A Domain Meta-wrapper Using Seeds for Intelligent Author List Extraction in the Domain of Scholarly Articles

Francesco Cauteruccio and Giovambattista Ianni

Dipartimento di Matematica e Informatica,
Università della Calabria,
I87036 - Rende (CS) - Italy

Abstract. In this paper we investigate about automated extraction of author lists in the domain of scientific digital libraries. It is given a list of known "seed" authors and we aim to extract complete lists of co-authors from Web pages in arbitrary format. We adopt a methodology embedding domain knowledge in a unique "meta-wrapper", not requiring training, with negligible maintenance costs and based on the combination of several extraction techniques. Such methods are applied at the structural level, at the character level and at the annotation level. We describe the methodology, illustrate our tool, compare with known approaches and measure the accuracy of our techniques with proper experiments.

1 Introduction and Motivation

The world research community has nowadays great interest towards numerically estimating the impact of whole institutions, research authors and individual papers thereof. Digital libraries specialized in archiving scholarly data play a significant role in many national processes of evaluation of research quality [18, 23, 20] and, in order to ease this process, involving also quantitative analysis, many search engines for scholar literature introduced bibliometric tools in their online interfaces [25, 26, 22, 21]. Also, this stimulated the recent focus towards extracting structured and semi-structured information from scholar digital libraries. For instance, there are many systems which perform automated bibliometrics estimates computed on top of the information provided by the Google Scholar search engine [1, 2, 24, 19] and this information is used daily by evaluators, experts and researchers worldwide.

The Google Scholar portal, despite its inaccuracies and vulnerabilities to spam [3], is renowned for its recall. Its service offers neither an API nor any automated data extraction means, however: specialized wrapping techniques for extracting information are thus necessary. Also, the available information is incomplete: for instance, author lists are partial (see Fig. 1). Incomplete lists prevent an accurate estimate of bibliometric indices which take into account normalizations based on the number of co-authors of a given paper, such as

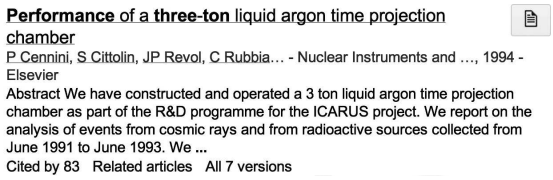

Fig. 1. An item from a Google Scholar output page

the *multi-authored h-index* [4]. However, this category of metrics is required by communities which tend to extend authorship to thousands of researchers[1].

Our Scholar H-Index Calculator (SHI in the following, [24]), allows users to interact using their browser with the Google Scholar web site and result pages are transparently enriched with bibliometrics information related to the displayed page hits, starting from the well-known h-index [7], to more specialized ones. The tool includes many specialized features for advanced bibliometrics analysis, including the possibility of defining custom formulas and the recently introduced capability to refine incomplete author lists, which is the subject of this paper. The problem of reconstructing full author lists can be solved by visiting web pages containing paper descriptions linked by Google Scholar: these description pages normally contain detailed author lists, but are in arbitrary HTML format, whose structure depends on the digital library the paper at hand has been indexed from (see Figure 2). This makes the issue of extracting complete author lists non-trivial and connected to the more general issue of reconstructing lists of similar items from documents in arbitrary format [8]. In order to face the above issue, we developed a sub-module of our tool, implementing what we called TPI-S techniques (*Tree Pattern Induction with Seeds*). The contributions of this paper are the following:

- we suggest a "meta-wrapper" approach aimed at completing partial information from scholar digital libraries. To this end we present an ensemble of unsupervised, domain-oriented information extraction techniques called TPI-S (Tree Pattern Induction with Seeds) targeted at reconstructing full author lists appearing in arbitrary paper description pages. These techniques need only a small amount of input information (seed data, i.e., incomplete author lists), do not require user interaction nor preliminary bootstrapping and are independent from the page structure of specific digital libraries. TPI-S work:
  1. at the structural level, by using a method called XPath Resolution (XPR), in which proper XPath expressions are automatically generated in order to reconstruct author lists;

[1] An analysis of the impact of multi-authorship in in the Physics field is reported in [6]. As an example, see the complete author list of [5] available at `http://iopscience.iop.org/1748-0221/3/08/S08003`.

   2. at the character level, by using a method called Text Node Resolution (TNR), in which, whenever author lists are detected to be within text content, appropriate regular expressions are generated;
- the ability to automatically reconstruct full author lists on top of a scholarly digital library with large recall, enables the possibility to perform accurate measurement of bibliometric impact indices, over a large corpus of indexed data, even when such measures depend on the number of co-authors and considerably reducing the burden of manual assessment;
- TPI-S techniques have been implemented as part of our bibliometric analysis tool; they are available to the end user and/or can be also exploited for the automated analysis of the scientific production of large sets of researchers; the accuracy of the approach is proven with appropriate experiments, showing that this type of analysis is feasible in practice, over very large corpuses of authors/documents.

Note that other approaches could be applied to the context at hand, like designing/training and maintaining a large collection of ad hoc wrappers. Such collection could complete information by accessing a few known online scholarly digital libraries. None of these libraries has however coverage comparable with Google Scholar at the moment of writing: we thus opted for designing a "meta-wrapper" which embeds domain oriented knowledge (like, generic knowledge about how author lists are usually encoded and how they appear within markup, etc.) and that is not tailored at specific digital libraries. The benefits of taking this approach are many and can be generalized to domains other than the one at hand:

1. Information is extracted at run-time, by generating on-the-fly a proper XPath or a regular expression working on the specific document at hand;
2. It is not necessary to take the burden of designing a battery of information extraction wrappers, nor it is necessary to allocate resources for training wrappers over sample instances at design-time;
3. When a new digital library enters the collection of interest, or it changes its HTML shape, it is not necessary to update the TPI-S module, nor it is necessary to undergo a new training stage at design time;

The structure of the paper is as follows: in the next Section we overview the problem we are addressing. Then we describe in apposite sections our extraction techniques and we discuss related work. Eventually we report about our evaluation and draw conclusions.

2 The TPI-S Methodology

We assume to work with strings from a fixed alphabet Σ. A *document* D is an *ordered tree* with fixed root r, for which each node (also called *element*) is a string $s \in \Sigma$, with attribute names and values $(k_1, v_1), \ldots, (k_m, v_m)$ and a ordered list of subnodes $n_1, \ldots, n_j (m, j \geq 0)$. This formal definition mimicks

actual DOM models: we assume nodes containing text (*textual nodes*) are part of D as well as *tag nodes* (i.e., elements associated to an allowed HTML tag). Whenever we refer to the *textual content* of a node $n \in D$, we intend the string obtained by concatenating all the textual sub-nodes of n ordered by means of a leftward depth-first visit of the subtree rooted at n. In the following we assume we are given a document D, a *seed list* of strings L and a *hidden list* $L^* \supseteq L$. L^* and L are connected by some semantic relationship, which is usually defined in a informal way only (e.g., *lists of recipes' ingredients, etc.*). In our setting, we look for lists of co-authors of scientific articles. In general, hidden lists are encoded within documents according to two main categories: either *i)* elements of L^* appear as individual nodes in a DOM structure (i.e., a HTML table, a list, etc.), or *ii)* L^* appears as encoded within the same textual node (e.g., a textarea, the text content of a single `<span>` node, etc.). Accordingly, our approach combines two techniques: (i) *XPath Resolution (XPR)*, in which we aim at reconstructing a XPath expression which captures all and only the elements of L^*; (ii) *Text Node Resolution (TNR)*, in which L^* is assumed to be encoded within a unique textual element of D, thus requiring to work at the character-level.

2.1 XPath Resolution

The XPR method aims at constructing a XPath expression E which should capture L^* over D. We argue that L^* is encoded in D respecting some *structural uniformity*. In particular, one can observe that usually *i)* lists of co-authors can be found in the textual content of a group of nodes each having the same distance from a common ancestor node, called *Lowest Common Ancestor* (LCA); and *ii)* among the sub-nodes of the LCA, those having the same level depth of seeds and correspond on a subset of their attributes values, usually contain remaining hidden authors names.

For simplicity, we assume $|L| = 2$, i.e., $L = \{l_1, l_2\}$. As a background domain information, we assume seeds are given in the same sequence as they should appear in D and that a single seed is in the format "ABC $\text{LastName}_1 \ldots \text{LastName}_N$", as it is the case when partial information is provided from Google Scholar output. For instance, we can have l_1 = "AM Turing" and l_2 ="J von Neumann". Given a seed string a, let $\mathit{fname}(a)$ and $\mathit{lname}(a)$ be respectively the initials of first names of a and its last name(s). In order to match occurrences of seeds within D, we take into account the fact that author name encodings might differ in how first names are presented (e.g., with initials only, with some of the first names only, etc.), while last names have usually a fixed appearance.

The XPR method is based on enumerating a set of XPath candidate expressions. Let L_1 and L_2 be the lists of nodes of D in which, respectively, $\mathit{lname}(l_1)$ and $\mathit{lname}(l_2)$ are contained in some text sub-node. Algorithm 1 (*computePath*) takes in input D and a couple of candidate elements $e_1 \in L_1$ and $e_2 \in L_2$. When it succeeds, *computePath* outputs a couple (p, R) where p is a candidate XPath expression and R is a candidate value for L^* (a list of textual values, obtained applying p to D). The outcome of the XPR method is a XPath expression p_{max} for which the condition $\mathit{score}(p_{max}) = \max_{e_2 \in L_2}^{e_1 \in L_1} \mathit{score}(\mathit{computePath}(D, e_1, e_2)) = 1$

```
Input  : document D, elements e1, e2 ∈ D
Output: A string path identifying a XPath expression and the result set of path when
        applied to D
route ← [ ];
t1 ← e1;
t2 ← e2;
balanced ← true;
while t1 ≠ t2 and balanced do
    push(route, t1);
    if hasFather(t1) and hasFather(t2) then
        t1 ← father(t1);
        t2 ← father(t2);
    else
        balanced ← false;
    end
end
if balanced then
    push(route, t1);
    head ← path from fixed root r to t1;
    while x ← pop(route) do
        append(path, nodeToXPath(x));
    end
    return (head · path, xpath(head · path));
else
    return null;
end
```

Algorithm 1: computePath

holds. The function *score* assigns a value w to a couple (p, R), returned by Algorithm 1, according to the fraction of seeds appearing in R. We set our threshold to 1 since we have found that XPath expressions not capturing all the seeds were of very low quality and thus worth discarding. Note that elements of R are compared with seeds only with respect to last names, after eliminating uppercases and after flattening accented letters.

In detail, *computePath* makes an upward traversal of D starting from e_1 and e_2: if e_1 and e_2 are at the same depth (i.e., they are "balanced") then we find their LCA, and we build a proper XPath expression p aimed at following a path which traverses the LCA and then captures neighbors at the same level and with similar attribute values of e_1 and e_2. The parts of p which describe the subtree below LCA are constructed with limited predicate filtering. For instance, *id* attribute values are usually different from an hidden element to another and thus excluded from filtering conditions, while *title*, *class* and *name* values can be used for filtering. The *head* sub-path is instead built using XPath index predicates and a stricter attribute filtering, in order to better tailor a unique path from the document root to the *LCA* of e_1 and e_2.

An example for XPR. An instance example is shown in Figure 2, which depicts a portion of a Web document encoded in HTML and containing a list of authors of a scientific publication. We have 2 seeds, l_1 = "AM Turing" and l_2 = "J von Neumann". The candidate elements e_1 and e_2 for which we achieve the maximum value of *score* are framed with a solid line, where the dotted frame

Fig. 2. An example for Algorithm 1.

encloses an unknown element h_1. When running *computePath* over the above document and e_1, e_2 as input, we obtain: (i) a stack *route* containing [`tbody`, `tr`, `td`, `a`], where $LCA(e_1, e_2)$ = `tbody` and (ii) a corresponding XPath expression `/tbody/tr/td/a[@title='Author']` which is then concatenated with the proper *head* expression. Note that unbalanced couples of candidates are filtered and excluded from scoring, consider e.g., the couple w_1 and e_2. Also, note that XPR is elastic with respect to changes in the substructure of elements candidate for matching seeds: for instance, e_2 would be unbalanced with respect to e_1 whenever surrounded by some additional markup (e.g., some nesting within a `<div>` element, etc.). In this case, we would achieve the maximum score with the couple of elements w_1 and w_2, still correctly extracting h_1. We found that, when XPR was applicable, structural uniformity was respected by about 100% of the experimental corpus (see Section 4). When not respected (think e.g., at the element e_2 surrounded by a `<div>` element), this corresponded to a scenario in which the culprit element was not actually an author item, thus correctly excluded. Note that optional footnotes (e.g., superscripts referring to the corresponding author) are usually appended using same-level nodes like `<sup>`: these follow structural uniformity and are correctly filtered out from text extracted by XPR.

2.2 Text Node Resolution

Whenever XPR can't be applied, we assume that L^* is not encoded in D according to structural uniformity, usually because the author list is encoded as raw text with little or no markup. We thus work at the character level, i.e., processing the textual content l_n of a single element of $n \in D$. Author lists have often a specific structure in which names are separated by some fixed delimiter, except the last element, which is easy to be confused with a portion of a last name: think e.g,. at the list "AM Turing, GW Leibniz, J von Neumann and N Tesla.". Note however that it is not possible to build a simple extraction module which

```
Input  : Textual element t, list L = {l_1, ..., l_n} of seeds
Output: An enumeration I of textual elements
t_b ← t;
result ← [ ];
er ← elementRegexp(t, L);
if er ≠ null then
    for |L| times do
        t_b = matchAndSubs(t_b, er, "#");
    end
    if occurrences("#", t_b) > 1 then
        let h_1, h_2 be the indices of the first and second match of "#" in t_b, respectively;
        gr ← "(.*?)" · buildRegexp(t_b[h_1 + 1, h_2 − 1]);
        extract(t, gr, er, result);
    end
end
return result;
```

Algorithm 2: computePattern

relies on looking for standard separators like "`,`" and "`and`". We found indeed that author lists are represented in a quite heterogeneous number of ways, think e.g,. at "G. Paratinik, M.Sc. and D. Knuth, Ph.D.", a usual case in fields other than computer science.

To this end, we generate and then try to apply two types of patterns, expressed in terms of a regular expression: one is aimed at describing author names (the *element regular expression*, or EREG), and the other aimed at describing groups of delimiters between names (the *glue characters regular expression* or GREG). We attempt to extract author names both by means of matches of the generated EREG, or extracting the text appearing in between two matches of a GREG.

Let t and s be two strings; we take advantage of the following notational conventions: (i) $|s|$ is the length of s; (ii) $s[i]$ denotes the character in the i-th position of s ($0 \leq i \leq |s|-1$); (iii) $s[i,j]$ denotes the substring $s_{ij} = s[i] \cdot \cdots \cdot s[j]$ ($i \leq j$); (iv) the function $search(t, s)$ returns a value $i, 0 \leq i \leq |t| - 1$ such that i is the starting position of the first match of s in t or is undefined otherwise; (v) $t \cdot s$ is the concatenation of two strings (or two regular expressions); we use the common fragment of the Perl and Javascript notation for denoting regular expressions.

Let $L = \{l_1, \ldots, l_n\}$ be the set of available seeds; we iterate over elements $t \in D$, which contain in their text content the value $lname(l_1)$. The algorithm *computePattern* takes in input a candidate string t and a list L of seeds, builds a EREG and a GREG, matches them properly and outputs a candidate hidden list. We take advantage of the intermediate function *buildRegexp*(t), which takes in input a string t, and outputs a regular expression r which matches t. r is generalized enough to match recurring patterns corresponding to the example string t: we build it by aggregating over alphabetical, numerical and spacing characters, while glue characters are aggregated independently. As an example, *buildRegexp*("`, Alan Turing`") = `/,\s[A-Z][a-z]+\s[A-Z][a-z]+/` . The behavior of Algorithm 2 is:

```
Input  : String t, elements a, b
Output: A regular expression
i_a ← search(t, lname(a));
if i_a > 0 then
    dright ← nextGlueCharacter(t, i_a);
    dleft ← lookBehind(t, dright, t[dright]);
    if dleft > 0 then
        s ← buildRegexp(t[dleft +1, i_a − 1]);
    else
        s ← buildRegexp(t[0, i_a − 1]);
    end
    s ← s · buildRegexp(t[i_a, i_a + |lname(a)|]);
    return s;
else
    i_b ← search(t, lname(b));
    if i_b > 0 then
        s ← buildRegexp(t, |lname(a)|);
        c ← previousGlueCharacter(t, i_b);
        r ← buildRegexp(t[|lname(a)|, c]);
        return s · r;
    end
end
return null;
```

Algorithm 3: elementRegexp

1. first a EREG *er* is obtained from the output of the function *elementRegexp*: as shown later, this function exploits seeds occurrences and looks for surrounding delimiters in order to build a regular expression which should match a possible occurrence of a name in t.
2. if *er* is not null, the algorithm uses it to iteratively match all the seeds $l_i \in L$ $(1 < i \leq n)$ in t_b. The function call $matchAndSubs(t_b, er, \text{“\#”})$ applies *er* to t_b replacing a match with a placeholder character “#”.

   For instance, suppose that we have t_b = “`Turing, Leibniz, von Neumann, Nash.`” and $L = \{$“`Turing`”, “`Leibniz`”$\}$. *elementRegexp*(“`Turing`”) returns a regular expression *er* = `/[A-Z][a-z]+/`; when applying the *matchAndSubs* function $|L| = 2$ times, t_b is modified and becomes “`#, #, von Neumann, Nash.`”; If *er* has been matched at least twice, we proceed as follows:
3. it is created a GREG *gr* in the form `/(.*?)`re'`/` where $re' = buildRegexp(t_b[h_1 + 1, h_2 - 1])$, for h_1 and h_2 be the position of the first and second “`#`” placeholder in t_b. *gr* is intended to match text between the two occurrences of `#` in t_b (excluding the character “`#`” itself). Following the previous example, we would have re = `/(.*?),\s/`.
4. the *extract* function eventually extracts the set of authors by iteratively applying either *gr* or, if *gr* fails to match, *er* to t and removing the match found from t itself.

The output of the Algorithm 2 is a vector *result* representing L^*, when t is assumed to be the element containing L^*.

The behaviour of the *elementRegexp* function is described in Algorithm 3 and is as follows:

Fig. 3. An input example for Algorithm 2 and the same input after application of *matchAndSubs*.

1. we search an occurrence of $lname(a)$ in t; if $lname(a)$ is a substring of t, we distinguish the two cases in which t starts with $lname(a)$ or not. Let $i_a = search(s, lname(a))$;
2. if $(i_a > 0)$ we assume that $lname(a)$ appears in t surrounded by two glue characters k_1 and k_2. Starting from the position i_a, we search rightwards for a *glue character*: let *dright* be the index of the first glue character found; then we scan t leftwards using the function call $lookBehind(t, dright, t[dright])$ which, given in input a string t, a position *dright* and the character $t[dright]$, searches, starting from position $dright-1$, a further occurrence of the same glue character. In the case $dleft > 0$, the positions *dleft* and $i_a + |lname(a)|$ give the boundaries of the pattern s we want to build, otherwise we use 0 as left boundary position.
3. when $i_a = 0$, we search the first occurrence of $lname(b)$; if $lname(b)$ is a substring of t starting at position i_b, we build the generalized regular expression s which covers the string $t[0, |lname(a)| - 1]$, then we look rightwards from i_b for the closest position c of a glue character and build a regular expression r which covers the string $t[|lname(a)|, c]$. We then return $s \cdot r$.

We chose as the search space of candidate nodes all elements $n \in D$ for which either the first or the second seed appears within the textual content of n. Among candidate nodes for which *computePattern* captures all the seeds, we then select the one yielding the largest number of hidden names.

An example for TNR. In order to exemplify the behavior of TNR, suppose we have extracted a textual content t from a node $d \in D$ and suppose we have a set of seeds $L =\{$ "`A Venditti`", "`A Battaglia`", "`F Buccisano`", "`L Maurillo`" $\}$. t is shown in Fig. 3: it is straightforward to see that each name is followed by a comma a space and a number, except for the last one. We apply Algorithm 2. The first step creates the regular expression for the first seed available; in this case, we have $l_1 =$ "`A Venditti`" and the generated EREG is `/[A-Z]\s[A-Z][a-z]+/`. Then we apply the *matchAndSubs* function 4 times and t_b is modified accordingly (see Fig. 3). The next step of the algorithm creates the GREG `/(.*?)[0-9],?\s?/`.

Eventually, the *extract* function returns the extracted list $I = \{$"`A Venditti`", "`A Battaglia`", "`F Buccisano`", ..., "`G Franconi`", "`S Amadori`"$\}$. Note how the last name is extracted by applying the EREG instead of the generated GREG.

3 Related work

Our contribution has clear points of contact with the vast literature concerning information extraction from the Web. We focus here at closer contexts and specifically at the problem of unsupervised extraction of collections of similar items from Web pages in the case in which *seed* information is available. We can categorize related work into two main streams: *a) structural approaches*, in which the goal is to induce an extraction expression (written in XPath or another tree extraction language) for extracting information. This approach category takes advantage of document markup and is related mostly to our XPR resolution method; and, *b) unstructured approaches*, in which documents are seen at the character level and is more related to our TNR method.

Concerning category *(a)*, the availability of seeds for extracting lists and tables, possibly containing homogeneous items, is exploited in [9–11]. The first approach which explicitly addresses a scenario similar to ours is the List Extractor module of the Knowitall system [8], in which the availability of "seeds" is used in order to automatically build a wrapper for extracting lists of named entities. In [12] it is briefly described a bottom-up method for reconstructing XPath espressions aimed at capturing lists, using seeds and implemented in the WebKnox system. Both the approaches follow the same principles of our XPR technique, with a number of technical differences, but are aimed at capturing general lists, thus not embedding domain knowledge. The unavailability to the public of both the Knowitall List Extractor and the WebKnox system prevented us to compare with XPR. A similar approach aimed at unsupervised table extraction, requiring training over example rows is [13].

Concerning our TNR technique, a character-based approach can be found in the SEAL system [14]. The system is aimed at automatic set building (in the spirit of the known Google Sets) and contains a character-based wrapper construction sub-module, which looks at groups of adjacent seeds in a document using a different technique and does not exploit domain knowledge. Note that our terminology is not to be confused with the notion of *entity lists* and *seeds list* used differently in the Machine Learning and Natural Language processing field [15], as we are looking for list of entities materially appearing in sequence within documents. Also note that none of the above contributions includes explicitly the idea of building a single meta-wrapping module embedding domain-knowledge, nor they can be straightforwardly reused in our context. To exemplify, recall that TPI-S looks for "list of items in the domain X", for a given X, while the list and set extraction techniques look for general lists/sets and do not consider the information about the fact that X is set beforehand. In this respect, our approach is similar in spirit to DIADEM [16], which has been, for instance, successfully applied in the domain of real estate web sites.

Origin	#	*XPR Acc.*	*TNR Acc.*	*Total Acc.*	*DCM*
ScienceDirect	1738	98.8%	0%	98.8%	0%
IEEE Xplore	542	99.8%	100%	100.0%	0%
Physical Review D	258	100.0%	N.A.	100.0%	0%
IOPscience	218	7.3%	100%	100.0%	100%
DSpace@MIT	184	0.0%	100%	100.0%	100%
SAO/NASA ADS	176	88.6%	0%	88.6%	100%
Taylor & Francis	135	100.0%	N.A.	100.0%	100%
Wiley Online	164	100.0%	N.A.	100.0%	0%
ACS Publications	111	98.2%	100%	100.0%	100%
SPIE Digital Library	55	100.0%	N.A.	100.0%	0%
Cambridge Journals	55	100.0%	N.A.	100.0%	0%
Others	466	59.7%	6%	62.2%	22.5%
Total	**4102**	**85.0%**	**65.0%**	**94.7%**	**22.6%**

Table 1. Summary of the evaluation for TPI-S techniques

4 Evaluation and conclusions

We tested our tool by exploiting, as seed information, the partial author lists reported when querying Google Scholar. The evaluation document collection (C in the following) has been constructed by taking the set of professors and researchers in Applied Physics officially enrolled in Italian Universities as of the end of 2012 and extracting from Google Scholar their list of publications. In order to reproduce experiments at will, we stored locally all the 4-tuples (t, u, c_u, l_s), for t a paper title, u the URL of the corresponding description sheet (hosted in an arbitrary digital library), c_u the content of u and l_s a list of seed authors, as reported by Google Scholar. Duplicates, broken URLs, documents in which Google Scholar already reported the full list of authors, documents whose content was other than text or HTML/XML, and documents not appearing in our reference library (see below) were excluded from the collection, obtaining a set of about 4000 documents.

We launched our tool on every 4-tuple $(t, l_s, u, c_u) \in C$, computing the corresponding hidden list l_h. C contained documents with an average of 145 co-authors per paper, with a peak number of 2887 co-authors. Given the practical difficulty to manually assess our performance on a per document basis, we pragmatically used the Microsoft Academic Search Portal (MSA Search, in the following) as an "imperfect gold standard" to compare our extracted data. Similarly to Google Scholar, the MSA Search portal allows the user to search for scientific publications, but it shows full lists of contributors. Note however, that although MSA has been used as a reference for experiments, it cannot be used for extracting in general full author lists, due to its currently lower coverage if one considers all the scientific areas outside computer science[2].

For each $(t, l_s, u, c_u) \in C$ we thus compared the l_h value obtained using our tool with a reference list of authors l_{ms}. l_{ms} has been obtained by searching and

[2] However note that, arguably, MSA is implemented using a collection of hard-wired wrappers, thus its expected accuracy is quite high.

extracting data about t on the MSA Search portal with an hard-wired wrapper. Whenever $l_{ms} = l_h$, we assumed both l_{ms} and l_h to be correct; we instead manually cross-checked the limited set of cases in which $l_{ms} \neq l_h$. In such cases, we manually inspected the content c_u and assessed the correctness of l_h, thus excluding situations in which the extracted l_{ms} value was incorrect, due to MSA inaccurate data extraction or to the MSA collection pointing to a document different than the one described in c_u. TPI-S computing times were a negligible fraction of the page loading times.

The outcome of our evaluation is shown in Table 1. We classified documents in C according to the digital library of provenance and we aggregated results accordingly: roughly, each row reflects a family of documents with similar structure. Per each document category we report: *i)* the number of documents belonging to the category at hand; *ii)* in the rows XPR and TNR, which percentage of author lists were computed correctly using the respective technique [3]; *iii)* the overall accuracy of TPI-S; *iv)* the percentage of documents that contained proper Dublin Core Metadata [17], thus potentially allowing automated machine reading of author lists. Our tool extracts author lists from Dublin Core Metadata whenever present: however, we disabled this possibility in order to test the performance of the TPI-S methods. The limited availability of annotated data suggests that uniform and standardized machine readability in the domain of scholar digital libraries is currently not widespread. Some metadata was often available (e.g., Bibtex, EndNote etc.), but with no standard access means and format.

The evaluation shows that the overall accuracy of TPI-S is satisfactory and that TNR is a fair complement of XPR. The "Others" category collects a number of documents rather unstructured and heterogenous, not coming from a public digital library, like lecture announcements, or conference schedules (often not containing an actual authors list at all) this explaining the lower performance of TNR and XPR. Although the goal of our contribution was not a comparison performance with MSA, our results denoted a fairly good performance in this respect, with only 10% of the cases in which l_h differed from l_{ms} because l_h was incorrect. The TPI-S module has been deployed in our tool enlarging its pool of data extraction and bibliometric analysis features. The Scholar H-Index Calculator, containing a publicly accessible version of TPI-S features is available from [24].

References

1. Harzing, A.: Publish or Perish. `http://www.harzing.com/pop.htm` (2007)
2. Kaur, J., Hoang, D., Sun, X., et al.: Scholarometer: A social framework for analyzing impact across disciplines. PloS One **7**(9) (2012)
3. Cyril, L.: Ike Antkare one of the great stars in the scientific firmament. International Society for Scientometrics and Informetrics Newsletter **6**(2) (2010) 48–52

[3] Note that TNR has been applied only *incrementally* on the relatively small fraction of documents which XPR has failed to find a match. For instance, it has not been applied at all when XPR reached 100% of accuracy.

4. Schreiber, M.: A modification of the h-index: The hm-index accounts for multi-authored manuscripts. Journal of Informetrics **2**(3) (2008) 211 – 216
5. ATLAS Collaboration, T.: The ATLAS experiment at the CERN large hadron collider. Journal of Instrumentation **3**(08) (2008)
6. Carbone, V.: Fractional counting of authorship to quantify scientific research output. arXiv preprint arXiv:1106.0114 (2011)
7. Hirsch, J.: An index to quantify an individual's scientific research output. PNAS **102**(46) (2005) 16569
8. Etzioni, O., Cafarella, M., Downey, D., et al.: Unsupervised named-entity extraction from the web: An experimental study. Artificial Intelligence **165** (2005) 91–134
9. Doorenbos, R.B., Etzioni, O., Weld, D.S.: A scalable comparison-shopping agent for the world-wide web. AGENTS 1997. 39–48
10. Cohen, W.W., Hurst, M., Jensen, L.S.: A flexible learning system for wrapping tables and lists in HTML documents. WWW 2002. 232–241
11. Cohen, W.W., Fan, W.: Web-collaborative filtering: recommending music by crawling the web. Computer Networks **33**(1-6) (2000) 685–698
12. Urbansky, D., Feldmann, M., Thom, J., Schill, A.: Entity extraction from the web with Webknox. Advances in Intelligent and Soft Computing **67** (2010) 209–218
13. Gupta, R., Sarawagi, S.: Answering table augmentation queries from unstructured lists on the web. VLDB Endow. **2**(1) (August 2009) 289–300
14. Wang, R.C., Cohen, W.W.: Language-independent set expansion of named entities using the web. ICDM 2007. 342–350
15. Talukdar, P.P., Brants, T., Liberman, M., Pereira, F.: A context pattern induction method for named entity extraction. CoNLL-X 2006. 141–148
16. Furche, T., Gottlob, G., Grasso, G., et al.: DIADEM: domain-centric, intelligent, automated data extraction methodology. WWW 2012. 267–270
17. Weibel, S., Kunze, J., Lagoze, C., Wolf, M.: Dublin Core metadata for resource discovery. Internet Engineering Task Force RFC **2413** (1998) 222
18. ASN: Italian National Scientific Habilitation (Abilitazione Scientifica Nazionale), `http://abilitazione.miur.it/` (2012)
19. Citations gadget for Google Scholar. `http://code.google.com/p/citations-gadget/`
20. ERA: Excellence in Research for Australia. `http://www.arc.gov.au/era/` (2012)
21. Google Scholar. `http://scholar.google.com`
22. Microsoft Academic Search. `http://academic.research.microsoft.com/`
23. REF: Research Excellence Framework, `http://www.ref.ac.uk/` (2012)
24. Scholar H-Index Calculator. `http://scholarcalculator.gibbi.com/` (2010)
25. Sciverse Scopus. `http://www.scopus.com/`
26. Thomson Reuters Web of Knowledge. `http://wokinfo.com/`