The VLDB Journal (2011) 20:209–226
DOI 10.1007/s00778-011-0223-0

SPECIAL ISSUE PAPER

Harvesting relational tables from lists on the web

Hazem Elmeleegy · Jayant Madhavan · Alon Halevy

Received: 11 August 2010 / Revised: 20 January 2011 / Accepted: 10 February 2011 / Published online: 2 March 2011

Abstract A large number of web pages contain data structured in the form of "lists". Many such lists can be further split into multi-column tables, which can then be used in more semantically meaningful tasks. However, harvesting relational tables from such lists can be a challenging task. The lists are manually generated and hence need not have well-defined templates—they have inconsistent delimiters (if any) and often have missing information. We propose a novel technique for extracting tables from lists. The technique is domain independent and operates in a fully unsupervised manner. We first use multiple sources of information to split individual lines into multiple fields and then, compare the splits *across* multiple lines to identify and fix incorrect splits and bad alignments. In particular, we exploit a corpus of HTML tables, also extracted from the web, to identify likely fields and good alignments. For each extracted table, we compute an extraction score that reflects our confidence in the table's quality. We conducted an extensive experimental study using both real web lists and lists derived from tables on the web. The experiments demonstrate the ability of our technique to extract tables with high accuracy. In addition, we applied our technique on a large sample of about 100,000 lists crawled from the web. The analysis of the extracted tables has led us to believe that there are likely to be tens of millions of useful and query-able relational tables extractable from lists on the web.

H. Elmeleegy (✉)
AT&T Labs - Research, Florham Park, NJ, USA
e-mail: hazem@research.att.com

J. Madhavan · A. Halevy
Google Inc., Mountain View, CA, USA
e-mail: jayant@google.com

A. Halevy
e-mail: halevy@google.com

Keywords Structured data on the web · Table extraction · List segmentation · WebTables · ListExtract

1 Introduction

The World Wide Web is a large, but as yet under-utilized, source of structured data. Consequently, managing structured data on the web has recently become the focus of many research efforts (e.g. [1,8,13,23,25,31]). Solutions have been proposed to find, extract, and integrate structured data. Building web scale structured data stores, and exposing them offers many advantages, e.g., more sophisticated querying of web data and performing *table search* to bootstrap other data management tasks. In addition, the analysis of large amounts of structured data on the web has enabled features such as schema auto-complete and synonymy discovery [8].

In recent work, Cafarella et al. [8] have shown that HTML tables are a particularly rich source of structured data. Their results indicate that there are more than 150 million HTML tables containing relational data on the web. In this paper, we consider a complementary, and equally plentiful, source of relational data—lists on the web.

The key challenge concerning HTML lists is that there is no clear notion of columns or cells, as is the case with tables. Each line in the list is largely unstructured text. Delimiters are typically very inconsistent, if at all existent, and hence cannot be relied upon to split each line into the correct fields. Moreover, information might be missing on some lines, and hence, not all lines can be split into the same number of fields.

For example, consider the cartoon listing in Fig. 1. Each cartoon has a serial number, the cartoon name, the production company, and the production year. Some cartoons are missing information, such as "6. Gertie The Dinosaur", where

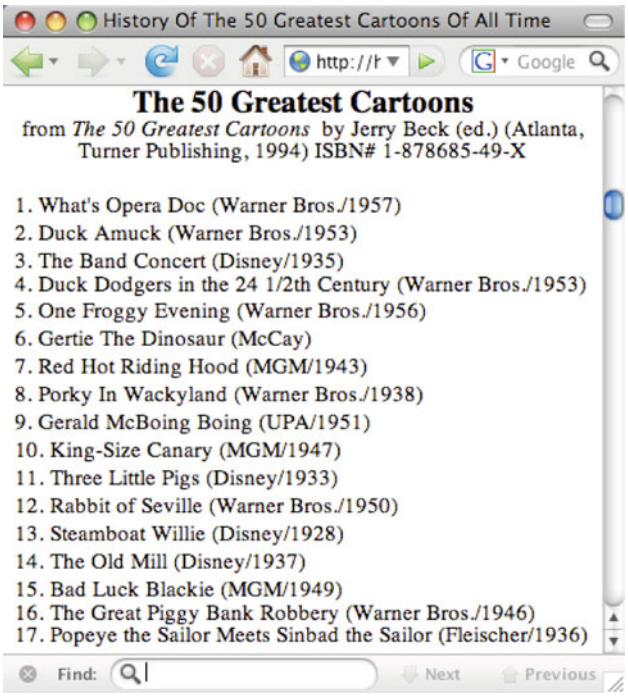

Fig. 1 List of the 50 greatest cartoons: an example of Web lists that contain structured data that can be extracted into relational tables

the production year is not specified. While it might seem that the list is uniformly delimited, a closer examination reveals several inconsistencies. First, a period is used both to separate the serial number from the cartoon name and to terminate abbreviations (e.g., the production company "Warner Bros."). Second, while the delimiter between the production company and year is typically a single slash ("/"), when abbreviations are used, the delimiter sequence is a period and a slash ("./"). Third, the slash also appears in the name of one of the cartoons ("Duck Dodgers in the 24 1/2th Century"). Finally, for cartoon "6. Gertie The Dinosaur", the slash delimiter is absent (along with the production year). These inconsistencies, while fairly easy for a human observer to detect, can be very confusing for an automated system.

The above example also demonstrates that the problem of segmenting lists is different from the traditional information extraction problem of wrapper generation [1,2,10,11,13,16,19,20,22,29–31]. The typical assumption in wrapper generation is that web pages (or parts of web pages) are automatically generated for each record in an underlying table using an HTML template. Thus, the layout of each record can be assumed to be *consistent* (with different data fields being separated by HTML tags), and hence, they can be inferred from multiple examples.

This paper proposes a technique for extracting tables from lists, ListExtract, which addresses the above challenges. Given an input list, ListExtract searches for the best possible table that the list can be segmented into. ListExtract is designed to be completely domain independent and hence, apply to any list found on the web.

The over-arching idea underlying ListExtract is that finding the best table involves interleaving local decisions *within* each line in the list and table-oriented decisions *across* lines of the list. Within the lines, ListExtract uses some typical signals such as the data types, syntax, and delimiters. ListExtract also uses two new sources: (1) a large-scale language model (e.g., like in [6]) that records word co-occurrence scores and (2) a large corpus of automatically extracted HTML tables [8]. The language model is used to identify candidate phrases that should not be split within a line, and the table corpus identifies phrases that occur elsewhere in table cells.

When looking across lines of the list, ListExtract identifies splitting errors by considering the cohesion of values across the column of the resulting table. Here too, the table corpus is helpful because it identifies values that have appeared in the same column in other tables. In addition, when a splitting error is found by ListExtract, it realizes that the error must affect a streak of values occurring to the left or to the right of the value. As we describe, ListExtract operates in several phases that interleave these two types of decisions.

In summary, we make the following contributions:

1. We present a novel technique for extracting tables from lists that is both domain independent and is completely unsupervised. These qualities are essential in making the technique applicable on a web scale.
2. We describe how language models, and a corpus of tables can be used to identify segments in lines that are well suited to be cell values in tables. We also show how the table corpus is instrumental in aligning segments across different lines in a list.
3. We present the results of an extensive experimental study based on real web lists, in addition to synthetic lists derived from HTML tables. The experiments demonstrate the effectiveness of our technique and the impact of its various components. We also show that existing information extraction techniques cannot be applied to our problem effectively.
4. We take a first step toward estimating the number of high-quality tables that can be extracted from lists on the web. From a sample of 100,000 web pages selected at random, we show that ListExtract can extract between 1,400 and 9,700 tables with more than one column from HTML lists, depending on the required quality threshold. The distribution of the number of columns in the extracted tables can be found in [15].

We note that to complete relational table extraction, we also need to assign column headers to the columns of output

tables. However, we only focus on the task of splitting the list's lines into the table's columns. Finding meaningful column headers is an area of future work, and some of the techniques in [9,28] can be directly applied.

This paper extends a previous conference publication [15] in the following ways: (1) we discuss the implementation challenges of ListExtract (Sect. 6), (2) we describe application scenarios for ListExtract (Sect. 8), (3) we provide additional details for the alignment phase (Algorithm 4.3) and the effects of selecting our line splitting algorithm (Sect. 7.8). We also updated our discussion of recent related work.

The rest of the paper is organized as follows. Section 2 presents our problem formulation and an overview of our approach. Sections 3, 4, and 5 describe the three phases of our algorithm-splitting, alignment, and refinement. Section 6 outlines our implementation of ListExtract while Sect. 7 presents an experimental evaluation. Section 8 discusses application sof ListExtract, Sect. 9 discusses related work, and Sect. 10 concludes.

2 Problem statement and overview

We begin by stating the problem we address and giving an overview of our solution.

2.1 Terminology and problem statement

Consider a list L of n lines, where the ith line l_i consists of m_i words $\langle w_{i1}, w_{i2}, \ldots, w_{im_i}\rangle$. Our goal is to extract a table T that contains n rows and some number of columns, say k.

We refer to each line in the list (that becomes a row in the table) as a *record* and each cell value as a *field*. Thus, the ith record in T contains the k fields $\langle f_{i1}, f_{i2}, \ldots, f_{ik}\rangle$. The field f_{ij} consists of m_{ij} successive words $\langle w_{ip_{ij}}, w_{i(p_{ij}+1)}, \ldots, w_{i(p_{ij}+m_{ij}-1)}\rangle$, where p_{ij} is the position of the first word in f_{ij}. We use the term *field candidate* to refer to a sequence of words that is being considered as a potential field.

In this work, we only consider records that are formed by a non-overlapping and complete splitting of a line in the list, i.e., each field is assumed to be disjoint and all words are assigned to some field.

Given a list L, our goal is to extract a table T that is the most likely representation of the underlying relational data. It is important to note that there is not necessarily a *single* right answer to the table extraction problem. Solutions may differ on how many columns they identify and how they deal with irregularities in the data. Ultimately, solution quality is subjective.

2.2 Algorithm overview

Our ListExtract technique executes as a sequence of operations over the input list (see Fig. 2). The underlying operations can be grouped into three main phases: an *independent splitting* phase, an *alignment* phase, and a final *refinement* phase. We use two scoring functions to decide where to split individual records. We use a *field quality score*, $FQ(f)$, to measure the quality of an individual field candidate f, and a *field-to-field consistency score*, $F2FC(f_1, f_2)$, to measure the likelihood of two field candidates f_1 and f_2 being in the same column. Both the scores take into consideration multiple sources of information.

Figure 3 shows the intermediate results of applying our technique on the first 17 rows in the cartoons list in Fig. 1.

Phase 1 (Splitting): Each line in the input list is split into a multi-field record. The splitting is performed independently hence the obtained records do not necessarily have the same number of fields.

As shown in Fig. 3a, after the independent splitting phase, 13 out of the 17 lines are correctly split into records of four fields representing the sequence number, cartoon's name, production company, and production year. Line 6 is also split correctly, though into a record of three fields only, as it is missing the year information. However, lines 4, 15, and 17 (highlighted) were incorrectly split. Interestingly, the cartoon's name in line 17 had two very common substrings ("Popeye the Sailor" and "Sinbad the Sailor"), which lead the splitting algorithm to assign high scores to them and thus, treat each of them as a separate field.

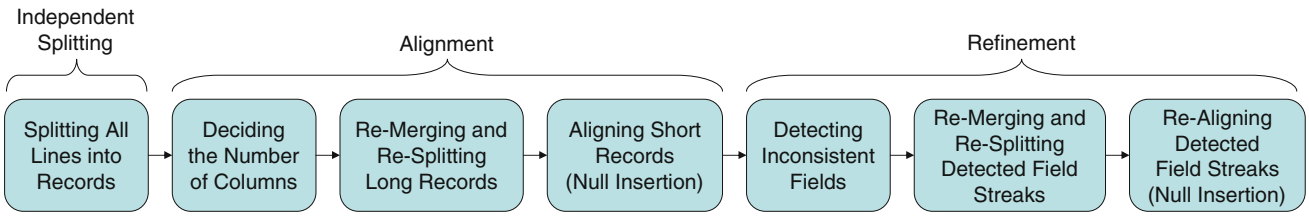

Fig. 2 ListExtract proceeds as a sequence of operations that can grouped into the independent splitting, the alignment, and the refinement phases

Fig. 3 Applying the LISTEXTRACT technique on the cartoons list in Fig. 1 **(a)** After independent splitting phase. **(b)** After re-splitting records given the number of columns. **(c)** After alignment phase (initial table T_I). **(d)** After refinement phase (final table T)

1 || What's Opera Doc || Warner Bros || 1957
2 || Duck Amuck || Warner Bros || 1953
3 || The Band Concert || Disney || 1935
4. Duck Dodgers in the 24 1/2th Century (Warner Bros || 1953
5 || One Froggy Evening || Warner Bros || 1956
6 || Gertie The Dinosaur || McCay
7 || Red Hot Riding Hood || MGM || 1943
8 || Porky In Wackyland || Warner Bros || 1938
9 || Gerald McBoing Boing || UPA || 1951
10 || King-Size Canary || MGM || 1947
11 || Three Little Pigs || Disney || 1933
12 || Rabbit of Seville || Warner Bros || 1950
13 || Steamboat Willie || Disney || 1928
14 || The Old Mill || Disney || 1937
15 || Bad Luck Blackie (MGM || 1949
16 || The Great Piggy Bank Robbery || Warner Bros || 1946
17 || Popeye the Sailor || Meets || Sinbad the Sailor || Fleischer || 1936

(a)

1 || What's Opera Doc || Warner Bros || 1957
2 || Duck Amuck || Warner Bros || 1953
3 || The Band Concert || Disney || 1935
4. Duck Dodgers in the 24 1/2th Century (Warner Bros || 1953
5 || One Froggy Evening || Warner Bros || 1956
6 || Gertie The Dinosaur || McCay
7 || Red Hot Riding Hood || MGM || 1943
8 || Porky In Wackyland || Warner Bros || 1938
9 || Gerald McBoing Boing || UPA || 1951
10 || King-Size Canary || MGM || 1947
11 || Three Little Pigs || Disney || 1933
12 || Rabbit of Seville || Warner Bros || 1950
13 || Steamboat Willie || Disney || 1928
14 || The Old Mill || Disney || 1937
15 || Bad Luck Blackie (MGM || 1949
16 || The Great Piggy Bank Robbery || Warner Bros || 1946
17. Popeye the Sailor Meets || Sinbad the Sailor || Fleischer || 1936

(b)

1	What's Opera Doc	Warner Bros	1957
2	Duck Amuck	Warner Bros	1953
3	The Band Concert	Disney	1935
		4. Duck Dodgers in the 24 1/2th Century (Warner Bros)	1953
5	One Froggy Evening	Warner Bros	1956
6	Gertie The Dinosaur	McCay	
7	Red Hot Riding Hood	MGM	1943
8	Porky In Wackyland	Warner Bros	1938
9	Gerald McBoing Boing	UPA	1951
10	King-Size Canary	MGM	1947
11	Three Little Pigs	Disney	1933
12	Rabbit of Seville	Warner Bros	1950
13	Steamboat Willie	Disney	1928
14	The Old Mill	Disney	1937
15	Bad Luck Blackie (MGM)		1949
16	The Great Piggy Bank Robbery	Warner Bros	1946
17. Popeye the Sailor Meets	Sinbad the Sailor	Fleischer	1936

(c)

1	What's Opera Doc	Warner Bros	1957
2	Duck Amuck	Warner Bros	1953
3	The Band Concert	Disney	1935
4	Duck Dodgers in the 24 1/2th Century	Warner Bros	1953
5	One Froggy Evening	Warner Bros	1956
6	Gertie The Dinosaur	McCay	
7	Red Hot Riding Hood	MGM	1943
8	Porky In Wackyland	Warner Bros	1938
9	Gerald McBoing Boing	UPA	1951
10	King-Size Canary	MGM	1947
11	Three Little Pigs	Disney	1933
12	Rabbit of Seville	Warner Bros	1950
13	Steamboat Willie	Disney	1928
14	The Old Mill	Disney	1937
15	Bad Luck Blackie	MGM	1949
16	The Great Piggy Bank Robbery	Warner Bros	1946
17	Popeye the Sailor	Meets Sinbad the Sailor (Fleischer)	1936

(d)

Phase 2 (Alignment): An initial candidate table (T_I) is constructed by first determining a single number of likely columns in the output table. Records with too many fields are re-merged and re-split to make sure that they fit into the output table. Records with too few fields are expanded by inserting fields with a *null* value. The fields are aligned into columns so as to ensure consistency among fields in the same column. Since the splitting decisions in the first phase were made independently for each record, the T_I can still have some incorrect fields.

In our example, the number of columns in the output table is set at four (the most common number of fields across all records). Since record 17 has more than four fields, it is merged and re-split to force it to have no more than four fields. As shown in Fig. 3b, it re-splits into exactly four fields. However, the split between the first and second fields was inaccurate, again because the common substring "Sinbad the Sailor" is recognized as a separate field candidate.

In the initial table T_I (Fig. 3c), note that a *null* field was correctly placed for the missing year in record 6. Moreover, the fields, which were correctly split in records 4 and 15, were also correctly positioned in T_I. In particular, the production year of record 4, and the serial number and production year for record 15 were placed in their correct columns.

Phase 3 (Refinement): The field assignments in T_I are analyzed to detect and fix those that are likely to be incorrect. This is done by marking fields that seem inconsistent with other fields in the same column. We observe that bad splitting decisions do not occur in isolation, but are likely to affect one or more adjacent fields. Hence, we detect *streaks*, i.e., sequence of fields in a single record, of such inconsistent fields that are then merged together and re-split.

Unlike the independent splitting in the first phase, splitting a re-merged streak of fields during refinement takes into account other fields occurring in the columns of T_I spanned by the streak. This enables us to identify higher-quality fields, which are consistent with their respective columns, and hence ultimately generate a higher-quality table, T.

In our example, in the result of the refinement phase, all the highlighted cells in Fig. 3c are correctly detected as fields that are inconsistent with their columns, by virtue of their low consistency scores with the other fields in their columns. The corresponding streaks are merged and re-split. This results in the correction of most of the fields, except for the cartoon name and production company in record 17. This is again attributed to the high popularity of the substring "Popeye the Sailor", which was also a good match for the cartoon name column.

We now look at each phase in detail.

3 Independent splitting phase

The first phase of ListExtract considers each line in a list independently and splits it into a record with multiple fields.

In order to measure the quality of a particular candidate field (as a cell value), we use the *field quality score* (FQ) for each candidate. We describe how we compute FQ shortly. In principle, every subsequence of words in a line l is a field candidate. For a line with m words, there are $\binom{m+1}{2}$ possible field candidates (the number of choices for the start and end words of the sequence).

We considered three alternate methods to select the best split for a line given the FQ scores of its candidate fields. Recall that we only consider splits that result in non-overlapping fields that together include all the words in the line.

The first method determines the best split to be the one that maximizes the *sum* of the FQ scores of the selected fields, while the second method maximizes the *average* FQ score. The third is a *greedy* method that at each step selects the field candidate with the highest score, while eliminating those that overlap with the selected field.

We found that maximizing the sum of FQ scores results in aggressive splitting, i.e., each line is split into too many fields. This is because increasing the number of fields typically leads to a larger sum. Maximizing the average FQ score on the other hand avoids such aggressive splitting, but is computationally expensive. While the sum can be maximized using a standard $O(m^2)$ dynamic programming segmentation algorithm [4], the average cannot be. Unlike sum, the average is not a *decomposable* objective function—in simple terms, "sum" can always be expressed as "sum of sums", while "average" cannot always be expressed as "average of averages". Hence, an exhaustive search is necessary.

In the interest of efficiency, we instead use a greedy method that does not result in aggressive splitting. Our greedy line splitting algorithm, **SplitLine**, is outlined in Algorithm 1 below. We create C_f, a ranked list of all field candidates sorted in descending order of their FQ scores. In each iteration of the loop, the candidate with the highest score, f_{top}, is removed from the ranked list and marked as selected (added to the output set r). All candidates that overlap with f_{top} are then removed from C_f to ensure that no two overlapping fields are selected. This process terminates when C_f becomes empty.

Surprisingly, in our experiments, we found that **SplitLine** generally yielded better results even compared to the "average" method. In particular, as reported in Sect. 7, we compared **SplitLine** against "average" implemented using exhaustive search and found it to have an average f-measure improvement of .05 over our data sets. It is likely that **SplitLine** is more re-silient to incorrect field scores that might

Algorithm 1 **SplitLine**(l: line)

```
r = {}
extract all subsequences from l as field candidates.
calculate FQ for each field candidate.
C_f = field candidates sorted in descending order of FQ.
while C_f is not empty do
    remove f_top, the field candidate with the highest FQ in C_f.
    add f_top to r.
    remove field candidates overlapping with f_top from C_f.
end while
return r
```

occur in practice. For example, consider that a correct field candidate f was given a low FQ score due to insufficient information, but there are other high scoring non-overlapping field candidates in the record. **SplitLine** will first pick the other candidates based on better information, thereby delaying the decision on f. At a later stage, f could be the best remaining candidate to fill the gap between previously selected fields. In effect, **SplitLine** delays the selection of f until it gains enough knowledge about its surrounding fields.

3.1 Field quality score

We now describe how we calculate the field quality score, $FQ(f)$, for a given field candidate f. One of the important aspects of ListExtract is that we compute these scores based on *multiple* sources of information. In our discussion below, we assume the candidate field, f, is composed of m words, $\langle w_i, \ldots, w_{i+m-1} \rangle$.

We obtain scores from three sources of information: *type support* (denoted $S_{ts}(f)$), *language model support* (S_{lms}) and *table corpus support*, (S_{tcs}). We assign each information source a weight, $a_{ts}s$, a_{lms}, and a_{tcs}, respectively, and compute $FQ(f)$ as their weighted combination:

$$FQ(f) = a_{ts} \times S_{ts}(f) + a_{lms} \times S_{lms}(f) + a_{tcs} \times S_{tcs}(f) \quad (1)$$

We now explain how each of the individual components is computed.

Type Score (S_{ts}): The type score reflects whether the field candidate can be recognized as a member of a type that commonly occurs in separate table columns. Our implementation currently recognizes numeric values, date-time values, currency values, URLs, emails, phone numbers, and zip codes. Type recognition is performed by matching f against regular expressions, which capture most of the possible instances of the considered types. We set $S_{ts}(f)$ to 1 if the type of f is recognized and to 0 otherwise.

Language model score (S_{lms}): A language model records the probability of occurrences of sequences of words. The probabilities are computed from the analysis of a large corpus of documents in that language, e.g., web pages resulting from a web crawl. Specifically, if $w_1, \ldots, w_i$ is a sequence of words, we use the language model to compute the conditional probability of $Pr(w_i|w_1, \ldots, w_{i-1})$, i.e., the probability that the word w_i follows the sequence $w_1, \ldots, w_{i-1}$.

Intuitively, we want the sequence of words *within* the cell to have a high probability and the sequence of words that *span* cells to have a low probability. We capture these intuitions with the internal cohesiveness score and the external in-cohesiveness score:

- *internal cohesiveness score*, S_{ic}, measures how likely a sequence of words is a single cell value. Specifically, it computes the average conditional probability of each word given the words before it.

$$S_{ic}(f) = \frac{\sum_{h=1}^{m-1} Pr(w_{i+h}|w_i, \ldots, w_{i+h-1})}{m-1} \quad (2)$$

- *external in-cohesiveness score*, S_{ei}, computes the inverse of the average probability of the boundaries of the field: $Pr(w_i|w_{i-1})$ (the probability that the first word in f follows the last word in the earlier field), and $Pr(w_{i+h+1}|w_{i+h})$, (the probability of the first word in the next field following the last word in f).

$$S_{ei}(f) = \frac{2}{Pr(w_i|w_{i-1}) + Pr(w_{i+h+1}|w_{i+h})} \quad (3)$$

The language model score is the weighted average of the internal cohesiveness and external in-cohesiveness scores:

$$S_{lms}(f) = a_{ic} \times S_{ic}(f) + a_{ei} \times S_{ei}(f) \quad (4)$$

where a_{ic} and a_{ei} are in the range [0, 1] and $a_{ic} + a_{ei} = 1$.

Table corpus support score (S_{tcs}): The table corpus support score reflects how well f is supported in a corpus of web tables [8]. Let $tc\_support$ is the number of times f occurs as a cell value in the table corpus. We use a simple scheme, where we set S_{tcs} to 1 if $tc\_support$ is greater than some threshold value $min\_tc\_support$ and 0 otherwise. This simple scheme proved to perform quite well (as discussed in Sect. 7).

Before calculating FQ and S_{lms}, we normalize and scale the component scores S_{ts}, S_{ic}, S_{ei}, and S_{tcs}. In order to bias the scores to prefer longer field candidates, the scores are scaled by the number of words in the sequence. In order to ensure that all the scores are between 0 and 1, each score is then divided by the maximum corresponding score achieved across all field candidates in L.

To see the benefit in preferring longer candidates, consider the two candidates: "Theodore Roosevelt" and "Theodore

Roosevelt, Jr.". Although both candidates would be recognized as likely fields, it is clear that the presence of ", Jr." immediately after "Theodore Roosevelt" makes the second candidate the more likely one.

4 Alignment phase

The second phase in ListExtract is alignment. The independently split records from the first phase are put together into an initial table that is then aligned.

Before a table can be assembled, we must address a crucial problem: what are the number of columns in the resulting tables? Since the lines were split independently, it is possible the resulting records have different numbers of fields. We use a simple majority voting scheme to determine the number of columns—suppose the first phase splits the row r_i into k_i fields. We pick the k_i that occurs the most number of times in the list. Observe that we are in essence assuming that a large number of the lines have in fact been split into the correct number of fields. This is a reasonable assumption provided the underlying relational tables that we are trying to extract are not sparse (do not have many null fields). Our experiments indicate that this is in fact the case.

Once the number k has been determined, records with exactly k fields are trivially aligned. However, there might be records with more than k fields and also those with fewer than k fields. In Sect. 4.1, we describe how we address longer records by re-splitting them with constraints on the number of fields. In Sect. 4.2, we consider shorter records and align them by inserting null fields that are appropriately placed between the non-null fields. In Sect. 4.3, we describe how we finally assemble the initial table and describe how we maintain *field summaries* that make the alignment process more efficient. Finally, in Sect. 4.4, we describe the *field-to-field consistency score* that we use during the null field insertion and alignment operations.

4.1 Aligning long records

We re-split the lines with more than k fields such that the new records have at most k fields. The re-split is achieved by the **BoundedSplitLine** algorithm, a modification of the original **SplitLine** algorithm. In addition to the input line, l, **Bounded SplitLine** takes an upper bound, $k_{\max}$, for the number of fields the output record r may contain.

The upper bound, $k_{\max}$, is enforced as follows: Before we include the field candidate f_{top} into r, we first ensure that it does not lead to a violation of the upper bound constraint. For this, we calculate the minimum number of fields that r will have if f_{top} was included in r, which we denote by $min\_fields(r, f_{\text{top}})$. We calculate $min\_fields(r, f_{\text{top}})$ as the sum of the number of fields already included in r (assuming f_{top} was added) and the number of "gaps" remaining in r, i.e., sequences of words in line l that are yet to be covered by r or f_{top}. f_{top} is included in r only if $min\_fields(r, f_{\text{top}})$ does not exceed $k_{\max}$.

Algorithm 2 **BoundedSplitLine**(l: line , k_{max}: upper bound)

```
r = {}
extract all subsequences from l as field candidates.
calculate FQ for each field candidate.
C_f = field candidates sorted in descending order of FQ.
while C_f is not empty do
  remove f_top, the field candidate with the highest FQ in C_f.
  estimate min_fields(r, f_top), i.e., the minimum number of fields
  if f_top were included in r.
  if min_fields(r, f_top) ≤ k_max then
    add f_top to r.
    remove field candidates overlapping with f_top from C_f.
  end if
end while
return r.
```

4.2 Aligning short records

We now have lines with k or fewer fields. Note that the re-splitting described above can, in theory, lead to a record with fewer than k fields. We address the problem of missing information by inserting *null* fields into short records. In order to decide where to insert the nulls, we need to identify the location of the missing information.

Suppose that we already have a partial table that includes records that have exactly k fields (and hence can be trivially aligned). The best alignment for a short record r with the partial table is one in which each non-null field aligns with the column it is most similar to and preserving the relative ordering of the non-null fields.

We use an adapted version of the classic Needleman–Wunsch dynamic programming algorithm, for sequence alignment [26] to align short records against a partial table. Algorithm **AlignShortRecord** describes the steps in aligning a record r with k^- fields $\langle f_1, \ldots, f_{k^-} \rangle$ with a partial table T with columns $\langle c_1, \ldots, c_k \rangle$ $(k > k^-)$.

As with a typical dynamic programming approach, we define a recursive objective function for the cost of the best possible alignment. The dynamic programming proceeds by computing a $k^- \times k$ cost matrix M. Suppose F_i represents the sequence of first i fields in r i.e., $\langle f_1, \ldots, f_i \rangle$, and C_j represents the sequence of first j columns in T, i.e., $\langle c_1, \ldots, c_j \rangle$, then $M[i, j]$ represents the cost of the best alignment of the fields in F_i with the columns in C_j. Let $A[i, j]$ be the alignment corresponding to $M[i, j]$. Note that the cost of the best possible complete alignment is $M[k^-, k]$, and the corresponding alignment is $A[k^-, k]$.

We observe that $A[i, j]$ can be constructed from the best alignments the subsequences of F_i and C_j. In fact, $A[i, j]$ can be constructed recursively, by just considering three possibilities: (1) f_i aligns with c_j and rest of the alignment is the same as $A[i-1, j-1]$, (2) f_i is un-matched and the rest of the alignment is the same as $A[i-1, j]$, and (3) c_j is un-matched and the rest of the alignment is the same as $A[i, j-1]$.

Step 10 in Algorithm 3 specifies the corresponding recursive definition for M. Steps 2–7 address boundary conditions. UnMatched(c_j) is the cost assigned to not matching column c_j with any field, i.e., inserting a null field; UnMatched(f_i) is the cost assigned to not matching field f_i with any column, and Matched(f_i, c_j) is the cost assigned to aligning the f_i with the column c_j.

Algorithm 3 **AlignShortRecord**(r: record with k^- fields, T: partial table with k columns)

1: $M[0, 0] = 0$
2: **for** $i = 1$ to k^- **do**
3: $M[i, 0] = M[i-1, 0] + \text{UnMatched}(f_i)$
4: **end for**
5: **for** $j = 1$ to k **do**
6: $M[0, j] = M[0, j-1] + \text{UnMatched}(c_j)$
7: **end for**
8: **for** $i = 1$ to k^- **do**
9: **for** $j = 1$ to k **do**
10: $$M[i, j] = \max \begin{pmatrix} M[i, j-1] + \text{UnMatched}(c_j), \\ M[i-1, j] + \text{UnMatched}(f_i), \\ M[i-1, j-1] + \text{Matched}(f_i, c_j) \end{pmatrix}$$
11: **end for**
12: **end for**
13: return best alignment $A[k^-, k]$ by re-tracing the computation of $M[k^-, k]$ back to $M[0, 0]$.

Since every field in the record r must match a column in the table, UnMatched(f) is set $-\infty$. To obtain a simple formulation, we would like to set UnMatched(c) to be a constant C for all columns c. In such a case, all possible valid alignments will have exactly $k - k^-$ un-matched columns and hence, a fixed additional cost of $(k - k^-) \times C$. Thus, UnMatched(c) can be set to any fixed value. Of course, a more sophisticated model might be possible where each column is assigned a different cost for being un-matched.

The term Matched(f, c) measures how well the field f aligns with other fields already in the column c, with a higher value indicating a better match. We use the *field-to-field consistency score*, $F2FC$, to estimate the quality of this match. Specifically, if f^c is a value that is already known to be in the column c (from rows that have been aligned), then $F2FC(f, f^c)$ estimates the consistency of f and f^c being in the same column. We discuss how we compute $F2FC$ shortly. We define $F2FC(f, c)$ as follows, where $f_1^c, \ldots, f_n^c$ are the values already known to be in c:

$$\text{Matched}(f, c) = F2FC(f, c) = \frac{1}{n} \times \sum_{i=1}^{n} F2FC(f, f_i^c) \tag{5}$$

Finally, to obtain the alignment $A[k^-, k]$, we trace the decisions taken in computing $M[k^-, k]$. The tracing process is done in reverse, i.e., from $M[k^-, k]$ back to $M[0, 0]$.

4.3 Constructing the initial table T_I

Algorithm **CreateTable** summarizes how we compute the initial aligned table T_I. At a high level, we first split the longer records. Then, we consider all the records in descending order of the number of fields (ties broken by descending average field scores FQ for the records). All the records with k fields (and appear at the top of the sorted order) are aligned trivially into T_I. **AlignShortRecord** is then invoked on each of the shorter records.

Our technique can be thought of as an *iterative* Multiple Sequence Alignment (MSA) technique [3]. MSA is a well-known hard problem, for which several approximate techniques were proposed (see [14,27] for surveys). Most such techniques are designed for the alignment of biological sequences, where each sequence is treated symmetrically. In our context however, we have a different level of confidence for each record (manifested in its average FQ score and the number of null fields). The iterative method allowed us to align records with high confidence first. Then, such records are used to align the ones with lower confidence and so on.

Observe that **AlignShortRecord**, as described in Sect. 4.2 compares each field against all the fields in the partial table T_I. This can be a fairly expensive operation, especially in lists with many lines. Hence, in the interest of efficiency, we maintain a table of *field summaries* for each of the columns in T_I. The field summary maintains representative fields that have already been aligned with that column. The configurable parameter max_n_reps determines the number of representative values in a field summary. The **AlignShortRecord** method only considers the field summaries (and not the entire T_I) while computing Matched(f, c) in Eq. 5. Note that as additional records are aligned, the field summaries are updated.

The main idea of the algorithm **UpdateFieldSummaries** is to select the field representatives such that they are approximately the most coherent set of values (measured by the average pairwise $F2FC$ scores) within a column. The rationale is that correctly extracted fields in the same column are expected to exhibit a high level of coherency.

Note that field summaries are maintained independently for each column. Thus, the field summaries for different columns can include fields from different records.

Algorithm 4 **CreateTable**(R: list of records)

```
for r_i in R do
  if number of fields in r_i > k then
    AlignLongRecord(r_i, k)
  end if
end for
T_I = {}
SF = {}
sort R in descending order of the number of fields.
for r_i in R do
  if number of fields in r_i < k then
    r_i = AlignShortRecord(r_i, SF)
  end if
  add r_i to T_I.
  SF = UpdateFieldSummaries(r_i, SF)
end for
return T_I
```

Algorithm 5 UpdateFieldSummaries(r_i, SF)

```
for j = 1...k do
  Add f_ij to SF_j
  if |SF_j| > max_n_reps then
    min_score_j = ∞
    worst_rep_j = null
    for h=1...|SF_j| do
      score_jh = F2FC(SF_jh, SF_j)
      if score_jh < min_score_j then
        min_score_j = score_jh
        worst_rep_j = SF_jh
      end if
    end for
    remove worst_rep_j from SF_j
  end if
end for
return SF
```

Table 1 Field summaries for the cartoons example

Column 1	Column 2	Column 3	Column 4
11	Steamboat Willie	Disney	1943
10	Rabbit of Seville	MGM	1935
15	Three little pigs	Disney	1949

The fields for each column can be drawn from different lines in the list

Continuing with the cartoons example, if $max\_n\_reps = 3$, then the field summaries for the four columns at the end of the alignment phase (shown in Fig. 3c) are as in Table 1. As desired, our selection method managed to find high-quality fields to serve as field summaries.

In Sect. 7, we consider the question of the size of the field summaries, i.e., max_n_reps. We found that the best value for real web lists in general was 3, but it can vary across domains.

4.4 Field-to-field consistency score

The field-to-field consistency score, $F2FC(f_1, f_2)$, measures the similarity between a pair of fields or of field candidates f_1 and f_2. As, with the field quality score, it is computed from multiple sources. In particular, the $F2FC$ has four components: *type consistency* S_{tc}, *table corpus consistency* S_{tcc}, *syntax consistency* S_{sc}, and *delimiter consistency* S_{dc}. The $F2FC$ is a linear combination of these factors.

$$F2FC(f_1, f_2) = a_{tc} \times S_{tc}(f_1, f_2) + a_{tcc} \times S_{tcc}(f_1, f_2) + a_{sc} \times S_{sc}(f_1, f_2) + a_{dc} \times S_{dc}(f_1, f_2) \quad (6)$$

where a_{tc}, a_{tcc}, a_{sc}, and a_{dc} are weights (in the range [0, 1]) assigned to each component, such that they sum to 1. We now consider each component in turn.

Type consistency score (S_{tc}): If the two fields f_1 and f_2 have the same type, then S_{tc} is set to 1. Otherwise, it is set to 0. The types recognized are the same as those for the type score (S_{tq}) component of FQ.

Tables corpus consistency score (S_{tcc}): Fields f_1 and f_2 are said to have high table corpus consistency if there are many columns in the table corpus in which the both f_1 and f_2 co-occur. For instance "Barack Obama" and "Nicolas Sarkozy" can have a high S_{wc} score, while "Barack Obama" and "France" would have a very low score. Formally, $S_{tcc}(f_1, f_2)$ is calculated as the average of the two conditional probabilities $Pr(f_1|f_2)$ and $Pr(f_2|f_1)$.

Syntax consistency score (S_{sc}): The syntax consistency measures if two fields have the same "appearance" (though they might not belong to the same recognized type or occur in the same column in the table corpus).

To calculate $S_{sc}(f_1, f_2)$, we first extract several numerical features from the strings forming both f_1 and f_2. The features we consider are as follows: (1) number of letters, (2) percentage of lower case letters, (3) percentage of upper case letters, (4) percentage of digits, and (5) percentage of punctuation.
Consistency scores are computed separately by comparing each of the above syntax features. Suppose v_1 and v_2 are the values for a particular feature for fields f_1 and f_2, then the corresponding feature consistency score is as follows: $1 - \frac{|v_1 - v_2|}{\max(v_1, v_2)}$. The feature consistency scores are all in the range [0, 1]. S_{sc} is the average of all the individual feature consistency scores.

Delimiters consistency score (S_{dc}): Delimiters consistency measures the similarity in the field delimiters. $S_{dc}(f_1, f_2)$ is set to 1 when the delimiters on both sides are identical for f_1 and f_2; to 0.5 when they are match on one side only; and to 0 otherwise.

Until now, we have not considered delimiters as being a part of fields (or field candidates). We assume delimiters to

belong to the set of letters " ,;:./\()<>&|#!?", and they are used to separate sentences into words (fields are sequences of words). A special class of delimiters are the different HTML tags that are encountered while processing web pages.

For example, the two fields (with their leading and trailing delimiters) `<b>Barack Obama</b>`, will have a higher value for S_{dc} when compared with `<b>Nicolas Sarkozy</b>`, than when compared with `<i>France</i>`.

5 Refinement phase

So far, the only information being passed between lines of the list is the number of likely fields that was used in the **Bounded SplitLine** algorithm to prevent the excessive splitting of some lines. However, our goal is to split an entire list of lines that are supposedly related to each other. In this section, we describe how we exploit the collective nature of our splitting task to fix errors resulting from the independent splitting.

Our algorithm is based on two observations. First, assuming that the number of correctly split fields is many more than incorrectly split ones, collective analysis of the records enables us to detect incorrectly split fields: such fields will align poorly with other fields in the initial table T_I. Second, incorrectly split fields do not occur in isolation within a record. By definition, if a field was incorrectly split, then one of its adjacent fields within the same record has to be incorrectly split. Thus, incorrect splits occur in *streaks*.

5.1 Detecting inconsistent streaks

We can detect incorrectly split fields by identifying those that are poorly aligned with their columns in T_I. We call such fields *inconsistent*, and we detect them using their $F2FC$ scores when compared to other fields in the same column. Rather than comparing each field with all the other fields in its column, we re-use the field summaries computed in the alignment phase. The field summaries are compact, yet representative of their columns and hence enable the efficient detection of inconsistent fields.

Specifically, for every field, f_i, in column c_i of T_I, we calculate its $F2FC$ score against its corresponding field summary, i.e., $F2FC(f_u, SF_i)$, where SF_i is the field summary for the column c_i (Eq. 5). We sort the fields in descending order of their consistency scores. We consider the fields in the bottom $P_{\text{inc}}\%$ to be the ones that are likely to be inconsistent. P_{inc} is a configurable parameter that reflects the percentage of inconsistent fields in T_I.

We consider all null fields to be inconsistent, i.e., they have $F2FC$ scores of zero. Thus, they are candidates for refinement. This is because the alignment phase might have inserted nulls between two adjacent fields that were incorrectly split.

Having detected individual inconsistent fields, we group them into *streaks*: a sequence of inconsistent fields within a single record. We ignore streaks that only consist of null fields. We also ignore streaks that only include a single field. This is consistent with our observation that incorrect splits do not occur in isolation. We denote a streak in record i that spans from the field in column j_1 to column j_2 as $F(i, j_1, j_2)$.

In our experiments, we found a P_{inc} value of 50% to work well. Note that this does not mean that we refine the splitting decisions for half the fields in the table. A number of streaks contain only nulls or have only one non-null field and are hence ignored.

5.2 Correcting inconsistent fields

For every detected streak of inconsistent fields $F(i, j_1, j_2)$, we apply the following three operations in sequence: *re-merge*, *re-split*, and *re-align*.

Re-merge: All fields within $F(i, j_1, j_2)$ are merged into a single field.

Re-split: The contents in the merged field are re-split using the **BoundedSplitLine** algorithm (Algorithm 2). The parameter k_{mas} is set as the number of columns spanned by $F(i, j_1, j_2)$, i.e., $j_2 - j_1 + 1$.
The splitting in this phase differs from the earlier splitting operations in one significant way. We exploit the collective nature of the splitting task by including an additional component in the computation of the field quality scores (FQ). The additional component, called the *List Support Score* S_{ls}, biases that field quality scores in the favor of field candidates that are more consistent with the columns spanned by the streak. We describe it in Sect. 5.3 below.

Re-align: The number of fields generated after the re-splitting step may be smaller than the number of columns spanned by $F(i, j_1, j_2)$. Therefore, we re-align the fields with their corresponding columns, placing nulls in the appropriate positions within the re-split fields. Alignment is achieved using the **AlignShortRecord** algorithm. We only consider the re-split fields. Further, we do not need to maintain or re-compute the field summaries, which can be simply re-used from the alignment phase.

Finally, we note that the refinement phase can be run repeatedly until the output table converges. In each invocation, some of the incorrect splits might be fixed, eventually leading to a stable split and alignment. However, our experiments indicate that a single invocation of the refinement phase typically suffices.

5.3 Field quality score—revisited

In order to exploit the collective nature of the splitting task, we consider an additional component in the field quality score for this phase. The **List Support Score**, $S_{\text{ls}}(f)$, of a field candidate f measures the consistency between f and the fields extracted from other lines in T_I.

Suppose f is a field candidate while re-splitting the streak $F(i, j_1, j_2)$. We compare the consistency of f against each of the columns between j_1 and j_2. The field f is deemed to have strong list support if it is consistent with *any* of the columns between j_1 and j_2. As before, we use the field summaries to estimate the consistencies.

$$S_{\text{ls}}(f) = \max_{h=j_1}^{j_2} F2FC(f, SF_h)$$

Note that the list support score is only applicable in the refinement phase because we have the initial table T_I that facilitates targeted consistency comparisons. Equation 1 for $FQ(f)$ is updated to include the list support score.

5.4 Table extraction score

Once a table T is extracted from the list L, we calculate its *Table Extraction Score*, $TE(T)$, by computing the average field quality score of all the fields in the table.

$$TE(T) = \sum_{i=1}^{n} \sum_{j=1}^{k} FQ(f_{ij}) \tag{7}$$

where f_{ij} is the field score of the jth field in the ith record. Null fields are assigned a field score of zero.

When applying the extraction algorithm to a large collection of lists, the TE score becomes very useful in ranking the extracted tables based on how well we think they were extracted. In fact, our experimental results show that our TE is able to accurately reflect the relative extraction quality of the lists into tables, and hence, can be used by applications that employ ListExtract.

6 Implementing ListExtract

ListExtract was implemented as described in the earlier sections as a Java library. It can be invoked on individual lists one at a time, e.g., to support a table extraction service or in a map-reduce where a large corpus of web pages is processed in parallel to extract potentially useful tables. We found one of the main challenges was in deploying and accessing the table corpus and the language model. Each of these was deployed as service backed by about 500 machines on a shared Google data center.

Table Corpus: We use the table corpus [8] to compute the table corpus support (in FQ) and the table corpus consistency ($F2FC$). The corpus includes 154 million tables. From the raw table corpus, we computed a single lookup table that had two kinds of entries. For each field value that occurred in some table in corpus, it had the count of the number of tables it occurred in. Likewise, for each pair of field values that co-occurred in a column in some table, it had the count of the number of such tables. The resulting lookup table was about 220 GB. When we restricted the entries to those that occurred in at least two tables, the resulting lookup table was about 44 GB.

To enable fast lookups to the table corpus, we use one of two alternative methods. First, we can reduce the size of the corpus by filtering tables based on quality measures such as the page rank of the web page and increasing the minimum occurrence count for entries ($min\_tc\_support$). Reducing the footprint of the corpus to below 10 GB will ensure that it can be loaded directly into memory, thereby enabling fast access. However, reducing the size of the corpus can lead to a loss in extraction quality.

We instead implemented the table corpus as a distributed service. The lookup table was partitioned into 500 parts using a hash function on the field/field pairs. Each of the partitions was served by a separate process. To compute the S_{tcs} and S_{tcc} scores, the occurrence counts for the field or field pair are looked up by making an RPC to the corresponding server (identified by the same hash function used for splitting).

Language model: We used a language model that recorded the number of occurrences of each word sequence in a crawl of many million web pages [6]. The language model was served by a Google-wide service that is used by a number of services. The service uses a splitting scheme similar to the one outlined above for the table corpus.

Making RPCs to access the table corpus and language model services can be very expensive. Take the table corpus for example: given a table with n lines each with m terms, there may be as many as $n \binom{m+1}{2}$ field count lookups, and if there are c columns, as many as $n \times c^2 \times max\_n\_reps$ field pair co-occurrence lookups. If the RPCs are made serially, then for a list with 10 lines each with 10 terms that is eventually split into 3 columns, this amounts to almost 800 RPC calls. Given that there are 500 servers for the lookup table, we make the calls more efficient by (1) grouping together the requests that are sent to the same server and implementing a batch lookup at each server and (2) by sending all requests to the different servers in parallel.

Table 2 Statistics about the WLISTS and TDLISTS data-sets

	WLISTS			TDLISTS		
	Range	Avg	SD	Range	Avg	SD
Rows	[7, 120]	52.16	26.18	[10, 50]	22.15	12.21
Columns	[2, 10]	4.28	1.84	[2, 7]	3.12	1.26
Words/Row	[2, 24]	7.54	2.87	[2, 34]	6.76	4.29
%Nulls	[0, 18]	4.76	6.72	[0, 43]	3.05	7.78

7 Experiments

We conducted an experimental study evaluating the performance of LISTEXTRACT. The goal of the study was to (1) understand in absolute terms the ability of our technique to correctly extract relational tables from lists, (2) understand the contributions of the various constituents in LISTEXTRACT, and (3) compare LISTEXTRACT with information extraction systems. Additional results estimating the potential for harvesting relational tables from the web at large are presented in [15].

7.1 Experimental setup

We start by describing our data sets and our general experimental setup.

Data sets: We considered two distinct data sets: one consisting of HTML lists from the web and the other consisting of lists constructed from tables. In both cases, we only consider English language lists.

Web Lists (WLISTS)*:* This is a collection of 20 HTML lists spanning 20 different domains, which we manually collected from the web. The lists span varied domains such as cartoons, airlines, lawsuits, and Emmy Award winners. We manually constructed tables from the list contents and used those tables as ground truth.

Tables-Derived Lists (TDLISTS)*:* This is a collection of 100 lists derived from 100 randomly selected HTML tables from the web. Note that these tables are not part of the web corpus we use in our experiments. We derived lists from tables by collapsing all the cells in a row into a single line (with white spaces separating the words). The original tables are used as the ground truth in our evaluation. Table 2 provides some statistics characterizing the two data sets.

Evaluation: We note that a direct comparison of the tables extracted by LISTEXTRACT and the corresponding ground truth can be tricky. In part, this is due to the fact that there may be more than one acceptable solution. For example, the data in a column, c_g, in the ground truth could be present in two columns c_1 and c_2 in the table output by LISTEXTRACT. Similarly, we may see that about half of the cells in c_g match exactly those in c_1, another half match exactly those in c_2, and a small number match neither.

We use the following rating method. First, we *match* the columns of the generated table T and the ground truth T_g based on the overlap between the values in their cells. We then consider the pair of columns c (from T) and c_g (from T_g) with the highest overlap. We declare the fields in c that match exactly with those in c_g (same row and same contents) to have been extracted correctly. The two columns c and c_g are excluded from further analysis. Then, we consider the pair with the second highest overlap, and so on, until either the columns of T or T_g are exhausted. We note that this is a rather *conservative* notion of correctness and hence, is likely to under-estimate the true utility of the extracted tables.

Let T^{total} and T_g^{total} be the total number of cells in T and T_g, respectively, and let $T^{correct}$ be the number declared to have been extracted correctly.

We estimate the precision (P), recall (R), and f-measure (F), as below:

$$P = \frac{T^{correct}}{T^{total}} \quad R = \frac{T^{correct}}{T_g^{total}} \quad F = \frac{2 \times P \times R}{P + R}$$

In the rest of this section, we only report the f-measure. In all our experiments, we observed that the number of columns in T and T_g is typically identical and occasionally off by only ± 1. Thus, the values of precision and re-call are very close to each other. For this reason, we found it sufficient to report the f-measure in our experiments. We compute the f-measure separately for each list and report the average f-measure over the entire data set.

Ranked performance curves: In the rest of the section, we present our results as follows. Recall that for every extracted table, T, we computed a score $TE(T)$ (Sect. 5.4). First, we rank the tables extracted in descending order of their TE scores. Then, we present performance curves in which we report the average f-measure for the top $X\%$ of the tables (X-axis), i.e., a point $\langle x, y \rangle$ on the curve indicates that the top $x\%$ of the tables sorted by their TE scores has an average f-measure of y. This analysis is interesting because it indicates that TE are in fact, closely related to the actual table quality.

System parameters: Unless otherwise specified, we apply the following strategy in all our experiments. FQ and $F2FC$ scores consider all their components (as described in Sects. 3.1 and 4.4). We assign equal weights to the different score components used in each case. The threshold

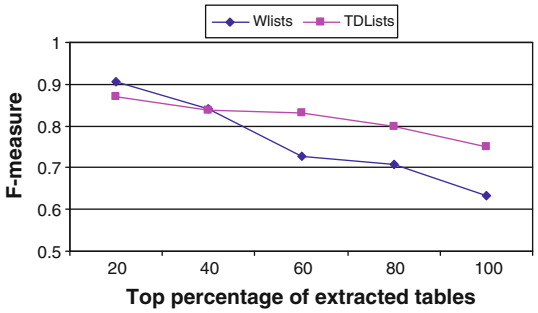

Fig. 4 Overall performance results for the WLISTS and TDLISTS data sets

min_tc_support used in calculating the table corpus support score (S_{tcs}) is set to 1. In the alignment phase, for the WLISTS data set, we maintain field summaries with *max_n_reps* as 3, while for the TDLISTS data set, we set it to 6. Finally, in the refinement phase, the fraction of fields considered inconsistent (P_{inc}) is set to 50%.

In general, the default values for the above parameters were set by conducting sensitivity analysis experiments (mostly reported in this section) to determine their best values.

7.2 Overall performance

Figure 4 shows the performance results for the WLISTS and TDLISTS data sets. The f-measure for WLISTS ranges from 0.90 to 0.65, while for TDLISTS it ranges from 0.95 to 0.75. Observe that for both the data sets, the f-measure decreases as we consider more lists whose extractions have poor TE scores. This indicates that TE is in fact a reasonable measure of the quality of table extraction. Thus, the TE scores can serve as a useful signal for any application that consumes tables automatically extracted from lists: Applications that need more precise extractions can restrict themselves to only those with very high TE scores.

The poorly performing lists in the WLISTS data set are the ones that have very inconsistent structure. For example, in the list titled *Complete list of Emmy Award winners*, the lines do not have parallel sentence structure: some awards are for performers, while others are for series; some mention character names, while others do not; and not all the lines mention the network name. The better performance on the TDLISTS data set is likely due to the fact that the underlying data were always constructed from a relational table.

Looking at our technique's performance from a different perspective, we made this observation: About one-third of all columns in WLISTS and more than half of TDLISTS' columns had over 90% of their fields correctly extracted.

7.3 Field quality score components

Figure 5a, b shows the performance for different configurations of FQ. Each configuration has a different combination of the component scores in FQ. All configurations include the list support score, S_{ls}, since it is essential in the refinement phase. `T`, `LM`, and `WT`, respectively, consider only type support, language model support, and table corpus support; `All` includes all the components. The configuration `All−T`, `All−LM`, and `All−WT` consider all components except the type, language model, and table corpus support, respectively.

For both data sets, combining all components achieves the highest f-measure. The most significant individual component appears to be the table corpus support. `WT` outperforms `T` and `LM`, while `All−T` and `All−LM` outperform `All−WT`. This clearly demonstrates that looking up other tables on the web helps identify field candidates that are more likely to be good cell values.

Interestingly, the language model performs very poorly when considered in isolation. However, `All` out-performs `All−LM` by as much as 20%. A closer look suggests that this is because the language model provides very sparse positive signals, i.e., for most field candidates it reports a low score. However, when it does report a high signal, it is very reliable positive signal and is able to complement type and table corpus support effectively.

7.4 Field-to-field consistency components

Figure 6a, b show the performance for different configurations of $F2FC$. Each configuration considers different components: only type consistency (`TC`), only table corpus consistency (`WC`), only syntax consistency (`SC`), and only delimiter consistency (`DC`). The other configurations are defined in the same spirit as in Sect. 7.3.

As with FQ, we note that, in general, combining multiple score components gives better results for both data sets. However, we note that including the delimiters consistency component sometimes leads to the degradation of the results. This is especially true in the TDLISTS data set. This is likely due to the fact that while deriving the lists from the table corpus, we uniformly insert white spaces as delimiters between adjacent fields. Hence, relying on the delimiters turns out to be misleading.

7.5 Effect of refinement

Figure 7a, b compare the performance with and without the refinement phase. For the WLISTS data set, the improvement is quite significant and ranges from 10 to 20%. Interestingly, the improvements are more in the top 20% and top 40% extracted tables. This is probably because in such cases a

larger number of fields are correctly split and hence, they are able to effectively aid in fixing errors.

On the other hand, the improvement for the TDLISTS is not as significant. In fact, the improvement is never more than 5%. This difference might again lie in the fact that the lists are derived from tables and hence the separation between fields are likely to be easier to detect even in the independent splitting phase.

7.6 Effect of field summaries

Figure 8a, b compare the performance of table extraction for different values of max_n_reps, that parameter that determines the size of field summaries maintained for each column (and used in the alignment and refinement phases).

For the WLISTS data set, smaller values of max_n_reps (i.e. 1 and 2) result in lower f-measure. However, as we increase max_n_reps to 3 and above, we get a significant improvement in f-measure. However, setting it to values higher than 3 does not seem to significantly improve the results. The results obtained for values 6 and 7 are marginally better when considering the entire data set. For the TDLISTS data set, the best results are obtained when max_n_reps is set to 7, but again the performance converges as max_n_reps increases. The results indicate that even small summaries suffice, thus making it un-necessary to perform expensive exhaustive comparisons.

7.7 Comparison with RoadRunner

Wrapper generation systems have the goal of extracting structured data from web pages. However, they typically assume that the data were created according to some template, and the goal of the system is to learn the template. We now compare LISTEXTRACT to one such system, RoadRunner [13], which is widely used in the research community, and show that LISTEXTRACT indeed offers superior performance. We were not able to obtain implementations of other wrapper generation systems [1,10,30].

RoadRunner makes three key assumptions: (A1) the template consists of only HTML tags, (A2) the template is consistent across all records, and (A3) data fields are separated by template elements.

We applied RoadRunner only on WLISTS. In TDLISTS, the fields are separated by white spaces, making them unsuitable for processing. To enable RoadRunner to learn templates, each list is presented as a set of web pages (one per line in the list).

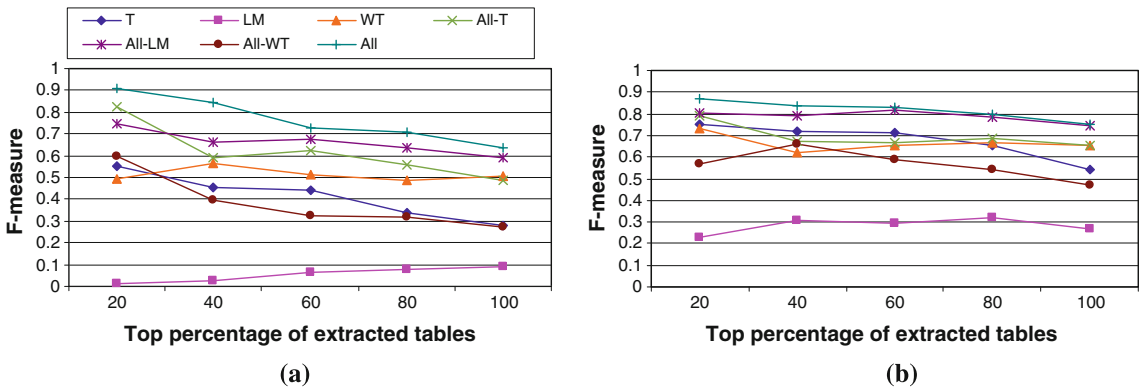

Fig. 5 Effect of different configurations for the field quality score on **a** WLISTS and **b** TDLISTS

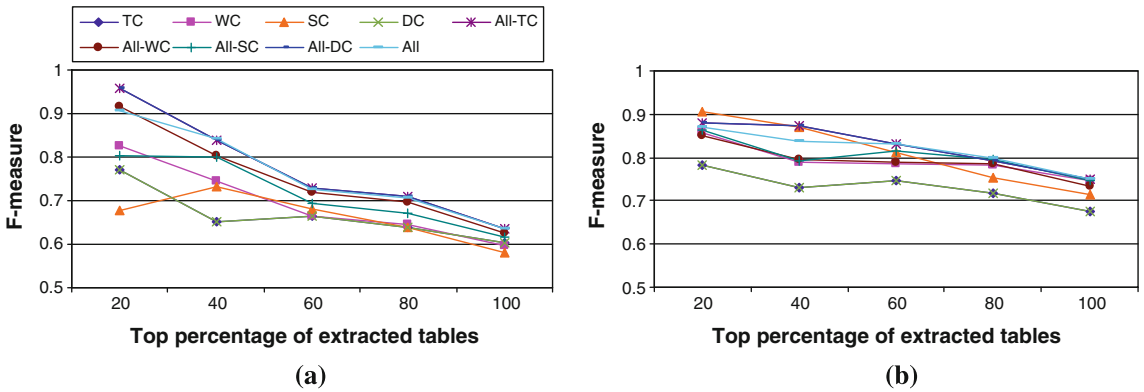

Fig. 6 Effect of different configurations for the $F2FC$ score on **a** WLISTS and **b** TDLISTS

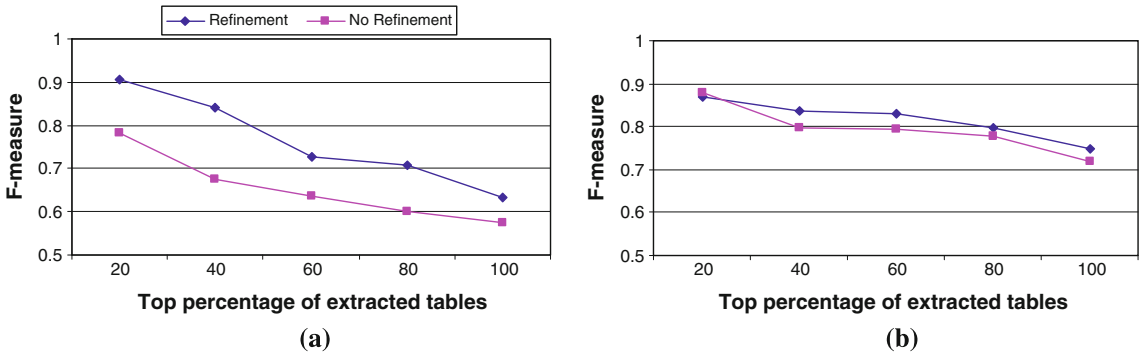

Fig. 7 Effect of the refinement phase on **a** WLISTS and **b** TDLISTS

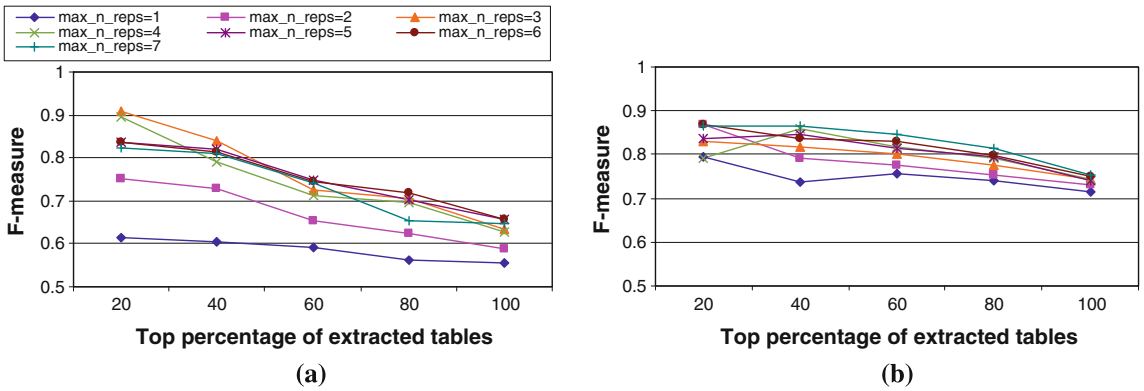

Fig. 8 Effect of number of column representatives on **a** WLISTS and **b** TDLISTS

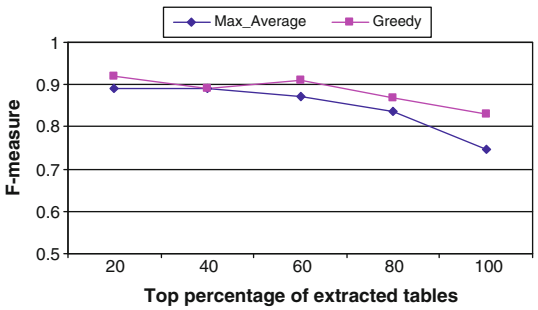

Fig. 9 Effect of **line** splitting search method on TDLISTS

Table 3 shows the precision, recall, and f-measure achieved by RoadRunner and LISTEXTRACT. As can be seen, LISTEXTRACT performs better. To understand RoadRunner's poor performance, consider the following: of the 20 lists, (1) all fields are correctly extracted in the 3 lists that had rich and consistent HTML tags; (2) no fields were extracted in 9 lists. Of those, in 5 lists there were no HTML tags, violating assumption A1 (e.g. cartoons in Fig. 1), and in 4 lists the tags were inconsistent, violating assumption A2 (some lists emphasize certain fields by formatting them differently; e.g. [21]); and (3) only a partial set of fields were extracted in the remaining 8 lists because some fields were not separated by HTML tags, violating assumption A3.

Table 3 Precision, recall, and f-measure for LISTEXTRACT and RoadRunner when applied to WLISTS

	Precision	Recall	F-measure
LISTEXTRACT	0.64	0.63	0.63
RoadRunner	0.39	0.28	0.32

There are other wrapper generation systems (though not publicly available for comparison) that do not make all of the three assumptions. For instance, ExAlg [1] does not restrict the template elements to HTML tags, but still makes assumptions A2 and A3. DEPTA [30] can tolerate some inconsistencies across records as they are reconciled by means of a partial tree alignment algorithm. However, assumptions A1 and A3 still must hold for DEPTA to work properly. Thus, neither of them is likely to work well across all the lists in our dataset.

7.8 Effect of the line splitting search method

The goal of this experiment is to evaluate the greedy method we use to search for the best split for each line in the list. For this purpose, we compare the greedy search method with

an exhaustive method whose objective is to maximize the average FQ scores of the extracted fields after the line split.

For this particular experiment, we only used the TDLISTS data set, and we ensured that the selected tables have at most 10 words per line. This way, we could run the exhaustive search method in a reasonable time.

As we discussed in Sect. 3, besides being more efficient, our greedy method proved superior to the exhaustive method. This is illustrated in Fig. 9, where an improvement of up to 11% is achieved when considering 100% of the extracted tables.

8 Applications of LISTEXTRACT

LISTEXTRACT is a generic technique that is usable in several contexts of Web data management. In this section, we briefly outline some of these contexts.

Question answering: Search engines try to provide factual answers when relevant (e.g., "Who was the American president in 1966?" and "When was the 'King Size Canary' cartoon produced?"). To answer such queries, the search engine needs to extract facts from the web, and many of these are embedded in lists.

Deep web crawling: Surfacing the deep web involves filling in web forms with guessed values and indexing the resulting HTML pages [23]. A key challenge in this approach is to surface as many hidden pages with as few form submissions as possible, and having access to high-quality guessed values is critical. HTML tables generated by LISTEXTRACT provide an excellent source of such guessed values. In particular, once some initial set of values is determined to be suitable for filling in a given web form (using the techniques described in [23]), this set can then be used to search for similar sets of suitable values in the table corpus.

Integration for the relational web: Integrating data from structured sources on the web require a workbench that also supports integrated search, extraction, and cleaning [7]. One of the important tasks such a workbench needs to support is creating tables from HTML lists (the split operator in [7]). For example, consider the task of creating a table of all the PC members of VLDB in the last 10 years. The data exist in lists spread out on 10 different web sites.

Table extraction web service: In addition to extracting tables from lists on the web, LISTEXTRACT can also be made available as a service for data management products such as Fusion Tables [17]. Users often upload data that is tabular but available to them only as lists. Hence, the ability to extract tables from these lists (which in some cases amounts to splitting fields into their subfields) is a valuable function of a data cleaning component of such systems.

9 Related work

In principle, extracting tables from lists is an information extraction task. The most closely related information extraction problem is that of *wrapper generation*, where fields are extracted from HTML documents [1,2,10,11,13,16,19,20,22,29–31]. In most cases, wrappers are used to encapsulate dynamically generated pages, where a collection of such pages would all have a fixed *template* and some varying data fields whose values are obtained from a back-end database. The wrapper should be able to identify the template and hence, extract the data fields from any new pages having the same template.

Supervised learning approaches such as WEIN [19], Stalker [2], Wrap [22], WL^2 [11,16], and [31], require a labeled set of web pages from which a template can be inferred. Our methods are intended to apply at web scale, and therefore, creating labeled training sets is infeasible.

Unsupervised approaches such as RoadRunner [13], ExAlg [1], DEPTA [30], IEPAD [10], DeLa [29], and [24] typically rely on the repetitive patterns in the HTML tags across multiple pages, or multiple records within the same page, to detect the template. In [20], it is assumed that every detected record in a web page is linked to a *detail page*. The co-occurrence of terms in a record and in its detail page is used to distinguish between terms in the template and the varying record-specific terms. All these approaches assume that the pages are dynamically generated, and hence, an underlying template exists.

Lists in static web pages, on the other hand, are not expected to be heavily structured using HTML tags. At best, the list items may contain a few delimiters and simple formatting tags. As already discussed, web lists are mostly hand-crafted and hence have inconsistent (or no) formatting, tagging, or field separation.

Some systems do not assume that their input data are structured with HTML tags (e.g. [5,12,18,25]). DataMold [5] uses domain-specific vocabulary and training examples to learn a Hidden Markov Model (HMM). The model can then be used in extracting fields from documents in a specific domain (e.g. publication lists or mailing addresses). However, this approach requires human supervision.

The WWT system [18] takes a few example data records as input and augments them with additional new records, extracted from a large corpus of lists on the web. The system starts by finding the most relevant lists to the input records. Then, an extraction algorithm based on conditional random fields (CRF) is used to extract more records from each such list. During extraction, WWT relies on the overlap between

lists. In particular, if previously extracted records also occur in a new list, then those records are used in training the extraction model for that new list. The set of extracted records is finally consolidated and augmented to the input table.

While similar to ListExtract in the notion of extracting tables from web lists, the goals of ListExtract and WWT are different. WWT quickly focuses on a few lists given a query. Our goal is to create a corpus of extracted tables, and hence, we consider all the lists that may be useful. As a result, they get to prune many lists immediately, whereas we do not. Moreover, while it is fairly easy to leverage the overlap between lists in ListExtract(similar to WWT), it is not necessarily desirable in our case. In particular, we can directly add every extracted table to the table corpus used by ListExtract, and hence, it can potentially help in the extraction of new tables from lists. However, since we already have access to a very large table corpus whose accuracy is expected to be higher than the tables extracted from lists, it is safer to only rely on those tables originally present in the corpus.

The system in [25] extracts data fields from text "posts", such as those on Craigslist, another example of hand-crafted content. The approach is unsupervised and does not depend on HTML tags. A collection of reference sets is used to recognize candidate fields. However, six reference sets are used for six specific domains. Similarly, the ONDUX system [12] relies on the use of a reference set for each domain with a small number of attributes in each set (ranging from 5 to 18 in their experiments). The system first attempts to identify field values in the input list and then matches each individual field to one of the few attributes in the reference set used. In a final "reinforcement" step, it builds a graphical model to verify (and potentially correct) the assignment of attribute labels to each field—which is comparable to our refinement phase. We note, however, that extending these approaches [12,25] to each new domain involves constructing a new reference set. On the other hand, ListExtract does not incur any per-domain costs as it relies on a corpus of many million raw HTML tables that span almost all conceivable domains.

10 Conclusions

In the quest to extract and leverage structured data on the web, we considered lists as a rich source of structured data. We addressed the key technical challenge concerning lists—splitting list entries into table rows. Our ListExtract is a completely unsupervised method and does not assume any domain knowledge. As such, it can be applied to lists on the web at large. ListExtract uses multiple sources of information to make splitting decisions within a line and across lines of the list. We described a set of experiments that validated the quality of tables that are created by ListExtractand suggested that a large number of high-quality lists can be extracted from the web.

References

1. Arasu, A., Garcia-Molina, H.: Extracting structured data from Web pages. In: SIGMOD (2003)
2. Barish, G., Shin Chen, Y., Dipasquo, D., Knoblock, C.A., Minton, S., Muslea, I., Shahabi, C.: Theaterloc: using information integration technology to rapidly build virtual applications. In: ICDE (2000)
3. Barton, G., Sternberg, M.: A strategy for the rapid multiple alignment of protein sequences: confidence levels from tertiary structure comparisons. J. Mol. Biol. **198**(2), 327–337 (1987)
4. Bellman, R.: On the approximation of curves by line segments using dynamic programming. Commun. ACM **4**(6), 284 (1961)
5. Borkar, V., Deshmukh, K., Sarawagi, S.: Automatic segmentation of text into structured records. In: SIGMOD (2001)
6. Brants, T., Popat, A.C., Xu, P., Och, F.J., Dean, J.: Large language models in machine translation. In: EMNLP-CoNLL (2007)
7. Cafarella, M., Halevy, A., Khoussainova, N.: Data integration for the relational Web. PVLDB **2**(1), 1090–1101 (2009)
8. Cafarella, M.J., Halevy, A.Y., Wang, D.Z., Wu, E., Zhang, Y.: WebTables: exploring the power of tables on the web. PVLDB **1**(1), 538–549 (2008)
9. Cafarella, M.J., Halevy, A.Y., Zhang, Y., Wang, D.Z., Wu, E.: Uncovering the relational Web. In: WebDB (2008)
10. Chang, C.-H., Lui S.-C.: IEPAD: information extraction based on pattern discovery. In: WWW (2001)
11. Cohen, W.W., Hurst, M., Jensen, L.S.: A flexible learning system for wrapping tables and lists in HTML documents. In: WWW (2002)
12. Cortez, E., da Silva, A.S., Goncalves, M.A., de Moura, E.S.: Ondux: on-demand unsupervised learning for information extraction. In: SIGMOD (2010)
13. Crescenzi, V., Mecca, G., Merialdo, P.: RoadRunner: towards automatic data extraction from large Web sites. In: VLDB (2001)
14. Edgar, R.C., Batzoglou, S.: Multiple sequence alignment. Curr. Opin. Struct. Biol. **3**, 368–373 (2006)
15. Elmeleegy, H., Madhavan, J., Halevy, A.: Harvesting relational tables from lists on the web. PVLDB **2**(1), 1078–1089 (2009)
16. Embley, D.W., Jiang, Y., Ng, Y.-K.: Record-boundary discovery in Web documents. In: SIGMOD (1999)
17. Gonzalez, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R., Shen, W., Goldberg-Kidon, J.: Google fusion tables: web-centered data management and collaboration. In: SIGMOD (2010)
18. Gupta, R., Sarawagi, S.: Answering table augmentation queries from unstructured lists on the web. PVLDB **2**(1), 289–300 (2009)
19. Kushmerick, N., Weld, D.S., Doorenbos, R.: Wrapper induction for information extraction. In: IJCAI (1997)
20. Lerman, K., Getoor, L., Minton, S., Knoblock, C.: Using the structure of Web sites for automatic segmentation of tables. In: SIGMOD (2004)
21. List of Indonesian floral emblems. http://en.wikipedia.org/wiki/List_of_Indonesian_floral_emblems
22. Liu, L., Pu, C., Han, W.: XWRAP: an XML-enabled wrapper construction system for Web information sources. In: ICDE (2000)
23. Madhavan, J., Ko, D., Kot, L., Ganapathy, V., Rasmussen, A., Halevy, A.Y.: Google's deep web crawl. PVLDB **1**(2), 1241–1252 (2008)
24. Miao, G., Tatemura, J., Hsiung, W.-O., Sawires, A., Moser, L.E.: Extracting data records from the web using tag path clustering. In: WWW (2009)

25. Michelson, M., Knoblock, C.: Unsupervised information extraction from unstructured, ungrammatical data sources on the World Wide Web. IJDAR **10**(3), 211–226 (2007)
26. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970)
27. Notredame, C.: Recent progresses in multiple sequence alignment: a survey. Pharmacogenomics **3**(1), 327–337 (2002)
28. Talukdar, P.P., Reisinger, J., Pasca, M., Ravichandran, D., Bhagat, R., Pereira, F.: Weakly supervised acquisition of labeled class instances using graph random walks. In: EMNLP (2008)
29. Wang, J., Lochovsky, F.H.: Data extraction and label assignment for web databases. In: WWW (2003)
30. Zhai, Y., Liu, B.: Web data extraction based on partial tree alignment. In: WWW (2005)
31. Zhai, Y., Liu, B.: Extracting Web data using instance-based learning. WWW **10**(2), 113–132 (2007)