

Bootstrapping for Example-Based Data Extraction

Paulo B. Golgher Altigran S. da Silva
Alberto H. F. Laender Berthier Ribeiro-Neto

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
31270-901 Belo Horizonte MG Brasil
{golgher,alti,laender,berthier}@dcc.ufmg.br

Abstract

The semiautomatic generation of wrappers for Web data sources is a crucial task if proper access to the huge amount of semi-structured data on the Web is to be granted. In particular, the development of strategies for wrapper generation based on user-given examples is currently one of the most promising research directions in Web data extraction. Further, there is great interest in solutions and tools that simplify the specification of examples by the user. In this paper we show how to use a pre-existing data repository to automatically generate examples for extracting data from multiple Web sources on the same domain. Thus, our solution provides human-independent bootstrapping for example-based data extraction on the Web. To demonstrate the feasibility of our approach we provide a number of results obtained from experiments we carried out, in the context of the DEByE tool for semi-structured data extraction. We also discuss how our ideas can be used to improve extraction rates and for providing resilience and adaptiveness for example-based generated wrappers.

1 Introduction

In the past few years, the perception that large portions of the Web can be regarded as data “containers” opened the perspective of using them in a variety of ways. However, these *data-rich* or *data-intensive* [6] Web sources provide HTML pages in which data of interest (e.g., data on featuring movies) are mixed inside the page text with markup tags, other strings, in-line code, etc. Further, the structure of the data is implicit and only suggested by presentation features. Such a structure is often loose, with the possibility that two similar items (e.g., entries on two distinct movies) present structural variations between them. Because of this, data available in Web sources have been termed *semi-structured* [1].

Many recent works in the literature have presented approaches for semiautomatically generating wrappers for extracting and structuring Web data, so that such data can be managed as a database [6, 7, 8, 11, 13, 16]. In most cases, examples are used to “learn” how the data of interest is presented in the sample pages, an essential step to allow using the textual surroundings of the data to generate a wrapper.

There are several important situations in which it is possible to exempt users from providing examples for generating a wrapper to a Web source of interest. This is important because the specification of examples might be tedious and time consuming, particularly if many Web sources are involved or if their layouts change frequently. In this paper, we discuss how to use a previous existing data repository to automatically generate examples for extracting data from Web sources on the same application domain. Thus, our solution provides human-independent bootstrapping for example-based data extraction on the Web.

As an example, suppose that an university librarian needs to populate a database with data on recent book releases available from several on-line bookstores. For this application, this user could generate a wrapper for extracting data from the pages of each one of these bookstores, using some example-based wrapper generation system (such as DEByE [12, 16]). Of course, this would require the user to provide a distinct example for each one of the bookstores. Even though few examples are generally needed [7, 14, 16], if the number of bookstores increases this might become a tedious work. Moreover, if the site of a bookstore changes, new examples need to be provided.

We propose an alternative strategy that uses, for instance, data on books (i.e., names of authors, titles, publishers and prices) available in a pre-existing repository (we call it *source repository*) to identify similar data in other bookstores of interest and automatically assemble examples for extracting data from these bookstores. The basic requirement in this strategy is that the intersection between the data available in the source repository and the data available in the bookstores of interest (the *target sites*) be non-empty.

One of the most challenging issues in our approach is recognizing the intersection among the data in the source repository and the target sites. For instance, the source repository might include the string “Machado de Assis” as the value of an author’s name, while in a target site this name appears as the string “Jose Maria Machado de Assis”. Although these two strings are not the same, they refer to the same person and thus, should be regarded as part of the intersection between source repository and target sites. To further complicate matters, there is no indication that the string “Jose Maria Machado de Assis” can even be considered as an author name. Indeed, it is just another ordinary string occurring in sample pages of the target site.

Despite the challenge imposed by this value and attribute matching problem, our strategy succeeds in properly recognizing the intersection of data between the source repository and the target sites, without generating any incorrect examples (as demonstrated by experimentation). Since few examples are generally needed for generating a useful wrapper, our procedure for the automatic generation of examples favors precision (i.e., generating correct examples) over recall (i.e., generating many examples). To support our claims, we provide a number of results obtained from experiments we carried out within the framework of the DEByE tool for semi-structured data extraction [12, 16].

To generate the source repository, a number of options are available. For instance, it can be generated from a legacy database or from the output of a program. However, the most interesting possibility is using a previously existing wrapper to extract data from a specific Web site and generate the source repository. This opens up the interesting alternative of using our bootstrapping approach to improve the data extraction process on a single site (starting with very few examples). In summary, we can use our strategy for important applications such as improving extraction rates, re-generating a wrapper for a source whose presentation features have changed, or automatically generating a wrapper for a Web source using a pre-existing wrapper for another Web source in the same domain.

The generation of wrappers for Web sources based on user-provided examples has been addressed in several recent works that use techniques from machine learning [7, 11, 15] or information retrieval [4, 16]. Although one of the first works in this area [9] suggests the use of an “oracle” for providing examples, most of the works later developed rely on humans for this task [7, 15, 16]. User intervention is required in another way in the work described in [4], where a method is described for recognizing plausible structures (records) in HTML pages. The structures found are ranked, so that a user can select the objects that must be used to generate the wrapper. As in our work, an existing data repository can be used to perform the role of the user, in an attempt to match the objects in the repository with the ones found in the target HTML pages. A bootstrapping framework functionally similar to the one we propose appears in [17]. However, that approach aims at assisting the user with text learning tasks that require large amounts of training data.

This paper is organized as follows. Section 2 overviews the problem of semi-automatic generation of wrappers through examples. Section 3 describes in details our bootstrapping framework. Section 4 presents experimental results. In Section 5, we discuss applications of our bootstrapping framework and show how it can be used to improve data extraction effectiveness. Finally, Section 6 presents our conclusions.

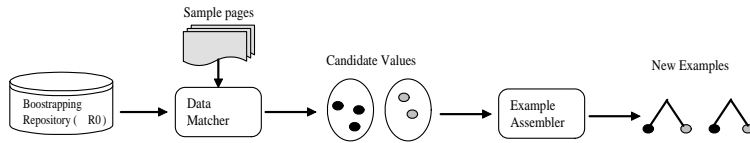


Figure 1: The general framework for automatic bootstrapping.

2 Example Based Extraction

The problem of wrapper generation can be represented as follows. Given a Web data source S containing a set of pages T , determine a mapping w that is capable of populating a repository R with a set O of objects (data items) extracted from the pages in T .

The mapping w is, in general, a set of rules or text patterns used to recognize (among other uninteresting pieces of text) attributes values for objects of interest, associating an appropriate semantics to them. Based on this definition, we can say that a wrapper is an implementation of the mapping w .

Recent works in the literature propose the semiautomatic generation of wrappers by deriving the mapping w from a given set of examples of the objects to be extracted. According to this approach, given a set $E \subset O$ of example objects, taken from a subset $T_0 \subset T$ of the pages of the source S , a wrapper generation procedure g generates the mapping w . That is, $g(E, T_0) = w$.

In general, the rules or patterns encoded in w are based on the textual surroundings (i.e., markups, symbols, keywords, etc.) of the data of interest, what makes the generated wrapper very specific, because it only works properly with pages *similar* to the pages that contain the examples. For practical purposes, we use the term *similar* in a very empirical sense. For example, it can be used to designate the pages of a same site or Web service, such as a Web bookstore. In the next section, we discuss how to automatically generate the set E of example objects, given an initial data repository.

3 Bootstrapping Framework

The framework for automatically generating example objects for the wrapper generation process is illustrated in Figure 1. Suppose that it is necessary to generate a wrapper for extracting data from a data source S (e.g, a Web site or on-line service) that contains objects (data) on a given domain (e.g, movies, books, vehicles, etc.). We assume the existence of a bootstrapping data repository R_0 , which we call the source repository, that contains a set of objects belonging to the same domain of S . The first step of the bootstrapping process consists in trying to recognize, in sample pages of S , matches for the values of the attributes of the objects in R_0 . This step is performed by the *Data Matcher* module (illustrated in Figure 1) and results in a set of candidate attribute-value pairs (avps) of the form $\langle \tau, v \rangle$, where τ is a type and v is a string taken from the text of the sample pages. Next, the *Example Assembler* module selects from this set of candidate attribute-value pairs a subset of avps that are likely to compose objects of the domain of S (according to a variety of heuristics), and uses them to assemble example objects. The candidate avps that do not fit in the assembled example objects are considered spurious and are discarded. We notice that the entire process is guided by the structure of the objects in the initial repository R_0 . In what follows, we discuss the *Data Matcher* and the *Example Assembler* modules in more detail.

3.1 Data Matcher Module

The data matcher module is responsible for recognizing strings (in the sample pages) that can be considered as values for the attributes of the objects in R_0 . These values will later be used to compose example objects.

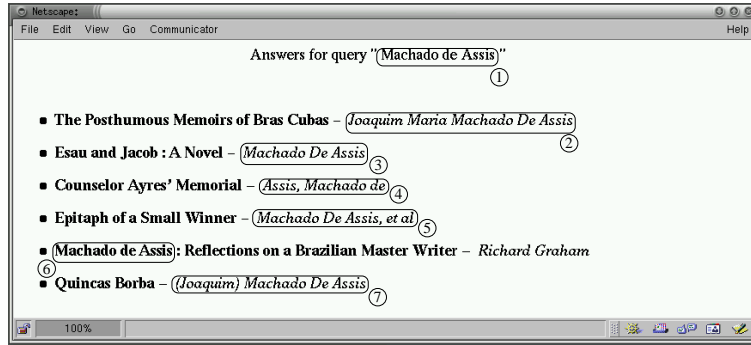


Figure 2: Possible matchings for “Machado de Assis”.

To illustrate, suppose that the avp $\langle \text{Author}, \text{“Machado de Assis”} \rangle$ is a component of one of the objects in R_0 . Consider the Web page presented in Figure 2. Our goal is to find examples for instances (values) of the type Author in this Web page. For this, we identify passages of text that contain approximate matches for the string “Machado de Assis” (because this is an attribute value of an object in R_0). The passages identified are shown in Figure 2 as oval boxes that are labeled with small circled numbers. All the approximate matches found correspond indeed to the name of the same person (a famous Brazilian writer). However, only passages 1, 3 and 6 match exactly the string “Machado de Assis”. For obtaining the other passages, a less strict matching strategy is used. Further, passages 1 and 6 cannot be considered as instances of Author, since they do not occur in a context implicitly associated with instances of the type Author in the sample page. If we use passages 1 and 6 as Author names, the generated wrapper will not be able to identify and extract proper values for Author in pages of the target Web source.

To eliminate passages 1 and 6 as plausible candidates, we observe that the context implicitly associated with the Author type is a pair of tags $\langle \text{it}, \text{/it} \rangle$ (they all occur in italics). Indeed, this is the most common context associated with possible Author instances in Figure 2 (five out of the seven marked passages share this context). The two remaining passages can be discarded using such observation.

To be considered as a candidate example for an instance of an atomic type τ , a string present in a sample page must satisfy two criteria: (1) it must provide an exact or approximate match for some known value of τ and (2) it must occur in a context implicitly associated with instances of τ in the sample page. In the immediately following we describe our strategy for locating candidate examples (i.e., examples that satisfy these criteria). We begin by providing some necessary definitions.

Definition 1 A character c is a **word symbol** iff, it belongs to the set $\{a, \dots, z, A, \dots, Z, 0, \dots, 9\}$. Otherwise, it is said to be a **non-word symbol**.

Definition 2 A string w is a **word** if (i) w is composed of three or more **word symbols**, and (ii) no substring of w is a word. Let s be any string. The **word set** of s , referred to as $WS(s)$, is the set of all the words that are substrings of s .

To exemplify, consider the following string $s = \text{“(Joaquim) Machado De Assis”}$. The word set of s , $WS(s)$, is equal to $\{\text{“Joaquim”}, \text{“Machado”}, \text{“Assis”}\}$.

Definition 3 Let t be the text in a Web page. A **passage** is any string p of t . We also define the **context-frame** of p as a pair $\langle f_i, f_f \rangle$, such that f_i occurs immediately before p and f_f occurs immediately after p , in the text t .

Definition 4 We say that a passage p of a text t is a **plausible passage** for association with a type τ , if there exists some known instance v of τ such that: (i) there is at least one context-frame for p that is composed only by non-word symbols, (ii) $WS(v) \subseteq WS(p)$, and (iii) there is no other passage p' in t , $\text{length}(p') \geq \text{length}(p)$ and $WS(p) \cap WS(p') \neq \emptyset$, such that $WS(v) \subseteq WS(p')$.

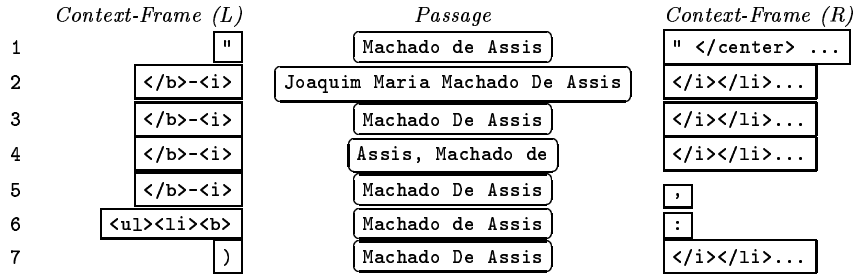


Figure 3: Plausible passages and their context-frames.

Intuitively, the set of plausible passages in a text t for association with a type τ is the set of shortest substrings of t , bounded only by non-word symbols, that contains all words of an proper instance v of the type τ .

Figure 3 presents the plausible passages occurring in the sample page of Figure 2. From this passages, only passages 2, 3 and 4 offer proper alternatives for examples of `Author`, while passages 1, 5, 6 and 7 do not. The reason for considering passages 2, 3 and 4 as proper examples is as follows. A fairly common assumption in the design of example-based wrapper generation procedures is that there exists a common textual context implicitly surrounding the instances of a same type in the pages of a data rich Web source [7, 11, 15, 16]. Indeed, this assumption is the most important hypothesis for such procedures to work. In the example of figures 2 and 3, for instance, all authors names are in italic, which implies that the pair of tags `<i>`, `</i>` form an appropriate common implicit context for instances of `Author` occurring in the Web page shown in Figure 2. By examining the plausible passages in Figure 3, we can determine that a large fraction of the plausible passages (in this case more than 40%) have the same context-frame. Thus, plausible passages 2, 3 and 4, which have this common context-frame, are considered proper or *candidate* passages, according to the following definition.

Definition 5 Let C_τ be a set of plausible passages for a type τ . We define the **support** of the context-frame $\langle f_i, f_f \rangle$ in C_τ , as the percentage of passages that share this context-frame. We say that a passage $p \in C_\tau$ is a **candidate** passage if it has a context-frame $\langle f_i, f_f \rangle$ with a support of at least k (where k is a constant determined empirically) and there is no other non-overlapping context-frame $\langle f'_i, f'_f \rangle$ of p with higher support. The context-frame $\langle f_i, f_f \rangle$ is called **common**.

In the case of our example, the context-frame $\langle \text{"<i>"}, \text{"</i>"} \rangle$ is considered common with a support of 40%. It is important to notice that we can have various common context-frames for a given type τ , since there may be multiple context-frames with support higher than k . In the current implementation, we adopt k equal to 15%.

The word-based matching criteria used for identifying plausible passages (Definition 4) is intentionally strict. Because of this, passages 5 and 7 were left out of the set of candidate passages, even though they are, in fact, part of a proper passages. Because of this, we revisit the set of passages that were not considered candidate passages and recheck if some of them are surrounded by common context-frames (even if they include non-word symbols). To do this, the passages are enlarged until (a) we reach a common context-frame (on both ends) or (b) the context-frame of the passages is composed solely by HTML tags not present in any of the common context-frames. We call this process *plausible passage revision*. After this, we can identify new candidate passages among the revised plausible passages. Figure 4 shows the revised plausible passages (i.e., the passages 1, 5, 6, and 7) for our example in Figure 3. As a result, passages 5 and 7 can now be considered as candidate passages.

In Figure 5, we show an algorithm that summarizes the whole matching process describe above.

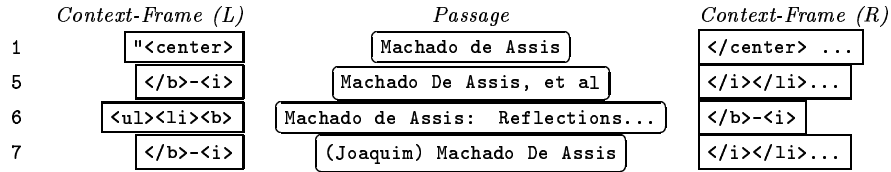


Figure 4: Revised plausible passages and their context-frames.

Match_Attributes

begin

Input:

A set of values of an atomic type τ ;
 A sample page g ;

Output:

Set $P(\tau)$ of candidate passages occurring in g ;

Find in g the set $C(\tau)$ of the plausible passages of type τ ;
 Let F be the set of common context-frames in passages of $C(\tau)$;
 $P(\tau) \leftarrow \{c \in C(\tau) \mid c \text{ has a context-frame } \langle f_i, f_f \rangle \in F\}$;

foreach $p \in C(\tau) - P(\tau)$ do

Enlarge p until we find a context-frame $\langle f_i, f_f \rangle \in F$ or other context-frame solely composed by HTML tags;

end

If a new common context-frame not in F has been found, add it to F ;

$P_r(\tau) \leftarrow \{c \in C(\tau) - P(\tau) \mid c \text{ has a context-frame } \langle f_i, f_f \rangle \in F\}$;

$P(\tau) \leftarrow P(\tau) \cup P_r(\tau)$;

end

Figure 5: Algorithm for finding attributes in the sample pages.

Finding Examples for Numeric Values

For generating examples for attributes that are essentially numeric (e.g., prices, dates, etc), we use a different matching strategy. This is needed because, in general, the intersection of numerical values present in the source repository and in the target web sites is usually small. Indeed, for instance, while it is expected that part of the books that Amazon.com and Barnes and Noble sell are the same, it is not expected that both companies practice the same prices. Thus, for matching numeric values, we generate numeric masks instead of using the real values. Also, we relax the definition of a “word” in this case and allow non-numeric values (such as ‘/’ and ‘;’) to compose the patterns being matched. For instance, for values like “10,76”, we generate a mask “dd,dd” for matching all sequences of two digits followed by a coma and again by two digits.

Matching Failures and Their Consequences

We do not expect that the Data Matcher module recognizes all proper values present in the sample pages given. Indeed, false negatives are expected to occur since the intersection in the contents of the source repository and of the pages in the target sites is usually partial. However, false negatives are usually acceptable since most example-based data extraction approaches require only a handful of examples to extract the data of interest (this is the basic motivation behind such approaches). Experiments presented in Section 5 confirm this assumption.

Our main goal is to avoid the generation of false positives, since this might lead to inaccurate data extraction. For instance, in the CDNOW web site page shown in Figure 6 we want to recognize CD titles and prices, discarding any price associated with the media vinyl. This goal is partially achieved by the support analysis done by the Data Matcher module. For those false positive values that share the same context-frames of true positive values, we propose to eliminate them by analyzing their relative positions, what is done during the assembling of examples described in the immediately following.

Vol. 1-1945-46 1998	CD \$14.49
Sketches Of Spain/Kind Of Blue* 1998	CD \$26.97
Panthalassa-Music Of Miles Dav 1998	CD \$11.49 Vinyl \$15.99

Figure 6: An excerpt of a CDNOW page.

3.2 Example Assembler Module

The goal of the Example Assembler module is to generate complex example objects (e.g., tuples) using the candidate passages found by the Data Matcher module in the sample pages. From now on, we refer to the values of these passages as *candidate attribute values*. The objects generated are similar to the objects in the source repository. As the Data Matcher module might produce false matches (as explained before), the Example Assembler module also has the burden of detecting and discarding such false matches. The assembling process is performed in two steps: (1) record boundary discovering and (2) the assembling of the complete objects, as we now discuss.

3.2.1 Record Boundary Discovery

The first step in the assembling process is the discovery of boundaries that allow determining which groups of attributes (found by the Data Matcher module) belong to a same complex object. These boundaries can be identified indirectly by looking for the occurrence of *separators*, which are string patterns that frequently occur before any components of an object of a given type. For instance, in the sample Web page of Figure 2, the HTML tag corresponding to the bullet before each book entry can be regarded as a separator. We argue that it is fairly reasonable to rely in the occurrence of separators in data rich HTML pages, based on our own experiments and in results published in other works (e.g., [3, 6]).

Generally, record boundary discovery methods rely on the representation of HTML pages as trees (called HTML trees). An HTML tree is a structure in which the tags present in the page are internal nodes and the visible text strings are leaves (we refer the reader to [6] for details).

Embley et al. [6] present a very effective approach for record boundary discovery in data rich Web pages. However, their method requires that the object (records) implicitly present in the page occur only in the subtree of the HTML tree with the highest number of children (fan-out), a fact that significantly reduces the applicability of their approach. We propose a method for record boundary discovery that does not have this restriction and that is also very effective as shown by our experiments. This method uses the data provided by the Data Matcher module to locate the record boundaries. As in Embley’s method, our approach first derives the HTML tree of the page. Using this tree, our method identifies record boundary separators by locating what we call a *minimal object holder subtree*, defined as follows.

Definition 6 Let $\tau : (\tau_1, \tau_2, \dots, \tau_n)$ be a tuple type and let C_i be the set of candidate attribute values of τ_i in a sample page g , such that each attribute $a \in C_1 \cup \dots \cup C_n$ is a leaf of a subtree of the HTML tree G of g . We define the **minimal object holder subtree** for τ as any subtree H of G that contains at least one instance of each type $\tau_1, \tau_2, \dots, \tau_n$, such that there is no subtree of H that contains the same instances.

Intuitively, each minimal object holder subtree corresponds to a text region that encompasses attribute values that will compose a single example object. Figure 7 illustrates examples of two minimal object holder subtrees for the CDNOW page presented in Figure 6. To avoid object holder subtrees that encompass more than a single object (which may occur due to failures in the data matching process), we discard any minimal object holder subtree for a type τ which has at

least 20% more descendants than the minimal object holder subtree with the smallest number of descendants for the type τ . The parameter 20% was determined empirically.

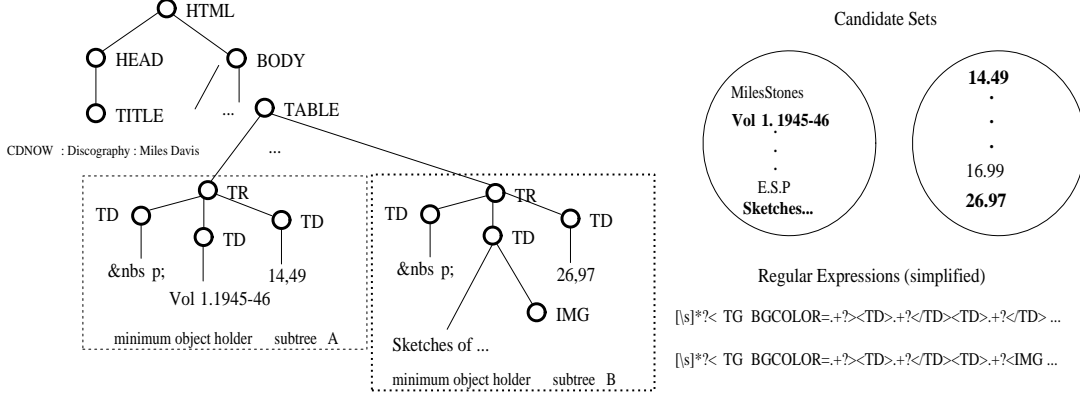


Figure 7: Locating record boundaries.

Once all minimal object holder subtrees have been identified, we derive (possibly many) regular expressions for matching such subtrees. For instance, analyzing Figure 7 we notice that each minimum object holder subtree may have some minor variations in its structure (notice the `` tag in subtree B). The algorithm derives one regular expression for each variation. These regular expressions are then used to determine separators in the sample page g (see Definition 6). Thus, the result of this step is a set s_1, \dots, s_m of separators. These separators guide the process of assembling the example objects, as described in the immediately following.

3.2.2 Assembling Examples

In this step, we take the sets of candidate attribute values and use them to assemble examples. For this, we use the separators found in the previous step as “guides”. In the simplest case, all that is required is identifying, for each separator, the attribute values occurring just after the separator and immediately before the position of the next separator. These attributes are then used to assemble a single example. As we are dealing with tuples, it is expected that at most one value for the attribute of a given type occurs between separators. There are other cases, however, in which more than one candidate attribute value occurs, because of possible false positives left by the Data Matcher module. Because of this, in the assembling step we also try to detect false positives that may have been generated by the Data Matcher module. For instance, when assembling tuples of the form $\langle \text{CDTitle}, \text{CDPrice} \rangle$, the value 15.99 shown in Figure 6 was mistakenly included as a value of CDPrice , and should be discarded in favor of the right value (11.49) for the CD instance being assembled. To detect such false positives, we rely on the regularity of the relative positions of the candidate avps and their respective separators. To measure such relative positions, we use the concept of *mark-up positions* as defined below.

Definition 7 Given an HTML source file f , the **mark-up position** of a string s , $\phi(s)$, is the actual offset of s in f minus the length of all visible text that occurs before s in f . In the same way, the **mark-up distance** between two strings s_1 and s_2 is given by $|\phi(s_1) - \phi(s_2)|$.

Figure 8 presents an example of how to calculate a mark-up position.

All the assembling is guided by the existence of the separators that were previously identified. Such separators play the role of *pivots* in the assembling process, as defined below.

Definition 8 Let $\tau : (\tau_1, \tau_2, \dots, \tau_n)$ be a tuple type and let C_i be the set of candidate attribute values of τ_i in a sample page g . For each attribute value $a \in C_1 \cup \dots \cup C_n$ we define the **pivot** for a as being the separator that immediately precedes it. More, precisely, a separator s is a pivot

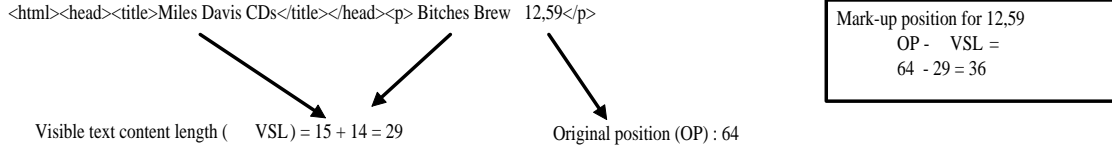


Figure 8: Calculating mark-up positions.

for an attribute value a if $\phi(s) < \phi(a)$ and if there is no other separator s' such that $\phi(s') < \phi(a)$ and $\phi(s) < \phi(s')$.

The assembling process is described by the algorithm in Figure 9. There, γ_i is the most common (in terms of the number of occurrences) mark-up distance between attributes of a given type τ_i and their pivots. If more than one candidate attribute value of the same type τ_i has a same pivot, we would assemble tuples with two attributes of the same type. To avoid this, we only consider one of them, which is the one presenting a mark-up distance more similar to γ_i . By doing so, we discard false positives that could have been generated by the Data Matcher module. In Figure 10, we illustrate this process. It presents an annotated version of the CDNOW page excerpt presented in Figure 6, where data to be assembled is disposed in columns. From this page, tuples of type CD must be assembled using candidate attribute values of types Title and CDPrice. In the top of the figure, we identify the mark-up distances between each candidate attribute value and their corresponding pivot. In this example, the candidate attribute value 15,99 of type CDPrice would be discarded in favor of the value 11,49, which presents a mark-up distance from their pivot that is more regular in comparison with the mark-up distance of the other CDPrice values to their pivots.

Example_Assemble
begin

Input:

A sample page g ;
 C_1, \dots, C_n , where each C_i is a set of candidate attribute values of type τ_i found in g ;
 s_1, \dots, s_m , where each s_j is a separator found in g ;

Output:

e_1, \dots, e_k , where each e_l is an example object;

Let $\gamma(\tau_i)$ be the most common mark-up distance from candidate attribute values of type τ_i to their pivots, $\forall i \in [1, n]$;

for $j = 1, \dots, m$ **do**

for $i = 1, \dots, n$ **do**

Let C_i^j be the set of candidate attribute values of type τ_i whose pivot is s_j ;

$a_i^j \leftarrow a \in C_i^j$ whose distance from s_j is more closer to $\gamma(\tau_i)$;

end

$e_j \leftarrow \langle a_1^j, \dots, a_n^j \rangle$;

end

end

Figure 9: Algorithm for assembling examples.

In the assembling process, there may be attributes that regularly present two or more values of the same type for a given pivot. This is specially true for numerical attributes. When faced with such a situation, the algorithm tries to infer the correct set of instances by comparing them (not their masks) with the ones present in the repository. If the direct comparison still does not provide enough evidence on which set of values to choose, the system generates examples with all attributes (by creating new, untyped attributes), and alerts the user of such situation.

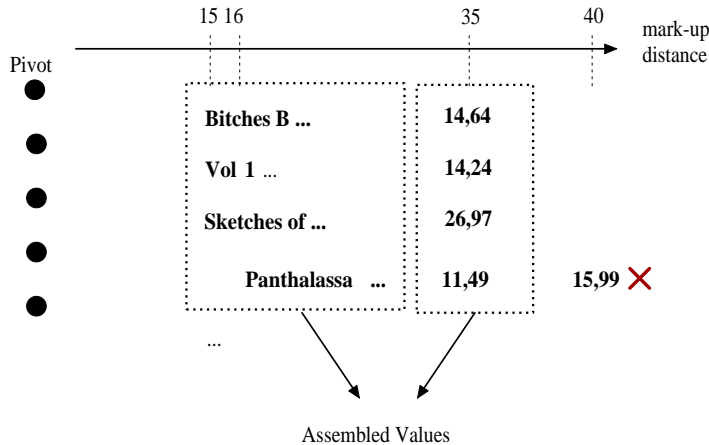


Figure 10: Assembling examples.

4 Experimental Results

To assess the effectiveness of our bootstrapping strategy, we performed experiments with 35 Web sites from 6 different domains: books, software, CDs, jobs, birds, and movies. Appendix A presents the complete list of the sites, which were selected according to the following criteria. Given a suitable domain, the top 5 sites listed in Google’s directory (which is ordered according to popularity) were chosen. In addition, two domains present in the RISE repository¹ were used to allow comparison with other strategies.

As we can see from Table 1, the results are very encouraging with an average of 97.5% of correct examples generated for each domain. Moreover, our record boundary discovery algorithm achieved an average of 98% of precision and 96.5% of recall for the web sites considered.

Domain	Number of Sites	Objects in the repository	Examples Generated		Boundaries Discovered	
			Incorrect	Correct	Precision	Recall
Software Stores	5	84	0	53 (100%)	100%	87%
Book Stores	5	53	0	148 (100%)	100%	100%
CD Stores	5	50	0	131 (100%)	99.8%	100%
Job Offerings	5	150	5	352 (98.5%)	88%	98%
Birds (RISE)	12	568	0	1784 (100%)	100%	100%
Movies (RISE)	3	1500	2	12 (86%)	100%	95%

Table 1: Results obtained with our bootstrapping and boundary discovery strategies.

In all experiments, for each domain, the repository was built by extracting data from one of the Web sites in that domain. Thus, the column “Objects in the repository” is the actual number of objects present in this site. Regarding the birds domain, we only consider the web sources containing entries with values for popular names and scientific names of birds (which were the attributes we extracted). From these, two were discarded because they had no content intersection with the other ones, thus our approach failed to generate any examples for those sites.

5 Applications

In this section, we present some applications of our bootstrapping strategy in the context of example-based data extraction. First, we characterize two important aspects of data extraction techniques.

¹<http://www.isi.edu/muslea/RISE/>

- *Target Page Set.* The set of pages that share the same formatting and structure characteristics of the ones used as sample pages when building a wrapper w . This set of pages can, therefore, be fed as input to w for data extraction.
- *Coverage.* The percentage of objects correctly extracted from a given target page set by a wrapper w .

5.1 Adaptive and Resilient Data Extraction

Using our bootstrapping strategy, we extended the DEByE approach [16] to provide resilience and adaptiveness in data extraction. Consider a wrapper $w = g(E, T_0)$ generated using a set of example objects E taken from a set of sample pages $T_o \subset T$, where T is a target page set taken from a Web source S . We define:

- *Resilience.* A data extraction process is said to be resilient if its coverage remains the same for any page set T' , composed by pages that are new versions of the pages in T , taken from the same Web source S . It is assumed that the HTML formatting features and the layout of the pages provided by S have changed and that the content (i.e., the objects of interest) of these pages remains the same.
- *Adaptiveness.* A data extraction process is said to be adaptive if its coverage remains approximately the same for any page set U , composed by pages taken from a Web source S' , distinct from S , but whose objects in it belong to the same domain of those present in T .

From the above definitions, we may say that the problem of adaptive data extraction is a generalization of the problem of resilient data extraction, since in both cases we have objects of a given domain implicitly present in pages with distinct layout and formatting features. In what follows, we present how we applied our bootstrapping strategy to provide adaptiveness and therefore resilience in our data extraction approach.

We can use our bootstrapping strategy to provide resilience in the following manner. Suppose that a wrapper was built for extracting data from Netcraft reports², such as the one shown in Figure 11(a). Using the data extracted from these reports as our source repository and a sample page from the new version of the report (shown in Figure 11(b)), our approach successfully generates new examples that can be then used for automatically building a wrapper for the new version³. Using the 71 objects present in the original report (Figure 11(a)), we managed to automatically generate a wrapper that could accurately extract all the 554 objects present in the new report (Figure 11(b)).

Moreover, using our bootstrapping strategy we can automatically adapt a wrapper built for a certain site for extracting data from other sites of the same domain. This is very useful, for instance, for electronic commerce brokers in which we want to compare data extracted from several sites of a given domain. We can start from a single wrapper and automatically build new wrappers for sites of the same domain by just providing sample pages from these new sites. Figure 12 illustrates this procedure for the bookstore domain.

To demonstrate the potential effectiveness of this approach, we performed an experiment using the bookstore and software store domains. First, using a pre-existing wrapper for the Amazon web site, we generated source repositories for books and software, with 53 and 84 objects respectively. Then, we applied our bootstrapping strategy to automatically generate examples and, therefore, new wrappers for four other bookstores web sites and four other software stores web sites. Table 2 presents the results achieved with these new automatically generated wrappers. From these results, we can see that the average extraction rates from these automatically generated wrappers are equivalent to typical levels achieved with wrappers built with user assistance [7, 14, 16].

²<http://www.netcraft.com>

³We did not address the *verification problem* [8, 10], which consists in determining that a given data source has changed its layout and that this change resulted in a poorly functioning wrapper.

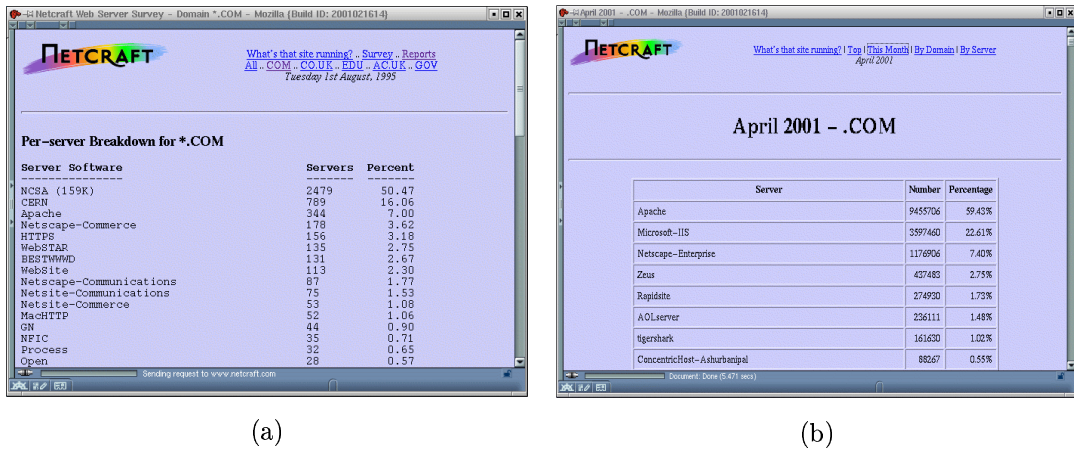


Figure 11: The two versions of the netcraft report.

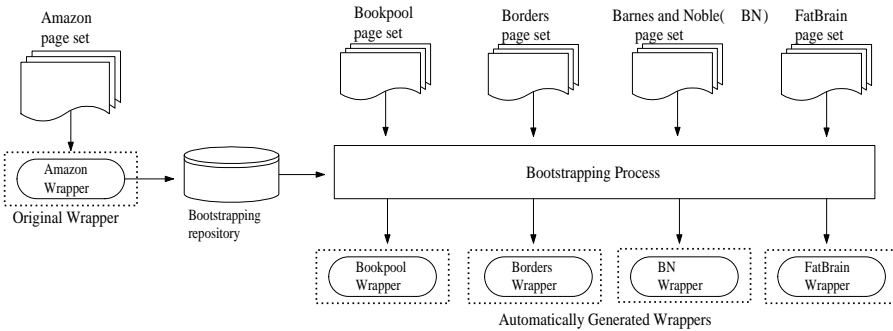


Figure 12: Automatically adapting wrappers for bookstores.

5.2 Improving Coverage Through Feedback

Our bootstrapping strategy can also be used for improving the coverage of an existing wrapper. Instead of requiring the user to provide more examples, we can automatically generate them by using a feedback process as described in the following. Consider a wrapper built for the Barnes and Noble (BN) bookstore presents low coverage. Using this wrapper, we can build a preliminary repository R_0 . Then, R_0 can be given as input to our bootstrapping process to build a wrapper for a new bookstore site (e.g., Borders), which can be used to generate a repository R'_0 . R'_0 is then fed back to our bootstrapping process along with the original BN pages, generating new examples and possibly a more accurate wrapper. Figure 13 illustrates the feedback process.

To demonstrate the effectiveness of such strategy, some experiments were performed using the bookstore domain. First, using the DEByE tool, we built a wrapper for each of the bookstores used in the previous experiments (Amazon, Barnes and Noble, Fatbrain, Bookpool and Borders) specifying just one example for each site. Then, we selected the wrappers that performed poorly. In this experiment, the BN and Fatbrain wrappers presented low levels of recall and precision⁴.

The feedback process was applied to these two sites using, for generating the source repository R'_0 , (a) only one site (Borders, whose wrapper performed best in our tests) and (b) all the sites. In each case, the generated a source repository R'_0 was used to build a new wrapper. By applying this process, we significantly raised the precision and recall rates in comparison to the initial wrappers. Figures 14(a) and (b) present the results for the BN and Fatbrain wrappers, respectively. Notice

⁴In this context, precision = number of correct objects extracted / total objects extracted and recall = correct objects extracted / total of correct objects

Domain	Source	Objects Extracted		
		Incorrect	Incomplete	Correct
Software Stores	EggHead	0	10	30 (75%)
	Software Outlet	0	0	60 (100%)
	Software and Stuff	0	0	5 (100%)
	Cdw	0	0	8 (100%)
Bookstores	Barnes and Noble	8	1	71 (90%)
	FatBrain	0	10	55 (85%)
	Borders	0	8	101 (93%)
	BookPool	0	0	5 (100%)

Table 2: Results obtained with automatically generated wrappers for software and book stores.

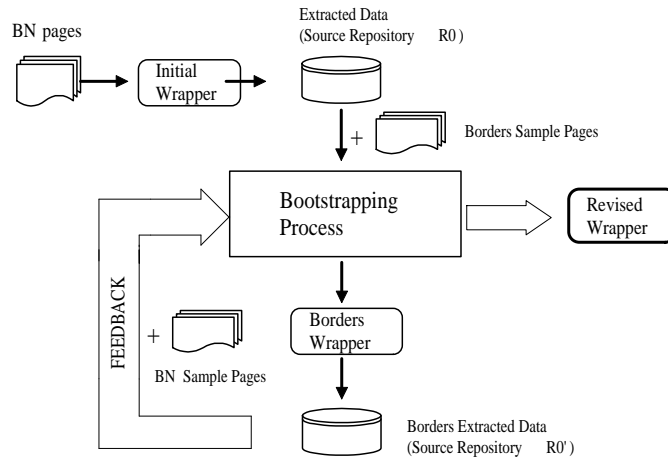


Figure 13: Feedback process for improving wrapper coverage.

that the benefit of using just one site (Borders) was identical to using all sites in both cases.

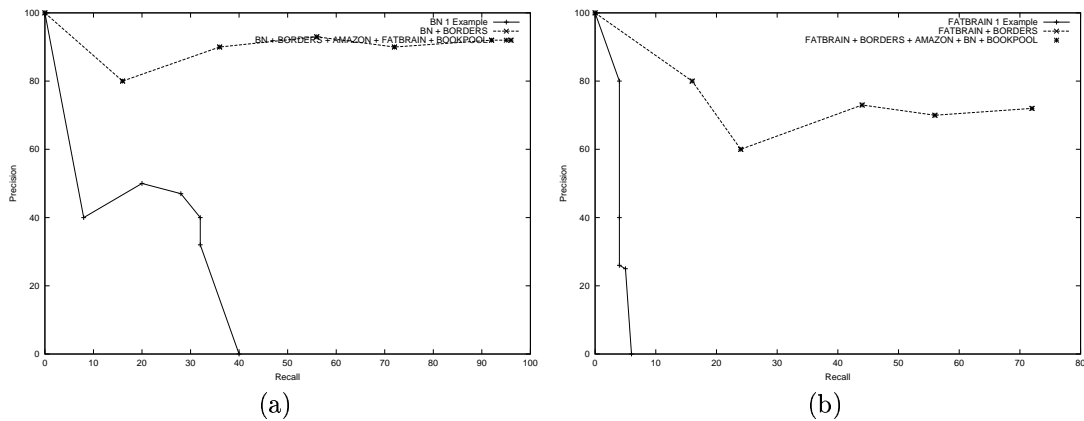


Figure 14: Precision and recall rates before and after the feedback process.

6 Conclusions and Future Work

In this paper, we describe a strategy for using a pre-existing repository containing data on a given domain to automatically generate examples for extracting data from Web sources on the same domain. We provide a number of results from experiments we carried out that show that this strategy performs very adequately in practical situations. We also discuss the application of our strategy for improving extraction rates and for providing resilience and adaptiveness for example-based generated wrappers.

It should be noted that, for the challenging issue of recognizing the data intersection between the initial source repository and the target Web sites, it is not possible to apply well accepted techniques for Web data integration (for example, the one presented in [5]) simply because in the target sites the attributes cannot be identified in advance. Similarly, phrase-based information retrieval techniques, such as the ones discussed in [2], cannot be applied because the necessary linguistic features are not usually present in data rich HTML pages.

In the present, paper we do not deal with nested structures that may appear in Web sources. As a future work, we intend to extend our strategy to treat such cases in a way similar to what is done in [12, 16].

Appendix A - Sites used in the experiments

- **Bookstores** Amazon.com (www.amazon.com), Barnes and Noble (www.bn.com), Bookpool (www.bookpool.com), FatBrain (www.fatbrain.com), Borders (www.borders.com).
- **Software Stores** Egghead (www.egghed.com), Software Outlet (www.softwareoutlet.com), Cdw (www.cdw.com.br), Software and Stuff (www.softwareandstuff.com), Amazon.com (www.amazon.com).
- **CD Stores** CD NOW (www.cdnow.com), CD Universe (www.cduniverse.com), Tower Records (www.cdw.com.br), 800.com (www.800.com), Amazon.com (www.amazon.com).
- **Job Offerings** Career Path (www.careerpath.com), Jobs.com (www.jobs.com), Career Site (www.careersite.com), Flip Dog (www.flipdog.com), Monster (www.monster.com).
- **Birds** 12 different sites available at the RISE repository.
- **Movies** 3 different sites available at the RISE repository.

References

- [1] ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 1999.
- [2] ARAMPATZIS, A. T., TSORIS, T., KOSTER, C. H. A., AND VAN DER WEIDE, T. P. Phase-based information retrieval. *Information Processing and Management* 34, 6 (1998), 1998.
- [3] BUTTLER, D., LIU, L., AND PU, C. A fully automated object extract system for the Web. In *Proceedings of the 21st International Conference on Distributed Computing (ICDCS-21)* (Phoenix, Arizona, USA, 2001).
- [4] COHEN, W. W. Recognizing structure in web pages using similarity queries. In *Proceedings of the 16th. National Conference on Artificial Intelligence (AAAI'99)* (Orlando, Florida, USA, 1999), pp. 59–66.
- [5] COHEN, W. W. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems (TOIS)* 18, 3 (2000), 288–321.

- [6] EMBLEY, D. W., CAMPBELL, D. M., JIANG, Y. S., LIDDLE, S. W., LONSDALE, D. W., NG, Y.-K., QUASS, D., AND SMITH, R. D. Conceptual-model-based data extraction from multiple-record web pages. *Data & Knowledge Engineering* 31, 3 (1999), 227–251.
- [7] HSU, C.-N., AND DUNG, M.-T. Generating Finite-State Transducer for Semi-Structred Data Extraction from the Web. *Information Systems* 23, 8 (1998), 521–538.
- [8] KNOBLOCK, C. A., LERMAN, K., MINTON, S., AND MUSLEA, I. Accurately and reliably extracting data from the web: A machine learning approach. *IEEE Data Engineering Bulletin* 23, 4 (2000), 33–41.
- [9] KUSHMERICK, N. *Wrapper Induction for Information Extraction*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1997.
- [10] KUSHMERICK, N. Regression testing for wrapper maintenance. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (Orlando, Florida, 1999), AAAI Press / The MIT Press, pp. 18–22.
- [11] KUSHMERICK, N. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence* 118, 1-2 (2000), 15–68.
- [12] LAENDER, A. H. F., RIBEIRO-NETO, B., DA SILVA, A. S., AND SILVA, E. S. Representing Web Data as Complex Objects. In *Electronic Commerce and Web Technologies*, K. Bauknecht, S. K. Mandria, and G. Pernul, Eds. Springer, Berlin, 2000, pp. 216–228.
- [13] LIU, L., PU, C., AND HAN, W. XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources. In *Proceeding of the 16th International Conference on Data Engineering* (San Diego, USA, 2000), pp. 611–621.
- [14] MUSLEA, I., MINTON, S., AND KNOBLOCK, C. Wrapper Induction for Semistructured, Web-based Information Sources. In *Proceedings of the Conference on Automatic Learning and Discovery CONALD-98* (Pittsburg, PA, 1998).
- [15] MUSLEA, I., MINTON, S., AND KNOBLOCK, C. An Hierarchical Approach to Wrapper Induction. In *Proceedings of the Third Annual Conference on Autonomous Agents* (Seattle, WA, 1999), pp. 190–197.
- [16] RIBEIRO-NETO, B., LAENDER, A. H. F., AND DA SILVA, A. S. Extracting Semi-Structured Data Through Examples. In *Proceedings of the Eighth ACM International Conference on Information and Knowledge Management - CIKM'99* (Kansas City, MO, 1999), pp. 94–101.
- [17] RILOFF, E., AND JONES, R. Learning dictionaries for information extraction by multi-level bootstrapping. In *Proceedings of the 16th. National Conference on Artificial Intelligence (AAAI'99)* (Orlando, Florida, USA, 1999), pp. 474–479.