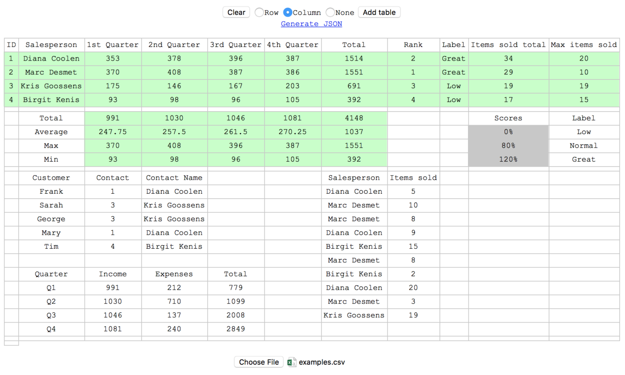
Instead, we divide this step into two parts: in the first part, we will not reason over individual (sub)blocks and their data, but rather over the properties of the maximal superblocks. Consider Table 1 and 2 in Figure 1a and the SUM_{col} constraint template. Instead of considering all possible subblocks of T1 and T2, we reason over the properties of the superblocks first, $B2$ and $B4$ are not numeric so can be immediately discarded. Looking at the sizes, $B1$ contains only one column and ~~$B5$ 2,~~ however, the rows in $T2$ have length 4 so we only need to consider superblocks of size at least 4, which only $B3$ satisfies.

In the second part, we start from an atom $c(B_1, \ldots, B_n)$ with maximal 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, each of the vectors of $B6$ will be considered as candidate for the left-hand side, and one can enumerate all subblocks for the right-hand side and verify the signature (e.g. at least size 4) and test the definition for each of the rows.

We now describe in more detail how the headerless tables are extracted and how the maximal superblocks are generated from them (step 1 and 2, Section 4.1), how the candidate superassignments are generated (step 3a, Section ~~4.2)~~ and how the actual assignments are extracted from that (step 3b, Section 4.3). We then describe a number of optimizations that we perform to speed up step 3 and 4 (Section 4.4).

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



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			Scores	Label
	Average	247.75	257.5	261.5	270.25	1037			0%	Low
	Max	370	408	396	387	1551			80%	Normal
	Min	93	98	96	105	392			120%	Great
	Customer	Contact	Contact Name			Salesperson	Items sold			
	Frank	1	Diana Coolen			Diana Coolen	5			
	Sarah	3	Kris Goossens			Marc Desmet	10			
	George	3	Kris Goossens			Marc Desmet	8			
	Mary	1	Diana Coolen			Diana Coolen	9			
	Tim	4	Birgit Kenis			Birgit Kenis	15			
						Marc Desmet	8			
	Quarter	Income	Expenses	Total		Birgit Kenis	2			
	Q1	991	212	779		Diana Coolen	20			
	Q2	1030	710	1099		Marc Desmet	3			
	Q3	1046	137	2008		Kris Goossens	19			
	Q4	1081	240	2849						

Fig. 2: This figure shows a simple table selection tool. A user selects rectangular ranges of cells that correspond to (headerless) tables, an explicit orientation can also be provided.

rows and columns is. However, detecting tables and headers is a complex problem on its own [4]. 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 (Figure 2). 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.

Algorithm 1 Learn tabular constraints

```
1: Input: $S$ – constraint templates, $\mathcal{B}$ – maximal blocks
2: Output: $C$ – learned constraints
3: procedure LEARNCONSTRAINTS($\mathcal{B}$, $S$)
4:     $C \leftarrow \emptyset$
5:     for all $s$ in $S$ do
6:         $A \leftarrow \mathit{SuperblockAssignments}(s, \mathcal{B})$
7:         for all $(B_1, \ldots, B_n) \in A$ do
8:             $A' \leftarrow \mathit{Subassignments}(s, (B_1, \ldots, B_n))$
9:             for all $(B'_1, \ldots, B'_n) \in A'$ do
10:                $C \leftarrow C \cup \{c_s(B'_1, \ldots, B'_n)\}$
11:    return $C$
```

To find row-blocks, each row is treated as a vector and must be type-consistent; similarly for 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 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 will use constraint satisfaction technology in the first step to efficiently enumerate superblock assignments that are compatible with the signature.

1) Generating superblock assignments: Given a constraint template s and the set of all maximal blocks $\mathcal{B}$, the goal is to find all combinations of maximal blocks that are consistent with the constraint signature. An argument assignment $(B_1, ...B_n)$ is *consistent* 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.

Furthermore, the choice of one argument can influence the possible candidates 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 *consistent* with the requirements defined in the signature.

TABLE II: Translation of signature requirements to superblock constraints. The requirements that do not need to be relaxed are: *length*$(B) = x$, *orientation*$(B) = x$, *table*$(B) = x$ and *size*$(B) \geq x$.

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

Table II shows how the conversion from requirements of the signature to CSP constraints happens: 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 subgroups might contains fewer vectors. Finally, restrictions on the number of rows or columns behave as length or size constraints based on the orientation of the group they are applied to. Subgroups of row-oriented (column-oriented) group 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 satisfying solutions. These solutions correspond to the valid superblock assignments for constraint template s.

Example 3. Consider the constraint template $B_r = \mathit{SUM}_{col}(\mathbf{B_x})$, 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 $B_r = \mathit{SUM}_{col}(\mathbf{B_x})$ (see Table I) will be translated into constraints:

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

2) Generating subblock assignments: 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, $B'_i \subseteq B_i$) that satisfy both the signature and the definition of template s.

Example 4. Consider the sum-over-rows constraint template $B_r = \mathit{SUM}_{row}(\mathbf{B_x})$. 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, B_r contains more than one vector and *size*$(\mathbf{B_x}) \neq$ *length*(B_r). 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: $\forall i, B_r[0][i] = \sum_j \mathbf{B_x}[j][i]$, where $B[i][j]$ indicates the ith value of the jth vector in B. In Figure 1a there is exactly one such satisfying 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 the signature of the definition may already not be satisfie~~d, or the first element in the data.~~ ?

Hence, a simple generate-and-test will usually suffice. Given a superblock assignment, all disjoint subassignments are generated taking into account the required size, columns or rows. The disjoint subassignments are then sequentially tested using a template-specific procedure *Test* and valid subassignments are returned. The *Test* procedure verifies the exact signature (e.g. subtypes will not have been checked yet) and checks whether the definition is satisfied for all data in the blocks.

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 $(B'_1, \ldots, B'_n) \in \{(B'_1, \ldots, B'_n) | \forall i : B'_i \sqsubseteq B_i \wedge \nexists i, j, i \neq j : B'_i \sqsubseteq B'_j\}$ do
        if $Test((B'_1, \ldots, B'_n))$ 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 and extensibility of our system as well as to eliminate some obvious redundant constraints.

1) Template dependencies: As discussed in Section II-D1, some constraints depend on other constraints. This is apparent when looking at the template signatures in Table I, where one can see that some signatures depend on other constraints, such as *LOOKUP* depending on *FOREIGNKEY*. If a constraint template s_1 includes template s_2 in its signature, we say that s_1 depends on s_2.

In Inductive Logic Programming, one often exploits implications between constraints to structure the search space [13]. 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 3 shows the dependency graph extracted from the signature definitions 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, the input templates S (see Algorithm 1) are reordered to agree with the partial ordering imposed by $\mathcal{D}$.

The dependency graph represents a partial ordering and is ~~used produce a~~ linear ordering of the given templates such that more general templates occur before templates that depend on them (e.g. *ALLDIFFERENT* before *FOREIGNKEY*). Considering Algorithm 1, this step can be seen as preprocessing the input S. Learning in this order allows templates to use constraints they depend on as these will have already been found. This is exploited in the *SuperblockAssignments* step to speedup the algorithm by reducing the search space. Let us illustrate this step with an example.

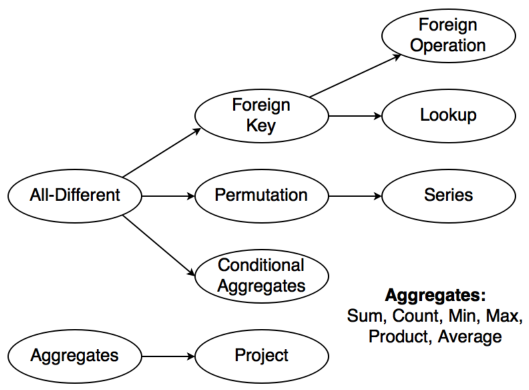


Fig. 3: Dependency graph; an arrow from s_1 to s_2 indicates that s_2 depends on s_2 (the Signature of s_2 includes s_1). The graph is extracted from the Signature column in Table I.~~SAMUEL: Correct aggregates~~

Example 5. Consider *FOREIGNKEY*(B_{fk}, B_{pk}) whose signature includes *ALLDIFFERENT*(B_{pk}) and assume that C are the constraints found for *ALLDIFFERENT*. Instead of generating assignments for B_{pk} using the given blocks $\mathcal{B}$, we will only consider (sub)blocks $P = \{B' | \mathit{ALLDIFFERENT}(B') \in C\}$. P can be seen as a set of partial assignments in which the blocks for certain arguments have been fixed.

The *SuperblockAssignments* implementation now takes into account previously found constraints to obtain partial assignments P for a template. Instead of generating one CSP to find all assignments, a CSP is generated for every $p \in P$ which finds all assignments completing p. The procedure to generate CSPs remains the same, except that the partial assignment is used to fix the domains of the dependent variables.

This approach has the additional benefit of ensuring that the (sub)blocks assigned to the dependent arguments already satisfy both the signature and definition of the original template. Therefore, the implementation of *Subassignments* does not have to take these blocks into account.

2) Redundancy: We consider two types of redundancy that we aim to eliminate during the 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, *TaCLe* currently contains only product ($B'_R = B'_1 \times B'_2$) and not division ($B'_1 = B'_R / B'_2$) as well as difference ($B_r = B_1 - B_2$) and not sum ($B_1 = B_r + B_2$). These choices are based on the prevalence of product and difference in financial spreadsheet data, however, division and sum could easily be ~~added to,~~ as can other constraints.

A last type of redundancy we aim to reduce is that there may be multiple *overlapping* subblocks that satisfy the definition of a constraint template. For example, consider the constraint

$B_r = SUM_{col}(B[1:n])$ where $B[k:n]$ consists of only zeros, then $B_r = SUM_{col}(B[1:j])$ will be true for all $k-1 \leq j \leq n$. In *TaCLe* we have, therefore, chosen to only consider *maximal* subblocks.

As a result, in some cases a maximal subblock might falsely include irrelevant columns. Consider $B_1 = [[200, 300], [200, 150], [1, 2]]$ and $B_2 = [200, 300]$, then *TaCLe* will find the constraint $B_2[1] = MAX(B_1[1:3])$. Should, the target constraint be $B_2[1] = MAX(B_1[1:2])$, then the last vector of B_1 must be split off into a separate block before the above approach can find it.

Alternatively, we can output all valid argument assignments instead of only maximal subblocks, e.g. $B_2[1] = MAX(B_1[1])$, $B_2[1] = MAX(B_1[1:2])$ and $B_2[1] = MAX(B_1[1:3])$, however, this might produce many redundant constraints and introduce constraints that accidentally cover too little data.

3) Constraints: The constraints that are currently supported in our system are shown in Table I. Their inclusion was based on the analysis of the most popular Excel constraints as well as the spreadsheets we encountered. All these constraints are available in popular spreadsheet software, except for the structural constraints (*ALLDIFFERENT*, *PERMUTATION* and *FOREIGNKEY*)~~. These constraints added so~~ they can be used in the signature of other constraint templates; they are also popular in constraint satisfaction [1].

IV. Evaluation

SERGEY: we need to add a summary of the dataset, avg number of constraints, cells, rows, columns In this section we experimentally validate our approach. We studied various questions, most notably with what accuracy our algorithm can find constraints.

The implementation is illustrated using a case study on the spreadsheet corresponding with the previously introduced example (figure 1a). In order to quantify the results and generalize our findings we also evaluate our algorithm on a benchmark of 30 (SAMUEL: check number) spreadsheets that we assembled from various sources.

There are three main categories of spreadsheets in the benchmark: spreadsheets from an exercise session teaching Excel, spreadsheets from tutorials online and publicly available spreadsheets such as crime statistics or financial reports that demonstrate more real world usage[1].

In this section we focus on *functional* constraints that could be used in spreadsheets, ignoring constraints such as all-different or foreign-key. All experiments were run on a Macbook Pro, Intel Core i7 2.3 GHz with 16GB RAM.

A. Case study

TIAS: Move explanation of origin to first time it is introduced Let us illustrate our implementation using the example presented in Figure 1a. This example combines several smaller exercise session examples into one spreadsheet.

[1](attached to to the submitted paper)

1) Results: *TaCLe* takes a few seconds to find the 18 constraints in Figure 1b. These include 5 spurious *RANK* constraints, such as $T_1[:,1] = RANK(T_1[:,5])$, that are true by accident. Moreover, there is 1 *LOOKUP* constraint that was is present in the original spreadsheet (looking up ID based on Salesperson).

For this example our primary goal of finding all original constraints is achieved, our accuracy or recall is 1.0. The implementation also returns 6 additional constraints, therefore, our precision is $12/18 = 0.67$. This precision is, as we will show, rather low compared to other spreadsheets. However, this example contains many short vectors which increases the chance for constraints to be true by accident.

B. Benchmark

There are three main categories of spreadsheets in the benchmark: spreadsheets from an exercise session teaching Excel at an affiliated University, spreadsheets from tutorials online and publicly available spreadsheets such as crime statistics or financial reports that demonstrate more real world usage[2]. The case study is also included.

All spreadsheets have been converted to CSV files. Definitions of tables were added manually, providing a way to compare results and overcoming shortcomings in the table detection algorithm (in some harder cases groups where also provided manually).

For every spreadsheet a ground-truth has been provided manually, specifying the essential (or original) constraints that are expected to be discovered. We consider three types of constraints:

Implemented Constraint that have been implemented and are expected to be found (all constraints in Table I)

Essential Constraint that have or have not been implemented but could be found using our algorithm (e.g. fuzzy conditional sum)

Non-trivial Constraints currently outside of the scope of this system (e.g. generic nested mathematical or logical formulas or n-ary constraints)

We examine three main questions concerning the accuracy (recall), redundancy (precision) and efficiency of our approach, our main focus being accuracy.

Q1. How accurate is the approach?: Fortunately, for the implemented constraint templates, our system achieves a recall of 1.0, i.e. it is able to find all of the supported constraints (90) on the benchmark suite. There are only three templates that have not been implemented yet, so w.r.t. the ground truth the total recall of our system is 0.97.

Q2. How many redundant constraints are discovered?: Our primary focus was, as mentioned before, accuracy, therefore we sometimes traded off less redundancy for more accuracy. The motivation is that solutions can still be pruned using entailment or heuristics afterwards. The constraints we considered to be redundant are either duplications (results that can be calculated in different ways, one of which seems superior) or constraints that hold by accident.

[2](attached to to the submitted paper)

	Total	**Average per spreadsheet**
Constraints	93	2.91
Accuracy	96.77%	94.27%
Redundant	23.14% (28)	8.33% (0.88)
Speed (s)	$16.12s \pm 0.62s$	$0.50s \pm 0.02s$

TABLE III: Overview evaluation on essential constraints of the collected Spreadsheet dataset

Across all spreadsheets our implementation finds 121 constraints, 28 (23.14%) of which are redundant. However, the average redundancy per spreadsheet is only 8.33%. Examining the redundant constraints reveals that many (12) of them are duplications occurring in a single spreadsheet. Many of the remaining constraints are duplications stemming from the overlapping role of difference and sum. Accidental constraints are limited to the 5 rank constraints that were discussed in the case study.

Redundant constraints can be reduced in different ways.samuel Duplications should be detected in a post-processing step that uses entailment or heuristics to filter constraints. Accidental constraints may be detected through heuristics, however, they can be also be removed by adding additional data.

Q3. How fast is the algorithm?: Concerning the speed of the algorithm we also prioritized accuracy when a trade-off had to be made. For the 32 spreadsheets in the benchmark our implementation ran in $16.12s \pm 0.62s$. The execution times vary widely though between spreadsheets, only four spreadsheets taking more than $0.2s$. Most of the execution time in these cases goes towards searching either aggregate constraints or conditional aggregates. The search for aggregates will be slow on spreadsheets containing larger groups of numeric data. For conditional aggregates the number of candidate primary keys (all-different) and numeric vectors determines the running time (e.g. the case study example).

SAMUEL: Expand summary SERGEY: would it make sense to specify the size of a spreadsheet in the number of non-zero cells?

a) Dependencies: In order for the algorithm to run efficiently it is crucial to use dependencies and find constraint incrementally whenever possible. This avoids some of the explosion of combinations for constraints that have many arguments. For example, using foreign keys as a base constraint to find conditional aggregates reduces the running time for the case study from about 3 seconds to below 1 second. Unfortunately, this assumption is sometimes too strong, when users are interested in aggregates for only some of the keys that are present in the data.

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 a snapshot of spreadsheet data to predict new values before they are written by the user. The 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. A *functional* constraint corresponds to a spreadsheet formula. Such a constraint, e.g. $B_r = \mathit{SUM}_{col}(\mathbf{B_x})$, is able to compute a result (B_r) based on its input ($\mathbf{B_x}$). If all the input vectors / blocks have been filled and the result is incomplete, the constraint can predict the result value(s).

Let us illustrate this on one of the spreadsheets from the gathered dataset. Consider Figure 4b containing two tables. By learning constraints on the tables excluding the last row, 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 last row. 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 $T_2[4, 3] = 428128$.

In case multiple constraints predict different values for the same cell, the algorithm will have to either prioritize constraints based on their specificity or a heuristic, not make a prediction, or present the choice to the user.

B. Error detection

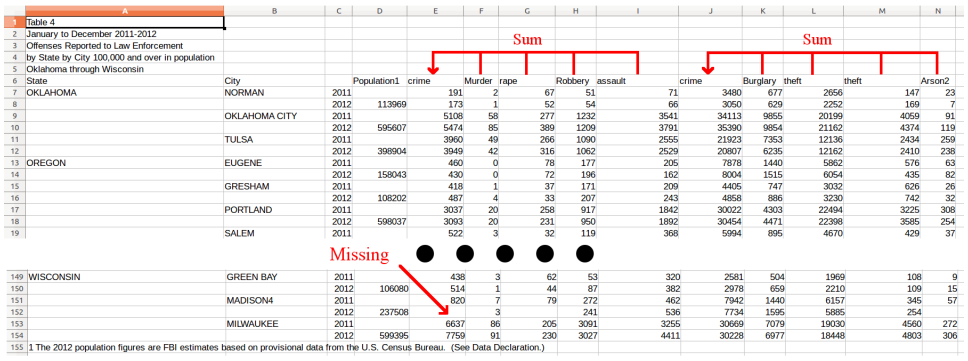
SERGEY: Either we need to elaborate on the online detection or remove offline-online and simplify the rest, otherwise it is not clear what is going on SAMUEL: Is this still the case?

We consider two cases of error detection. The first setting is the *online* detection of errors which is similar to the autocompletion setting. However, we look at *conflict* constraints, i.e. constraints that were learned on the snapshot data but do not agree with the newly added data ΔD. Such conflict constraints can indicate errors in the input. However, such conflicts can be wrongly caused by spurious constraints. Therefore, conflicts should only be detected if the snapshot data is large enough.

The second type of error detection is *offline* and attempts to detect errors in a spreadsheet. Consider, for example, a use case concerning crime statistics provided by the FBI. An extract of the spreadsheet is depicted in Figure 4a. When run on this sheet, our system is able to detect the second sum constraint, however, a missing value prevents the first sum constraint from being learned. A possible approach to detect such missing or wrong values is to look at a sample of the table that excludes the erroneous row, e.g. trying to leave out each row, and compare the constraints learned on this sample with the constraints learned by adding the row containing the mistake. The sample needs to large enough such that we have enough confidence that the constraints are correct. For example, in Figure 4a all but one row satisfy the constraint, which is a strong indication that there is a constraint violation. If we define the sample to be the snapshot data and the added row to be the additional input ΔD, this task can be formulated as online error detection and dealt with in the same way.

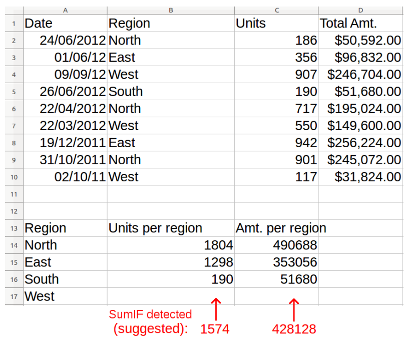
VI. Related Work

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



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Table 4													
2	January to December 2011-2012													
3	Offenses Reported to Law Enforcement													
4	by State by City 100,000 and over in population													
5	Oklahoma through Wisconsin													
6	State	City		Population1	crime	Murder	rape	Robbery	assault	crime	Burglary	theft	theft	Arson2
7	OKLAHOMA	NORMAN	2011		191	2	67	51	71	3480	677	2656	147	23
8			2012	113969	173	1	52	54	66	3050	629	2252	169	7
9		OKLAHOMA CITY	2011		5108	58	277	1232	3541	34113	9855	20199	4059	91
10			2012	595607	5474	85	389	1209	3791	35390	9854	21162	4374	119
11		TULSA	2011		3960	49	266	1090	2555	21923	7353	12136	2434	259
12			2012	398904	3949	42	316	1062	2529	20807	6235	12162	2410	238
13	OREGON	EUGENE	2011		460	0	78	177	205	7878	1440	5862	576	63
14			2012	158043	430	0	72	196	162	8004	1515	6054	435	82
15		GRESHAM	2011		418	1	37	171	209	4405	747	3032	626	26
16			2012	108202	487	4	33	207	243	4858	886	3230	742	32
17		PORTLAND	2011		3037	20	258	917	1842	30022	4303	22494	3225	308
18			2012	598037	3093	20	231	950	1892	30454	4471	22398	3585	254
19		SALEM	2011		522	3	32	119	368	5994	895	4670	429	37
149	WISCONSIN	GREEN BAY	2011		438	3	62	53	320	2581	504	1969	108	9
150			2012	106080	514	1	44	87	382	2978	659	2210	109	15
151		MADISON4	2011		820	7	79	272	462	7942	1440	6157	345	57
152			2012	237508		3		241	536	7734	1595	5885	254	
153		MILWAUKEE	2011		6637	86	205	3091	3255	30669	7079	19030	4560	272
154			2012	599395	7759	91	230	3027	4411	30228	6977	18448	4803	306
155	1 The 2012 population figures are FBI estimates based on provisional data from the U.S. Census Bureau. (See Data Declaration.)													

(a) Real world tabular constraint reconstruction: FBI crime statistics



	A	B	C	D
1	Date	Region	Units	Total Amt.
2	24/06/2012	North	186	$50,592.00
3	01/06/12	East	356	$96,832.00
4	09/09/12	West	907	$246,704.00
5	26/06/2012	South	190	$51,680.00
6	22/04/2012	North	717	$195,024.00
7	22/03/2012	West	550	$149,600.00
8	19/12/2011	East	942	$256,224.00
9	31/10/2011	North	901	$245,072.00
10	02/10/11	West	117	$31,824.00
11				
12				
13	Region	Units per region	Amt. per region	
14	North	1804	490688	
15	East	1298	353056	
16	South	190	51680	
17	West			

(b) SERGEY: Samuel, Tias have a look at it, should be corrected and extended, like with groups indications and formulas?

First, it borrows techniques from logical and relational learning [13], as it finds constraints in multiple relations (or tables). However, unlike typical logical and relational learning approaches, it focuses on data in spreadsheets and on constraints that can be column or row-wise. 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 [14, 9]. Still 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 [15, 11] although this line of research has focused on more specialized techniques for specific types of constraints. Furthermore, the constraints used in spreadsheets are often quite different than those in databases.

Second, there exist several algorithms in the constraint programming community that induce constraint (programs) from one or more examples and questions. Two well-known lines of research 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 tables from the vector. 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 with our technique are that we assume the tables are given (a reasonable assumption if one starts from a spreadsheet), and also that ModelSeekers constraints are the typical ones from the constraint programming communityTIAS: which is all over integers, we also over strings and real-valued, AND, not one vector but we have many indep. blocks, which are aimed at combinatorial optimization and scheduling rather than what is typically used in spreadsheets. Quacq [2] and its predecessor Conacq [3] acquire a conjunction of constraints using techniques inspired on Mitchells version spaces to process examples, some variants also asking informative questions to the user. As the conjunction for typical constraint programming problems (such as Sudoku) can be quite large, a key problem is to cope with redundant constraintsTIAS: it looks only at pairwise constraints, so an important factor is how fast the algorithm can converget to the entire constraint set. Redundant and spur also issue for these methods. As Modelseeker, these systems focus on the types of constraints in combinatorial optimization but unlike ModelSeeker and our approach, they do not assume the variables are organized in tabular form.

Third, our work borrows from the seminal work of Gulwani et al. [6] on program synthesis for tabular data that 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 would generate a program that produces the output "T.B." for the inputs "Blair" and "Tony", would also apply it to automatically generate values for 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." TIAS: only strings?, FlashFill learns correct programs from very few examples. Flashfill has been extended to e.g. FlashExtract [10] to produce a set of normalized database tables TIAS: can be more precise, does it add formulae? or this is real DB database, e.g a schema and PK/FK constraints? starting from naively filled out spreadsheets. 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 [5], 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 a large set of constraints from CSV data. Moreover, the amount of redundant constraints found by the system is limited despite the lack of more sophisticated post-processing steps.

a) Future work: There are multiple directions for future work on this topic. Integrating the system in an interactive setting would offer users the possibility to easily receive and provide feedback. Moreover, the system can be adapted to deal with additional types produce less redundant constraints through the use of (heuristic) filtering or post-processing.

Currently, the system only learns single constraints, however, extending our approach to nested constraints would allow more expressive concepts to be learned. This shift would bring the approach in line with programming by example.

Aside from finding errors, the system can be extended to deal with noise and the ability to learn soft constraints. Soft constraints are the (potentially weighted) constraints that hold only on some of the data. This would extend the approach to new application domains as well as provide more native error detection.

TIAS: write limitations: single cons, complete rows/cols only, (what else?) and deal with it as future work.

References

[1] Nicolas Beldiceanu and Helmut Simonis. *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, 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, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.

[3] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, pages 23–34, 2005.

[4] Jing Fang, Prasenjit Mitra, Zhi Tang, and C. Lee Giles. Table header detection and classification. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[5] 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.

[6] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[7] 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.

[8] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, 1(section II):45–52, 2010.

[9] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 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 '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 55, 2014.

[11] Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *Data Knowl. Eng.*, 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, 2015.

[13] Luc De Raedt. *Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies)*. Springer-Verlag New York, Inc., 2008.

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

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