TaCLe: Learning Constraints in Tabular Data

Sergey Paramonov*
Samuel Kolb*
sergey.paramonov@kuleuven.be
samuel.kolb@kuleuven.be
KU Leuven, Belgium

Tias Guns
Vrije Universiteit Brussel, Belgium
KU Leuven, Belgium
tias.guns@vub.be

Luc De Raedt
KU Leuven, Belgium
luc.deraedt@kuleuven.be

ABSTRACT

Spreadsheet data is widely used today by many different people and across industries. However, writing, maintaining and identifying good formulae for spreadsheets can be time consuming and error-prone. To address this issue we have introduced the TaCLe system (Tabular Constraint Learner). The system tackles an inverse learning problem: given a plain comma separated file, it reconstructs the spreadsheet formulae that hold in the tables. Two important considerations are the number of cells and constraints to check, and how to deal with multiple formulae for the same cell. Our system reasons over entire rows and columns and has an intuitive user interface for interacting with the learned constraints and data. It can be seen as an intelligent assistance tool for discovering formulae from data. As a result, the user obtains a spreadsheet that can automatically recompute dependent cells when updating or adding data.

KEYWORDS

Relational Learning; Constraint Learning; Spreadsheets

1 INTRODUCTION

Spreadsheets are one of the most widely used data processing tools today. Working with tables is often intuitive, and using macros and formulae for basic tasks is considered a standard skill among office employees even without a technical background. But when the structure of the data and formulae are getting complex, spreadsheets become harder to maintain, extend and develop.

The problem gets even more complicated when importing and exporting from special-purpose software, e.g. Enterprise Resource Planning (ERP) systems. Often, data from such and other systems is exported into a plain textual format, called Comma Separated Values (CSV). On each data transformation to CSV and back, all created formulae and relations are lost. Even in a spreadsheet-only environment, maintaining consistent formulae across multiple tables and spreadsheet files can quickly become problematic in a large company.

For example, the influential "Growth in a Time of Debt" paper [6] had some of its claims contested [2] because the used Excel sheets contained mistakes in formulae.
We argue that an intelligent assistance tool that automatically learns formulae in tabular data, can overcome such problems.

Our work is, on the one hand, inspired by FlashFill [1], which learns a string-transformation function in Excel from very few examples. Flashfill has been extended and generalized into FlashMeta [5] and FlashExtract [4]. On the other hand, it is inspired by equation discovery research [7], where one aims at finding numeric dependencies that hold in the data. However the the bias they have is incompatible with formulae in Excel and not designed to work in spreadsheet environment, i.e., with ranges of cells.

The problem setting is unconventional for machine learning, since columns and rows no longer represent variables and data-points. Here everything is mixed: the formulae can range both over rows and over columns. We work directly with CSV files, and therefore, there is no input-output information, i.e., the setting is unsupervised. The problem is unconventional for data mining as well, since the data is relational (i.e., there are multiple connected relations) and also contain textual and numeric data.

We investigated an automated constraint learning algorithm named *TaCLe* (from: Tabular Constraint Learner, pronounced "tackle") for discovering row-wise and column-wise constraints [3]. *Constraints* include both functional relations (e.g. formulae) and non-functional relations between rows and columns. The focus on constraints over entire rows and columns rather than individual cells is partly out of necessity but also very natural in spreadsheet setting: their elements are usually related and formulae are often *dragged* across an entire range.

The key technical difficulty for constraint learning is the large number of constraints and the exponential number of row/column combinations to try for each constraint. At a high level, the learning proceeds in a three step approach: first, headerless tables and their rows and columns are extracted from the tabular input; then impossible constraints and table combinations are filtered out after which all valid constraints over rows and columns are computed. In this demo paper we showcase the workings of the system and show how it can be used as a "smart import" tool to automatically replace values by formulae. It requires transforming the individual constraints into a set of constraints that is collectively usable

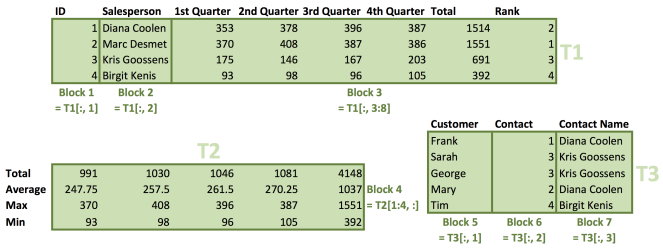
*equal contribution

CIKM'17, Singapore, November 6–10, 2017, Singapore.

ACM ISBN ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3133193

in spreadsheets. This includes choosing among equivalent or redundant constraints and breaking cyclic dependencies among the formulae.

2 FROM TABULAR DATA TO SPREADSHEET FORMULAE



(a) Example spreadsheet. A green background indicates headerless tables, green text indicates the table and block names and short notation.

$SERIES(T_1[:,1])$
$T_1[:,1] = RANK(T_1[:,5])$
$T_1[:,1] = RANK(T_1[:,6])$
$T_1[:,8] = RANK(T_1[:,7])$
$T_1[:,8] = RANK(T_1[:,3])$
$T_1[:,8] = RANK(T_1[:,4])$
$T_1[:,7] = SUM_{row}(T_1[:,3{:}6])$
$T_2[1,:] = SUM_{col}(T_1[:,3{:}7])$
$T_2[2,:] = AVERAGE_{col}(T_1[:,3{:}7])$
$T_2[3,:] = MAX_{col}(T_1[:,3{:}7])$
$T_2[4,:] = MIN_{col}(T_1[:,3{:}7])$
$T_3[:,2] = LOOKUP(T_3[:,3], T_1[:,2], T_1[:,1])$
$T_3[:,3] = LOOKUP(T_3[:,2], T_1[:,1], T_1[:,2])$

(b) Constraints extracted by *TaCLe* for the above tables.

Figure 1: Spreadsheet (top) and learned constraints (bottom).

An important assumption to make learning constraints in tabular data more tractable is to search for constraints over entire rows and columns only. We use the generic term *vector* to refer to either a row or a column and only consider vectors whose cells all contain data of the same type, i.e., numeric or textual. Our method groups adjacent vectors of the same type and orientation into contiguous *blocks*, enabling us to reason about *typed* data more efficiently.

Using this concept of blocks, our approach proceeds in four stages, the first three being *TaCLe*'s constraint learning stages [3] and the last stage is specific to the smart import use case. The stages are 1) table extraction and block detection; 2) filtering out impossible constraint/block combinations; 3) generating all valid constraint assignments and 4) producing a compact set of constraints that together are valid in a spreadsheet setting.

The example tabular data that we use is shown in Figure 1a. This example is based on exercises in the book "MS Excel 2010" [8], and is chosen to be representative for the type of formulae that experienced spreadsheet users should know.

We use notation $B = T1[:,1]$ to indicate that block B is a column-oriented block consisting of column 1 (across all rows) of table $T1$. Likewise, $T2[1:4,:]$ is a row-oriented block consisting of rows 1 to 4 of Table $T2$. Bold-face names ($\mathbf{B}$) indicate blocks that can consist of multiple rows or columns, while italic names (B) indicate blocks consisting of at most one row or column.

The *TaCLe* system consists of a set of constraints including the most popular formulae in Excel. Table 1 shows a selection of them. Note that *TaCLe* can not only learn functions (such as block B_2 represents the rank of block B_1) but also non-functional relations (such as block B is a series).

Table 1: A subset of constraints implemented in *TaCLe* and their intuitive meaning. For constraints with † (called *aggregate* and *conditional aggregate* constraints; the table shows variants for *SUM*) *TaCLe* also supports *MAX*, *MIN*, *AVERAGE*, *PRODUCT* and *COUNT*.

Syntax	Intuitive Meaning
$SERIES(B)$	Values increase by 1.
$B_2 = RANK(B_1)$	Computes a ranking, incl. ties.
$B_3 = B_1 \times B_2$	Vector dot product.
$B_3 = B_1 - B_2$	Elementwise difference.
$B_3 = B_1 - B_2 + PREV(B_3)$	Sliding difference.
$B_2 = SUM_{row}(\mathbf{B}_1)^\dagger$	Sum over rows
$B_2 = SUM_{col}(\mathbf{B}_1)^\dagger$	Sum over columns
$B_2 = SUMIF(B_{fk}, B_{pk}, B_1)^\dagger$	Group-by like sum
$B_2 = LOOKUP(B_{fk}, B_{pk}, B_1)$	Lookup mapping

We now describe each of the phases in turn, using the above example data and constraints.

2.1 Table extraction and block detection

The first step of our approach is, given a CSV file, to group cells into tables and remove any headers. If the tables contain no empty cells inside and are surrounded by empty cells outside then extracting them is trivial. Detecting and removing headers is more involved as tables can also contain textual data. A simple solution which is implemented in our webtool is to let users select or modify the selection of the headerless tables, as this task is often trivial to them.

Block detection. The second step of our approach is to partition tables into smaller blocks. In the ideal case, blocks group vectors into units of related vectors. An estimate of such meaningful units is obtained by joining all adjacent vectors with the same type (numeric or textual) into a block. In almost all cases this partition can be easily computed automatically, the only exceptions are tables which are formatted in a way that they, for example, include empty columns.

Example 2.1. In Figure 1a there are three tables, T_1, T_2 and T_3 whose headers have been excluded. The first table T_1 contains 11 vectors consisting of its columns. Since some columns are numeric, e.g., ID, and some textual, e.g., Salesperson, the rows of T_1 are not type-consistent and thus not considered. These vectors are grouped by their type into 5 blocks, blocks 1, 2 and 4 consist of one vector each, while blocks 3 and 5 contain multiple vectors.

2.2 Constraint learning

Given a CSV file and the definitions of the tables and type-consistent row or column blocks, *TaCLe* learns what constraints hold in the data. To formalize what constraints our system should discover, a set of *constraint templates* is given to the system as part of the implementation. A constraint template contains the syntax, signature and definition of a constraint. For the column-wise sum constraint, the syntax is $B_2 = SUM_{col}(\mathbf{B}_1)$.

Given an assignment of blocks to $\mathbf{B}_1$ and B_2, the signature checks whether the *properties* of the assigment fulfil the requirements for the constraint, while the definition checks whether the *actual values* correspond to the constraint. For example for the sum, the *signature* specifies the following properties: that $\mathbf{B}_1$ should contain at least two columns, that B_2 must consist of a single vector whose length must be the same as the number of columns in $\mathbf{B}_1$ and that both arguments must be numeric. The *definition* computes whether the actual values in $\mathbf{B}_1$ sum up to the values in B_2.

Our algorithm is able to search over all possible constraints efficiently with a two step approach [3].

Filtering out impossible constraint/block combinations based on signature. First, *TaCLe* reasons over entire type-consisted and row or column-oriented blocks as detected in the previous phase. It precomputes the properties of every such block and solves for every constraint template a constraint-satisfaction problem that computing what assignments of blocks to constraint arguments are *compatible* with the requirements imposed by the signature. An assignment of input blocks to arguments is compatible if an assignment of subblocks, i.e., subsets of the input blocks, could fulfill the signature. This allows to eliminate entire blocks that are incompatible with certain constraints, rather then having to check this for each row, column and combinations of rows or columns separately.

Constraint templates are implemented as Python classes and they specify the required types for their arguments as well as a set of objects that specify their signature properties. The implementation creates a Constraint Satisfaction Problem (CSP) for each constraint template. CSP solvers are generic and highly-efficient search algorithms for finding satisfying assignments to the variables. For us, the variables represent the possible blocks and each of the signature properties is automatically converted into CSP constraints that express compatibility of the blocks.

Generating all valid constraint assignments. Second, for every input-block assignment, *TaCLe* generates all subblock assignments and tests whether they fulfill the signature and the definition. That is, it tries each individual row or column in case of single-length arguments (such as B_2 in $B_2 = SUM_{col}(\mathbf{B}_1)$), and each combination in case of wider blocks (such as $\mathbf{B}_1$ in $B_2 = SUM_{col}(\mathbf{B}_1)$). By exploiting dependencies between constraints it also cuts down on the number of candidate assignments.

In practice, we introduce an engine class which provides code for generating and testing assignments, given a set of subblock assignments and previously found constraints. Our own internal Python engine implements the spreadsheet constraints listed in Table 1 and more. Generic CSP solvers could also be used for this task, however, especially for numeric constraints where CSP support is faint, our imperative engine is able to provide much better performance.

2.3 Producing a valid set of formulae

In order to use the set of learned constraints, they need to be translated into formulae that can be used by spreadsheet software. Spreadsheet formulae are functional and describe how to calculate the value of one cell based on other cells. *TaCLe* supports additional constraints that cannot be translated into spreadsheet formulae, i.e., alldifferent, permutation, foreign key and ascending. These additional constraints are ignored in the smart import process and in our webtool.

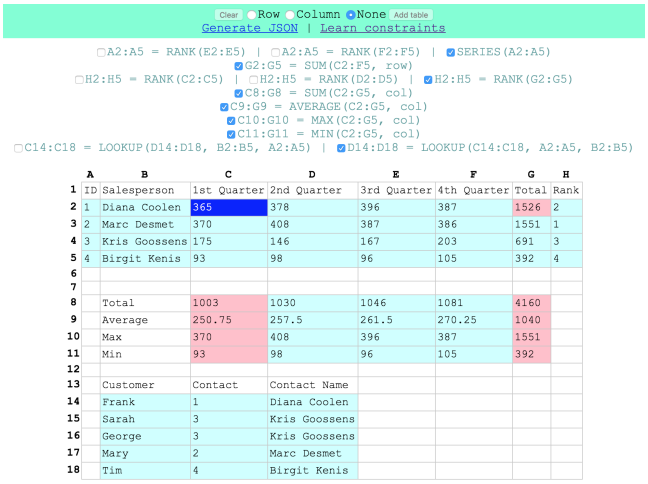
Translation. In order to translate the functional constraints over blocks, a corresponding spreadsheet formula is created for every cell in the output block. Instead of addressing cells with their relative location within a table, such as $T_1[:, 1]$, spreadsheets usually use the name of the sheet-wide row and column, e.g. $A1, C2$, etc. This is a matter of translating the relative location with respect to the table location.

Note that one of the constraints, i.e., series, is unary. To translate the series constraint, the first cell is kept intact and every subsequent cell is a formula that increments the value of the previous cell by one.

Example 2.2. Consider headerless tables T_1 and T_2 in Figure 2 at absolute locations $A2$ and $C8$ and a constraint $T_2[1, :] = SUM_{col}(T_1[:, 3{:}7])$ meaning the first row of T_2 is the result of summing columns 3 to 7 of T_1. We can translate the constraint into 5 spreadsheet formulae for each cell in row $T_2[1, :]$. The Excel formula of the first cell is then $C8 = SUM(C2{:}C5)$, the last cell is $G8 = SUM(G2{:}G5)$. We will denote the formulae for this row compactly as $C8{:}G8 = SUM(C2{:}G5, col)$.

Resolving conflicts. It is possible that *TaCLe* produces multiple constraints that compute the same vectors, for example the SERIES and two RANK constraints for $T_1[:, 1]$. By detecting these and asking the user which constraint to prioritize, these conflicts can be resolved. A default resolution choice is offered by heuristically selecting a constraint to prioritize. Our current heuristic is to select the most likely constraint considering the 'distance' between the output and input vector locations in the spreadsheet. More specifically, to prioritizing vectors within the same table and for alternative vectors within the same table, considering the Manhatten distance to the output vector.

Additionally, in the western world spreadsheets are often built top-to-bottom and left-to-right, resulting in aggregations occurring more often to the right of, or below, the input data. Ideally, a system could store each modification of the user to the default proposed constraint and automatically learn the preferences of its user over time.



	A	B	C	D	E	F	G	H
1	ID	Salesperson	1st Quarter	2nd Quarter	3rd Quarter	4th Quarter	Total	Rank
2	1	Diana Coolen	365	378	396	387	1526	2
3	2	Marc Desmet	370	408	387	386	1551	1
4	3	Kris Goossens	175	146	167	203	691	3
5	4	Birgit Kenis	93	98	96	105	392	4
6								
7								
8		Total	1003	1030	1046	1081	4160	
9		Average	250.75	257.5	261.5	270.25	1040	
10		Max	370	408	396	387	1551	
11		Min	93	98	96	105	392	
12								
13		Customer	Contact	Contact Name				
14		Frank	1	Diana Coolen				
15		Sarah	3	Kris Goossens				
16		George	3	Kris Goossens				
17		Mary	2	Marc Desmet				
18		Tim	4	Birgit Kenis				

Figure 2: Our web-based demo shows candidate formulae when given the tables of Figure 1a as input (top). The selected constraints are enforced and when a cell value is changed (blue), dependent cells are automatically recomputed (pink).

Resolving cycles. Finally, constraints may have cyclic dependencies. For example $T_3[:,2] = LOOKUP(T_3[:,3], T_1[:,2], T_1[:,1])$ and $T_3[:,3] = LOOKUP(T_3[:,2], T_1[:,1], T_1[:,2])$. By constructing a dependency graph where each node represents the columns and rows of the tables, and the directed edges represent which vectors depend on which, we can detect such cycles. To break the cycles we have to remove edges (and hence constraints) in the graph. In case cycles can be broken in multiple equivalent ways, we can use the heuristics from the previous step to propose a default constraint to keep in the user interface.

3 SOFTWARE DEMO

Our web-based tool demonstrates the *smart import* capabilities, where our method learns the formulae that are present in a CSV file of raw data. Our demo is available online at: **http://bear.cs.kuleuven.be/tacle** and example CSV data with a usage guide at: https://github.com/SergeyParamonov/TaCLe. The workflow of the demo consists of three steps:

1. Import CSV file and select headerless tables. The user selects a file on his computer and it is rendered by the tool. The user then selects tables by clicking consecutively on two opposite corner cells. The selected cells should not contain any headers that are not part of the actual data. Optionally, the user can also select an orientation for the table. By clicking 'Add table', the selected area is added as a table.

This looks similar to Figure 1a (black text only) whereby the three green areas are identified as tables.

2. Instruct the system to learn and output the formulae. After clicking 'Learn constraints', the system will execute the four phases described in Section 2 and show the candidate formulae. This is shown at the top of Figure 2. Each line represents one possible constraint. In case of multiple conflicting constraints they are all shown on the same line and the user can change the default choice suggested by the system.

3. The formulae are put in place. The raw cell values are now replaced by the learned formulae, meaning that if the value of one cell is changed all the cells that depend on it will recompute there value. This is shown at the bottom of Figure 2 (blue/pink colors).

4 CONCLUSIONS AND FUTURE WORK

Learning formulae is important because spreadsheets are used by many people across the world. The automation offered by our smart import tool can save them time when working with raw data files as well as help users that are less familiar with available spreadsheet functions.

This demo extends our *TaCLe* system for constraint learning [3]. An inspiration to our work was FlashFill [1], which learns string transformation functions in tables. It is a supervised system that needs input/output transformation examples, while ours is unsupervised. Our system can handle both numeric and textual data and data across different tables.

Currently our method can extract individual functions, which covers a large part of the use of formulae. An interesting direction of future work is to learn arbitrary compositions of functions, though this makes the space of candidate constraints much larger again.

There are various other possible applications for *TaCLe*. *Auto-completion* can help users to fill in missing values in a spreadsheet by suggesting values based on learned formulae. *Error checking* can indicate that a previously discovered constraint is violated by a newly inserted value. These applications can help users that do not feel confident using spreadsheet formulae, by indicating why a particular value should or shouldn't be in a cell. Also, *formula suggestion* can help users that are unsure of a formula or of the syntax.

Acknowledgment. This work has been supported by the FWO and by the ERC-ADG-201 project 694980 SYNTH funded by the European Research Council.

REFERENCES

[1] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *ACM SIGPLAN-SIGACT (POPL)*. 317–330.
[2] Thomas Herndon, Michael Ash, and Robert Pollin. 2013. Does high public debt consistently stifle economic growth? A Critique of Reinhart and Rogoff. *Cambridge Journal of Economics* (2013).
[3] Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. 2017. Learning constraints in spreadsheets and tabular data. *Machine Learning* (Jun 2017).
[4] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *ACM SIGPLAN PLDI*. 55.
[5] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *ACM SIGPLAN*. 107–126.
[6] Carmen M. Reinhart and Kenneth S. Rogoff. 2010. *Growth in a Time of Debt*. Working Paper 15639. NBER.
[7] Ljupčo Todorovski. 2010. Equation Discovery. In *Encyclopedia of Machine Learning*, Claude Sammut and Geoffrey I. Webb (Eds.). Springer US, Boston, MA, 327–330.
[8] Eddy Van den Broeck and Erik Cuypers. 2011. *MS Excel 2010*. Uitgeverij De Boeck.