MUPPETS: Multipurpose Table Segmentation

Gust Verbruggen[1], Lidia Contreras-Ochando[2], Cèsar Ferri[2], José Hernández-Orallo[2], and Luc De Raedt[1]

[1] KU Leuven, Department of Computer Science, Leuven, Belgium
Leuven.AI — KU Leuven Institute for AI, Leuven, Belgium
{gust.verbruggen,luc.deraedt}@kuleuven.be
[2] Valencian Research Institute for Artificial Intelligence (vrAIn)
Universitat Politècnica de València, Spain
{liconoc,cferri,jorallo}@dsic.upv.es

Abstract. We present MUPPETS, a framework for partitioning cells in a table in segments that fulfil the same semantic role or belong to the same semantic data type, similar to how image segmentation is used to group pixels that represent the same semantic object in computer vision. Flexible constraints can be imposed on these segmentations for different use cases. MUPPETS uses a hierarchical merge tree algorithm, which allows for efficiently finding segmentations that satisfy given constraints and only requires similarities between neighbouring cells to be computed. Three applications are used to illustrate and evaluate MUPPETS: identifying tables and headers, type detection and discovering semantic errors.

Keywords: table analysis, segmentation, error detection, type detection

1 Introduction

Tables allow users to store and represent information in a general and familiar way. Data that is often stored in tables includes measurements, data science pipelines, logs, relational and non-relational databases. Such tables are intrinsically more complex than the grid of their values, as they may include parts that play different roles or are associated with different domains. Unfortunately, they rarely come with metadata describing these roles and domains.

An example of a table with cells of different roles and data types is shown in Figure 1. Humans easily recognise the syntactic and semantic structure in this table, and automatically identify the coloured segments that we see on the right. In data processing and analysis, understanding how a table is segmented in different regions is a prerequisite for processing it. Unfortunately, this is still mostly done manually.

The problem of segmenting a table takes inspiration from image segmentation. Images are usually composed of different objects that we would like to delineate automatically. The goal is to assign a label to every pixel, such that those with the same label share the same features and are different from the rest [22, 19]. The imposed contiguity constraint makes image segmentation a specific kind of

This Line is being used as a header					
ID	Date	Amount	Quantity	Status	
0042	16-Oct-17	$23.99	123	Closed	Jansen
7731	15-Jan-17	$49.99		Pending	Rho
8843	9-Mar-17	129	45		Gupta
3013	12-Feb-17		15	Pending	Harrison
4431	1-Jul-17	$99.99	1	Closed	Yang

(a) Segmentation of table in data, header and metadata.

This Line is being used as a header					
ID	Date	Amount	Quantity	Status	
0042	16-Oct-17	$23.99	123	Closed	Jansen
7731	15-Jan-17	$49.99		Pending	Rho
8843	9-Mar-17		45		Gupta
3013	12-Feb-17		15	Pending	Harrison
4431	1-Jul-17	$99.99	1	Closed	Yang

(b) Segmentation in data domains. It is easy for humans to see that the last two columns are semantically different domains.

Fig. 1: Examples of different structures of tables.

clustering. Additional constraints make these segments go from small *superpixels* to other more complex regions of the image. Image segmentation is a very clear and distinctive problem with many applications, such as object detection, scene understanding and compression. Table segmentation is similar to image segmentation, but instead of clustering pixels, we are clustering cells in segments that belong to the same data type or play the same semantic role.

Given that so much data is stored in tables and spreadsheets, and that image segmentation has been studied for decades, it is surprising that the problem of table segmentation has not received much attention outside of specific applications. The key contribution of this paper is that we introduce a generic, multipurpose framework for table segmentation. It can use any distance function between cells and can represent a wide range of constraints on the shape and position of segments. More specifically, we make the following contributions:

1. We introduce the problem of table segmentation subject to a set of constraints.
2. We combine a concrete class of constraints, a heuristic algorithm for finding segmentations that satisfy these constraints and a method for scoring them into a flexible framework called MUPPETS.
3. We show how table segmentation, and MUPPETS in particular, can be applied to a range of problems, such as identifying tables and headers, type detection and discovering semantic errors.

The following three sections respectively describe these contributions.

2 Table segmentation

A table T is an $n \times m$ grid of cells. We define a *segment* $S \subseteq T$ as a set of orthogonally connected cells in this table. A segmentation $\mathbf{S}$ of T is a partitioning into mutually disjoint segments $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ such that $\cup_i S_i = T$. This corresponds to a clustering of the cells of the table in which the elements in each cluster are orthogonally connected.

Assumptions can often be made about structural properties of tables. In spreadsheets, for example, cells of the same type are typically arranged in

rectangular regions [2]. These assumptions are encoded as Boolean constraints on the segmentations.

Multiple segmentations of a table can satisfy any given constraint, but the target segmentation will have some additional, desired properties. For example, in our MUPPETS framework, we will aim for the cells within each segment to be *similar* according to some criterium. Analogous to internal evaluation methods for determining the number and quality of clusters, we thus want to assign a score to each segmentation that represents how well it exhibits these properties. The problem of table segmentation can now be defined as follows.

Given
- a table
- a constraint
- a scoring function

Find a set of segmentations for the table that satisfy the constraint and that are ranked according to *score*.

In the next section, we present a versatile framework for table segmentation by describing a class of constraints, a scoring function and a heuristic algorithm for finding segmentations that satisfy the constraint and have a high score.

3 The MUPPETS framework

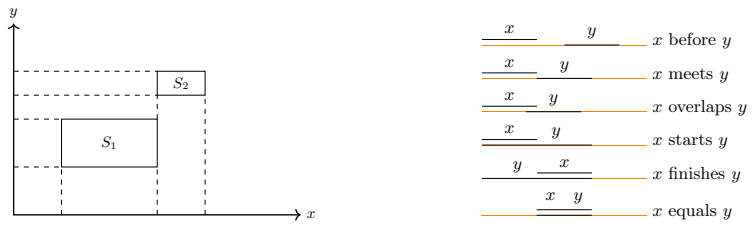
Our MUPPETS framework consists of three key ingredients: (1) a class of supported constraints, (2) a scoring function based on the similarity between cells and (3) a heuristic search algorithm. The following sections describe these ingredients.

3.1 Constraints

A constraint C in MUPPETS consists of two parts. First, there are constraints on the possible shapes each segment in a segmentation is allowed to take. This is formalised as a Boolean function $shape(S)$ which yields true if segment S satisfies the constraint and false otherwise. All segments in the output segmentation must satisfy the constraint. The most common shape constraint is a rectangular one, which is true if the bounding box around a segment is equal to the segment itself.

Second, MUPPETS supports constraints on the spatial configuration of segments in a segmentation. This is formalised as a position constraint that must hold between pairs of segments S_i and S_j with $i \neq j$. To specify these position constraints, we resort to the well-known qualitative spatial relationships [1] that are illustrated in Figure 2a. Each segment S_i is projected on its x- and y-coordinates, which yields two intervals on which the relations listed in Figure 2b can be specified. In this paper, we consider first-order logic formulae over qualified variables, as shown in Example 2. This can be extended to constraints on particular segments, for example, to allow interactively specifying constraints [9].

Example 1. The spatial configuration of two segments S_1 and S_2 is shown in Figure 2a. We can see that on the x axis it holds that S_1 meets S_2 and on the y axis it holds that S_1 before S_2. If no axis is specified, it should hold for both.



(a) Two segments and their projection. (b) Interval algebra relations.

Fig. 2: Illustration of positional constraints using interval algebra.

Example 2. We can define a *tabular* constraint by allowing only rectangular segments and prohibiting any of {overlaps, starts, finishes, during} to hold, or

$$\forall S_1, S_2 : \neg((S_1 \text{ overlaps } S_2) \vee (S_1 \text{ starts } S_2) \vee (S_1 \text{ finishes } S_2) \vee (S_1 \text{ during } S_2))$$

All segments then either align in one dimension or are completely disconnected. Similarly, a less strict *subtabular* constraint can be defined as follows.

$$\forall S_1, S_2 : \neg((S_1 \text{ overlaps } S_2) \vee (S_1 \text{ during } S_2))$$

3.2 Score

The scope of possible values in table cells is virtually endless and deciding whether two values belong to the same segment is heavily dependent on context. We therefore allow any distance function $d(c_1, c_2)$ between two cells c_1 and c_2 to be supplied as an argument. As in clustering, our goal is then to have a segmentation in which similar cells are in the same segment and vice versa. Internal cluster evaluation methods can be used to score segmentations.

Some of these distances are expensive to compute, for example, because they require search queries [6]. We present a scoring function that exploits the spatial configuration of tables and only performs $(n-1)(m-2)$ distance computations between neighbouring cells, as opposed to the $\mathcal{O}(n^2m^2)$ computations required for popular evaluations such as the silhouette index [20].

Let $\langle c_1, c_2 \rangle$ represent the edge between two neighbouring cells c_1 and c_2. A boundary $b(S_1, \ldots, S_n)$ between segments $\{S_1, \ldots, S_n\}$ is the set of edges $\langle c_i, c_j \rangle$ such that $c_i \in S_k$, $c_j \in S_l$ and $k \neq l$. We can assign a score to a boundary b as the average distance between cells on either side of all edges that it contains

$$s(b) = \operatorname*{avg}_{\langle c_i, c_j \rangle \in b} d(c_i, c_j). \tag{1}$$

where avg denotes the mean of the values that it ranges over and $\operatorname{avg} \varnothing = 0$.

We compute the inter-segment score $s_e(\mathbf{S})$ of a segmentation $\mathbf{S}$ as the score of the boundary between all segments. The intra-segment score $s_a(\mathbf{S})$ is computed

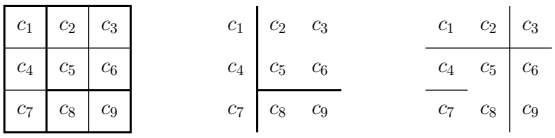


Fig. 3: (left) Table and segmentation, (middle) boundary between segments of the segmentation and (right) edges between neighbouring cells within each segment.

as the average distance of neighbouring cells within each segment. These are combined in the segmentation score

$$s(\mathbf{S}) = s_e(\mathbf{S}) - s_a(\mathbf{S}) = s(b(\mathbf{S})) - \operatorname*{avg}_{S \in \mathbf{S}} \operatorname*{avg}_{\langle c_i, c_j \rangle \in S} d(c_i, c_j) \tag{2}$$

that we want to maximise.

Example 3. A segmentation $\mathbf{S}_e$ and its boundary are shown in Figure 3. The inter- and intra-segment scores are computed as follows.

$s_e(\mathbf{S}_e) = \frac{1}{5}\left(d(c_1, c_2) + d(c_4, c_5) + d(c_7, c_8) + d(c_5, c_8) + d(c_6, c_9)\right)$
$s_a(\mathbf{S}_e) = \frac{1}{7}\left(d(c_1, c_4) + d(c_4, c_7) + d(c_2, c_5) + d(c_3, c_6) + d(c_2, c_3) + d(c_5, c_6) + d(c_8, c_9)\right)$

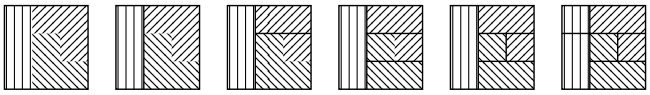
3.3 Algorithm

We combine a divisive and an agglomerative step to heuristically search for segmentations that satisfy the given constraints and have high scores. Let $\mathbf{S}_t$ be the unknown target segmentation—a segmentation that ideally maximises Equation 2. Our algorithm consists of three main steps. First, in the divisive step, we look for a segmentation $\mathbf{S}_o$ such that (an approximation of) $\mathbf{S}_t$ can be obtained by joining as few segments of $\mathbf{S}_o$ as possible. Second, in the agglomerative step, we merge segments to search for segmentations that satisfy the given constraints. Finally, the resulting segmentations are ranked.

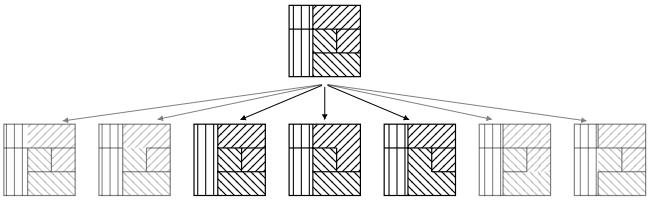
Divide Inspired by *superpixels* in image segmentation, we want to find small groups of highly similar cells by iteratively splitting one segment. Let S be an $n_s \times m_s$ segment in the current segmentation $\mathbf{S}_i$. There are $(n_s - 1)(m_s - 1)$ straight boundaries that divide S in two segments. The best candidate boundary for splitting S is the one for which Equation 1 is maximal, indicating that cells on both sides of the boundary are dissimilar. Let $\mathbf{b}(S)$ be the set of these candidate boundaries. We define the *splitting score* of a boundary $b \in \mathbf{b}(S)$ as

$$s_{split}(b) = s(b) - \operatorname*{avg}_{b' \in \mathbf{b}(S)} s(b') + \operatorname*{std}_{b' \in \mathbf{b}(S)} s(b') \tag{3}$$

where the boundary score $s(b)$ is adjusted for variations in the similarity between values of different types. This allows for comparing of splitting scores across



(a) Example of the divisive step. The final split and any subsequent splits will be based on a very slow splitting score.



(b) Graphical example of a single step of the merge algorithm with $w = 3$. Only candidate joins that satisfy the shape constraint are part of the merge tree. Of those, the three best candidates are added to the stack and will be expanded in the future.

Fig. 4: Graphical example of oversegmentation and merging steps using a tromino shape constraint and no position constraint.

different segments, as we expect the distance between cells of different types to have varying magnitudes. In other words, we look for a boundary with an unusually large boundary score *for its segment*. The divisive algorithm then works by iteratively splitting the segment with the highest scoring boundary along this boundary. It stops when the score of the chosen boundary becomes too small.

Merge Starting from the segmentation $\mathbf{S}_o$ that was found in the previous step, we can iteratively merge segments to look for segmentations that satisfy the constraints. This corresponds to a *merge tree* as used for object segmentation in images [18]. In this tree, each node is a segmentation and its children are obtained by merging two neighbouring segments. Rather than full enumeration, we perform a heuristic beam stack search [21] with constraint checking.

Let $\mathbf{S}_i$ be the current segmentation popped from the stack. Using breadth-first search, we search the neighbourhood of every segment $S \in \mathbf{S}_i$ for the lowest number $k < k_m$ of segments that can be merged with S to create a new segment that satisfies the shape constraint, where k_m is a hyperparameter. Merging k segments at once amounts to following $k - 1$ edges in the merge tree, but the intermediate nodes are never explicitly considered. Each of these combinations of $k + 1$ candidate segments is scored using Equation 1 and the w best ones are pushed to the stack, with w another hyperparameter. The algorithm is initialised with only $\mathbf{S}_o$ on the stack and stops when the stack is empty.

Example 4. An example of both divide and merge of a segmentation problem is shown in Figure 4. The shape constraint is that every segment must consist of exactly three orthogonally connected cells, there is no position constraint.

Similarity	Description
embedding	Cosine similarity between sum-of-word-vectors.
alignment	Global alignment similarity between strings with lowercase, uppercase and digits substituted `a`, `A` and `0`, respectively.
compressed alignment	Same as alignment, but with subsequent, identical characters compressed into a single one.
prefix	Longest prefix similarity.
postfix	Longest postfix similarity.
longest common substring	Longest common substring similarity.

Table 1: Individual similarity functions used for training a mixed similarity.

4 Evaluation

We now evaluate applicability and flexibility of MUPPETS on three use cases. In these experiments, we use the ranking of segmentations produced by MUPPETS.

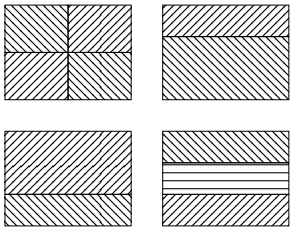
4.1 Single column type detection

Automatically discovering the statistical and semantic types of data in tables is a valuable tool in data preparation and information retrieval. Accordingly, methods have been presented that predict the type of a column [3, 4]. These methods expect the values in a column to have the same type. If this is not the case, they will not work or perform worse.

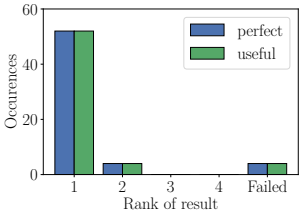
By generating tables where data of the same type is not in one column, we show that MUPPETS is capable of detecting segments in this context. Figure 5a shows configurations of such tables. Segments of the same pattern are populated with values of a single type, randomly sampled from half of the columns used to evaluate the Sherlock [13] type detection system.

The other half of these domains was used to train a mixed syntactic and semantic distance function. For two cells c_1 and c_2, we first compute a feature vector $\vec{d}(c_1, c_2)$ from k individual distance functions $\{d_1, \ldots, d_k\}$ and train a probabilistic classifier to predict whether c_1 and c_2 are taken from the same domain. Given two new values, their similarity is the probability of classifying them as being from the same domain. All considered similarity functions are shown in Table 1.

We show two results of running MUPPETS with this trained similarity measure and a tabular constraint on 60 generated problems in Figure 5b. First, we show the rank of the perfect segmentation. Second, we also show the rank of the first segmentation in which no values of different types are in the same segment. Each segment then contains values of a single type, their types can be detected, and the segmentation is thus useful. In almost all cases, the correct segmentation is obtained as the highest ranked one. When this does not happen, we see that the similarity fails to distinguish some domains, because they contain syntactically distinct values or embeddings do not capture their semantics.



(a) Example table configurations where type detection fails without segmentation.



(b) Distribution of ranks of perfect segmentations and first segmentation in which all values of the same domain are in the same segment.

Fig. 5: Example of table configurations and results of detecting segments when generating tables from these configurations.

4.2 Semantic error detection

Error detection in spreadsheets typically happens on the basis of formulas, as these are prone to mistakes [11]. Such errors are called *spreadsheet smells* and approaches to find them often detect blocks of data that are influenced by the same formula [14]. Other errors can occur as well, however, such as copy and paste errors or artefacts of misaligned data. We call these *semantic errors*, as they require understanding of the semantics of a cell. This experiment shows how a noisy semantic similarity between words can be used by MUPPETS to detect such semantic errors in spreadsheets.

After finding a segmentation, we compute the average distance $\bar{d}(c)$ of every cell $c \in S$ to all other cells in the same segment. Additionally, we compute the average similarity $\bar{d}(S)$ between all pairs of cells in the segment. The error score of a cell $c \in S$ is then $\bar{d}(c) - \bar{d}(S)$. A high error score indicates that the cell is not like other elements in the same segment, and thus probably an error. Figures 6b and 6c show a heat map of the error scores for all cells in Table 6a in case of absence and presence of the segmentation as a 2×2 checkerboard.

The experiment is then performed as follows. We generate tables with errors by filling a table template with data from a pair of data domains and randomly replacing a single cell with a value from the other domain. Domain pairs are selected either two columns from the same dataset, such as movies and genres, or from the same column but with a distinct property, such as athletes from different sports. The full list of domain pairs is shown in Table 2. They were chosen to be syntactically indistinguishable. Three fairly weak semantic similarities are considered: the normalised web distance [6] using either Wikipedia or ChatNoir as search engines—that can be queried for free—and embedding similarity with spaCy [12] using the `en_core_web_lg` model.

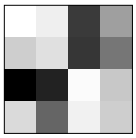
All cells are then ranked by their error score. The distribution of ranks of the actual error for 60 iterations is shown in Figure 7. All similarity measures are able to correctly identify close to a third of errors. We also show the lowest rank obtained for every iteration using either of the three similarities, correctly

Different columns		Different property	
Domain 1	Domain 2	Domain 1	Domain 2
baseball players	football players	male gymnasts	female gymnasts
soccer players	soccer clubs	American gymnasts	Russian gymnasts
car manufacturer	car type	fencers	boxers
movie studio	movie genre		
Pokémon name	Pokémon type		
countries	flavours		

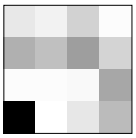
Table 2: Domain pairs for semantic error detection.

Belgium	Netherlands	Vanilla	Strawberry
Spain	Poland	Pistachio	Banana
Cookie dough	Peppermint	Germany	Russia
Greece	Cherry	Austria	Spain

(a) Table following a 2×2 checkerboard pattern with a semantic error.



(b) no segmentation



(c) segmentation

Fig. 6: Table with a semantic error and its error scores (b) without and (a) with the segmentation using embedding similarity.

identifying 43 out of 60 errors. In cases where the actual error was not ranked first, unexpected values often confused the similarity measure. For example, in the car manufacturers and types domain, the word "pickup" yields high error scores due to its double meaning.

4.3 Table and header detection

A popular tool for working with tables is a Python package called `pandas`. It provides functions `read_csv` and `read_excel` for reading tables from their respective formats. Important parameters are `skiprows` that controls what part of the file to load and `header` and `index_col` for selecting the structural properties of the table. For example, the table in Figure 1 requires `skiprows=2`, `header=0` and `index_col=0` for loading it correctly. Searching for the tags `[pandas]` and `[csv]` on StackOverflow, we found 20 questions about loading a table, where the appropriate values for these parameters was the solution.

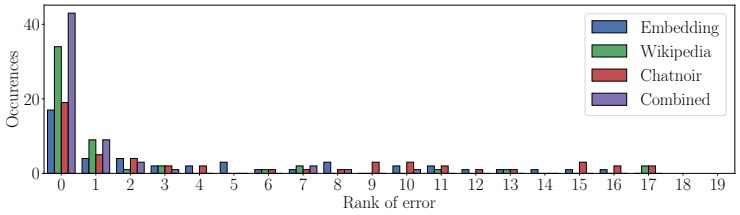


Fig. 7: Distribution of the ranks of errors when sorting by their error score.

Parameter	Tab	Sub	Rec	Any
`skiprows`	19	20	20	20
`header`	17	17	18	19
`index_col`	12	16	19	19
All	12	15	17	

Table 3: Parameter detection results.

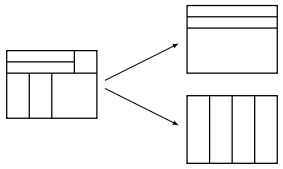


Fig. 8: Cuts based on boundaries.

We use MUPPETS to estimate the parameters for these tables as follows. First, we use the compressed alignment similarity from Table 1 and one of the tabular, subtabular or rectangular constraints to obtain the highest ranked segmentation. From this segmentation, horizontal and vertical *cuts* are made along all boundaries of segments, as visualised in Figure 8. The `header` and `skiprows` parameters are respectively chosen to be the lowest horizontal cut and the lowest horizontal cut above which is a row that is mostly empty. On the remaining rows, `index_col` is chosen as the leftmost vertical cut such that the columns left of it are a valid index and thus contain unique rows.

For the different position constraints, Table 3 shows how often each individual parameter and all parameters together were correctly recovered, and whether it was estimated correctly using any of the constraints. Only two parameter values were never recovered. Imposing more position constraints results in coarse segmentations and information is lost. Estimating `index_col` either fails because both index and subsequent column are numerical and the distance is to small, or because the ground truth has no `index_col` and there is a vertical cut between columns of different types. Similarly, `header` either fails because data and headers are too similar or because empty cells cause superfluous segments in the data regions. In these unsuccessful cases, it is the similarity that fails to capture semantic meaning.

5 Related Work

A first line of related work is layout detection in spreadsheets, where the goal is to infer the layout of a spreadsheet and use it to extract data. Rather than distinguishing between cells, these systems take a predictive approach and try to detect their functional role—for example, whether they contain data, metadata, derived values or headers. One approach is to train a classifier using a manually curated set of syntactic and stylistic features on annotated data [16]. A more recent approach first trains context and style embeddings on unsupervised data and then uses these embeddings to make predictions with a recurrent architecture [10].

These cell roles can then be used to infer the layout using heuristics [8], graphs [15] or a genetic approach [17]. Segmentation is complementary to these approaches for detecting finer grained layouts in the *data* region, which we used to perform semantic error detection and improve semantic role detection. On a related note, the problem of detecting tables in spreadsheets [7] or CSV files [5] has recently received some attention.

Table segmentation is related to statistical and semantic type detection, where the goal is to find the data type of a set of values. Unlike our unsupervised segmentation approach, type detection generally works in a predictive setting, where the goal is to classify the statistical type of columns or to annotate them with semantic types [4, 3]. As data is assumed to be grouped in sets of values that share a distinctive type, table segmentation can serve as a preprocessing step.

6 Conclusion and future work

We presented the flexible MUPPETS framework for the new problem of partitioning a table in segments that fulfil the same role. The framework is parametrised by a distance function between cells, which allows it to be used for different use cases. Three use cases were introduced in which MUPPETS either solves a new problem or complements existing approaches: detecting the types of cells, detecting semantic errors and easily loading tables.

Two direct pointers for future work are learning a general similarity measure between cells and learning the constraints from annotated tables. Both are aimed at making MUPPETS applicable for new use cases, such as data wrangling. Different search strategies can also be explored. For example, an evolutionary approach might be less likely to suffer from local errors that prevent the correct segmentation from being found—at the cost of performance.

Acknowledgements

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models). This research received funding from the Flemish Government (AI Research Program), the EU (FEDER) and the Spanish MINECO RTI2018-094403-B-C32 and the Generalitat Valenciana PROMETEO/2019/098. LCO was also supported by the Spanish MECD grant (FPU15/03219).

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM **26**(11), 832–843 (1983)
2. Barowy, D.W., Berger, E.D., Zorn, B.: Excelint: automatically finding spreadsheet formula errors. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 148 (2018)
3. Ceritli, T., Williams, C.K., Geddes, J.: ptype: probabilistic type inference. Data Mining and Knowledge Discovery pp. 1–35 (2020)
4. Chen, J., Jiménez-Ruiz, E., Horrocks, I., Sutton, C.: Colnet: Embedding the semantics of web tables for column type prediction. Proceedings of the of the 33th AAAI Conference on Artificial Intelligence (AAAI19) (2019)
5. Christodoulakis, C., Munson, E.B., Gabel, M., Brown, A.D., Miller, R.J.: Pytheas: pattern-based table discovery in csv files. Proceedings of the VLDB Endowment **13**(12), 2075–2089 (2020)

6. Cilibrasi, R.L., Vitanyi, P.M.: The google similarity distance. IEEE Transactions on knowledge and data engineering **19**(3), 370–383 (2007)
7. Dong, H., Liu, S., Han, S., Fu, Z., Zhang, D.: Tablesense: Spreadsheet table detection with convolutional neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 69–76 (2019)
8. Eberius, J., Werner, C., Thiele, M., Braunschweig, K., Dannecker, L., Lehner, W.: Deexcelerator: a framework for extracting relational data from partially structured documents. In: Proceedings of the 22nd ACM international conference on Information & Knowledge Management. pp. 2477–2480 (2013)
9. Gautrais, C., Dauxais, Y., Teso, S., Kolb, S., Verbruggen, G., De Raedt, L.: Human-machine collaboration for democratizing data science. arXiv preprint arXiv:2004.11113 (2020)
10. Gol, M.G., Pujara, J., Szekely, P.: Tabular cell classification using pre-trained cell embeddings. In: 2019 IEEE International Conference on Data Mining (ICDM). pp. 230–239. IEEE (2019)
11. Hermans, F., Pinzger, M., van Deursen, A.: Detecting code smells in spreadsheet formulas. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM). pp. 409–418. IEEE (2012)
12. Honnibal, M., Montani, I.: spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing (2017), to appear
13. Hulsebos, M., Hu, K., Bakker, M., Zgraggen, E., Satyanarayan, A., Kraska, T., Demiralp, Ç., Hidalgo, C.: Sherlock: A deep learning approach to semantic data type detection. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 1500–1508 (2019)
14. Koch, P., Hofer, B., Wotawa, F.: On the refinement of spreadsheet smells by means of structure information. Journal of Systems and Software **147**, 64–85 (2019)
15. Koci, E., Thiele, M., Lehner, W., Romero, O.: Table recognition in spreadsheets via a graph representation. In: 2018 13th IAPR International Workshop on Document Analysis Systems (DAS). pp. 139–144. IEEE (2018)
16. Koci, E., Thiele, M., Romero, O., Lehner, W.: Cell classification for layout recognition in spreadsheets. In: International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management. pp. 78–100. Springer (2016)
17. Koci, E., Thiele, M., Romero, O., Lehner, W.: A genetic-based search for adaptive table recognition in spreadsheets. In: 2019 International Conference on Document Analysis and Recognition (ICDAR). pp. 1274–1279. IEEE (2019)
18. Liu, T., Seyedhosseini, M., Tasdizen, T.: Image segmentation using hierarchical merge tree. IEEE transactions on image processing **25**(10), 4596–4607 (2016)
19. Minaee, S., Boykov, Y., Porikli, F., Plaza, A., Kehtarnavaz, N., Terzopoulos, D.: Image segmentation using deep learning: A survey. arXiv preprint arXiv:2001.05566 (2020)
20. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics **20**, 53–65 (1987)
21. Zhou, R., Hansen, E.A.: Beam-stack search: Integrating backtracking with beam search. In: ICAPS. pp. 90–98 (2005)
22. Zhu, H., Meng, F., Cai, J., Lu, S.: Beyond pixels: A comprehensive survey from bottom-up to semantic image segmentation and cosegmentation. Journal of Visual Communication and Image Representation **34**, 12–27 (2016)