Automatic Web Spreadsheet Data Extraction

Zhe Chen
University of Michigan
Ann Arbor, MI 48109-2121
chenzhe@umich.edu

Michael Cafarella
University of Michigan
Ann Arbor, MI 48109-2121
michjc@umich.edu

ABSTRACT

Spreadsheets contain a huge amount of high-value data but do not observe a standard data model and thus are difficult to integrate. A large number of data integration tools exist, but they generally can only work on relational data. Existing systems for extracting relational data from spreadsheets are too labor intensive to support ad-hoc integration tasks, in which the correct extraction target is only learned during the course of user interaction.

This paper introduces a system that automatically extracts relational data from spreadsheets, thereby enabling relational spreadsheet integration. The resulting integrated relational data can be queried directly or can be translated into RDF triples. When compared to standard techniques for spreadsheet data extraction on a set of 100 random Web spreadsheets, the system reduces the amount of human labor by 72% to 92%. In addition to the system design, we present the results of a general survey of more than 400,000 spreadsheets we downloaded from the Web, giving a novel view of how users organize their data in spreadsheets.

1. INTRODUCTION

Spreadsheets have become a critical data management tool [3, 20]. They allow non-experts to perform tasks we traditionally associate with relational systems: selection, projection, sorting, etc. Spreadsheets make up some of the most expensive data available to us, because they have been constructed by hand by well-paid professionals. They are a standard tool for many researchers, scientists, and policymakers and have found especially wide adoption in financial and clinical research settings [15]. In short, spreadsheets make up an important dataset but live outside mainstream data management practices.

Spreadsheets often contain data that are roughly relational, but the schema is often designed for human consumption and entirely implicit. As a result, spreadsheets cannot benefit from society's huge investment in data management tools that work on relational databases. In particular, spreadsheets lack data integration operations. For example, it is easy to imagine an analyst who wants to combine a spreadsheet about company sales with a government-produced spreadsheet about economic performance to predict future sales. But in practice, the analyst would likely have to write custom code to integrate the two spreadsheets. If we could automate this burdensome integration procedure, users could make vastly more effective use of the spreadsheet data on the Web, intranets, and elsewhere. Given the high value of data stored in spreadsheets, such a system would likely have a large impact on the lives of analysis-minded Web users.

Extracting relational data from spreadsheets would enable traditional data integration methods to unlock the latent value in spreadsheet data. Recent studies [1, 2, 8, 11] attempted to transform spreadsheet data into the relational model, making further integration among spreadsheets possible. Some extraction systems require explicit sheet-specific user-provided rules [2, 11], which might yield good results for a single spreadsheet. But they are not feasible for our setting: the corpus is large and users are not aware of the target spreadsheets to be processed ahead of time. It is impractical to manually transform all of them to relations. Abraham and Erwig[1] and Cunha et al. [8] automatically infer some spreadsheet structure, but they cannot process *hierarchical* spreadsheets. This type of spreadsheet is commonplace and extracting metadata from such spreadsheets presents the central technical challenge of this paper. We will illustrate these challenges using an example of a Web spreadsheet downloaded from the U.S. Census Bureau.

Challenges – The spreadsheet in Figure 1 shows a spreadsheet about the smoking rate downloaded from the government's Statistical Abstract of the United States.[1]. Each row clearly represents a different configuration of the smoking rate; for example, 13.7 in the *value region* is the rate for people with constraints Male, White, 65 years and over in the *attribute region*, and it yields an annotating *relational tuple* at the bottom. But there are two main problems here. First, the spreadsheet only implicitly indicates which cells carry *values* versus *attributes*. Often a spreadsheet is a mix of attributes, values, and other elements such as titles and footnotes. These elements are not easily distinguished from each other. Second, the spreadsheet does not explicitly indicate which *attributes* describe which *values*. If the leftmost column is processed naïvely, rows 25, 31, and 37 will yield three tuples that have different smoking rates for 65 years and older. All three extracted tuples are incorrect, as none

[1] http://www.census.gov/compendia/statab/2012/tables/12s0204.xls

SSW'13, August 30, 2013, Riva del Garda, Italy.

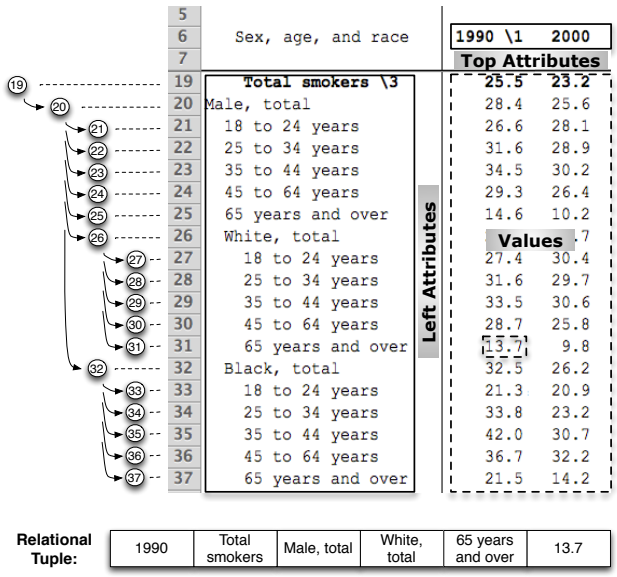

Figure 1: A portion of one spreadsheet from the U.S. Census Bureau.

will contain any mention of the attribute Male. In summary, Figure 1 shows a clean, high-quality spreadsheet, but extracting relational data from it requires us to: (1) detect *attributes* and *values*, (2) identify the hierarchical structure of left and top attributes, and (3) generate a *relational tuple* for each value in the spreadsheet.

Background – We implemented SENBAZURU [5], a prototype spreadsheet database management system (SSDBMS). SENBAZURU focuses on *data frame* spreadsheets, which are one of the most popular types in the Web. SENBAZURU searches a large number of Web crawl spreadsheets, and it automatically transforms spreadsheets into relations while allowing users to fix the extraction errors effectively and efficiently. Finally, it supports selection queries on the resulting relations and join queries to integrate arbitrary spreadsheets. In summary, SENBAZURU is able to extract relational information from a large number of Web spreadsheets, which makes it possible to directly manage spreadsheet data in databases; doing so also opens up opportunities for data integration among spreadsheets and with many other relational data sources.

Contributions – In this paper, we present the first automatic, domain-independent spreadsheet extractor, which is the first step in building SENBAZURU. First, we analyzed our Web crawl spreadsheets, 410,554 Microsoft Excel files from 51,252 distinct Internet domains. Our findings on those Web spreadsheets help us to better understand numerous spreadsheet usage scenarios on the Web and motivate our goal to design a practical SSDBMS. Second, our spreadsheet extractor automatically recognizes spreadsheet's layout, discovers the implicit metadata structure and emits relational data for a given spreadsheet. In contrast to previous research, our extraction does not require users to explicitly provide extraction rules. In our follow-up work, we will explore how to effectively and efficiently incorporate users' efforts to fix extraction errors, thus generating perfect relational tables from spreadsheets.

Applications – The central application goal of this work is a system that can combine spreadsheet data from many different sources, including the Web, intranets, and local storage. But there are a number of byproducts from our extraction procedure that could be useful for other semantic Web applications. For example, the procedure recovers a large amount of hierarchical metadata that is implicit in the spreadsheet and may not exist in a more formal database. Consider the organizational hierarchy of a small company; the management relationships among employees may not exist in a formal LDAP system but could plausibly be recovered from a stray spreadsheet. There may also be business-specific hierarchies (e.g., families of industrial materials, or manufacturing stages) that are otherwise not recorded. These hierarchies are critical for our integration and search application but could also be useful on their own merit when combined with other semantic tools.

Organization – The paper is organized as follows.

- We describe a general survey of spreadsheet data practices based on 410,554 spreadsheets we downloaded from a general Web crawl (Section 2).
- We present our domain-independent extractor that obtains relational tuples from raw spreadsheets without any human intervention (Section 3).
- We evaluate the system's accuracy on a random sample of 100 Web hierarchical spreadsheets. We find that our methods can accurately obtain relational tuples from a spreadsheet; compared to a standard technique, our method reduces the amount of work a human must perform between 72% and 92% on average (Section 4).

In Section 5, we differentiate our work from the previous related work. Finally, we conclude and discuss the future work in Section 6.

2. THE WEB SPREADSHEET CORPUS

We obtained 410,554 Microsoft Excel files from 51,252 distinct Internet domains from our Web crawl. We call this collection the WEB dataset. We located the spreadsheets by looking for Excel-style file endings among roughly ten billion URLs in the ClueWeb09 Web crawl [6]. However, to the best of our knowledge, there is no other study that has surveyed a large number of spreadsheets in the Web. Therefore in this section, we aim to create the first portrayal of the WEB dataset in order to design our spreadsheet extraction pipeline. But first, we introduce notation to describe spreadsheets.

2.1 Spreadsheet Notations

In this paper, we focus on a two-part spreadsheet structure that we call a **data frame**. Data frame spreadsheets represent a type of spreadsheet, and this structure consists of two semantic components: a block of numeric values as a *value* region and accompanied *attribute* or *metadata* regions on the *top* or to the *left*. For attributes on the *right*, we treat them as an extension of *left* attributes. For example, Figure 1 shows a *data frame* with a value region indicated by the dashed rectangle and attribute regions on the *top* and to the *left* indicated by the solid rectangles.

A **hierarchical** spreadsheet is a *data frame* spreadsheet with either a hierarchical *left* attribute region or a hierarchical *top* attribute region. An example of a hierarchical *left* region can be found in Figure 1, and an example of hierarchical *top* region is shown in Figure 5. In contrast, *flat*

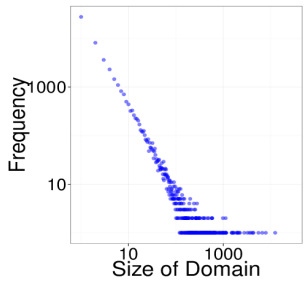

Figure 2: The distribution of Web spreadsheets.

Domain	# files	% total	data frame	h-top	h-left
www.bts.gov	12435	3.03%	99%	30%	40%
www.census.gov	7862	1.91%	94%	72%	70%
www.stat.co.jp	6633	1.62%	x	x	x
www.bankofengland.co.uk	5520	1.34%	98%	77%	35%
www.ers.usda.gov	4328	1.05%	95%	77%	70%
www.agr.gc.ca	4186	1.02%	87%	77%	81%
www.wto.org	3863	0.94%	96%	61%	77%
www.doh.wa.gov	3579	0.87%	81%	53%	64%
www.nsf.gov	2770	0.67%	96%	53%	76%
nces.ed.gov	2177	0.53%	98%	55%	92%
average	**5335**	**1.30%**	**93.78%**	**61.67%**	**67.33%**

Figure 3: The top 10 domains in our Web spreadsheet corpus. h-top **and** h-left **are percentages of spreadsheets with a hierarchical *top* or *left* region.**

spreadsheets refer to those spreadsheets without any hierarchical structure.

For each value in the *value* region, there is usually (but not necessarily) at least one annotating string in the *top* and *left* regions, generating a **relational tuple**. For example, in Figure 1, the value 13.7 is annotated by 65 years and over, White total, Male total, and Total smokers in the *left* region and by 1990 in the *top* region, yielding a relational tuple.

2.2 Web Spreadsheets Survey

To better design our extraction system, we answer the following critical questions about the general properties of the Web spreadsheets and the popularity of the *data frame* spreadsheets on the Web:

1. Where are those Web spreadsheets from? The Web spreadsheets cover a huge range of topics and show wide variance in cleanliness and quality. Most of the spreadsheets are statistical data, with a heavy emphasis on government, finance, transportation, etc. We are also interested in the distribution of the spreadsheets from different Internet domains. Figure 3 shows the top 10 Internet domains that host the largest number of spreadsheets in the WEB corpus. Nine of the top 10 domains are sites run by the U.S., Japanese, UK, or Canadian governments. Figure 2 shows the distribution of spreadsheets among hosting domains. We rank the domains according to the size of their hosting spreadsheets in descending order. The plot indicates that the spreadsheets follow a strongly skewed distribution, with a large number of spreadsheets from relatively few domains and with a large number of domains hosting relatively few spreadsheets.

2. How many of the Web spreadsheets consist of* data frame *structures? To better understand the structure of the WEB spreadsheets, we randomly chose 200 samples and asked a human expert to mark their structures. We found 50.5% of the spreadsheets consist of *data frame* components and 32.5% have hierarchical top or left attributes. The other 49.5% non-*data frame* spreadsheets belong to the following categories: 22.0% are Relation spreadsheets that can be converted to the relational model almost trivially (we can simply translate each spreadsheet column into a relational table column and translate each spreadsheet row into a relational tuple); 10.5% are Form spreadsheets that are not for data storage and are designed to be filled by a human; 3.5% are Diagram spreadsheets for visualization purposes, and they are often data-intensive without any schema information; and 3% are List spreadsheets that consist of non-numeric tuples. The 10.5% Other spreadsheets are schedules, syllabi, scorecards, or other files whose purpose is unclear. Although there are a variety of categories of spreadsheets on the Web, in this paper, we only focus on *data frame* spreadsheets.

3. How many of the Web spreadsheets are hierarchical like the example shown in Figure 1? Are those hierarchical spreadsheets spread uniformly across the Web? As just mentioned, 32.5% of the 200 sample Web spreadsheets have hierarchical *top* or *left* attributes in a *data frame*. To better understand how the hierarchical spreadsheets are distributed in different domains, we randomly selected 100 spreadsheets from each of the top 10 domains, yielding 900 spreadsheets in total.[2] Figure 3 shows the fraction of spreadsheets with *data frames* or hierarchical attributes in the top 10 domains. The ratios are much higher than the fractions we obtained from the general Web sample. We also randomly selected 100 spreadsheets from domains hosting fewer than 10 spreadsheets. We found 19% with *data frame* structures, 4% of which have hierarchical *top* attributes and 6% of which have hierarchical *left* attributes. These results suggest that the number of hierarchical spreadsheets differs greatly by domain and may be linked to the domain's popularity or degree of professionalism. Computing the exact distribution of hierarchical spreadsheets among domains would be useful but requires a huge amount of labeled data; we will explore this question in future work. Even without computing that distribution, we have found a huge number of hierarchical spreadsheets: 32.5% of all spreadsheets on the Web and more than 60% in popular domains. Therefore, to extract relational data from spreadsheets, we believe our system must process hierarchical-style metadata.

Overall, we observe that: (1) the Web contains a huge variety of spreadsheets from a large range of data sources, and (2) the *data frame* spreadsheets, especially the hierarchical ones, are highly popular in the Web. Therefore, to design a system to extract relational data from spreadsheets, the system should be able to process *data frame* spreadsheets, especially those hierarchical-style spreadsheets.

3. SYSTEM PIPELINE

In this section, we describe our spreadsheet extraction pipeline. The goal of the extraction pipeline is to create a relational model of the data embedded in *data frame* spreadsheets: it takes in a *data frame* spreadsheet and emits relational tuples. It should be able to work on both *flat* and *hierarchical* spreadsheets (we treat *flat* spreadsheets as a

[2]www.stat.co.jp is excluded because it is in Japanese.

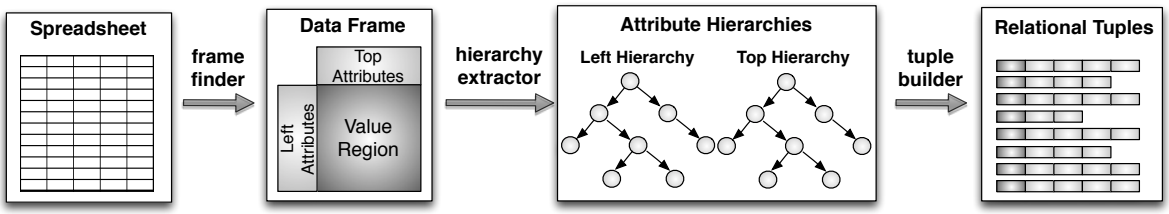

Figure 4: The system pipeline to process a single spreadsheet.

special case of *hierarchical* ones). As shown in Figure 4, the extraction pipeline consists of three components: the **frame finder**, the **hierarchy extractor**, and the **tuple builder**. The **frame finder** identifies the *data frames*, locating *attribute* regions and *value* regions. The **hierarchy extractor** recovers the hierarchical metadata from spreadsheets, and the **tuple builder** generates a relational tuple for each value in the *value region*.

Notice that the extraction pipeline does not explicitly distinguish *data frame* spreadsheets from non-*data frame* spreadsheets. We assume that given a spreadsheet, if the pipeline could process the spreadsheet and output a set of relational tuples in a good quality, then the spreadsheet is a *data frame* spreadsheet; otherwise it is not. Of course, this approach might yield false positive *data frame* spreadsheets. However, a post-processing stage could be added to the end of the pipeline to further filter the non-*data frame* spreadsheets. We skip details of this post-processing component because it is beyond the scope of this paper.

3.1 Frame Finder

The **frame finder** identifies the *value region* and the *top* and *left* attribute regions. It receives a raw spreadsheet as input and emits geometric descriptors of the *data frame*'s three rectangular regions. We define the problem as follows:

DEFINITION *1.* (Frame Finder) Let a spreadsheet be a grid of cells $\mathbf{c} = \{c_{ij}\}$, where i represents the row index and j represents the column index. The **frame finder** assigns each cell $c_{ij} \in \mathbf{c}$ with a label $l_{ij} \in \mathrm{L} = \{$top, left, value, other$\}$, where top represents *top attributes*, left represents *left attributes*, value represents *values*, and other represents everything else.

To simplify the problem, we assume that the structure of the spreadsheets has the following property: there may be multiple *data frames* in a spreadsheet, but they only stack in the vertical dimension. In fact, we found less than 2% of the 900 spreadsheets in Figure 3 violate the assumption. This assumption allows us to treat *data frame*-finding as a problem of row labeling. Therefore, we start with the **row labeler** task, which assigns each row in a spreadsheet to one of the following four categories: title, header, data, or footnote. The label title represents a spreadsheet title, header represents a row that contains *top* attributes only, data represents a row that contains *left* attributes or *values*, and footnote is information that annotates the main contents. As in Figure 1, rows 5-7 are labeled header and rows 19-37 are labeled data. A formal definition is as follows:

DEFINITION *2.* (Row Labeler) Let $\mathbf{r} = \{r_1, r_2, ..., r_N\}$ be a set of variables representing the non-empty rows in a spreadsheet. The **row labeler** assigns each $r_i \in \mathbf{r}$ with a label $l_i \in L = \{$title, header, data, footnote$\}$.

1	Table 5-8: Active Aviation Pilots and Flight Instructors: 2000[1]							
2								
3				Airplane pilots[2]				
4	State	Total	Students	Private	Commercial	Airline transport	Misc.[3]	Flight instructor[4]
5	Alabama	7,262	1,170	3,065	1,649	1,084	294	920
6	Alaska	8,638	833	3,686	2,130	1,906	83	1,118
7	Arizona	17,429	2,329	6,508	3,345	4,654	593	2,617
8	Arkansas	4,988	776	2,153	1,206	788	65	634
9	California	71,053	10,173	31,571	13,448	12,786	3,075	8,984
54	Wisconsin	11,275	1,768	5,682	1,884	1,830	111	1,455
55	Wyoming	1,812	254	901	354	273	30	195
56	United States, total	593,218	87,319	244,389	112,092	134,024	15,394	78,686

Figure 5: An example of hierarchical top attributes.

We observe the following two types of signals that the **row labeler** should use to automatically assign semantic labels to each non-empty row: (1) the properties of each non-empty row indicate its semantic label, such as its fonts and keywords; and (2) the labels assigned to adjacent rows are highly related. For example, if we know the current row is a header row, it is highly probable that the next row is a header or data row. Therefore, we employ an approach based on a linear-chain, conditional random field (CRF) [13] to exploit these two types of signals. Pinto et al. [16] used linear-chain CRFs to obtain labels for textual tables in government statistical text reports. We also use the linear-chain CRFs to obtain the semantic labels for each row of a spreadsheet, and our training and inference procedure is the same. However, with the access to spreadsheet APIs, we are able to build the CRFs with a richer set of features, such as the alignment and indentation information that is hard to obtain from plain text reports. Our extraction features fall into two main categories: *layout features* test visual characteristics of a row, and *textual features* test the contents of the row. The details of the features can be found in Appendix A.

Once we have labels for each row in a spreadsheet, we can construct the correct *data frame* regions. The vertical extent of a *value region* is described by the set of rows marked data, and its horizontal extent is determined by finding regions of numeric values. The *top* attribute region is delimited by all header rows, and the *left* attribute region is everything to the left of the *value region*.

3.2 Hierarchy Extractor

The **hierarchy extractor** recovers the *attribute hierarchies*. This step receives a *data frame* with *top* and *left* regions as input and emits hierarchies as output: one for *left* and one for *top*. These trees describe the hierarchical annotation relationship among attributes in the *top* and *left* regions. For example, in Figure 1, row 31 is annotated by attributes at rows 26, 20, and 19. An example of a *top* hierarchy can be found in Figure 5, where the *attribute* Airplane pilots annotates the *attribute* Airline transport. Now we formally describe the problem of recovering the *attribute hierarchy* for a single region as follows:

DEFINITION *3.* (Hierarchy Extractor) Let $A = \{a_1, ..., a_N, root\}$ represent the set of cells in an attribute region. Given $a_i, a_j \in A$, we say (a_i, a_j) is a *ParentChild* pair if a_i is a parent attribute of a_j in the hierarchy. The **hierarchy extractor** identifies all the *ParentChild* pairs in the attribute region.

For example, consider the hierarchy on the left of Figure 1 where each node represents an attribute in the corresponding row. (20, 26) is a *ParentChild* pair, while (20, 31) and (24, 25) are not. For a top hierarchy as shown in Figure 5, (Airplane pilots, Airline transport) is a *ParentChild* pair.

The goal of the **hierarchy extractor** is to find all such *ParentChild* pairs in an attribute region, thus describing a hierarchical tree. It may seem that a simple heuristic can recover the *annotation hierarchies*, but in practice, a correct heuristic is extremely hard to obtain and to generalize to a large number of spreadsheets. As in Figure 1, a simple heuristic, such as the indentations of *left* attributes, may identify some *ParentChild* pairs but fails to recognize *(19, 20)* as a *ParentChild* pair. Therefore, to obtain a correct hierarchy, we need to use a probabilistic method to exploit a variety of signals. In the rest of the section, we discuss our proposed two *classification-based* algorithms.

3.2.1 Algorithm 1: Classification

To obtain all the *ParentChild* pairs in an attribute region, our **classification-based** approach first generates a series of *ParentChild* candidates and then finds all the true *ParentChild* pairs through classification.

A straightforward way to generate *ParentChild* candidates for an attribute region $A = \{a_1, ..., a_N, root\}$ is to create a (a_i, a_j) for each $a_i, a_j \in A$. This simple method will yield thousands of nodes for every single attribute region A with N attributes. The size of created candidates potentially could be very large, but in practice, we can greatly shrink the number by leveraging our observations on the spreadsheets to filter the unlikely candidates.

For ***left* attributes**, given a *ParentChild* pair candidate (a_i, a_j) in a *left* attribute region, we observe that a candidate in two cases is not likely to be a *ParentChild* pair.

1. The formatting styles of the parent and child attributes are the same. We believe that formatting style of an attribute cell is a strong indication of the hierarchical structure. Given a *ParentChild* candidate (a_i, a_j), a_i and a_j are determined to share the same formatting style or not based on a predefined *style* feature vector. The *style* feature vector for each attribute, style(a), is a combination of the unary extraction features shown in Table 5 and the attribute's font size and indentation information. If we have $style(a_i) = style(a_j)$, then (a_i, a_j) is not likely to be a true *ParentChild* pair. For example in Figure 1, the *left* attribute at row 23 has the same formatting style as the one at row 21: $style(23) = style(21)$, and so we do not consider this a *ParentChild* candidate $(21, 23)$.

2. We prioritize the similar ParentChild *pairs which are geometrically close to each other in the spreadsheet.* For example, as shown in Figure 1, we create two *ParentChild* pair candidates $(32, 33)$ and $(26, 33)$, and the two candidates are considered similar because 32 (Black, total) and 26 (White, total) are both talking about race. As a result, we only keep the pair $(32, 33)$ with attributes geometrically closer and remove the *ParentChild* candidate $(26, 33)$.

Algorithm 1 EnforcedTreeInference

Input: The attributes in an attribute region $A = \{a_1, ..., a_N\}$
Output: The attribute hierarchy $P = \{p_1, ..., p_M\}$

```
Initiate P
for each parent ∈ A do
    maxprob ← 0
    maxparent ← Root
    for each child ∈ A do
        Create a ParentChild candidate p = (parent, child)
        Compute the probability cprob for L(p) = true
        if cprob > maxprob then
            maxprob ← cprob
            maxparent ← parent
        end if
    end for
    Create a ParentChild pair p' = (maxparent, child)
    P ← P ∪ p'
end for
Break cycles in P
```

For ***top* attributes**, given a *ParentChild* candidate (a_i, a_j) in a *top* attribute region, we believe (a_i, a_j) is not likely to be a true *ParentChild* pair if the row of the child attribute a_j is lower than the row of the parent attribute a_i. Of course, more heuristics could be found to further filter the unlikely *ParentChild* candidates, but in practice, this simple rule is already effective enough to greatly shrink the size of the created *ParentChild* candidates.

Now we formally describe the classification process: given a set of *ParentChild* pair candidates $P = \{(a_i, a_j)\}$ in an attribute region, the classifier assigns each $p = (a_i, a_j) \in P$ with a label from $L = \{true, false\}$ s.t. the predicted *ParentChild* pairs $\{(a_i, a_j) \mid L(a_i, a_j) = true\}$ construct an attribute hierarchy in the given attribute region. If the classification is entirely correct, the produced *ParentChild* pairs represent a tree. However any error in the classification might yield an inaccurate result.

Our *classification-based* method exploits a variety of signals in a spreadsheet to extract *attribute hierarchies*. For *left* and *top* attributes, we use a different set of features. For *left* attributes, the classifier utilizes two types of features: *unary features* and *binary features*; for *top* attributes, we mainly utilize *layout features*. The features we used for both *left* and *top* regions are discussed in detail in Appendix B.

3.2.2 Algorithm 2: Enforced-tree Classification

One weakness of the *classification-based* approach is that it does not guarantee that the emitted *ParentChild* pairs construct a tree. Thus, our second proposed technique is **enforced-tree classification**, which embeds simple heuristics into the classification results to ensure the produced pairs construct a strict hierarchical tree. Of course, in a tree structure, each node has only one parent (except the root). Thus for each attribute, we select the one with the maximal probability as its parent attribute. We obtain the probability associated with each *ParentChild* pair during the classification. This *one-parent* constraint does not guarantee that the output will be a tree, as cycles may still exist in the results. We then iteratively break cycles by deleting the pairs with the minimal probability until there are no more cycles in the output. We assume one's parent attribute is Root by default. Therefore, the two steps, the *one-parent* constraint and the *breaking-cycles*, enforce the classification results to generate a strict tree. The classifier uses the same

Algorithm 2 TupleBuilder

Input: The left hierarchy H_l, the top hierarchy H_t, the set of values in the value region $V = \{v\}$
Output: The relational tuples $T = \{t\}$

```
1: Initiate T
2: for each v ∈ V do
3:     Initiate t
4:     Get annotating attributes for v from H_l as {a_l}
5:     Get annotating attributes for v from H_t as {a_t}
6:     t ← v ∪ {a_l} ∪ {a_t}
7:     T ← T ∪ t
8: end for
```

set of features as the **classification** method, and the details of the algorithm are shown in Algorithm 1.

3.3 Tuple Builder

The **tuple builder** is straightforward, as long as the previous steps are accurate. We generate a relational tuple for each value in the *value region*, annotating each one with relevant attributes from the *attribute hierarchies*. For example, Figure 1 shows the full six-field tuple we want to recover for the highlighted value 13.7. The **tuple builder** is also algorithmically straightforward. It processes the extracted *attribute hierarchies* and the *value region* to generate a series of relational tuples. As described in Algorithm 2, for each value v, we find the its annotating attributes along the path to the root in the *attribute hierarchies* for both *left* and *top* attribute regions. The **tuple builder** relies entirely on the **frame finder** and **hierarchy extractor** for correctness.

4. EXPERIMENTS

We can now quantify the performance of the system by evaluating its individual components. In particular, we present the performance of the **frame finder** and the **hierarchy extractor**. We do not directly evaluate the **tuple builder** because it entirely relies on the correctness of the **hierarchy extractor**, and it will yield the ideal results as long as it receives accurate hierarchies.

In the following experiments, we use 100 random hierarchical *data frame* spreadsheets (*data frame* spreadsheets with hierarchical *top* attributes or hierarchical *left* attributes). We obtained this dataset by randomly sampling spreadsheets from WEB and only keeping those with a hierarchical *data frame*. For *top*, the average hierarchy depth of the dataset is 2.14, with a maximum depth of 5; for *left*, the average hierarchy depth is 2.61, with a maximum depth of 9. The training and testing procedures for both the **row labeler** and the **hierarchy extractor** are as follows: we randomly split the dataset into equal-sized training and testing sets, repeating this process 10 times. Then we report the average `Precision, Recall`, and `F1` measure for each class. We also present the `mean` and standard deviation (`std`) for *errors per sheet*, which is defined as follows:

Definition *4.* (Errors per sheet) A classification task produces two types of errors, false positive (fp) and false negative (fn), on N spreadsheets. We define *errors per sheet*:

$$Errors_{sheet} = (fp + fn)/N$$

For the experiment setup, we used several open-source packages: for **frame finder**, our CRFs were implemented on CRF++ [7]; for **hierarchy extractor**, we used the SVM library from the LIBSVM package [4] and the Weka package [10] for the logistic regression and naive Bayes method.

		Performance			Errors	
		Precision	Recall	F1	Mean	Std
title	Base-CRF	0.561	0.605	0.582	3.534	4.532
	Full-CRF	0.818	0.734	0.774	0.872	0.150
header	Base-CRF	0.624	0.606	0.615	2.348	0.621
	Full-CRF	0.812	0.740	0.774	1.316	0.343
data	Base-CRF	0.995	0.970	0.982	6.526	5.239
	Full-CRF	0.995	0.993	0.994	1.528	0.330
footnote	Base-CRF	0.550	0.786	0.647	4.208	3.414
	Full-CRF	0.843	0.826	0.834	1.208	0.223

Table 1: Performance of the row labeler.

		Performance			Errors	
		Precision	Recall	F1	Mean	Std
top	SVM	0.921	0.918	0.919	1.834	0.398
	EN-SVM	0.920	0.920	0.920	1.829	0.395
left	SVM	0.852	0.700	0.769	19.554	5.107
	EN-SVM	0.811	0.811	0.811	16.154	4.332

Table 2: Performance of the hierarchy extractor.

4.1 Frame Finder

We now evaluate the performance of the **frame finder** described in Section 3.1 by evaluating the **row labeler**. Our experiment spreadsheets contain 27,531 non-empty rows that are correctly assigned with semantic labels by a human expert. In [16], CRFs were used to label the lines of tables in plain-text government statistical reports using *textual* features. Our **row labeler** also uses CRFs but incorporates richer features: both *textual* and *layout* features. The *layout* features, such as bold font and alignment, are hard to obtain from plain text but accessible in spreadsheets through the Python xlrd library. Therefore, we compare two CRFs with different sets of features: Base-CRF for *textual* features and Full-CRF for *textual + layout* features.

As shown in Table 1, Full-CRF performs significantly better than Base-CRF on all the metrics, including precision, recall, and errors per sheet. According to precision and recall, both methods do a decent job of predicting all the labels, but they show a large difference in the number of errors, especially for the label data. For data, the two methods have very close precision and recall, but Base-CRF produces many more errors than Full-CRF. This occurs because of the huge number of data rows in the dataset, and a small difference in the F1 measure will make a big difference to the absolute number of errors. Overall, Table 1 shows that the Full-CRF method is superior to the baseline Base-CRF method and can work effectively as a part of the system. Full-CRF predicts each category fairly accurately, and the number of errors produced by Full-CRF is tolerable for all the labels, with about one error per sheet.

4.2 Hierarchy Extractor

We evaluate the performance of the *hierarchy extractor* discussed in Section 3.2 by measuring its accuracy in retrieving correct *ParentChild* pairs from a spreadsheet. The hierarchical metadata in spreadsheets is unique, and we are not aware of any previous method to automatically extract such hierarchical metadata. Therefore, we create a baseline approach Human to ask a user to manually enumerate all the *ParentChild* pairs in an attribute region, which is exactly what the hierarchy extractor infers automatically. We first evaluate the performance of our two approaches:

		Repairs
top	Human	22.469
	SVM	1.834
	EN-SVM	1.829
left	Human	58.598
	SVM	19.554
	EN-SVM	16.154

Table 3: User repair # for the hierarchy extractor.

SVM for *classification* and EN-SVM for *enforced-tree classification*. We then compare our two methods with the baseline method, Human, on the metric *user repair #*.

DEFINITION *5.* (User Repair #) We assume that a user reviews every *ParentChild* pair candidate with an assignment, true or false, in an attribute region. **User repair #** is the number of assignments the user must modify in order to obtain the correct hierarchy.

For SVM and EN-SVM, *user repair #* equals *errors per sheet* in a given attribute region. For Human, *user repair #* is the total number of true *ParentChild* pairs in an attribute region (we assume that all the *ParentChild* pair candidates are labeled false by default).

Table 2 shows the performance of our two methods. As seen in Table 2, EN-SVM performs the best, especially on *left*. Note that for *left*, EN-SVM has a higher recall than SVM but a slightly lower precision. The reason is that given an attribute, all the *ParentChild* candidates containing its parent attribute may be labeled false by the classifier, and then the attribute will not have any parent attribute. But EN-SVM is able to recover its parent attribute by selecting the most probable *ParentChild* pair from the false group. Table 3 presents the *user repair #* for the three methods and shows that both SVM and EN-SVM require a much smaller number of user repairs than Human. We also tried logistic regression and naive Bayes for classification. Overall, SVM is comparable to logistic regression but performs the best of the three. Therefore, we conclude that our method EN-SVM is superior to the baseline Human, as EN-SVM predicts the *ParentChild* fairly accurately and it beats Human on *user repair #*.

One limitation of our system lies in the fact that the absolute number of required repairs on *left* is not trivial. According to Table 3, the number of repairs on *top* is almost negligible, but not on *left*. We will try to reduce the user burden even further in future work.

5. RELATED WORK

There are three main types of existing approaches to transform spreadsheet data into databases. The first is a *schema-based* approach. For spreadsheet data already in a relational format, traditional schema mapping systems, such as Clio [9] and Clip [17], could potentially be used to convert the spreadsheet data into databases by specifying the source and target attribute mapping. The second is a *rule-based* approach [11], which requires explicit user-provided conversion rules. Finally, a *visualization-based* approach provides users with an interactive visualization interface to convert or manage the underlying data [12, 18, 19, 21]. However, these existing approaches all suffer from two common drawbacks: (1) it is challenging to handle *hierarchical* spreadsheets; (2) the transformation process cannot be accomplished automatically: most of the approaches require users to learn a new language or predefined operators to describe the transformation rules. In fact, the work of Abraham and Erwig [1] and Cunha et al. [8] are the most similar to ours. The former attempts to infer spreadsheet metadata, but they do not address the *hierarchical* structures in spreadsheets. The latter tries to convert spreadsheets into relational databases, but their primary technical focus is to address the lack of data normalization in spreadsheets that use very conventional layouts. In addition, the approach does not address *hierarchical* spreadsheets.

There is other research on spreadsheets attempting to build *database-like operators on a spreadsheet-style interface* [14, 22, 23, 25], but these systems cannot be directly used to manage the large number of spreadsheets that already exist on the Web. The QueryByExcel project [23, 24, 25] uses a spreadsheet as a front end of the relational database. It translates Excel formulas using an extension of SQL relational operations and performs on RDBMS tables. Liu et al. [14] implemented an extended set of database functions operating on spreadsheets, and the operations are executed by a classic database engine in the background. Tyszkiewicz [22] also attempted to combine SQL with spreadsheets but implemented the functionality inside spreadsheets instead of using an additional database engine.

6. CONCLUSIONS AND FUTURE WORK

We have described a domain-independent spreadsheet extraction system for converting spreadsheet data into relational tuples. Our system consists of three components that detect the structure of a spreadsheet, extract hierarchical metadata, and generate relational tuples. Our experiments show that our proposed methods are superior to the baseline approaches and can work effectively as a part of the whole framework. As a result, the system can help bring relational-style data management techniques to the mass of data currently locked in spreadsheets. The extraction procedure also emits various semantic byproducts in the form of hierarchies that could be useful in a range of other applications, such as schema integration and design tools.

One area of future work lies in how to best incorporate manual repairs to further reduce users' burden. Another lies in the integration application itself; extraction is a necessary first step, and we have not yet rigorously addressed search ranking quality and join findability in the end user tool.

7. ACKNOWLEDGMENTS

This project is supported by National Science Foundation grants IIS-1054913 and IIS-1064606, as well as gifts from Dow Chemical, Yahoo!, and Google. Special thanks to Robert Vogel for advice and assistance.

8. REFERENCES

[1] R. Abraham and M. Erwig. Ucheck: A spreadsheet type checker for end users. *J. Vis. Lang. Comput.*, 18(1):71–95, 2007.

[2] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A type system for statically detecting spreadsheet errors. In *ASE*, pages 174–183, 2003.

[3] P. Blattner and L. Stewart. Microsoft excel 2000 functions in practice. *QUE*, 1999.

[4] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[5] Z. Chen, M. Cafarella, J. Chen, D. Prevo, and J. Zhuang. Senbazuru: A prototype spreadsheet database management system. *VLDB Demo*, 2013.

[6] 2009. ClueWeb09, http://lemurproject.org/clueweb09.php.
[7] 2009. http://crfpp.sourceforge.net.
[8] J. Cunha, J. Saraiva, and J. Visser. From spreadsheets to relational databases and back. In *PEPM*, pages 179–188, 2009.
[9] A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: Schema mapping reloaded. In *VLDB*, pages 67–78, 2006.
[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.
[11] V. Hung, B. Benatallah, and R. Saint-Paul. Spreadsheet-based complex data transformation. In *CIKM*, pages 1749–1754, 2011.
[12] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. Zellweger. Fluid visualization for spreadsheet structures. In *VL*, pages 118–125, 1998.
[13] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, pages 282–289, 2001.
[14] B. Liu and H. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, pages 417–428, 2009.
[15] M. Nahm and J. Zhang. Operationalization of the ufurt methodology for usability analysis in the clinical research data management domain. *Journal of Biomedical Informatics*, 42(2):327–333, 2009.
[16] D. Pinto, A. McCallum, X. Wei, and W. B. Croft. Table extraction using conditional random fields. In *SIGIR*, pages 235–242, 2003.
[17] A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández. Clip: a visual language for explicit schema mappings. In *ICDE*, pages 30–39, 2008.
[18] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.
[19] R. Rao and S. K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *CHI*, pages 318–322, 1994.
[20] J. Simon. Excel 2000 in a nutshell. *O'Relly Media*, 2000.
[21] M. Spenke, C. Beilken, and T. Berlage. Focus: The interactive table for product comparison and selection. In *UIST*, pages 41–50, 1996.
[22] J. Tyszkiewicz. Spreadsheet as a relational database engine. In *SIGMOD*, pages 195–206, 2010.
[23] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in rdbms for olap. In *SIGMOD Conference*, pages 52–63, 2003.
[24] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Business modelling using sql spreadsheets. In *VLDB*, pages 1117–1120, 2003.
[25] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold. Query by excel. In *VLDB*, pages 1204–1215, 2005.

APPENDIX

A. FRAME FINDER FEATURES

The features we used in the **frame finder** include *layout* and *textual* features. Each of the features is a binary function, taking in a given row in a spreadsheet as the input and emitting a 0/1 Boolean value as the output. The features attempt to test whether the properties of a row are an indication of a certain category in {title, header, data, footnote}. The features are listed in Table 4.

B. HIERARCHY EXTRACTOR FEATURES

Left Attributes – For *left* attributes, given a *ParentChild* pair candidate (a_i, a_j), we employ a set of features to characterize its properties, thus determining whether it is a true *ParentChild* pair. The testing features include *unary features* and *binary features*, as shown in Table 5. The unary features apply on each of the child and parent attributes, and the binary features apply on the attribute pair.

Top Attributes – For *top* attributes, given a *ParentChild* pair candidate (a_i, a_j), we utilize a set of layout features to characterize the properties of the attribute pair, thus determining whether it is a true *ParentChild* pair. The features we used are shown in Table 6.

	Layout Features
1	Has a bold font cell
2	Has a cell reaching the left bound
3	Has a cell reaching the right bound
4	Has a cell with indentations
5	Has a center-aligned cell
6	Has a left-aligned cell
7	Has a merged cell
8	Has only one column
	Textual Features
1	Contains colon
2	Contains punctuations
3	Has a cell with with a word count > 40
4	Numeric cells within year range ratio > 0.6
5	Row is blank
6	With all words in lowercases
7	With all words capitalized
8	With all words starting with capitals
9	With numeric cells ratio > 0.6
10	With words starting with "table"

Table 4: Extraction features for the frame finder.

	Unary Extraction Features
1	Attribute has underline
2	Attribute contains keywords like "total"
3	Attribute contains colon
4	Attribute is bold
5	Attribute is center aligned
6	Attribute is italic
7	Attribute is numeric
8	Attribute letters are all capitalized
9	Is the first attribute
10	Is the last attribute
	Binary Extraction Features
1	Attribute pair is adjacent
2	Attribute pair's indentation is equal
3	Attribute pair's style is adjacent in the region
4	Child's font size is smaller than parent's
5	Child's indentation is greater than parent's
6	Child's row index is greater than parent's
7	Child's style is the same as the first attribute
8	Has blank cells in the middle
9	Has middle cell with indentation between the pair's
10	Has middle cell with indentation larger than the pair's
11	Has middle cell with indentation less than the pair's
12	Has middle cell with style different from the pair's
13	Has middle cell containing keywords like "total"
14	Parent is the root

Table 5: Extraction features for the hierarchy extractor on left attributes.

	Layout Extraction Features
1	Child has no cell right above
2	Child is at the uppermost header row
3	Has a cell in the middle
4	Parent cell covers child's column
5	Parent is on the left of child
6	Parent is on the right of child
7	Parent is right above child
8	Parent is the root

Table 6: Extraction features for the hierarchy extractor on top attributes.