Abstract

Many online data sources, such as product catalogs, online directories, etc, are available on the web. Extracting information from such sources is a hard task since these sources are designed to be presented to human users. Many researchers have tackled the problem of building wrappers for such sources. The state of the art approach is to use machine learning techniques based on fully labeled example pages. In this paper we propose and study an approach based on example instances. This allows the user to build a wrapper using only a handful of examples of the whole source allowing to take into account structural differences. The patterns obtained allow to extract the instances of the relation described by the examples and contained in the same data source.

1. Introduction

The main motivation to wrapper generation for web sources is to give computer programs access to generated web content. Once extracted and restructured the wrapped data can be used by any other data-based application such as mediators, online agents, etc, or to new applications giving a new and higher level view of web content. The need for automatic wrapper generation methods comes from the fact that the web is in constant change and that hand-crafting wrappers is known to be a tedious task. Therefore many automatic information extraction methods such as wrapper induction, [6, 7, 5], structure extraction [3, 4, 1] or relation extraction from the web [2] have been studied.

Existing wrapper construction techniques are based on techniques such as labeled page based induction, unsupervised structure discovery, and knowledge-based wrappers. We propose an approach which is example based. However the examples are not labeled pages but rather example instances of the relation to extract. Therefore all the examples of a page need not be given. We will see in fact that only two or three examples of the different formats which appear in a page are required. This relieves the user to fully labeling pages making the wrapper construction more efficient. In the case of pages containing 30 results we only require the user to give 3 to 4 instances where approaches requiring 3 to 4 fully-labeled pages would require having 90 to 120 instances.

In this paper we present and analyze an approach to wrapper construction which is based on the generalization of the contexts of occurrence of a small set of example instances the user wishes to extract from a given web source. We compare this work to other existing approaches. Furthermore we give an indication on how to choose examples in order to rapidly obtain an effective wrapper.

This paper is organized as follows. Section 2 describes an approach based on the generalization of the context of occurrence of example instances. Section 3 gives an evaluation of our method. We compare this work to existing techniques in section 4 and conclude in section 5.

2. Context generalization

The approach we take is based on the generalization of contexts. We consider the user wishes to extract a given relation from a given source. To describe this relation the user only needs to give a small set of example instances. Having instances as examples relieves the user from having to fully tag example pages while still precisely defining the desired relation.

Building the extraction patterns allowing to extract a given relation for a source can be done in a three phase process: (1) a preprocessing phase, (2) a context extraction phase and (3) a pattern generalization phase. In the preprocessing step the selected documents of the source are
cleaned and encoded. This allows to have a more uniform format to work on in the discovery phase. In the context extraction phase the user given example instances are searched for in the documents of the target source. Their contexts are then extracted as the first patterns of the pattern generalization phase. The set of resulting patterns is initialized by the contexts of the given example instances. Once the result set is initialized a mining step can be entered which consists in generalizing these contexts into more general patterns representing the relation the user wishes to extract.

2.1. Basic Concepts

The basic concepts behind wrapper generation are described here. A web document \( D \) is a sequence of tokens. Each token is a string of characters. A token is either a tag or non-tag. A web data source \( S \) is a set of web documents formatted in the same manner. A relation \( R \) is a set of tuples representing a relationship between objects of the universe of discourse. In our case this universe is the set of objects described by the source (for example books, prices, people, e-mails, …). Each tuple \( t \) is a finite, ordered sequence of objects. An instance of \( R \) is one of the tuples belonging to \( R \). An occurrence of a tuple \( t = (v_1, \ldots, v_n) \) in a document \( D \) is a tuple \( o = (s_1, \ldots, s_n) \) where each \( s_i \) can be represented by a pair \((a_i, b_i)\) such that the substring of \( D \) between positions \( a_i \) and \( b_i \), noted \( D[a_i, b_i] \) is equal to \( v_i \). This allows to have a representation whose size is independent of that of the value length. Since a same substring can occur more than once, the instance of a relation can have zero or more occurrences. An instance for which an occurrence can be found in \( D \) is said to appear in \( D \). Furthermore a set of instances for which all its elements appear in a document \( D \) is also said to appear in a document \( D \). A \( k \)-pattern \( p \) is an expression representing a set of occurrences. It is a \((k+1)\)-tuple of parts. Each part represents the text found before the first value between two values or after the last value of an occurrence. Each part of a pattern is divided into tokens. A token can either be a string of text which should match exactly in the document, a tokenizer which matches any substring of content text, or a tag which should match exactly. As we will see later we distinguish content text from tag text. A part of a pattern can either match content text or a tag, but not a substring containing both at a time. The objective of a pattern is twofold: firstly find the occurrences in the page as whole pieces and secondly delimit the different values.

Formally a wrapper \( W_{S, R} \) can be considered as function having as its input a document \( D \in S \) and as output the subset of instances \( R_D \) of a relation \( R \) which appears in \( D \). We consider that the source \( S \) is simply a set of documents sharing the same structure, and that our relation is the union \( R_S = \bigcup_{D \in S} R_D \) of the set of instances appearing in reach of the documents \( S \).

With these definitions, a multi-pattern wrapper \( W_{S, R} \) can then be defined as a set of patterns \( P \) which each allow to extract subsets of \( R_S \) from the documents in \( S \). Each pattern of \( P \) is said to cover a subset of instances of \( R_S \). A multi-pattern wrapper is perfect when the generated patterns together cover all the instances of \( R_S \). It is said coherent when no more than the instances of \( R_S \) are covered. Respectively recall and accuracy allow to measure the perfection and coherence of a multi-pattern wrapper.

2.2. Preprocessing phase

A first step in our method, is to put the documents in a normalized format by cleaning white spaces and abstracting tags into unique tokens. Furthermore in this step low level tags such as \((b, i, em, a, etc.)\) are removed. Such tags can be removed since they do not contain useful structural information. This step results in an encoded document.

2.3. Context extraction phase

In order to build a wrapper we consider the user has given a set \( E \) of examples of a relation he wishes to extract from a set of document \( S \). For each example tuple \( t = (t_1, \ldots, t_n) \), each encoded document string \( D \) is searched linearly to find each of the \( t_i \) items of \( t \). The context of \( t \) is tuple formed of the substrings of \( D \) respectively before \( t \), between each \( t_i \) and \( t_{i+1} \) in the same order, and after \( t \). Each substring can be represented by a pair of indexes representing the beginning and end of the substring in the encoded input document. For execution speed reasons and space usage, not all the text before and after need to be saved and further more if the text between two parts is exceeds a given length, then current occurrence of \( t \) is considered to be erroneous. In theory, these optimizations would not extract all the contexts and the front and back strings are returned incomplete. However in practice this does not change the limits provided the maximum context length be big enough.

Let us consider the space of \( k \)-patterns \( P \). We can define a generalization operator over this space. If \( p = \langle p_0, \ldots, p_k \rangle \) and \( q = \langle q_0, \ldots, q_k \rangle \) are \( k \)-patterns then generalizing \( p \) with \( q \) consists in generalizing \( p_i \) with \( q_i \) for each \( i \in [0, k] \). Recall that the parts \( p_i \) are sequences of tokens. In order to generalize these parts we need to define a generalization over the token space. Let \( g_T \) be the generalization operator over the token space. The generalization of two tokens \( t \) and \( t' \) can be defined as follows. If \( t = t' \) then \( g_T(t, t') = t \). If \( t \) and \( t' \) are both tag tokens but not equal then \( g_T(t, t') = \top \) where \( \top \) represents failure. If \( t \) and \( t' \) are both text tokens but are not equal then \( g_T(t, t') = * \) where * represents any text token. The generalization of parts depends on the position of the part in the pattern to which
they belong. In order to generalize two front (resp. back) parts we only need to find a sequence which generalizes a suffix (prefix) of each of the two parts. However this suffix (resp. prefix) may not be empty. In the case of middle parts they need to be of the same length. The generated generalized part will be a sequence of tokens of the same length. The token found at a position $j$ in the generalized sequence is the result of the generalization of the tokens found at the same position in the two sequences being generalized.

The pattern generalization method is reflexive, associative and commutative. Therefore generalizing a set of patterns is can be done by generalizing pairs in any order. This allows us to have an efficient algorithm consisting in building the most general patterns different from $\top$. To do this we choose a first context and generalize it with all the contexts for which the generalization does not lead to $\top$. Then another context is chosen an generalized with the remaining contexts. This is repeated until all contexts have generalized with all other contexts.

3. Evaluation

We have implemented our method in a system called IERel (Information Extraction of Relations). The results we obtain are given in figure 1. The columns respectively contain the source names (Source), the number of given examples (Ex.), instances considered (Inst.), instances retrieved (Retr.) and finally the recall (Rec.) and the accuracy of the generated wrapper (Acc.). In all cases every extracted instance is correct. This can be explained by the fact that our generalization methods prefers multiple specific patterns over too general ones. The recall is less than 1.00 when the given examples were not sufficient to learn a wrapper extracting all the data. This is due to the multiplicity of formats in which a source may present its results.

In order to have an idea of the sensitivity of our method to the number of examples we estimated the number of examples instances necessary to extract all the instances from a set of 7 pages containing 99 instances from the Amazon DVD source. We proceeded with the following experiment. We setup a program which randomly chooses one instance from the instance set and uses it as an example, applies the generalization method and uses the obtained patterns to extract the covered instances. These instances were removed from the instance set. Then another example is chosen from the remaining set and added as an example. We further generalized the patterns we had previously obtained. The procedure stops when a perfect wrapper has been built (ie. when no remaining instances are available). We run this program four times generating the results show in figure 1. This plot shows the evolution of the number of extracted instances in function of the number of given examples.

We also added to the plot the results obtained by randomly choosing a new example out of the whole instance set (ie. without removing covered instances). This gives an idea of the role the quality of the examples play on our generalization method. The plateaus on this last plot correspond to the addition of useless examples. Such plateaus do not appear in the other four plots. This shows that once an instances is covered it is useless to add it as an example. The plots show that in the case of the Amazon source about 13 to 20 examples are necessary to learn a perfect wrapper. This corresponds to about twice the number of formats present on the page.

In each of the 5 runs of the previous experiment generated a specific set of examples allowing to obtain a perfect
wrapper. Given the property that our generalization gives the same patterns for a given set of examples, we wanted to show that however, some examples are of a better quality than others. To show this we took the set of examples used in the fourth run (the one having the biggest number of examples) and proceeded to evaluate the generalizations on random orders of the used examples. The plot of figure 2 results from this experiment. The plot named recall is the same as that of the plot of the fifth run in figure 1. This experiment shows that the quality of examples influence how fast the pattern will be found. Run 3 shows that out of the 9 examples only 5 are necessary to build the perfect wrapper.

![Figure 2. Quality of the examples](image)

### 4. Comparison with other work

Our work mostly compares to wrapper induction based on labeled pages. Indeed we also use an example based approach. However our approach differs to existing methods [6, 7, 5] in that it does not require giving fully tagged pages as examples. We claim that it is only necessary to give a handful of example *instances* for each format appearing in a web source. Therefore pages with only one format such as Okra only four or five examples instances are necessary to learn a wrapper. On such a source the current state of the art wrapper induction method reported using two example pages. Since each page contains 50 results this means hand labeling 100 instances instead of just five in our case.

Structural discovery based methods [4, 3] are *a priori* unsupervised. However such methods generate a set of patterns from which the user has to choose the ones which respond to his needs. Furthermore the patterns may extract data which the user is not interested in. In such cases the user has to further set up a procedure allowing to select the data he wishes to extract. Also the structured data which is extracted only has anonymous attributes. It is left up to the user to label these attributes. In our method the informational needs of the user are expressed as example instances of the relation he wishes to extract. Once the wrapper built, no further treatment is necessary. Indeed the built wrapper extracts the data in a known format (the same as the example instances) and no pattern selection is required.

### 5. Conclusion

In this paper we presented a method to build a wrapper for a given web source. The proposed method is based on a three phase process of document preprocessing, context extraction and pattern generalization. The gain of the method when compared to existing wrapper induction methods is that we do not require the user to manually label a set of examples. We only require the user to give a handful of example instances of a relation to extract from the source. The proposed method has nice properties allowing for an efficient algorithm. We also show that some examples are useless in the wrapper construction phase. We show that using an incremental learning strategy also to build the pattern set making the wrapper in an efficient manner.

### References


