

# Finding similar papers using ontologies

Francesco Gaetano, Luigi Lomasto, Marco Mecchia, Andrea Soldá

Gennaio 2016

## 1 Introduzione al problema

Il problema di trovare lavori scientifici simili scritti in linguaggio naturale é un compito molto difficile dal punto di vista informatico: Tali documenti infatti non hanno una struttura fissa, e sono pieni di elementi non facilmente confrontabili come formule, notazioni ed immagini. Inoltre, nonostante una netta predominanza dell'inglese, i documenti sono scritti in lingue diverse. Tutte queste caratteristiche, insite in lavori di ricerca di questo tipo, rendono gli approcci basati sul confronto testuale non utilizzabili. Il nostro lavoro, basato sulle meta informazioni dei documenti e sull'utilizzo di tecnologie del Semantic Web, mira a fornire un approccio alternativo ai metodi tradizionali, nonché un'infrastruttura riutilizzabile anche in altri ambiti.

Il resto del lavoro é organizzato come segue: nella sezione 2 viene presentata l'analisi da noi condotta ed i vari step che hanno portato al risultato finale. Nella sezione 3 vengono analizzati nel dettaglio le tecnologie del Web Semantico da noi utilizzate. Nella sezione 4 vengono presentate nel dettaglio le varie componenti introdotte nella sezione 2, illustrando e commentando estratti di codice del progetto. Infine, nella sezione 5 verranno commentati i risultati ottenuti ed eventuali applicazioni ed estensioni di quanto fatto.

## 2 Workflow

Il lavoro é stato suddiviso in diverse fasi.

### 2.1 Selezione del database ed estrazione delle meta-informazioni

In questa fase preliminare del lavoro, abbiamo costruito il dataset sul quale basarci per tutto il resto del lavoro. Per prima cosa abbiamo scelto il database dal quale attingere i lavori da confrontare. La nostra scelta e' ricaduta su DBLP[**DBLP**] in quanto punto di riferimento centrale per la sottomissione dei paper nella comunita' informatica. DBLP inoltre mette a disposizione ampi dataset di meta-informazioni sugli articoli scaricabili in

formato standard xml, con relativi url alle pagine web degli articoli. Tramite gli url, abbiamo estratto dalle pagine web gli abstract di ogni articolo tramite *scraping*. Nel nostro dataset finale, ogni articolo ha quindi i seguenti campi:

- Titolo del lavoro
- Autori del lavoro
- Anno di pubblicazione
- Abstract
- Topics (se presenti)
- Keyword (se presenti)
- Rivista
- URL

Inoltre, attraverso l'utilizzo del estrattore di keywords AlchemyAPI, dagli abstract sono state estratte le parole chiave con relativa rilevanza all'interno dell'articolo. Queste keyword sono state aggiunte a quelle già esistenti.

## 2.2 Progettazione e popolamento dell'ontologia

In questa fase, é stata studiata la progettazione di un'ontologia adatta a gestire le informazioni estratte nella fase precedente. Il passaggio ad un'ontologia é stato necessario per almeno due motivi:

1. La possibilità di interrogare il database di meta informazioni tramite query semantiche.
2. La possibilità di collegare il lavoro a strumenti già esistenti per i *linked data*, in modo da rendere lo strumento integrabile.

Per l'ontologia, la nostra scelta é ricaduta su CIDOC/CRM[CIDOC]. Il modello concettuale CIDOC/CRM é un stato progettato per la gestione di contenuti relativi alla storia ed alle opere d'arte, quindi si é rivelato adatto al nostro scopo. Per definizione, un'ontologia é CRM compatibile se rispetta lo schema di base (illustrato in seguito) proposto dagli autori. Per rappresentare in modo corretto le informazioni estratte da DBLP é stato necessario aggiungere, allo schema di base del CRM, altre classi e proprietà.

—————Qui cé da appronfondire sullo schema che abbiamo progettato noi a partire dalle meta informazioni Francesco————

E1	CRM Entity
E2	- TemporalEntity
E4	- - Period
E5	- - - Event
E7	- - - - Activity
E11	- - - - - Modification
E12	- - - - - - Production
E13	- - - - - Attribute Assignment
E65	- - - - - Creation
E63	- - - - - Beginning of Existence
<i>E12</i>	- - - - - <i>Production</i>
E65	- - - - - Creation
E64	- - - - - End of Existence
E77	- PersistentItem
E70	- - Thing
E72	- - - Legal Object
E18	- - - - Physical Thing
E24	- - - - - Physical Man-Made Thing
E90	- - - - - Symbolic Object
E71	- - - Man-Made Thing
<i>E24</i>	- - - - <i>Physical Man-Made Thing</i>
E28	- - - - Conceptual Object
E89	- - - - - Propositional Object
E30	- - - - - - Right
E73	- - - - - - Information Object
<i>E90</i>	- - - - - <i>Symbolic Object</i>
E41	- - - - - - Appellation
<i>E73</i>	- - - - - - <i>Information Object</i>
E55	- - - - - Type
E39	- - Actor
E74	- - - Group
E52	- Time-Span
E53	- Place
E54	- Dimension
E59	Primitive Value
E61	- Time Primitive 3
E62	- String

## 2.3 Interrogazione dell'ontologia

Una volta popolata l'ontologia é stata necessaria la progettazioni di query adatte al contesto del progetto. Le query su cui é stata dedicata maggiore attenzione sono due:

1. A partire dal titolo di un articolo, restituire keywords e topics.
2. A partire da keywords e topic ottenuti dalla query precedente, restituire la lista degli articoli che hanno un'sottoinsieme di keywords e topics in comune.
3. A partire dal titolo di un articolo, restituire tutte le informazioni quali: Autori, anno di pubblicazione, rivista, url ...

## 2.4 Presentazione dei risultati

Una volta progettate ed eseguite le query, abbiamo studiato come proporre i risultati in modo elegante, ma che allo stesso tempo ponesse enfasi sullo strato semantico che lega i documenti. Per fare ciò, abbiamo generato in maniera ricorsiva un grafo centralizzato: la radice é il documento di partenza, ed il solo nodo presente nel grafo. Il livello  $i + 1$ -esimo del grafo viene generato semplicemente lanciando la query principale su tutti gli articoli del livello  $i$ -esimo. Il processo viene reiterato finché non si arriva alla profondità desiderata.

## 3 Strumenti utilizzati

Le tecnologie utilizzate per lo sviluppo di questo progetto sono molteplici:

- Java 8 per gran parte del backend, cioè lo scraping e la popolazione dell'ontologia. Abbiamo usato le seguenti librerie:
  - jsoup per l'utilizzo di espressioni Xpath nella fase di scraping.
  - Apache Jena per la popolazione dell'ontologia e la creazione del file .owl.
  - AlchemyAPI per l'estrazione delle keywords da ogni topic.
- Abbiamo utilizzato le seguenti tecnologie del Semantic Web:
  - Protege per creare ed estendere lo schema ontologico.
  - OWL come linguaggio per definire l'ontologia.
  - SPARQL come linguaggio di query per interrogare l'ontologia.
  - Apache Fuseki come server per gestire le query.
- PHP7 per la formulazione e la sottomissione delle query lato server.

- Javascript per la parte di frontend, utilizzando le seguenti librerie:
  - vis.js per il rendering del grafo.
  - JQuery per gestire meglio le richieste ajax agli script Php lato server.
  - Bootstrap per la gestione dell'aspetto della pagina.

### 3.1 Jsoup

Libreria Java che permette di lavorare con documenti HTML. Fornisce delle API molto semplici con le quali é possibile estrarre e manipolare i dati a partire dal DOM di una pagina mediante l'uso di espressioni XPath.

#### Esempio

```
Document doc = Jsoup.connect(URL).timeout(50*1000).get();
Elements elemsAbstract = doc.select("p.Para");
```

### 3.2 Apache Jena

Francesco

### 3.3 AlchemyAPI

AlchemyAPI utilizza algoritmi per l'apprendimento automatico che permettono di estrarre meta-dati semantici dal contenuto desiderato, come ad esempio informazioni su persone, luoghi, aziende, gli argomenti, i fatti, le relazioni, gli autori e le lingue. I meta-dati possono essere restituiti in formato XML, JSON e RDF.

Ad ogni parola estratta viene associato una relevance (valore numerico compreso tra 0 e 1), che indica l'incidenza della parola all'interno del testo. Nel nostro caso sono state usate parole con relevance maggiore o uguale di 0.6.

—————Qui possiamo inserire sottosezioni con la pala se vogliamo descrivere nel dettaglio alcune delle tecnologie utilizzate—————

## 4 Implementazione

### 4.1 Costruzione del dataset

Per costruire il dataset di meta-informazioni dei documenti presenti su DBLP, é stato sviluppato un package java chiamato *scraper*, costituito dalle seguenti classi:

**Count.java** Contiene il main. Si occupa, preso in input il dataset di meta-informazioni degli articoli scaricabile da DBLP, di eliminare

le informazioni superflue e di invocare le classi di scraping quando si trova l'elemento `url`.

**SuperScraper.java** Superclasse astratta degli scraper, utile per il polimorfismo.

**FactoryScraper.java** Implementazione del Factory Method per gli scraper.

**IJIEMScraper.java** Istanza di SuperScraper.

**JDisplaScraper.java** Istanza di SuperScraper.

**JkdbScraper.java** Istanza di SuperScraper.

**StandardScraper.java** Istanza di SuperScraper.

```
Count
1 public class Count {
2     public static void main(String [] args) throws
        FileNotFoundException, IOException {
3         BufferedReader reader = new BufferedReader(new FileReader(
            args[1]));
4         FileWriter w = new FileWriter(args[2]);
5
6         parsing(reader, line, w);
7         w.close();
8         reader.close();
9     }
10
11     private static void parsing(BufferedReader reader, FileWriter
        w) throws IOException {
12         int nArticoli=0;
13         SuperScraper scraper;
14         FactoryScraper f = new FactoryScraper();
15
16         do{
17             line = reader.readLine();
18             if(line.contains("<article")){
19                 ++nArticoli;
20             }
21             else
22                 if(line.contains("<ee>")){
23                 scraper = f.createScraper(line);
24                 scraper.scrape(w, line);
25                 }
26             if(GENERATE)
27                 w.write(line+"\n");
28         }while(line!=null && nArticoli <=5000);
29         w.flush();
30         System.out.println(nArticoli);
31     }
32 }
```

---

Listing 1: Count.java

Il main prende due argomenti da linea di comando: il primo corrisponde al percorso del file xml di informazioni di DBLP, ed il secondo al nome del file dove ricopiare il dataset aggiornato. Il cuore della classe é costituito dal metodo statico parsing: Tale metodo legge il file XML riga per riga, fino a trovare la riga contenente l'URL dell'articolo; una volta trovato, la factory crea uno scraper apposito in base al contenuto della linea. Lo scraper si occupa di effettuare la connessione all'url, di estrarre l'abstract dalla pagina web e di aggiornare il file di output. Grazie al Factory Method ed al polimorfismo, aggiungere nuovi scraper e' semplicissimo, e non richiede l'intervento diretto sul main. Il numero di articoli é stato impostato a 5000 poiché tale numero si é rivelato sufficiente per costruire il nostro dataset.

**StandardScraper** La classe StandardScraper é un'istanza della super-classe SuperScraper, e si occupa di scrivere in output le informazioni estratte da un articolo in un file xml.

```
1 public class StandardScraper extends SuperScraper {
2
3     @Override
4     public void scrape(FileWriter w, String line) throws
5         IOException {
6         if (GENERATE)
7             w.write("<abstract>\n");
8
9         String URL[]=line.split("<ee>");
10        String URL2[]=URL[1].split("</ee>");
11        try{
12            Document page=returnPage(URL2[0]);
13            Elements elemsAbstract = page.select("p.Para");
14            for(Element elem: elemsAbstract){
15                if(DEBUG)
16                    System.out.println(elem.text().toString());
17                if(GENERATE)
18                    w.write(elem.text().toString());
19            }
20            if(GENERATE)
21                w.write("\n</abstract>\n");
22            Elements elemsKeyWord = page.select("ul.abstract-about-
23                subject > li > a");
24            for(Element elem: elemsKeyWord){
25                if(DEBUG)
26                    System.out.println(elem.text().toString());
27                if(GENERATE)
28                    w.write("<topic>");
29                w.write(elem.text().toString());
30                w.write("</topic>\n");
31            }
32        }
```

```

30 }
31 }
32 catch (ConnectException e) { w.write("\n</abstract>\n");
    failureConnect++; if (DEBUGException) { System.out.print("
    Connessione rifiutata "+URL2[0]+" "); System.out.println(
    failureConnect); } }
33 catch (HttpStatusException e) { w.write("\n</abstract>\n");
    failureConnect++; if (DEBUGException) { System.out.print("
    Status=404 "+URL2[0]+" "); System.out.println(
    failureConnect); } }
34 catch (SocketTimeoutException e) { w.write("\n</abstract>\n");
    failureConnect++; if (DEBUGException) { System.out.print("
    Socket timeout "+URL2[0]+" "); System.out.println(
    failureConnect); } }
35 catch (UnknownHostException e) { w.write("\n</abstract>\n");
    failureConnect++; if (DEBUGException) { System.out.print("dx.
    doi.org "+URL2[0]+" "); System.out.println(failureConnect)
    ; } }
36 catch (UnsupportedMimeTypeException e) { w.write("\n</abstract
>\n"); failureConnect++; if (DEBUGException) { System.out.
    print("jsoup "+URL2[0]+" "); System.out.println(
    failureConnect); } }
37 }
38 }
39 }
40 }

```

Listing 2: StandardScraper.java

Questo tipo di output e' lo stesso per tutte le istanze degli scraper. Quello che differenzia ogni scraper e' la struttura della pagina che vanno ad esaminare, per cui sono richiesti controlli ed espressioni Xpath diverse per estrarre le informazioni giuste. Se il documento contiene delle keyword, esse vengono aggiunte in un apposito tag dopo i topic, prima della chiusura del tag relativo all'articolo.

```

<article mdate="2011-01-11" key="journals/acta/Saxena96">
<author>Sanjeev Saxena</author>
<title>Parallel Integer Sorting and Simulation Amongst CRCW
    Models.</title>
<pages>607-619</pages>
<year>1996</year>
<volume>33</volume>
<journal>Acta Inf.</journal>
<number>7</number>
<url>db/journals/acta/acta33.html#Saxena96</url>
<abstract>In this paper a general technique for reducing
    processors in simulation without any increase in time is
    described.</abstract>
<topic>Logics and Meanings of Programs</topic>

```



```

<topic>Computer Systems Organization and Communication
  Networks</topic>
<topic>Software Engineering/Programming and Operating Systems<
  /topic>
<topic>Data Structures, Cryptology and Information Theory</
  topic>
<topic>Theory of Computation</topic>
<topic>Information Systems and Communication Service</topic>
<ee>http://dx.doi.org/10.1007/BF03036466</ee>
</article>

```

Listing 3: Esempio di xml prodotto dallo StandardScraper

## FactoryScraper

```

1 public class FactoryScraper {
2
3     private SuperScraper jdispla;
4     private SuperScraper jkdb;
5     private SuperScraper ijiem;
6     private SuperScraper standard;
7
8     public FactoryScraper(){
9         jdispla = new JDisplaScraper();
10        jkdb = new JkdbScraper();
11        ijiem = new IJIEMScraper();
12        standard = new StandardScraper();
13    }
14
15    public SuperScraper createScraper(String line){
16
17        if(line.contains("<ee>")){
18            if(line.contains("j.displa.") || line.contains("j.compind")
19                ){
20                return jdispla;
21            }
22            else{
23                if(line.contains("jkdb")){
24                    return jkdb;
25                } else if(line.contains("IJIEM.")){
26                    return ijiem;
27                }
28                else{
29                    return standard;
30                }
31            }
32        }
33        return null;
34    }
35 }

```

Listing 4: FactoryScraper.java

La FactoryScraper alla sua creazione crea un tipo di oggetto per ogni specializzazione della classe SuperScraper; in questo modo, quando c'è bisogno di fare il parsing di un nuovo documento non si crea ogni volta un nuovo oggetto, ma si utilizza sempre lo stesso. Così facendo, gli oggetti vengono riutilizzati e vengono risparmiati memoria e processore, in quanto la Garbage Collector di Java non deve essere invocata di continuo. Il metodo createScraper si deve solamente occupare di verificare le condizioni per cui deve essere creato un tipo di Scraper piuttosto che un altro.

## 4.2 Estrazione delle Keyword

Per l'estrazione delle keyword a partire dall'abstract è stato fatto uso delle API messe a disposizione dal software Alchemy[alchemy], scaricabili gratuitamente. Tale software è composto da molti strumenti utili per l'analisi linguistica, tra cui l'estrazione di parole chiave da un testo con rilevanze normalizzate nell'intervallo [0,1]. Gli unici limiti riscontrati sono stati il fatto di doversi registrare per ottenere una chiave per utilizzare le API ed il relativo limite di chiamate giornaliere.

Il pacchetto *keyword* sviluppato nel progetto è composto da due classi:

**KeywordExtractor** è una classe di wrapper per la chiamata al metodo di AlchemyAPI che dato un testo restituisce le keyword rankate.

**ScraperForKeyword** è la classe contenente il metodo main del pacchetto. Si occupa di aggiungere al dataset generato con il pacchetto scraper le keyword estratte con la classe KeywordExtractor.

### ScraperForKeyword

```

1 captioncaption
2 public class scraperForKeyword {
3     private static String getStringFromDocument(Document doc)
4         throws IOException {
5         try {
6             DOMSource domSource = new DOMSource(doc);
7             StringWriter writer = new StringWriter();
8             StreamResult result = new StreamResult(writer);
9             String toReturn="";
10            TransformerFactory tf = TransformerFactory.newInstance();
11            Transformer transformer = tf.newTransformer();
12            transformer.transform(domSource, result);
13
14            String [] split=writer.toString().split("<keyword>");
15
16            for(int i=1;i<split.length;i++){
17                String [] relevance1=split[i].split("<relevance>");
18                String [] relevance2=relevance1[1].split("</relevance>");
19                String [] text1=split[i].split("<text>");
20                String [] text2=text1[1].split("</text>");

```

```

21         double relevance = Double.parseDouble(relevance2[0]);
22         if(relevance*10 >= 6){
23             toReturn+="<keyword>\n";
24             toReturn+="<relevance>"+relevance+"</relevance>\n";
25             toReturn+="<text>"+text2[0]+"</text>\n";
26             toReturn+="</keyword>\n";
27         }
28     }
29
30     return toReturn;
31 } catch (TransformerException ex) {
32     ex.printStackTrace();
33     return null;
34 }
35 }
36
37 public static void main(String [] args) throws IOException,
    XPathExpressionException, SAXException,
    ParserConfigurationException{
38     int nArticoli = Integer.parseInt(args[0]);
39     int numeroRigheLette;
40     int articoliGialetti=nArticoli;
41     int articoliDaLeggere=Integer.parseInt(args[1]);
42     boolean abstractB=false;
43     BufferedReader reader = new BufferedReader(new FileReader(
        args[2]));
44     String line = reader.readLine();
45     String abstract_Text="";
46     FileWriter w=new FileWriter(args[3]);
47     String api_path = args[4];
48     int counter=0;
49
50     while(counter<=nArticoli){
51         line = reader.readLine();
52         if(line.contains("<article>"))
53             ++counter;
54     }
55     while(line!=null && nArticoli<articoliDaLeggere+
        articoliGialetti) {
56         if(line.contains("<article>")){
57             ++nArticoli;
58             if(nArticoli<articoliDaLeggere+articoliGialetti){
59                 w.write(line+"\n");}
60             abstractB=false;
61         }
62         else
63             if(line.contains("<abstract>")){
64                 abstract_Text+=line;
65                 abstractB=true;
66
67             }
68         else if(abstractB && line.contains("</abstract>")){
69             abstractB=false;
70             abstract_Text=abstract_Text.replace("<abstract>","");

```

```

71         abstract_Text=abstract_Text.replace("</abstract>","");
72         if(!abstract_Text.equals("")){
73             try{
74                 Document s=KeywordExtractor.extractKeyword(
75                     abstract_Text, api_path);
76                 String keywordDocument = getStringFromDocument(s);
77                 w.write(keywordDocument);
78             }
79             catch(IOException e){
80                 System.out.println("Error making API call:
81                                     unsupported-text-language.");
82             }
83         }
84         abstract_Text="";
85     }
86     else if(abstractB){
87         abstract_Text+=line;
88     }
89     else
90         w.write(line+"\n");
91     line = reader.readLine();
92 }
93 w.close();
94 System.out.println("#Articoli letti: "+ --nArticoli);
95 }
96 }

```

./src/keywords/scrapperForKeyword.java

Il main prende in input 5 argomenti: il numero di articoli già letti, il numero di articoli da leggere, il percorso del file di input<sup>1</sup>, il percorso del file di output ed il percorso della chiave per utilizzare le API di Alchemy. Il numero di articoli già letti e quelli da leggere sono parametri necessari introdotti dal limite di chiamate di AlchemyAPI; in questo modo il lavoro è stato diviso tra i componenti del team in maniera facilmente ricostruibile.

Dopo aver saltato gli articoli già letti, il parser per ogni articolo trova l'abstract ed esegue una chiamata al metodo statico ExtractKeyword della classe KeywordExtractor. Grazie al metodo di utilità getStringFromDocument, il documento xml contenente le keyword dell'abstract viene serializzato. La stringa corrispondente al documento serializzato a questo punto viene concatenata alle informazioni del documento e viene quindi scritta in output.

**KeywordExtractor** La classe é composta dal solo metodo statico ExtractKeyword. Tale metodo prende in input il testo da cui estrarre le parole chiave ed il percorso del file contenente la chiave fornita dal sito web di Alchemy. Il metodo restituisce un documento xml sotto forma di oggetto

<sup>1</sup>Nel file ci sono già gli abstract, estratti con il pacchetto scraper.

Document; in questo modo é facilmente serializzabile, oltre che navigabile con gli strumenti del DOM.

```
1 public class KeywordExtractor {  
2  
3     public static Document extractKeyword(String text, String  
        api_path) throws IOException, XPathExpressionException,  
        SAXException, ParserConfigurationException {  
4         AlchemyAPI alchemyObj = AlchemyAPI.GetInstanceFromFile(  
            api_path);  
5         return alchemyObj.TextGetRankedKeywords(text);  
6     }  
7  
8 }
```

Listing 5: KeywordExtractor.java

```
<keywords>  
<keyword>  
<relevance>0.986122</relevance>  
<text>tree schemas</text>  
</keyword>  
<keyword>  
<relevance>0.868575</relevance>  
<text>NP-complete problems</text>  
</keyword>  
<keyword>  
<relevance>0.830641</relevance>  
<text>certain NP-complete problems</text>  
</keyword>  
</keywords>
```

Listing 6: Esempio di documento restituito dalla classe KeywordExtractor

### 4.3 Popolamento dell'ontologia

### 4.4 Query SPARQL

### 4.5 Front-end

Il lavoro viene presentato come una pagina web interattiva alla quale é possibile sottomettere query...

## 5 Conclusioni

Un altro utilizzo interessante, anche se non pienamente pertinente con gli obiettivi del lavoro, é la possibilità di prevedere query che prendano in input dei topic e restituiscano tutti gli articoli che hanno quel topic e le relative

parole chiave. In questo modo, cambiando il contesto (e.g. l'ontologia viene popolata con un set diverso di articoli, magari provenienti da una serie di conferenze), si può mostrare come lo stesso argomento viene trattato in maniera diversa a seconda del contesto. Se ad esempio il topic é *Bioinformatica*, in un contesto di proceedings di conferenze informatiche le parole chiave potrebbero essere *Algoritmi*, *Strutture Dati*, *Complessità Computazionale*, etc. Mentre lo stesso topic in un contesto di proceedings di conferenze biologiche potrebbe avere come parole chiave *Ribosomi*, *Proteine*, etc.