Noname manuscript No.
(will be inserted by the editor)

Learning constraints in spreadsheets and tabular data

Samuel Kolb* · **Sergey Paramonov*** · **Tias Guns** · **Luc De Raedt**

the date of receipt and acceptance should be inserted later

Abstract Spreadsheets, comma separated value files and other tabular data representations are in wide use today. However, writing, maintaining and identifying good formulas for tabular data and spreadsheets can be time consuming and error-prone. We investigate the automatic learning of constraints (formulas and relations) in raw tabular data in an unsupervised way. We represent common spreadsheet formulas and relations through predicates and expressions whose arguments must satisfy the inherent properties of the constraint. The challenge is to automatically infer the set of constraints that is present in the data, without labeled examples or user feedback. We propose a two-stage generate and test method where the first stage uses constraint solving techniques to efficiently reduce the number of candidates, based on the predicate signatures. Our approach takes inspiration from inductive logic programming, constraint learning and constraint satisfaction. We show that we are able to accurately discover constraints in spreadsheets from various sources.

1 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 relations 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 all tables. For example, the influential Reinhart-Rogoff economical paper "Growth in a Time of Debt" [18] had some of its claims contested [9] after an investigation showed that the used Excel sheets contained mistakes in formulas.

Such issues and mistakes could be overcome through the development of intelligent user-assisting tools that would be able to automatically discover what constraints and for-

KU Leuven
* Shared first author

mulas hold in a spreadsheet. This is the problem we study in this paper. We envision that automatic constraint discovery can enable: auto-completion, error checking, formula suggestion, rich import and data compression. Auto-completion can help users quickly populate spreadsheets by suggesting values for unfilled cells based on constraints that have previously been detected within the spreadsheet. Error checking can identify erroneous values in spreadsheets, for example, while typing, by noticing that previously discovered constraint are violated by newly entered values, or offline by selectively leaving out values. Both of these applications can facilitate the use of spreadsheets even for users who do not explicitly use formulas at all. Furthermore, these applications are able to explain what constraints are responsible for suggesting or rejecting a value. Other potential applications include formula suggestion, which is a variant of auto-completion that does not suggest a *value* but provides a formula which computes the user provided value. This setting can help users in finding the formula to perform a certain task or finding the right syntax for a formula. Rich import denotes importing semi-structured data from another source such as CSV files and extracting the formula's that were used implicitly or explicitly. A similar approach could be used to compress the data: instead of storing all data in a file, one can identify rows or columns generated by formula and store just that formula instead of the values.

To this aim, we investigate whether machine learning and knowledge discovery techniques can be used to learn constraints (formulas and other relations) in spreadsheet data in an unsupervised way. From a machine learning point of view this is an unconventional problem because the data is in tabular form, but the constraints we wish to learn can involve both rows and columns of the table. Being unsupervised, there is no labeled information either, although for every possible function one can verify whether a certain input satisfies the definition. From a data mining point of view, the data is relational on the one hand, since one can have multiple tables with relationships between them, but also consists of mixed textual and numeric types. More closely related is clausal discovery [17, 12], learning CSP constraints [2, 3, 1] and dependency discovery in databases [19]. But an important difference remains that we want to learn constraints on both columns and rows over integer, floating point as well as textual data. Our inspiration comes from work on program synthesis, in particular Flashfill [8], where the definition of a string-manipulation function is learned in spreadsheet data from very few examples.

The question that we ask in this paper is: is it possible to learn the set of constraints present in a spreadsheet, from just tabular spreadsheet data? The main challenge is the number of possible constraints and combinations of rows and columns that need to be considered as input to the constraints. To answer this challenge we propose a general-purpose method and system, named *TaCLe* (from: Tabular Constraint Learner, pronounced "tackle"), for discovering row-wise and column-wise constraints. Constraints on contiguous rows and columns are most common in spreadsheets, with semantically related items stored in rows or columns and formula's usually ranging over contiguous ranges, as formulas parallel to a table are often *dragged* over its entire span. Our contributions are as follows:

- we define the tabular constraint learning problem, where the goal is to find constraints that range over entire rows or columns in an unsupervised way;
- we propose an effective two-stage generate-and-test method where the first stage reasons only over properties of contiguous blocks of rows/columns, and the second stage continues to investigate individual rows and columns and their content;
- furthermore, in the first stage we use a constraint solver to efficiently enumerate all combinations of maximally contiguous blocks compatible with the arguments of the candidate constraints

– experiments on different publicly available spreadsheets show that the system is able to extract constraints with high precision and recall.

This paper is organized as follows. Section 2 discusses related work. Section 3 introduces the relevant concepts and defines the problem formally while Section 4 presents our approach. This is evaluated in Section 5 after which we show how to realize two of the motivating applications using our system in Section 6, and we conclude in Section 7.

2 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 discovers constraints in multiple relations or tables. Unlike typical logical and relational learning approaches, it focuses on data in spreadsheets and constraints involving both columns and rows in the sheets. Furthermore, it derives a set of simple "atomic" constraints rather than a set of rules that each consist of a conjunction of literals as in clausal discovery [17, 12]. 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. Tabular constraint learning also connects with the logical generalization of given formulas in a spreadsheet [11], where the existing formulas are abstracted by introducing parameters in the place of particular cell references. The key difference here is that we discover constraints/formulas in the data, we do not have or use any prior information on the formulas in a spreadsheet.

Second, there exist a number of algorithms in the constraint programming community that induce constraints 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 that hold given a large constraint library. 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, $\ldots$ and Modelseeker would then look for both column, row and even diagonal constraints. Key differences with our work are that we use spreadsheets with *multiple* tables and that ModelSeeker's constraints are restricted to integer values and constraints that are typical for the constraint programming community, e.g., aimed at combinatorial optimization and scheduling. Conacq [3] acquires a conjunction of constraints using techniques inspired by Mitchell's version space algorithm to process examples; its successor Quacq uses active learning to query the user. 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. In a similar spirit the field of equation discovery [20] focuses, guided by heuristics based on the structure of physical equations and unit measures, on the discovery of physical formulas over numbers and times series. However, they focus specifically on noisy physical data and formulas, e.g., a single variable can take values from a series of measurements to get a regression parameter estimate or they apply specific data transformation such as numerical calculation of time and partial derivatives. In other words, the method works with approximations and the equation structures are heavily biased towards parameter estimation, while our goal is to discover hard constraints that hold over a subset of the rows and columns.

Third, our work borrows from the seminal work of Gulwani et al. [8] on program synthesis for string manipulation in tabular data, which is incorporated in Microsoft's Excel. In FlashFill, the end-user might work on a table containing the surnames (column A) and

names (column B) of scientists. The first row might contain the values A1 = “Curie” and B1 = “Marie”, and the user might then enter in cell C1 the value “M.C.”. At that point FlashFill generates a program that produces the output “M.C.” for the inputs “Curie” and “Marie”, and also applies it to the other scientists (rows) in the table. While it might need an extra example to deal with names such as “Bell”, “Alexander Graham” and determine whether it should output “A.B.” or “A.G.B.”, FlashFill learns correct programs from very few examples. Flashfill has been extended to, e.g., FlashExtract [13] to produce a set of tables starting from an unstructured document (e.g., 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. Furthermore, FlashFill handles only single tables with textual data to obtain a string transformation program, while the examples we consider contain multiple tables with mixed numeric and textual data and multiple constraints in them. An 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.

Fourth, also related to this line of research is the work on deriving constraints in databases such as functional and multi-valued dependencies [19, 14] although that line of research has focused on more specialized techniques for specific types of constraints. Many of the discovery techniques rely on the database schema [6], which is not available in our case. Techniques that work directly on the data have been investigated for the discovery of functional dependencies between attributes [10]. The constraints and formulas we wish to learn go beyond that, including arithmetic, conditional arithmetic and *fuzzy* lookups, for example.

Finally, worth mentioning is BayesDB [15] and Tabular [7], 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 the “hard” constraints that hold in spreadsheets.

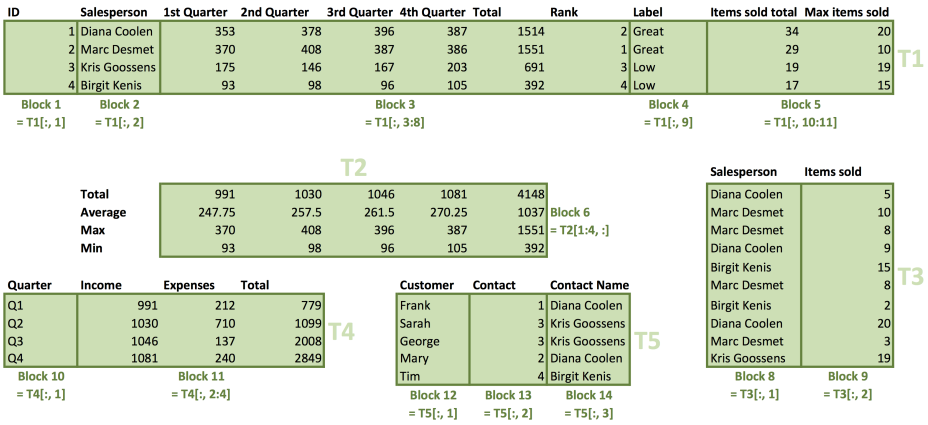
3 Formalization

Our goal is to automatically learn the constraints that hold between the rows and/or columns in a spreadsheet. This is applicable not just to data from spreadsheets, but to any data in tabular form.

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

3.1 Terminology

Spreadsheets and tabular data may conceptually consist of multiple tables, as in Figure 1a. Note that a table can contain one or more headers; however, we wish to reason over entire rows and columns of data, and hence we will consider **headerless tables** only. Headerless tables contain only data entries and are obtained by stripping away all contextual information such as headers.



(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. This example combines several Excel sheets based on exercises in the book "MS Excel 2010" [21].

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

(b) Constraints learned for the tables above, except 19 *ALLDIFFERENT*, 2 *PERMUTATION* and 5 *FOREIGNKEY* and 5 *ASCENDING* constraints not shown. Constraints marked with $*$ were not present in the original spreadsheets.

Fig. 1: Running example

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 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 or 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: $B = T[a{:}b, :]$ 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 independently of their orientation.

Blocks are used to reason about multiple rows or columns simultaneously. Requiring blocks to be contiguous is quite natural in the spreadsheet setting, as they logically group neighboring rows or columns. This matches the way people commonly construct formulas, by selecting a contiguous ranges in a sheet. For example, constraints such as summation often occur over contiguous blocks.

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 ($b_1 = T_1[:, 1]$, $b_2 = T_1[:, 2]$, $b_3 = T_1[:, 3{:}8]$, ...).

Definition 2 Block containment $\sqsubset$. A block B' is contained in a block B, $B' \sqsubset B$, 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 it means that: $B' \sqsubset B \Leftrightarrow B = T[a{:}b, :] \wedge B' = T[a'{:}b', :] \wedge a \leq a' \wedge b' \leq b$ 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 of block containment is $T_1[:, 3{:}6] \sqsubset T_1[:, 3{:}8]$, which contains the sales numbers of all employees for the four quarters.

3.2 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; each of those arguments will correspond to a block. Thus, a constraint is viewed as a relation or predicate of arity n in first order logic.
  Functions can be represented in a similar way. Any function $B_{n+1} = s(B_1, ..., B_n)$ that has n arguments and computes B_{n+1} can be represented with an $(n{+}1)$-ary predicate $s'(B_1, ..., B_n, B_{n+1})$.

Table 1: Constraint templates implemented in *TaCLe*. Only blocks in **bold** may contain more than one vector; *discrete*(x) is a shortcut for: *textual*$(x) \vee$ *numeric*(x). Templates marked with $*$ are *structural*, i.e. they are not functional and not available in most spreadsheet software. For templates marked with † (called *aggregate* and *conditional aggregate* constraints), the table shows variants for *SUM* but *TaCLe* also supports *MAX*, *MIN*, *AVERAGE*, *PRODUCT* and *COUNT*, e.g., $B_2 = MAX_{row}(\mathbf{B}_1)$ or $B_2 = COUNTIF(B_{fk}, B_{pk}, B_1)$.

Syntax	**Signature**	**Definition**
ALLDIFFERENT$(B)^*$	*discrete*(B)	Only different values in B. $i \neq j$: $B[i] \neq B[j]$.
PERMUTATION$(B)^*$	*numeric*(B) and *ALLDIFFERENT*(B)	The values in B are a permutation of the numbers 1 through *length*(B).
ASCENDING$(B)^*$	*numeric*(B)	$i < j$: $B[i] \leq B[j]$.
SERIES(B)	*integer*(B) and *PERMUTATION*(B)	$B[1] = 1$ and $B[i] = B[i-1] + 1$.
FOREIGNKEY$(B_{fk}, B_{pk})^*$	B_{fk} and B_{pk} (*p*rimary *k*ey) are *discrete*, from different *table*s, same *type* and *ALLDIFFERENT*(B_{pk})	Every value in B_{fk} also exist in B_{pk}.
$B_2 =$ *LOOKUP*(B_{fk}, B_{pk}, B_1)	B_{fk} and B_{pk} are *discrete*; arguments $\{B_{fk}, B_2\}$ and $\{B_{pk}, B_1\}$ within the same set have the same *length*, *table* and *orientation*; B_1 and B_2 have the same type; and *FOREIGNKEY*(B_{fk}, B_{pk}).	$B_2[i] = B_1[j]$ where $B_{pk}[j] = B_{fk}[i]$.
$B_2 =$ *LOOKUP*$_{fuzzy}(B_{fk}, B_{pk}, B_1)$	*Same as lookup*; except *ASCENDING*(B_{pk}) instead of *FOREIGNKEY*(B_{fk}, B_{pk})	$B_2[i] = B_1[j]$ where $B_{pk}[j] \leq B_{fk}[i]$, j maximal.
$B_3 = B_1 \times$ *LOOKUP*(B_{fk}, B_{pk}, B_2)	Arguments $\{B_3, B_1, B_{fk}\}$ are *numeric*, arguments $\{B_{pk}, B_2\}$ are *discrete* and within both sets all arguments have the same *length*, *table* and *orientation*; also *FOREIGNKEY*(B_{fk}, B_{pk}).	$B_3[i] = B_1[i] \times$ *LOOKUP*$(B_{fk}, B_{pk}, B_2)[i]$.
$B_3 = B_1 \times B_2$	B_3, B_1, B_2 are all *numeric* and have the same *length*	$B_3[i] = B_1[i] \times B_2[i]$.
$B_3 = B_1 - B_2$	Arguments $\{B_3, B_1, B_2\}$ are all *numeric* and have the same *length* and *orientation*	$B_3[i] = B_1[i] - B_2[i]$.
$B_3 = \frac{B_1 - B_2}{B_2}$	*Same as* $B_3 = B_1 - B_2$	$B_3[i] = (B_1[i] - B_2[i])/B_2[i]$.
$B_2 =$ *PROJECT*$(\mathbf{B}_1)$	Arguments $\{B_2, \mathbf{B}_1\}$ all have the same *length*, *orientation*, *table* and *type*; $\mathbf{B}_1$ contains at least 2 vectors; and $B_2 = SUM_{orientation(\mathbf{B}_1)}(\mathbf{B}_1)$	At every position i in 1 through *length*(B_2) there is exactly one vector v in $\mathbf{B}_1$ such that $v[i]$ is a non-blank value, then $v[i] = B_2[i]$.
$B_2 =$ *RANK*(B_1)	*integer*(B_2); *numeric*(B_1); and *length*(B_2) = *length*(B_1)	The values in B_2 represent the rank (from large to small) of the values in B_1, including ties.
$B_3 = B_1 - B_2 +$ *PREV*(B_3)	Arguments $\{B_3, B_1, B_2\}$ are all *numeric* and have the same *length* ≥ 2, $PREV()$ denotes previous value	$B_3[i] = B_1[i] - B_2[i] + B_3[i-1]$.
$B_2 = SUM_{row}(\mathbf{B}_1)^\dagger$	B_2 and $\mathbf{B}_1$ are *numeric*; *columns*$(\mathbf{B}_1) \geq 2$; and *rows*$(\mathbf{B}_1)$ = *length*(B_2)	$B_2[i] = \sum_{j=1}^{columns(\mathbf{B}_1)}$ *row*$(i, \mathbf{B}_1)[j]$
$B_2 = SUM_{col}(\mathbf{B}_1)^\dagger$	B_2 and $\mathbf{B}_1$ are *numeric*; *rows*$(\mathbf{B}_1) \geq 2$; and *columns*$(\mathbf{B}_1)$ = *length*(B_2)	$B_2[i] = \sum_{j=1}^{rows(\mathbf{B}_1)}$ *column*$(i, \mathbf{B}_1)[j]$
$B_2 =$ *SUMIF*$(B_{fk}, B_{pk}, B_1)^\dagger$	B_{fk}, B_{pk} are *discrete*; B_2, B_1 are *numeric*; within the sets $\{B_1, B_{fk}\}$ and $\{B_{pk}, B_2\}$ arguments have the same *length* and *orientation*; B_{fk} and B_1 have the same *table*; B_{fk} and B_{pk} must have different *table*s but the same *type*; and *ALLDIFFERENT*(B_{pk})	$B_2[i] = \sum_{j=1}^{length(B_1)} \begin{cases} B_{val}[j] & \text{if } B_{fk}[j] = B_{pk}[i] \\ 0 & \text{otherwise} \end{cases}$
$B_3 =$ *SUMPRODUCT*(B_1, B_2)	Arguments $\{B_3, B_1, B_2\}$ are *numeric*; *length*(B_1) = *length*$(B_2) \geq 2$; and *rows*(B_3) = *columns*$(B_r) = 1$	$B_3[1] = \sum_{j=1}^{length(B_1)} B_1[j] \times B_2[j]$.

- *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 a conjunction of the properties that the arguments must have for the constraint to be well-formed. We denote this bias for a template s using the 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.

  The constraint templates implemented in *TaCLe* are defined in Table 1.

Example 2 Consider, for example, the constraint template for the row-based sum:

- Syntax: $B_2 = SUM_{row}(\mathbf{B}_1)$, for arguments B_2 and $\mathbf{B}_1$.
- Signature: B_2 has to be a single vector ($size = 1$) while $\mathbf{B}_1$ can be a block ($size \geq 1$), 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 $\mathbf{B}_1$ equals the length of B_2. Moreover, the bias specifies that the number of columns to sum over is at least 2. This avoids that $\mathbf{B}_1$ contains only a single column.
- Definition: each value in the vector B_2 is obtained by summing over the corresponding row in $\mathbf{B}_1$.

SUM_{row} and SUM_{col} are part of a family of constraints that we refer to as *aggregate constraints* that range over multiple rows or columns and are also available for aggregate operators *MIN*, *MAX*, *AVERAGE*, *PRODUCT* and *COUNT*. *SUMIF* is part of a family constraints that we refer to as *conditional aggregate constraints* which aggregate over a vector and only include those cells that satisfy a condition on a related cell in that table, e.g., for the spreadsheet in Figure 1a: $T_1[:, 10]$ and the data in T_5 for matching salesperson names. Conditional aggregates are also available for the above aggregate operators, except for product. Note that *SUMPRODUCT* is not considered part of the family of aggregate constraints in this paper as it is not defined for other aggregate operators. See Section 4.4.3 for a more detailed discussion of the constraints used in *TaCLe*.

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 $B_2 = RANK(B_1)$ can be seen as a predicate $RANK(B_2, B_1)$ where *RANK* is the name of the predicate and its arguments B_2 and B_1 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 $T_1[:, 8] = RANK(T_1[:, 7])$, then the value of B_2 is the 8th vector in T_1: $B_r = T_1[:, 8] = [2, 1, 3, 4]$ and the value of B_1 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 of a constraint template s with n arguments is satisfied by the blocks $(B_1, ..., B_n)$ if $Sig_s(B_1, ..., B_n)$. Likewise a definition is satisfied if $Def_s(B_1, ..., B_n)$ is. The template is hence satisfied if both the signature and definition are satisfied; in logic programming, we would define the predicate s using a Prolog like clause: $s(B_1, ..., B_n) \leftarrow Sig_s(B_1, ..., B_n) \wedge Def_s(B_1, ..., B_n)$. Under this interpretation, the term constraint and constraint template can be used interchangeably.

Definition 3 A **valid argument assignment** of a constraint 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.

3.3 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 represent 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 4 we will discuss how these can be extracted from a spreadsheet. Figure 1b shows the solution to the tabular constraint learning problem when applied to the blocks of Figure 1a and constraint templates listed in Table 1.

3.4 Other considerations

3.4.1 Dependencies

In Table 1 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 *PERMUTATION*(B) holds for a block B, then *ALLDIFFERENT*(B) also holds.

While dependencies are optional, they simplify the specification of signatures and definitions. Moreover, in Section 4 we will see how such dependencies can be used to speed up the search for constraints.

3.4.2 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 *PERMUTATION*(B), then *ALLDIFFERENT*(B) 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, $B_3 = B_1 \times B_2 \equiv B_3 = B_2 \times B_1$, 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: $B_3 = B_1 \times B_2 \equiv B_1 = B_3/B_2$.

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 4 we first explain our generic method for finding all constraints, before describing some optimizations that avoid obvious equivalences (Section 4.4). In the experiments we will investigate the impact of redundancy further.

4 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 (Algorithm 1):

Algorithm 1 The *TaCLe* approach

Step 1 Extract headerless tables from tabular data
Step 2 Partition the tables into contiguous, type-consistent blocks (*input blocks*) using the *BlockDetect* tool
Step 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 an input block and the B_i are *compatible* (defined below) with the signature of s.
(b) For each generated assignment $(B_1, \ldots, B_n)$, find all constraints $s(B'_1, \ldots, B'_n)$ such that $B'_i \sqsubseteq B_i$ holds for all 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 input blocks and testing each combination of blocks for each of the arguments of the constraints. However, each input block 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 input blocks. Consider table T_1 in Figure 1a and the SUM_{row} constraint template. Instead of considering all possible subblocks of T_1 (b_1, b_2, b_3, b_4, b_5), we first reason over the properties of the input blocks. For example, blocks b_2 and b_4 are not numeric so they can be immediately discarded. Input blocks b_1, b_3 and b_5 are valid candidates for the second argument of the constraint, i.e., the functionally defined block. However, since they all have length 4, the first argument, which is the block to sum over, needs at least 4 columns which is only satisfied by b_3. Hence, the assignments generated for SUM_{row} 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 input blocks B_i and generate and test all possible (sub)block assignments to the 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 input

blocks, this is typically an easier problem to solve. For SUM_{row} in the above example and assignment (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, 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 4.1), how input blocks are generated from them (step 2, Section 4.2), and in Section 4.3 how the candidate input block assignments are generated (Step 3a) and how the actual assignments are extracted from that (Step 3b). In Section 4.4 we describe a number of optimizations designed to improve the effectiveness algorithm.

4.1 Step 1: 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, our algorithm will 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 tools that can be used for the specification of the tables:

Automatic detection (AutoExtract) 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. Our implementation of this approach is called the *AutoExtract* tool.

Visual selection (VisualExtract) The above assumptions do not hold for many tables. However, since the specification of tables is usually easy for humans, we employ a second approach which allows users to indicate tables using a visual tool. Screenshots are available in the accompanying GitHub repository with meta-data[1] and in Appendix B. Users select tables excluding the header and optionally specify an orientation. The *VisualExtract* tool then generates the specification of the tables automatically.

4.2 Step 2: Block detection with *BlockDetect*

The goal of the block detection step is to partition tables into maximally type-consistent (all-numeric or all-textual) blocks. First, the *BlockDetect* tool preprocesses 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 and 85% as 0.85). Then, each table is partitioned into maximal type-consistent blocks.

[1] `https://github.com/SergeyParamonov/TaCLe`

Algorithm 2 Learn tabular constraints

```
1: Input: S – constraint templates, B – maximal blocks
2: Output: C – learned constraints
3: procedure LEARNCONSTRAINTS(B, S)
4:     C ← ∅
5:     for all s in S do
6:         A ← InputBlockAssignments(s, B)
7:         for all (B_1, ..., B_n) ∈ A do
8:             A' ← Subassignments(s, (B_1, ..., B_n))
9:             for all (B'_1, ..., B'_n) ∈ A' do
10:                C ← C ∪ {c_s(B'_1, ..., B'_n)}
11:    return C
```

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 blocks.

4.3 Step 3: Constraint learning algorithm

Our learning method assumes that constraint templates and input blocks are given and solves the Tabular Constraint Learning problem by checking for each template: what combination of input blocks can satisfy the signature (*input block assignment*) and which specific *subblock assignments* satisfy both the signature and the definition. The pseudo-code of this approach is shown in Algorithm 2.

This separation of checking the *properties* of input block 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 input block assignments that are compatible with the signature.

Step 3a: Generating input block assignments

Given a constraint template s and the set of all input blocks $\mathcal{B}$, the goal is to find all combinations of input blocks 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 input block 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 to one of the input 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 input block, namely its type, table, orientation, size, length and number of rows and columns.

The actual CSP constraints must enforce that an input block assignment is *compatible* with the requirements defined in the signature. Table 2 shows how the conversion from

Table 2: Translation of signature requirements to input block constraints. The following need not be relaxed: *length*$(B) = x$, *orientation*$(B) = x$, *table*$(B) = x$ and *size*$(B) \geq x$.

Requirement	**Input block constraint**
type$(B) = x$	$basetype($*type*$(B)) = basetype(x)$
size$(B) = x$	*size*$(B) \geq x$
columns$(B) = x$	*if orientation*$(B) = column$: *columns*$(B) \geq x$
	if orientation$(B) = row$: *columns*$(B) = x$
rows$(B) = x$	*if orientation*$(B) = column$: *rows*$(B) = x$
	if orientation$(B) = row$: *rows*$(B) \geq x$

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, namely 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.

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

Example 3 Consider the constraint template $B_2 = SUM_{row}(\mathbf{B}_1)$, then the generated CSP will contain two variables V_2, V_1 corresponding to arguments B_2 and $\mathbf{B}_1$. Given maximal blocks $\mathcal{B}$, the domain of these variables are $D(V_1) = D(V_2) = \{1, \ldots, |\mathcal{B}|\}$. Finally, the signature of $B_2 = SUM_{row}(\mathbf{B}_1)$ (see Table 1) will be translated into constraints:

$$numeric(V_2) \wedge numeric(V_1) \wedge columns(V_1) \geq 2$$
$$(orientation(V_1) = column) \Rightarrow (rows(V_1) \geq length(V_2))$$
$$(orientation(V_1) = row) \Rightarrow (rows(V_1) = length(V_2))$$

Step 3b: Generating subblock assignments

Given an input block assignment $(B_1, \ldots, B_n)$ from the previous step, the goal is to discover valid *subassignments*, i.e., assignments of subblocks $(B'_1, \ldots, B'_n)$ with 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 template $B_2 = SUM_{row}(\mathbf{B}_1)$ again; an example input block assignment from Figure 1a is $(B_2, \mathbf{B}_1) = (T_1[:, 3{:}8], T_1[:, 3{:}8])$. Note that this assignment does not satisfy the signature yet, because B_2 contains more than one vector. This step aims to generate subassignments $(B'_2 \sqsubseteq B_2, \mathbf{B}_1{}' \sqsubseteq \mathbf{B}_1)$ that do satisfy the signature and test whether they satisfy the definition: $\forall i, B_2[i] = \sum_{j=1}^{length(\mathbf{B}_1)} row(i, \mathbf{B}_1)[j]$. 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 such blocks may not even satisfy the signature or just the first element in the data may already not satisfy the definition.

Algorithm 3 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}

On the other hand, a simple generate-and-test approach, as illustrated in Algorithm 3, will usually suffice. Given an input block assignment, all disjoint subassignments are generated taking into account the required size, columns or rows. 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.

Based on the assumptions that most subassignments will not satisfy the definition of the constraint template, the implementation of the definition check is geared to *fail-fast* when possible. Specifically, for many constraints the entries of every subblock are fetched one by one, and as soon as one does not hold failure is returned. Often, the first entry will already not satisfy the definition (e.g. if $B_3[0] \neq B_1[0] \times B_2[0]$ then $B_3 = B_1 \times B_2$ does not hold). As a result, many subassignments can be discarded by looking at just one or a few entries, and considering all subassignments that will be checked, the runtime of the checks will typically not be influenced much by the length of the vectors.

4.4 Optimizations

This section discusses various design decisions and optimizations aimed at improving the extensibility and effectiveness of our method, as well as reducing the redundancy in the output.

4.4.1 Template dependencies

As discussed in Section 3.4.1, some constraint templates depend on others by including templates they depend on in their signature (see Table 1).

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. Dependencies are provided to the system as part of the specification of the constraint templates. 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 1. 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 2 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

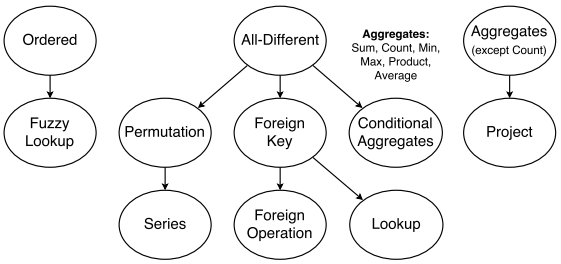


Fig. 2: Dependency graph of Table 1; an arrow from s_1 to s_2 indicates that s_2 depends on s_1: its signature includes s_1.

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 *InputBlockAssignments* on line 6.

The previously learned constraints are used to speed up the *InputBlockAssignments* step. For constraint templates where some arguments are also part of a constraint that it depends on, it is not needed to search for input blocks and subblocks from scratch. Instead, one can start from all and only the actual argument assignments of the base constraint, and only search matching input blocks for the remaining arguments. Therefore, the number of candidate assignments for a constraint template can be reduced by adding a dependence on a lower arity constraint.

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 *InputBlockAssignments* only needs to check which input blocks for B_{fk} are compatible (different table, same type) with these 18 assignments to B_{pk}.

In this case, instead of generating one CSP to find all assignments, a CSP is generated for every known assignment of the depending constraint, which then searches for all assignments completing this partial assignment. The procedure to generate CSPs remains the same otherwise.

4.4.2 Redundancy

We consider two types of redundancy that we aim to eliminate during search. Specifically, these redundancies are taken into account in the implementation for finding subassignments for a constraint. As noted in Section 3.4.1, for some constraint templates there are *symmetries* over the arguments that lead to trivially equivalent solutions, for example, $B'_3 = B'_1 \times B'_2 \Leftrightarrow B'_3 = B'_2 \times B'_1$. 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 ($B'_3 = B'_1 \times B'_2$) and division ($B'_1 = B'_3 / B'_2$) we only added product, as well as only difference ($B_3 = B_1 - B_2$) and not sum ($B_1 = B_3 + B_2$). However, division and sum could easily be added, and they could equivalently be added in post-processing based on the matching product and difference constraints.

A last type of redundancy we chose to avoid during search is that there may be multiple *overlapping* subblocks that satisfy the definition of a constraint template. For example,

consider the constraint $B_2 = \mathit{SUM}_{col}(T[:, 1{:}n])$ where $T[:, k{:}n]$ consists of only zeros, then $B_2 = \mathit{SUM}_{col}(T[:, 1{:}j])$ will be true for all $k - 1 \leq j \leq n$. In *TaCLe* we have, therefore, chosen to only consider *maximal* subblocks. This can be seen as a bias of our system.

In some cases a maximal subblock might falsely include irrelevant columns. Consider input block $\mathbf{B}_1 = T[:, 1{:}3] = [[200, 300], [200, 150], [1, 2]]$ and $B_2 = [200, 300]$, then *TaCLe* will find the constraint $B_2 = \mathit{MAX}(T[:, 1{:}3])$. Should the target constraint be $B_2 = \mathit{MAX}(T[:, 1{:}2])$, then *TaCLe* is only able to find exactly that constraint if $T[:, 3]$ is split off into a separate block in the block detection phase or by post-processing the detected constraints.

4.4.3 Constraints

The 33 constraints that are currently supported in our system are shown in Table 1. We included most formulas that we encountered in tutorial spreadsheets including the popular *SUM* and *LOOKUP* constraints.[2] We also added four structural constraints (*ALLDIFFERENT*, *PERMUTATION*, *ASCENDING* and *FOREIGNKEY*) so that they can be used in the signature of other constraint templates; they are also popular in constraint satisfaction [1]. In our current system, the signature for these structural constraints was chosen to be *strict*, i.e. only support those types that are required by depending constraints. Therefore, *ALLDIFFERENT* does not support floating point numbers and *ASCENDING* does not support strings.

For *LOOKUP*, aggregate constraints and *PROJECT* we use optimized implementations to find subassignments instead of generic generate-and-test.

Given an input block assignment, the implementation for $B_2 = \mathit{LOOKUP}(B_{fk}, B_{pk}, B_1)$ first generates candidate vectors for B_{pk} and B_1, then populates a hash table that allows it to quickly look up keys, afterwards it generates candidate vectors for the remaining arguments and, finally, checks which subassignments are valid.

For aggregates (e.g. *SUM*) and *PROJECT*, a custom implementation is used that tries to find maximal subblocks that satisfy the constraint templates. Instead of generating all possible subblocks, given an inputblock assignment, these implementations start from an input block and repeatedly generate and test smaller subblocks. Therefore, once a match is found, all smaller subblocks do not need to be considered anymore.

Finally, for *FOREIGNKEY* and conditional aggregates, generate-and-test is used, but in combination with caches that store intermediate results and a preparation step to populate some of the caches. For example, for conditional aggregates, input block assignments are analyzed to find vectors with overlapping values. These results are then cached and used to quickly reject subassignments where there is no overlap between B_{pk} and B_{fk}.

4.4.4 Limited precision

One of the challenges of discovering mathematical functional constraints, such as *product*, is that values in spreadsheets often have limited precision and might have been rounded. Testing whether such a constraint holds for a given subassignment, e.g., $B_3 = B_1 \times B_2$, can be done by calculating the *expected* result ($B_1 \times B_2$) first and testing if it corresponds to the actual result (B_3). For the values $B_1 = [1.7], B_2 = [1.8], B_3 = [3.1]$, we would compute first $B_1 \times B_2 = [3.06]$ and conclude $[3.06] \neq [3.1]$ unless the expected result is

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

first rounded. Therefore, *TaCLe* analyzes how long the fractional part of an actual result is and rounds the expected result accordingly.

5 Evaluation

In this section we experimentally validate our approach. We first explain the experimental setup and illustrate it on the example in Figure 1a, we then investigate the effectiveness of our method on synthetic spreadsheets, after which we evaluate our method on spreadsheets from various sources.

5.1 Experimental setup

All spreadsheets used in the experiments are in CSV format and for each spreadsheet the tables are specified. The tables were obtained by running the table extraction tool, *AutoExtract*. Manual intervention of the simple automatic detection was needed only in the case of *None* values (e.g. to merge split-up tables caused by the *None*'s) and in cases where the headers are ambiguous (e.g. headers containing textual and numeric cells). In these cases the tables specifications were generated using the *VisualExtract* tool.

Blocks are then detected automatically using the block detection algorithm, *BlockDetect*. Manual intervention was required for tables that contain *None* values (e.g. if the type of empty cells cannot be inferred).

All spreadsheets also have a set of ground-truth of constraints, we call these the **intended** constraints, that are expected to be (re-)discovered. These were determined by us using either the formulas of the original sheet if present, or inferring them based on, for example, the header names in the sheet. Five spreadsheets contained rows or columns that were entirely identical in which case results can be computed in multiple ways and there are multiple valid constraints. For these spreadsheets, we used the original formulas when present and otherwise inferred the intended constraints based on the headers and the location within the sheet. However, some of the intended constraints are currently outside of the scope of *TaCLe*, in particular nested mathematical or nested logical formulas. We denote by **supported** constraints the subset of intended constraints that the system can find in theory.

In the following experiments, we focus on spreadsheets specifically and hence only include *functional* constraints, constraints that can be expressed as formulas. Therefore, we filter out the structural constraints *ALLDIFFERENT*, *FOREIGNKEY*, *PERMUTATION* and *ASCENDING* in the output. All constraints are stored in their canonical form.

We will use recall and precision to measure how well our tool is performing. Recall is the fraction of intended constraints actually discovered by the system: $\frac{\text{intended discovered}}{\text{all intended}}$, while precision is the fraction of constraints found by the system that are indeed intended: $\frac{\text{intended discovered}}{\text{all discovered}}$.

We include a *no-CSP* baseline to compare our approach to. This baseline follows a similar approach as *TaCLe*, however, it does not prune input blocks in step 3a and generates subassignments directly. Therefore, the signature is also only tested on the subassignment level. While the modified approach still benefits from the reuse of earlier solutions, it does not generate partial assignments for dependent constraint templates.

All experiments were run using Python 3.5.1 on a Macbook Pro, Intel Core i7 2.3 GHz with 16GB RAM. The constraint solver that is used by the input block assignment phase (Step 3a) is python-constraint [16].

5.2 Results on the running example

For the example presented in Figure 1a, *TaCLe* takes a few seconds to find the constraints listed in Figure 1b. These include 5 spurious *RANK* constraints, e.g. $T_1[:,1] = \textit{RANK}(T_1[:,5])$, that are true by accident. Moreover, there is one *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).

Hence, for this example we achieve a recall of 100%, that is, all intended constraints are discovered, and a precision of $12/18 = 67\%$. All intended constraints are supported. The recall is perfect, but the precision is rather low. This is mostly due to the spurious RANK constraints. We note that the tables are quite short, which increases the chance of a constraint like *RANK* to be true by chance; we expect less spurious constraints on larger datasets.

We now investigate three main questions that influence the quality of the solutions found by a constraint learning method:

- Q_1: the method may fail to find intended constraints if it takes too much time to find them; what are the factors that influence the runtime of the method most?
- Q_2: the method may fail to find intended constraints because it does not support such constraints; how does our method perform on real spreadsheets?
- Q_3: the method may find non-intended constraints; how many and what type of non-intended constraints are found on real spreadsheets?

We first investigate Q_1 on synthetic spreadsheets and investigate Q_1, Q_2 and Q_3 on a collection of real spreadsheets.

5.3 Effectiveness on synthetic data

In this section we investigate the factors that influence the runtime of *TaCLe* the most; we do this by investigating the impact on the running time of 1) the number of vectors; 2) the length of the vectors; and 3) the size of the blocks. This analysis is accomplished by testing our system on synthetic spreadsheets. The timeout for all experiments was set to 200 seconds.

Spreadsheet generation. The synthetic sheets should trigger the checking of many different constraints, both in the first phase (input block assignment) and in the second phase (sub-block assignment), in order to clearly study the effect on the constraint checking. As many constraints require numeric or discrete data, we will use integers in the synthetic sheets as these are both numeric and discrete.

We generate spreadsheets with random data. The result is that there are no intended constraints in the data. This is reasonable, as even in real data most of the constraints will not hold. Additionally, for vectors in random data the *ALLDIFFERENT* constraint is likely to hold, which is a requirement for many other constraint templates.

The synthetic spreadsheets contain two tables, because a number of constraints require that their arguments are picked from different tables. For one table the number of vectors and block size is fixed to two, while for the other table these properties are varied according to the experimental setup. The reported number of vectors and the block size in the experiments refer to those of the second table. The length of the vectors can also be adapted and applies to both tables, since equal length vectors are a required property of many constraints. Unless otherwise mentioned, the 'adaptable' table consists of one block of 8 vectors of length 8, and hence 2 vectors of length 8 in the 'fixed' table.

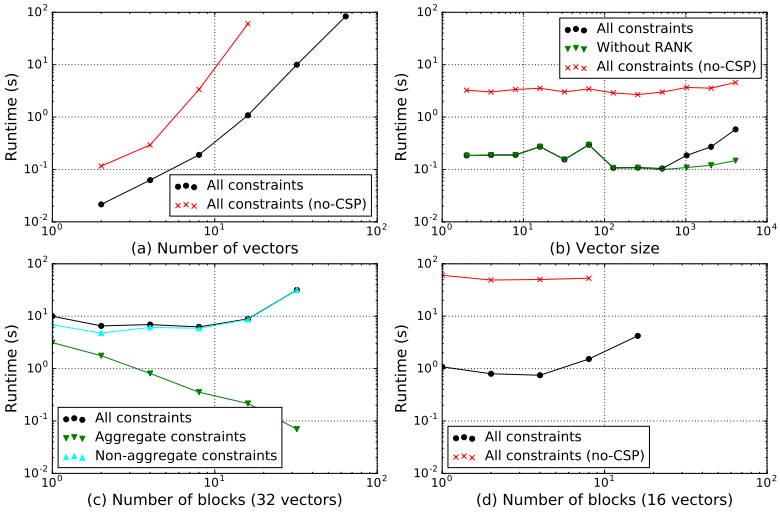


Fig. 3: Loglog plots of the experiments with synthetic (random) data. In plot a the number of vectors is increased; in plot b the size of the vectors is increased; and in plots c and d the number of blocks is increased (the block size is decreased). Missing values for the no-CSP baseline indicate that it timed out.

Effect of number of vectors Figure 3 (a) shows the results of our experiments, in which *TaCLe* was run on synthetic spreadsheets containing an increasing number of equally sized vectors. As shown in the figure, the number of vectors in a spreadsheet can have a significant impact on the running time. Moreover, *TaCLe* performs favorably to the no-CSP baseline whose running time not only starts off slower but also grows faster.

The number of vectors is expected to impact the running times, especially when few input block assignments can be discarded in the first phase. This occurs, for example, when most vectors belong to the same block or all vectors have the same type and size as we artificially enforced in this experiment. In this case, for a constraint template that requires n single vector arguments and a spreadsheet with v vectors, up to $\frac{v!}{(v-n)!}$ subassignments are generated and checked by the algorithm.

Typically, we expect the number of generated subassignments to be much smaller in real world spreadsheets and, therefore, the running time to be less sensitive to the number of vectors. This is because in the first phase entire blocks can be disqualified that do not match the signature requirements, e.g. blocks of vectors with different lengths.

Effect of length of vectors To measure the effect that the length of the vectors has on the running time, we ran our system on spreadsheets in which the length of the vectors was gradually increased. Figure 3 (b) summarizes our results and shows that the length of the vectors has a small impact on the performance for most templates. The no-CSP baseline performs similarly, however, its runtime is higher by a large margin.

As discussed in Section 4.4.3, our implementation attempts to detect as fast as possible if a subassignment does not satisfy a constraint template. Our experiments satisfy the as-

sumption that the large majority of possible subassignments do not satisfy most constraints. Therefore, the results confirm that our implementation is able to severely limit the impact of the length of vectors.

Most of the increase in running time can be attributed to the *RANK* constraint. The implementation is often not able to quickly detect that subassignments do not satisfy *RANK*, meaning that it will have to analyze many if not all entries in the vector.

Effect of block size To measure the effect of changing the block size, the number of vectors is set to 32 and the vectors are explicitly split into varying number of blocks/tables (up to 32). Figure 3 (c) shows the running times of *TaCLe* for various settings. As the blocks become smaller, the total running time first decreases before increasing again. Looking at the running times for aggregate constraints and non-aggregate constraints separately reveals that they are affected differently by the block size. For aggregate constraints (green line) the running time decreases with the block size, while for non-aggregate constraints (red line) the running time first stays more or less constant but then increases more strongly as the size of the blocks becomes small.

Aggregate constraints, e.g. sum or max, have an argument that allows subblocks of varying sizes while non-aggregates only allow single vector arguments. Hence, for aggregate constraints our method will try to find any subblock in the given input block that satisfies the constraint. Since a input block of size m has $m(m+1)/2$ contiguous subblocks, the search for aggregate constraints becomes faster if there are more but smaller blocks[3].

Non-aggregate constraints have only single vector arguments. By splitting a set of vectors into many blocks, however, there will be a larger number of input blocks given as input to the first phase. In the synthetic spreadsheets, the first phase, which looks at properties of input blocks alone, can disqualify few or no input block assignments, leading to some increase in runtime as more blocks are added.

In order to compare to the baseline, we had to lower the number of vectors from 32 to 16. Figure 3 (d) shows that here, too, our system performs better. Both approaches profit from lower running times for aggregates and the no-CSP baseline can offset some of the slowdown as the block size grows small.

These experiments with synthetic data have given us a better understanding of what effect the properties of vectors and blocks can have on runtime and hence on the ability of the algorithm to finish within reasonable time. We next investigate the behavior of our method on real spreadsheets.

5.4 Effectiveness on real spreadsheets

In order to test our approach on real data we assembled a benchmark of spreadsheets from three sources: 1) spreadsheets from an exercise session for teaching Excel at the Brussels Business School based on the book [21], covering the most popular Excel formulas; 2) spreadsheets from online tutorials on Excel formulas; and 3) publicly available *data* spreadsheets such as real-world crime statistics (FBI:UCR) or financial reports (US BEA). The data and its extended description are available in Appendix A in Table 4 and at the same repository as before with the links to all original publicly available spreadsheets used in the experiments.

[3]Consider, for example, v vectors distributed over b blocks of size v/b, the number of subblock candidates then is $O(b * v^2/b^2) = O(v^2/b)$

Table 3: 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 (all)	0.88	0.85	0.88	0.87	1.00	1.00
Recall (supported)	1.00	1.00	1.00	1.00	1.00	1.00
Precision	0.94	0.96	0.69	0.90	1.00	1.00
Runtime *TaCLe*	0.91	0.10	0.87	0.04	0.79	0.20
Runtime no-CSP baseline	58.36	6.48	36.84	1.75	19.73	4.93

Table 3 gives an overview of the spreadsheets in the different categories and summarizes the results of our experiments. The data is also visualized in Figure 4.

Q_1: Effectiveness Our first question, Q_1, is about the inability to find intended constraints due to excessive runtime. As Table 3 (last row) shows, the runtime of our method on these spreadsheets is always fast. The average runtime across all spreadsheets is $0.08s$. Across categories the average running times fluctuate in the range between $0.04s$ to $0.20s$. While most spreadsheets can be processed very quickly, running times can go up to around $0.61s$ for others. In the previous section we analyzed the effect of various factors that can help explain runtime behavior on data generated to allow many candidate subassignments (e.g. using only vectors of the same type and length). Our benchmark experiments show that our algorithm is able to perform well on real spreadsheets, even for larger numbers of vectors. Moreover, these experiments validate our two-step approach, since *TaCLe* is able to find constraints 20 to 60 times faster than the no-CSP baseline.

Looking at the runtime per constraint template individually, the slowest by far is fuzzy lookup using $\pm 21\%$ of the running time. It is followed by product ($\pm 7\%$), relative difference ($\pm 6\%$), running total ($\pm 5\%$), difference ($\pm 5\%$) and various row-aggregates ($\pm 5\%$).

Dependencies are optional and do not affect precision or recall. However, our method can exploit dependencies in order to find constraints incrementally and increase the systems efficiency. To illustrate the effect this can have, we ran the benchmarks without using *ASCENDING* as base constraint for fuzzy lookup. The total running times per category increase by 74% (exercises), 84% (tutorials) and 13% (data). This shows that the use of dependencies can have a strong effect on running times by reducing the number of candidate assignments for depending constraints and sharing the computation and pruning done for one constraint with another.

Q_2: Intended and supported constraints The overall recall achieved by our system, *TaCLe*, is 0.88 $(81/92)$. When looking at the recall per category, *TaCLe* is able to consistently obtain high values for both overall and average recall, as shown in Table 3. The intended constraints not discovered correspond to constraints that are not supported by the system, hence the supported recall is 1. Therefore, *TaCLe* is able to find all constraints for supported constraint templates, which shows that our method is able to learn all constraints using the bias chosen, that is, the formalism of signature and definition.

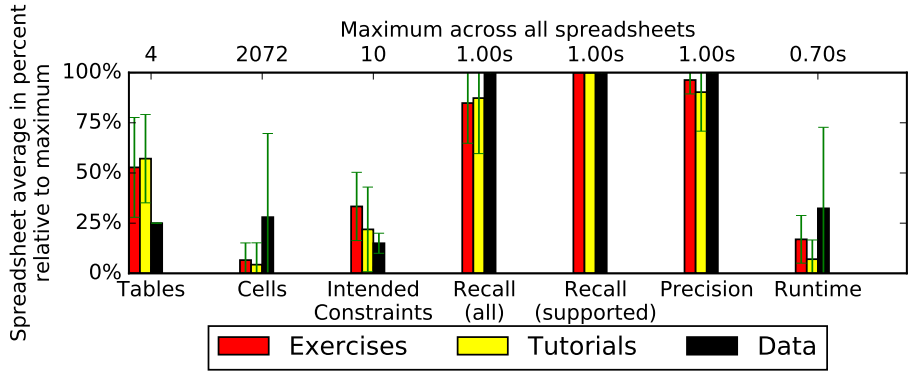


Fig. 4: This figure visualizes the data from Table 3, i.e., the average properties and performance of spreadsheets within every benchmark category. Values are scaled relatively to the *maximum value across all individual spreadsheets*. Green bars indicate standard deviation within the category.

The intended constraints that are not discovered correspond with nested constraints (e.g., $R = \frac{W}{(L/100)^2}$), variants of supported constraints (e.g., series that do not start at 1) or more complex versions of supported constraints (e.g. conditional aggregates whose conditions are constant values).

Q_3: Precision The final question, Q_3, is about the amount and type of **non**-intended constraints found. Across all spreadsheets *TaCLe* achieves a precision of 0.79 $(81/103)$. While the average precision per spreadsheet is high, over 0.90 in every category, we observe that the overall precision is much higher for the exercises and data categories (0.94 and 1.00) than for the tutorials category (0.69).

Examining the tutorial spreadsheets confirms that there are a few spreadsheets that have a high number of additional constraints, e.g., one spreadsheet computes aggregates on inventory data, but copies the data column for every aggregate. Since every aggregate can be calculated based on any of the equal columns, our method will find all of these constraints. However, for every aggregate only one constraint is considered to be *intended*, the others will be considered as redundant and decrease the precision.

Such redundant or unintended constraints are the result of multiple ways to calculate the same result, e.g., if there is duplicate data as in the example above. One way to increase precision for this case is to post-process the constraints to detect equivalences and implications, and heuristically select one among multiple ways to compute the same vector.

6 Applications

In this section, we illustrate how two of the motivating applications, auto-completion and error checking, can build on our method for solving the tabular constraint learning problem.

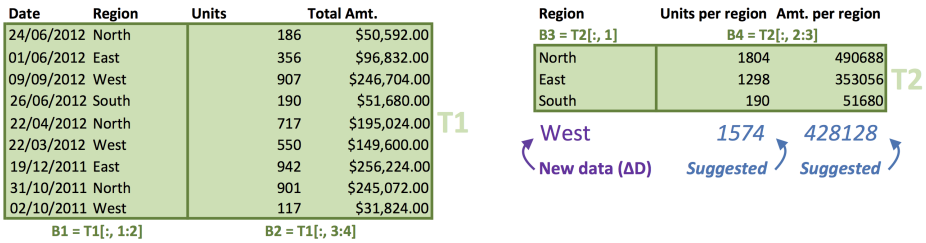


Fig. 5: Auto-completion works by learning constraint on a snapshot (green tables T_1 and T_2) and combining those constraints with new data ΔD (purple). For example, in this figure a user types a new value *West*. The system can then suggest values (blue) for the other cells in that row.

6.1 Auto-completion

Auto-completion can be seen as using knowledge derived from the current data to predict new values before they are written by the user. We can decompose this process into three phases: 1) learn constraints on a snapshot of the spreadsheet; 2) new data (ΔD) is added by the user; and 3) the system uses the learned constraints to predict missing values.

Step 1 corresponds to running *TaCLe* on the snapshot to discover constraints in the spreadsheet. In step 2, the user extends one or more vectors in the spreadsheet with new values. These extended vectors now stretch beyond the original table they are situated in. Therefore, the remaining vectors and blocks in the same table are marked as *incomplete*, indicating they are missing some values required to complete the table. Finally, in step 3, *functional* constraints discovered in step 1, are used to predict missing values. A functional constraint, such as $B_r = SUM_{col}(\mathbf{B}_x)$, calculates a result ($B_r$) based on its *input* ($\mathbf{B}_x$). Hence, a functional constraint can predict missing values if the result block is incomplete and none of the input blocks is incomplete.

Let us illustrate this application on the two-table spreadsheet shown in Figure 5. First, we run *TaCLe* on tables T_1 and T_2 (in green) and discover the constraints $T_2[:,2] = SUMIF(T_1[:,2], T_2[:,1], T_1[:,3])$ and $T_2[:,3] = SUMIF(T_1[:,2], T_2[:,1], T_1[:,4])$. Afterwards, vector $T_2[:,1]$ is extended with a new value *West* and vectors $T_2[:,2]$ and $T_2[:,3]$ are marked as incomplete. Finally, the *SUMIF* constraints are used to complete their incomplete result blocks, suggesting $T_2[4,2] = 1547$ and $T_2[4,3] = 428128$.

In an interactive setting, auto-completion can be performed whenever new values are entered, however, to update the learned constraints, *TaCLe* would need to consider only assignments containing modified vectors.

6.2 Error detection

We discuss two cases of error detection. The first setting is the *online* detection of errors and is similar to the auto-completion setting. Constraints are learned on a snapshot before new data is added. Functional constraints whose result block was modified are checked on the new data. If such a constraint does not hold anymore, it could indicate errors such as typing errors in the data just entered. For example, filling in $T_2[4,2] = 1572$ in Figure 5 would violate the corresponding *SUMIF* constraint and likely indicate an error.

The second setting is *offline* error detection, that is, on a given file. First, *TaCLe* is run to detect all constraints (S) in the given spreadsheet. Then, from this spreadsheet, one can repeatedly remove one or multiple rows or columns (v) and detect all constraints (S_v) that hold in the modified spreadsheet. Every constraint in S_v that did not occur in S is violated by a value in the rows or columns v that were removed. Therefore, values in the rows or columns v that occur in such a violated constraint can be marked as potential errors to be reviewed by a user. This approach allows constraint discovery to be treated as a black-box, however, it requires a scheme for choosing which row(s) or column(s) are removed.

Alternatively, we propose to extend current constraint definitions to tolerate up to a given number of wrong values and report these. This approach would be more robust, however, it requires definition tests to be able to deal with wrong values.

7 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 a limited number of redundant constraints and is sufficiently efficient for normal and interactive use. Our approach also allows new constraint templates to be easily added by specifying their signature in terms of properties and providing an implementation that finds subassignments satisfying the definition.

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. $B_4 = (B_1 + B_2)/B_3$), 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 consistent (noise-free) data, typically generated by some external system. We do support the detection of limited precision calculations as this is a type of noise generated by exporting data. Also, our approach can be used for error correction in case some noise/errors are assumed. However, in inductive logic programming there has been much work on how to handle constraints over noisy data such as allowing a tolerance level ϵ when a rule or a constraint $\sum_{i=1}^{n} b_i = c$ matches up to ϵ: $|\sum_{i=1}^{n} b_i - c| \leq \epsilon$, while in constraint satisfaction a popular alternative is to consider *soft* constraints that can be violated at a cost, which is a real value, and then the task is to find the top-k best matching constraints. Threshold bounded error tolerance can be supported by adapting constraint definitions to allow such noise. This could extend the approach to new application domains, beyond traditional spreadsheets.

Acknowledgements

This work has been partially funded by the ERC AdG SYNTH (Synthesising inductive data models) and a PhD and Postdoctoral Fellowship of the Research Foundation-Flanders.

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. Peter A. Flach and Iztok Savnik. Database dependency discovery: A machine learning approach. *AI Commun.*, 12(3):139–160, August 1999.
7. 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.
8. 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.
9. 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.
10. Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42:100–111, 1999.
11. Tomás Isakowitz, Shimon Schocken, and Henry C. Lucas, Jr. Toward a logical/physical theory of spreadsheet modeling. *ACM Trans. Inf. Syst.*, 13(1):1–37, Jan 1995.
12. Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. *ICTAI*, 1(section II):45–52, 2010.
13. 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.
14. Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *DKE*, 12(1):83–99, 1994.
15. 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.
16. Gustavo Niemeyer. `https://labix.org/python-constraint`.
17. Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.
18. Carmen M. Reinhart and Kenneth S. Rogoff. Growth in a time of debt. Working Paper 15639, National Bureau of Economic Research, January 2010.
19. Iztok Savnik and Peter A. Flach. Discovery of multivalued dependencies from relations. *Intell. Data Anal.*, 4(3-4):195–211, 2000.
20. Ljupčo Todorovski. Equation discovery. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 327–330. Springer US, 2010.
21. Eddy Van den Broeck and Erik Cuypers. *MS Excel 2010*. Uitgeverij De Boeck, 2011.

Table 4: Constraint occurrence in the collected dataset in absolute values by category

Constraint	Exercises	Data	Tutorials
average (col)	0	0	2
average (row)	0	1	0
average-if	1	0	0
count (col)	0	0	1
count-if	1	0	0
difference	2	1	0
equal	0	0	10
foreign-product	1	0	0
fuzzy-lookup	2	0	1
lookup	1	0	0
max (col)	0	0	2
max-if	1	0	0
min (col)	0	0	3
min-if	1	0	0
percentual-diff	5	1	0
product	2	0	3
project	1	0	0
rank	1	0	0
series	2	1	0
sum (col)	5	0	10
sum (row)	2	2	3
sum-if	2	0	9
sum-product	0	0	2

A Spreadsheet Dataset Overview

We collected spreadsheets from three main sources.

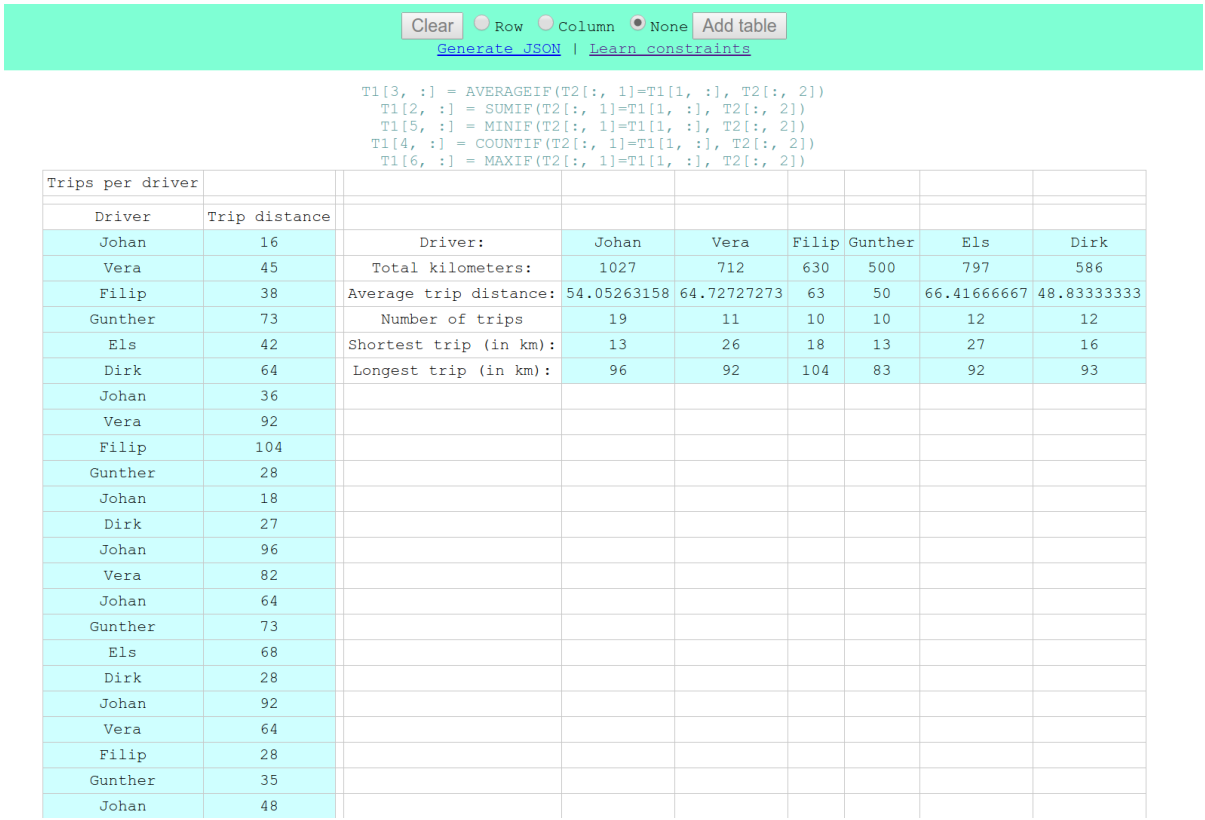
- After identifying popular Excel functions, the MS Office web page has an overview of popular functions (see the link in Subsection 4.4.3), we searched for online tutorials about these functions and collected them under the category **Tutorials**. A link to each webpage found and used is provided in the accompanying GitHub repository.
- We have collected the exercises under the category **Exercises** from the introductory Excel book [21] that focused 1) on popular Excel functions and 2) on datatypes supported by our system.
- We have collected under the category **Data** spreadsheets reporting data. More specifically, economic data, crime reporting data and data from runtime experiments. The spreadsheets on economic data and crime reporting are publicly available and originate from the U.S. Bureau of Economic Analysis (BEA), the RWE annual report 2014 and the U.S. FBI Uniform Crime Reporting (UCR) Program. The links are also provided in the accompanying GitHub repository) and data summary spreadsheets.

The overview of constraint distributions per category is presented in Table 4.

The file *links.txt*, available in the accompanying GitHub repository, contains the links to all original publicly available spreadsheets used in the experiments.

B Examples of Learned Constraints

This section provides examples of actual constraints learned by *TaCLe* on real spreadsheets from our benchmark dataset. Figure 6 shows constraints learned on a dataset containing conditional aggregates. In Figure 7 a dataset is used that contains arithmetic constraints. More examples and screenshots can be found in the accompanying GitHub repository:
`https://github.com/SergeyParamonov/TaCLe`.



Trips per driver								
Driver	Trip distance							
Johan	16	Driver:	Johan	Vera	Filip	Gunther	Els	Dirk
Vera	45	Total kilometers:	1027	712	630	500	797	586
Filip	38	Average trip distance:	54.05263158	64.72727273	63	50	66.41666667	48.83333333
Gunther	73	Number of trips	19	11	10	10	12	12
Els	42	Shortest trip (in km):	13	26	18	13	27	16
Dirk	64	Longest trip (in km):	96	92	104	83	92	93
Johan	36							
Vera	92							
Filip	104							
Gunther	28							
Johan	18							
Dirk	27							
Johan	96							
Vera	82							
Johan	64							
Gunther	73							
Els	68							
Dirk	28							
Johan	92							
Vera	64							
Filip	28							
Gunther	35							
Johan	48							

Fig. 6: This screenshots shows the five constraints learned from a spreadsheet containing conditional aggregates. The first table extends below the depicted area. (Translated spreadsheet from exercises [21], originally in Dutch.)

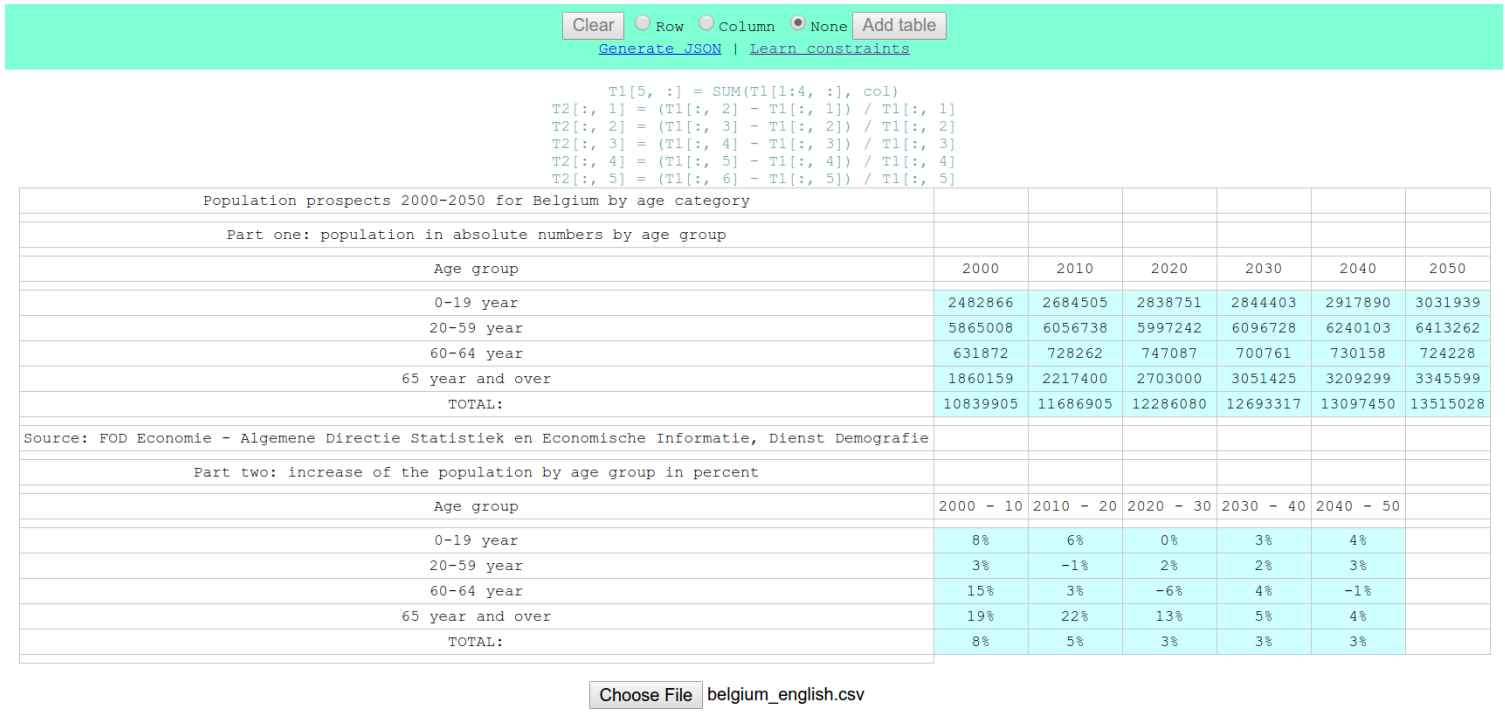


Fig. 7: This screenshot shows the six constraints learned on a spreadsheet containing arithmetic constraints: sum and relative difference. (Translated spreadsheet from exercises [21], originally in Dutch.)