Università degli Studi
di Salerno

Studio e implementazione di un estrattore di ontologie fuzzy

Progetto per il corso di
Web Semantico

Studenti:
Giovanni G. Costa
Demia Massaro
Antonio Sanfelice

Docente:
Sabrina Senatore

Anno Accademico 2012 - 2013

1 Introduzione

La ricerca di informazioni sul Web è un problema complesso, la cui risoluzione ha portato alla nascita di diverse compagnie che hanno migliorato di anno in anno la qualità dei risultati forniti. Tuttavia, l'uso del testo puro come criterio di ricerca porta inevitabilmente alla scoperta di informazioni legate solo alle parole cercate e non al loro *significato*: ciò ha portato all'avvento del *Web Semantico*. Grazie a linguaggi specifici, come RDF o OWL, è possibile associare ai dati pubblicati sulla rete una struttura che ne descriva il contenuto semantico, permettendo di effettuare ricerche per *contesto*. Ciò porta ad un nuovo problema: cosa fare di tutte le informazioni pubblicate in rete *prima* della nascita del Web Semantico? Milioni e milioni di pagine di testo puro che, di certo, non possono essere aggiornate manualmente. Nasce quindi l'esigenza di trovare un metodo automatico per estrarre dal puro testo i concetti in esso contenuti e le loro relazioni, ottenendo quindi *un'ontologia*.
Ci poniamo lo scopo di fornire un'implementazione in Java dell'algoritmo descritto in [Lau et al., 2007], pubblicazione di riferimento per questo progetto, il quale consente di estrarre ontologie *fuzzy* a partire da una collezione di testi. L'uso della logica fuzzy permette di ottenere una descrizione dei concetti più "umana", grazie all'introduzione di un *grado di correlazione* fra i concetti. Nel seguito di questa sezione saranno fornite alcune definizioni necessarie per la piena comprensione del lavoro svolto. Nella sezione 2 verrà esaminato l'algoritmo proposto da [Lau et al., 2007], nella sezione 3 sarà descritta la nostra implementazione dell'algoritmo e verranno spiegate le scelte implementative fatte, per poi concludere con la spiegazione dei test effettuati nella sezione 4 e le conclusioni in 5.

1.1 Definizioni

Cominciamo con il definire il concetto di *ontologia*. L'ontologia è una delle principali branche della filosofia, essa si occupa delle categorie fondamentali dell'essere. Dal punto di vista informatico, l'ontologia ha trovato applicazione come metodo di *rappresentazione della conoscenza*. Essa può essere definita nel modo seguente:

Definizione 1 (Ontologia) *Un'ontologia è una tupla*

$$\mathcal{O} =< X, C, R_{XC}, R_{CC} >$$

dove

- X *è un insieme di oggetti;*
- C *è un insieme di concetti;*

- $R_{XC} : X \times C \mapsto \{0, 1\}$ *è una funzione che associa ogni oggetto a un concetto* ($x \in X$ *è un* $c \in C$)*;*
- $R_{CC} : C \times C \mapsto \{0, 1\}$ *è una funzione che modella le relazioni presenti fra i concetti, nota anche come* ***tassonomia****.*

Da questa definizione si evince che un'ontologia è un modello capace di esprimere l'appartenenza degli oggetti a delle categorie, fornendo al tempo stesso una gerarchia fra le categorie stesse. Il problema di questo modello però è che non è capace di esprimere *sfumature*, in quanto le due relazioni R_{XC} e R_{CC} hanno entrambe come codominio l'insieme $\{0, 1\}$. La *logica fuzzy* nasce proprio per risolvere problemi simili, passando da una logica bivalore dove una proposizione è vera o falsa, ad una multivalore dove si considera *quanto la proposizione è vera*. Pertanto il valore di verità di una proposizione non è più nell'insieme $\{0, 1\}$, ma nell'*intervallo* $[0, 1]$. Detto ciò, possiamo introdurre il concetto di *ontologia fuzzy*:

Definizione 2 (Ontologia Fuzzy) *Un'ontologia fuzzy è una tupla:*

$$\mathcal{O} =< X, C, R_{XC}, R_{CC} >$$

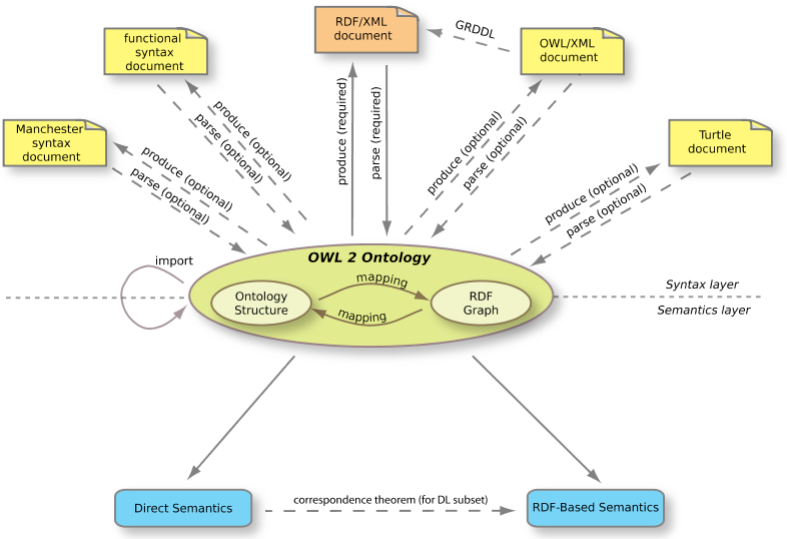
dove

- X *è un insieme di oggetti;*
- C *è un insieme di concetti;*
- $R_{XC} : X \times C \mapsto [0, 1]$ *è una funzione che esprime* ***quanto*** *ogni oggetto* $x \in X$ *appartiene ad ogni concetto* $c \in C$*;*
- $R_{CC} : C \times C \mapsto [0, 1]$ *è una funzione che rappresenta* ***quanto*** *due concetti sono correlati fra loro, nota anche come* ***tassonomia fuzzy****.*

1.2 Strumenti

OWL 2

OWL 2 (Ontology Web Language) è un linguaggio di markup per la specifica formale della semantica, il cui fine è descrivere delle basi di conoscenze, effettuare delle deduzioni su di esse e integrarle con i contenuti delle pagine web. Le ontologie OWL 2 sono un'estensione ed una revisione dell'OWL, sviluppato dal W3C Web Ontology Group, le quali forniscono classi, proprietà, individui e valori che è possibile memorizzare e condividere come documenti del Semantic Web.



La precedente figura fornisce una visione generale della struttura del linguaggio OWL 2, in particolare:

- l'ellisse centrale rappresenta il concetto astratto di ontologia, il quale può essere visto come una struttura astratta oppure come un grafo in notazione RDF;
- nella parte superiore della figura, sono mostrate le possibili notazioni sintattiche che possono essere utilizzare per la definizione delle ontologie;
- nella parte inferiore, invece, vi sono due specifiche semantiche che definiscono il sIgnificato delle ontologie.

Rispetto ad OWL 1, OWL 2 aggiunge alcune caratteristiche di tipo sintattico (ad esempio disjoint union di classi) ed altre che offrono maggiore espressività (ad esempio nuove tipologie di annotation), in particolare:

- chiavi;
- concatenamento di proprietà;
- ulteriori datatype e data ranges;
- proprietà asimmetriche, riflessive e disgiuntive;
- altre possibili annotation.

Fuzzy Ontology

Fuzzy OWL 2 nasce dalla crescente necessità di trattare informazioni con un certo grado di verità nei linguaggi per il Semantic Web. [Bobillo and Straccia, 2011] hanno proposto di utilizzare OWL 2 per rappresentare ontologie fuzzy, in particolare le *annotation properties*.

```
<AnnotationAssertion>
    <AnnotationProperty IRI="#annotationProperty"/>
    <IRI>#className</IRI>
    <Literal datatypeIRI="&rdf;PlainLiteral">
        annotationValue
    </Literal>
</AnnotationAssertion>
```

Tale meccanismo consente di estendere ontologie esistenti con concetti fuzzy, utilizzare editor testati e raffinati per creare le ontologie e continuare ad usare reasoner non-fuzzy, i quali scarteranno semplicemente le annotazioni.
L'idea alla base del Fuzzy OWL 2 è di utilizzare una propria annotation property: `fuzzyLabel`.

```
<AnnotationAssertion>
    <AnnotationProperty IRI="#fuzzyLabel"/>
    <IRI>#Vecchio</IRI>
    <Literal datatypeIRI="&rdf;PlainLiteral">
        <fuzzyOwl2 fuzzyType ="datatype">
            < Datatype type ="leftshoulder" a ="5" b ="7"/>
        </fuzzyOwl2 >
    </Literal>
</AnnotationAssertion>
```

Un elemento può avere al massimo un'annotazione fuzzy, la quale sarà delimitata dai tag `<fuzzyOwl2>` e `</fuzzyOwl2>` e l'attributo `fuzzyType` specifica il tipo di elemento annotato.

2 L'Algoritmo

Per estrarre le ontologie a partire dal puro testo, verrà usato l'algoritmo proposto da [Lau et al., 2007]. Tale algoritmo si divide in tre passi principali:

Calcolo delle frequenze: In questo passo, dopo alcune operazioni di pulitura, vengono calcolate le frequenze dei termini presi singolarmente e a coppie;

Creazione dei vettori di contesto: In questo passo si usano i risultati del passo precedente per decidere quali sono i termini che con più probabilità rappresentano concetti chiave del contesto;

Creazione della tassonomia: I concetti trovati vengono messi in relazione creando una gerarchia, questa gerarchia viene poi ripulita dalle relazioni che non sono "abbastanza significative".

Ai passi appena elencati, ne verrà aggiunto uno che dovrà tradurre l'ontologia così ricavata in **fuzzy OWL2**. Nel prosieguo di questa sezione verranno studiati uno per uno i passi precedentemente elencati.

2.1 Calcolo delle frequenze

Come anticipato prima, in questo passo vanno calcolate le frequenze singole e mutue dei termini che compongono i documenti del dominio facendo in modo che i termini che lo caratterizzano vengano messi in evidenza e i termini poco significativi vengano scartati. Per ottenere questo risultato è necessario sfruttare al massimo la conoscenza a priori del dominio: se sappiamo che molti dei concetti che caratterizzano il dominio sono formati da una coppia di sostantivi, allora l'algoritmo dovrebbe tenerne conto. A questo scopo, come prima operazione viene effettuato il *Part Of Speech tagging*. Il *Part Of Speech tagging* è un'operazione che consiste nell'etichettare ogni parola con il suo ruolo nella frase (soggetto, verbo, predicato, ...): per effettuarla abbiamo utilizzato la libreria sviluppata da [Toutanova et al.,].

Effettuato il *POS tagging* è necessario ripulire il testo da informazioni inutili. In primo luogo bisogna eliminare le *Stop Words*, ovvero quei termini di uso comune che non sono prorpri del contesto (congiunzioni, disgiunzioni, avverbi, ecc.), e i segni di punteggiatura, quindi si passa all'operazione di *word stemming*: tale processo riduce le parole alla loro radice, in modo da identificare più facilmente concetti. Ad esempio, i termini "fishing" , "fished", "fisher" vengono sostiuiti dalla loro radice, "fish". Una volta ripulito il testo si può procedere con il conteggio delle frequenze. Viene utilizzata una finestra scorrevole di n parole (come suggerito in [Lau et al., 2007]) in modo da diminuire il rumore dovuto a termini non significativi. La frequenza di un termine è quindi

$$\text{frequenza}(t_i) = \frac{\text{Numero di finestre in cui compare } t_i}{\text{Numero totale di finestre}}$$

Mentre la frequenza mutua di due termini è

$$\text{frequenza}(t_i,\, t_j) = \frac{\text{Numero di finestre in cui compaiono } t_i \text{ e } t_j}{\text{Numero totale di finestre}}$$

Una volta che le frequenze sono state calcolate, viene operato un filtraggio atto a scartare come candidati concetti tutti quei termini la cui frequenza

Algoritmo 1 Calcolo delle frequenze

1: $X \leftarrow \emptyset$
2: **Per ogni** documento $d \in$ dominio
3: Effettua il POS tagging
4: **Per ogni** frase $\in d$
5: Elimina le Stop Words
6: Elimina i segni di punteggiatura
7: Effettua il Word Stemming
8: **Per ogni** Finestra di testo $w \in$ frase
9: numOccorrenze$[t_i] \leftarrow$ numOccorrenze$[t_i] + 1 \qquad \forall t_i \in w$
10: numOccorrenze$[t_i, t_j] \leftarrow$ numOccorrenze$[t_i, t_j] + 1 \quad \forall t_i, t_j \in w$
11: **Fine Ciclo**
12: **Fine Ciclo**
13: **Fine Ciclo**
14: **Per ogni** termine t_i
15: frequenza$[t_i] \leftarrow \frac{\text{numOccorrenze}[t_i]}{\text{totale finestre}}$
16: frequenza$[t_i,\, t_j] \leftarrow \frac{\text{numOccorrenze}[t_i, t_j]}{\text{totale finestre}}$
17: **Se** lower $\leq$ frequenza$[t_i] \leq$ upper **allora**
18: $X \leftarrow X \cup t_i$
19: **Fine Se**
20: **Fine Ciclo**

si trova al di fuori di un dato range. I termini la cui frequenza è al di sotto della soglia minima risultano insignificanti per il contesto, quelli che sono al di sopra della soglia massima invece possono essere termini sfuggiti alla rimozione di stop words, o comunque termini semplicemente troppo comuni che non aiutano a definire il contesto.

2.2 Creazione context vectors

In questo passo si decide quali termini rappresentano concetti significativi per il contesto. Vengono creati dei *Vettori di contesto*, ovvero delle strutture che relazionano ogni concetto candidato ai termini che lo descrivono. Ad ogni termine t_j è associato un grado $\mu_{c_i}(t_j)$ che esprime quanto quel termine descrive il concetto c_i. Per calcolare tale valore possono essere utilizzate diverse funzioni: ad esempio è possibile utilizzare la *mutua informazione puntuale*. La *mutua informazione* è definita come

$$MI(t_i,\, t_j) = \log_2 \left(\frac{p(t_i,\, t_j)}{p(t_i)\, p(t_j)} \right) \tag{2.1}$$

In [Lau et al., 2007], viene definita un'ulteriore metrica, la *Balanced Mutual Information*, che rispetto alla MI, tiene conto sia della presenza che

dell'assenza dei termini. Essa è definita come segue:

$$BMI(t_i, t_j) = \beta(Pr(t_i,\, t_j)\, MI(t_i,\, t_j) + Pr(\neg t_i,\, \neg t_j)\, MI(\neg t_i,\, \neg t_j)) - (1-\beta)\,(Pr(t_i,\, \neg t_j) MI(t_i,\, \neg t_j) + Pr(\neg t_i, t_j)\, MI(\neg t_i,\, t_j)) \tag{2.2}$$

Il parametro $\beta \in [0,\, 1]$ permette di decidere quanto deve pesare il fatto che un termine appaia quando l'altro è assente. Una volta che tutti i gradi di appartenenza sono stati calcolati è necessario scartare quelli che non forniscono abbastanza informazione: vengono quindi scartati i vettori di contesto in cui tutti gli elementi hanno grado inferiore ad una certa soglia. Questo processo ha la stessa funzione dell'α-cut utilizzato in logica fuzzy.

Algoritmo 2 Creazione dei vettori di contesto

1: **Per ogni** termine $t_i \in X$
2: **Per ogni** termine $t_j \in X,\ t_i \neq t_j$
3: Calcola il grado di appartenenza di t_j a t_i
4: **Fine Ciclo**
5: **Se** Tutti i gradi di appartenenza sono minori di α **allora**
6: t_i non è un concetto
7: **Fine Se**
8: **Fine Ciclo**

2.3 Creazione delle tassonomie

Ottenuti i vettori di contesto bisogna collegare i diversi concetti tra loro tramite relazioni *is-a*. come spiegato nella sezione 1.1. Definiamo *Specificity* il grado con il quale un concetto è sottoconcetto di un altro. Se un concetto c_x è sottoconcetto di un concetto c_y, lo è con un grado pari a:

$$\mu_{C\times C}(c_x,\, c_y) \approx Spec(c_x,\, c_y) = \frac{\sum_{t_x \in c_x,\, t_y \in c_y, t_x = t_y} \mu_{c_x}(t_x) \otimes \mu_{c_y}(t_y)}{\sum_{t_x \in c_x} \mu_{c_x}(t_x)} \tag{2.3}$$

L'operatore $\otimes$ è un operatore di congiunzione fuzzy equivalente alla funzione minimo. Ottenuta la *specificity* per ogni coppia di concetti bisogna scartare le relazioni superflue: in primo luogo vengono conservate le relazioni tali che $Spec(c_x,\, c_y) > Spec(c_y,\, c_x)$ e $Spec(c_x,\, c_y) > \lambda$. Il parametro λ è una soglia stimata tramite esperimenti empirici, che permette di scartare le relazioni che non sono abbastanza signifcative. Inoltre viene eseguito un ulteriore processo di potatura: se $\mu_{C\times C}(c_1,\, c_2) < \min(\mu_{C\times C}(c_1,\, c_i),\, \ldots,\, \mu_{C\times C}(c_i,\, c_2))$ dove $c_1,\, c_i,\, \ldots,\, c_2$ formano un path da c_1 a c_2, allora $R(c_1,\, c_2)$ viene rimosso dalla tassonomia, in quanto vi è una catena di relazioni più forti che collega i due concetti.

Algoritmo 3 Creazione della tassonomia

Per ogni coppia di concetti $c_i, c_j \in C$
  $R(c_i, c_j) \leftarrow Spec(c_i, c_j)$
  Se $R(c_i, c_j) > \lambda$ **allora**
    $R \leftarrow R \cup R(c_i, c_j)$
  Fine Se
Fine Ciclo
Per ogni $R(c_i, c_j) \in R$
  Se $R(c_i, c_j) < R(c_j, c_i)$ **allora**
    $R \leftarrow R - R(c_i, c_j)$
  Fine Se
  Se $\exists$ path p da c_i a c_j e $R(c_i, c_j) < min_{\mu_{C\times C}}(p)$ **allora**
    $R \leftarrow R - R(c_i, c_j)$
  Fine Se
Fine Ciclo

3 Implementazione

In questa sezione sarà mostrata la nostra implementazione dei concetti precedentemente esposti.

3.1 Strutture Dati

Le seguenti strutture sono pensate per conservare i risultati dei singoli passi dell'algoritmo: frequenze dei termini, vettori di contesto e tassonomia.

3.1.1 `DTMatrix`

La classe `DTMatrix` implementa una matrice triangolare a espansione dinamica. Per l'esecuzione dell'algoritmo è necessario conservare dati come le frequenze mutue dei termini e i gradi di appartenenza dei termini ai concetti. Tali relazioni sono simmetriche, pertanto ci è sembrato corretto utilizzare una matrice triangolare come struttura dati principale. La classe è generica, così che vi si possa conservare qualsiasi tipo di dati e contiene due attributi:

```
1  private ArrayList<E> terms;
2  private double[][] values;
```

L'arraylist serve ad avere una corrispondenza oggetto/indice, mentre la matrice di double contiene gli effettivi valori da memorizzare. Il metodo `checkSize` controlla se la grandezza dell'arraylist e della matrice sono consistenti: se non lo sono invoca il metodo `growSize` che raddoppia la dimensione della matrice I metodi `getValue` e `setValue` consentono l'accesso ai dati contenuti dalla matrice. Il metodo `normalizeBy` consente di dividere

tutti i valori della matrice per un valore passato per parametro. Le implementazioni di queste funzioni sono molto banali pertanto non verranno esaminate.

3.1.2 `TermFrequencies`

Questa classe eredita da `DTMatrix` e conserva sia le frequenze singole che quelle mutue. Oltre agli attributi `terms` e `values` ereditati dalla classe padre, `TermFrequencies` ha due ulteriori attributi:

```
1  private int totWindows;
2  private boolean finalValues;
```

Ricordiamo che le frequenze devono essere relative, non assolute: il numero di occorrenze viene diviso per il numero di finestre di testo ottenute, che viene memorizzato nell'attributo `totWindows`. L'attributo `finalValues` serve da lock, per fare in modo che i dati nella matrice non vengano più modificati una volta che sono stati normalizzati. La classe offre i seguenti metodi:

`void addOccurence(E term)`: metodo che aumenta di uno il numero di occorrenze del termine `term`;

`void addOccurence(E firstTerm, E secondTerm)`: metodo che aumenta di uno il numero di occorrenze mutuo dei termini `firstTerm` e `secondTerm`;

`void augmentWindows()`: aumenta di uno il numero di finestre di testo totali;

`double getFrequency(E term);` metodo che restituisce la frequenza del termine `term`, solo se sono state calcolate le frequenze relative;

`double getFrequency(E firstTerm, E secondTerm);` metodo che restituisce la frequenza mutua dei termini `firstTerm` e `secondTerm`, solo se sono state calcolate le frequenze relative;

`void filterTerms(double lower, double upper)`: se le frequenze sono state calcolate, elimina i termini la cui frequenza è compresa fra `lower` e `upper`;

`ArrayList<E> getTerms()`: metodo che restituisce la lista di termini di interesse.

La struttura è pensata in modo da avere sulla diagonale principale le frequenze singole, mentre ogni entrata $[i, j]$ con $i \neq j$ rappresenta la frequenza mutua dei termini i-esimo e j-esimo.

3.1.3 `ContextVectors`

Questa classe contiene i vettori di contesto. Anche essa eredita da `DTMatrix`, dato che le metriche utilizzate per calcolare il grado di appartenenza di un concetto ad un altro sono simmetriche. La classe offre i seguenti metodi:

`void setMembership(E term, E concept, double membership)`: assegna alla coppia (`term`, `concept`) il grado `membership`;

`double getMembership(E concept, E term)`: restituisce la membership di `term` a `concept`;

`ArrayList<E> getConcepts()`: restituisce un arraylist dei concetti trovati;

`ArrayList<E> getTerms()`: metodo che restituisce la lista di termini di interesse;

`void normalizeMemberships()`: metodo che normalizza i dati scalandoli all'interno dell'intervallo [0, 1].

I concetti che non passano il processo di filtraggio descritto nell'algoritmo 2 vengono eliminati tramite il metodo `void deleteConcept(E term)`, che assegna il valore -1 alla membership del termine con se stesso.
Il metodo `normalizeMemberships` utilizza la scalatura lineare, quindi dopo aver trovato il minimo e il massimo ricalcola ogni valore nel seguente modo:

```
1 values[i][j] = (values[i][j]- min)/(max - min);
```

3.1.4 `Taxonomy`

Questa classe modella la tassonomia, ovvero la relazione di concetto - sotto-concetto descritta nella sezione 1.1. Al contrario delle relazioni precedenti, la (2.3) non è simmetrica, quindi in questo caso viene utilizzata una normale matrice quadrata. La classe usa due attributi:

```
1   private double[][] relations;
2     private ArrayList<E> concepts;
```

`relations` è la matrice contenente i gradi di relazione, mentre `concepts` è un arraylist contenente i concetti, che serve a mappare concetti a indici della matrice. I principali metodi forniti da questa classe sono i seguenti:

`void setSpecificity(E subConcept, E concept, double spec`: imposta `spec` come grado di speficifctà di `subConcept` a `concept`;

`double getSpecificity(E subConcept, E concept`: restituisce il grado con cui `subConcept` è sottoconcetto di `concept`;

`void taxonomyPruning()`: esegue il processo di potatura descritto nella sezione 1.1.

Il metodo `taxonomyPruning` merita una spiegazione più accurata In primo luogo, data una coppia di concetti i e j, il metodo scarta il più piccolo fra $Spec(i, j)$ e $Spec(j, i)$.

```
int numConcepts = concepts.size();

for(i = 0; i < numConcepts; i++){
    for(j = i; j < numConcepts; j++){
        if( relations[i][j] != relations[j][i])
            if( relations[i][j] > relations[j][i])
                relations[j][i] = 0;
            else
                relations[i][j] = 0;
        }
    }
```

Quindi si passa al vero e proprio processo di potatura: ricordiamo che per ogni coppia di concetti i e j, bisogna trovare un path che li colleghi e controllare se $Spec(i, j)$ è minore del minimo arco del path. A questo scopo, abbiamo leggermente modificato l'algoritmo di *Floyd-Warshall*, in modo da includere tale controllo. L'idea è la seguente: controllare se il peso dell'arco (i, j) è minore del minimo arco del path da i a j, equivale a controllare se esiste un path composto da soli archi con peso maggiore dell'arco (i, j). Quindi facciamo in modo che l'algoritmo cerchi uno *shortest path* fra i e j senza tener conto:

- dell'arco diretto (i, j);
- di tutti gli archi il cui peso è inferiore a quello dell'arco (i, j).

La modifica è implementata tramite la seguente condizione, che tiene conto degli archi (i, k) e (k, j) solo se questi hanno peso maggiore dell'arco diretto, indicato da `relations[i][j]`.

```
if(relations[i][k] > relations[i][j] &&
   relations[k][j] > relations[i][j]){
        sumDist = distance[i][k] + distance[k][j];
        if( sumDist < distance[i][j])
            distance[i][j] = sumDist;
}
```

Successivamente, per ogni coppia di concetti, se i due concetti sono relazionati ed è stata trovata una catena di relazioni più forti, la relazione diretta viene scartata:

```
if(relations[i][j] > 0 && (distance[i][j] > relations[i][j]))
            relations[i][j] = 0;
```

3.2 Interfacce ed implementazioni

3.2.1 `CorrelationFunction`

Per rendere più semplice la scelta della metrica da usare per il calcolo dei vettori di contesto, abbiamo deciso di fornire un'interfaccia, chiamata `CorrelationFunction`, le cui implementazioni dovranno essere passate all'algoritmo per la creazione dei vettori di contesto.

```
1  public interface CorrelationFunction {
2
3    public double calculateCorrelation(double freq_x, double freq_y,
         double freq_xy );
4
5  }
```

3.2.2 `PMI`

Forniamo due implementazioni di questa interfaccia: la prima consente di calcolare la correlazione tra due termini utilizzando la mutua informazione puntuale.

```
1  public class PMI implements CorrelationFunction{
2    private static final int num = 1024;
3
4    public double calculateCorrelation(double p_x, double p_y, double
         p_x_y) {
5
6      int isZero = new Double(p_x_y).compareTo(0.0);
7      int isOne = new Double(p_x_y).compareTo(1.0);
8
9      if (isZero == 0)
10       return -num;
11
12     if (isOne == 0)
13       return num;
14
15     return (Math.log(p_x_y) - Math.log(p_x) - Math.log(p_y))/Math.
           log(2);
16   }
17 }
```

La (2.1) è stata risistemata in modo da limitare problemi numerici dovuti a prodotti e divisioni fra double:

$$\begin{aligned}\log_2\left(\frac{p(x,\,y)}{p(x)\,p(y)}\right) &= \log_2(p(x,\,y)) - \log_2(p(x)\,p(y)) \\ &= \log_2(p(x,\,y)) - (\log_2(p(x)) + \log_2(p(y))) \\ &= \log_2(p(x,\,y)) - \log_2(p(x)) - \log_2(p(y))\end{aligned}$$

Il controllo sull'uguaglianza della probabilità congiunta a 0 è necessario per evitare la presenza di `NaN`, dovuti all'esecuzione di operazioni con $-\infty$ quando viene passato 0 al logaritmo. Se ci troviamo in questa situazione, facciamo sì che invece di $-\infty$ venga restituito un intero abbastanza piccolo, scelto arbitrariamente. Il controllo sull'uguaglianza della probabilità congiunta a 1 viene fatto perché, come spiegato da [Lau et al., 2007], se due termini hanno probabilità congiunta pari a 1 sono totalmente correlati, quindi invece del risultato ottenuto normalmente dalla (2.1), restituiamo un intero abbastanza grande che sottolinei il peso di questa relazione.

3.2.3 BMI

La seconda implementazione consente il calcolo della correlazione secondo la (2.2):

```
1  public class BMI implements CorrelationFunction{
2
3    private double beta;
4    private static final double def_beta=0.51f;
5    private static PMI mi = new PMI();
6
7    public BMI(){
8      this(def_beta);
9    }
10
11   public BMI(double b){
12     this.beta=b;
13   }
14
15   public double calculateCorrelation(double p_x, double p_y, double
         p_x_y) {
16     double p_nx_ny = 1 - (p_x + p_y - p_x_y);  // Pr(!x,!y)
17     double p_x_ny = p_x - p_x_y;       // Pr(x, !y)
18     double p_nx_y = p_y - p_x_y;     // Pr(!x, y)
19     double p_nx  = 1 - p_x;         // Pr(!x)
20     double p_ny  = 1 - p_y;         // Pr(!y)
21     double mi_x_y, mi_nx_ny, mi_x_ny, mi_nx_y;
22     double bmi;
23
24
25     mi_x_y = mi.calculateCorrelation(p_x, p_y, p_x_y);
26
27     mi_nx_ny = mi.calculateCorrelation(p_nx, p_ny, p_nx_ny);
28
29     mi_x_ny = mi.calculateCorrelation(p_x, p_ny, p_x_ny);
30     mi_nx_y = mi.calculateCorrelation(p_nx, p_y, p_nx_y);
31
32     bmi = beta * ( p_x_y * mi_x_y + p_nx_ny * mi_nx_ny)
33         - (1 - beta) * ( p_x_ny * mi_x_ny + p_nx_y * mi_nx_y);
```

```
34
35
36      return bmi;
37    }
38
39  }
```

Nelle righe dalla 16 alla 20 le probabilità passate per parametro vengono utilizzate per calcolare le probabilità rimanenti. Ad esempio abbiamo

$$\begin{aligned} Pr(\neg x \cap \neg y) &= 1 - Pr(x \cup y) \\ &= 1 - (Pr(x) + Pr(y) - Pr(x, y)) \end{aligned}$$

Successivamente vengono calcolati i valori della mutua informazione puntuale per ogni combinazione di x, y, $\neg x$, e $\neg y$. Quindi viene calcolata la BMI secondo la (2.2) e restituita in output.

3.3 La classe principale: FuzzyOntologyMiner

Questa classe contiene quattro metodi con le effettive implementazioni degli algoritmi descritti nella sezione 2.La classe è stata pensata come statica, e pensata in modo da favorire la sostituzione dei singoli passi.

3.3.1 Metodo: `extractTermFrequencies`

Questo metodo implementa l'algoritmo 1. Esaminiamo passo passo il codice

```
1  public static TermFrequencies<String> extractFrequencies(
2       String path, int windowSize, String[] searchPattern)
3           throws IllegalArgumentException, ClassNotFoundException,
                IOException{
```

Il metodo prende i seguenti parametri:

`String path`: percorso contenente i file da cui estrarre l'ontologia;

`int windowSize`: grandezza della finestra scorrevole;

`String[] searchPattern`: array di tag descriventi il pattern di ricerca desiderato.

In primo luogo controlliamo se la lunghezza del pattern è consistente con la grandezza della finestra:

```
1  if( searchPattern.length > windowSize)
2       throw new IllegalArgumentException("Il numero di tag deve
            essere minore della grandezza della finestra.");
```

Quindi inizializziamo le variabili e controlliamo se il path fornito è effettivamente una directory: in caso positivo recuperiamo la lista dei file.

```
    TermFrequencies<String> freq = new TermFrequencies<String>();
    File dir = null;
    File[] listOfFiles;
    StopWords sw = new StopWords();
    MaxentTagger tagger = new MaxentTagger("left3words-wsj-0-18.
        tagger");
    dir = new File(path);

    if(!dir.isDirectory())
      throw new IllegalArgumentException(path + " is not a directory.
          ");

    listOfFiles = dir.listFiles();
```

La classe MaxentTagger fa parte della libreria per il *POS tagging* sviluppata da [Toutanova et al.,]. Essa permette sia di instanziare un tagger, il quale si occupa di taggare una frase tramite il metodo `tagSentence`, sia di tokenizzare il testo tramite il metodo statico `tokenizeText`.

```
for(File file: listOfFiles){
      if(file.isFile()){

        List<List<HasWord>> sentences = null;
        ArrayList<TaggedWord> tSentence = new ArrayList<TaggedWord>()
            ;
        Morphology stemmer = new Morphology();
        TaggedWord curr;
        String term;

        sentences = MaxentTagger.tokenizeText(new BufferedReader(new
            FileReader(file)));
```

La classe `Morphology`, sempre appartenente alla libreria di Stanford per il *POS tagging*, fornisce i metodi per effettuare il *word stemming*. A questo punto abbiamo nella variabile `sentences` il testo tokenizzato diviso in frasi. Andiamo ora a lavorare sulla singola frase, dapprima effettuando il *POS tagging*, e creando un arraylist che conterrà la frase ripulita da parole inutili.

```
for(List<HasWord> sentence: sentences){
         tSentence = tagger.tagSentence(sentence);

         ArrayList<TaggedWord> filteredSentence = new ArrayList<
             TaggedWord>();
```

Il processo di filtraggio avviene nel codice seguente: abbiamo dovuto portare tutte le parole in lower case, in quanto i vari metodi sono case sensitive. Come è possibile vedere dal codice, dopo essere state portate in lower case, le parole sono soggette a stemming. Solo le parole ottenute che non sono stop words e rispettano l'espressione regolare \w+ (che è una

shortcut per l'espressione regolare [A-Za-z0-9]+, la quale consente di scartare i caratteri di punteggiatura e caratteri singoli) vengono aggiunte alla lista di termini filtrati.

```
1
2  for(TaggedWord word: tSentence){
3
4      word.setWord(word.word().toLowerCase());
5      word.setWord(stemmer.lemma(word.word(), word.tag()));
6      if(!sw.isStopWord(word.word()) && word.word().matches("\\w+")){
7        filteredSentence.add(word);
8      }
9  }
```

Ora bisogna implementare la finestra scorrevole: dato che la finestra scorre di una parola alla volta, il numero di finestre totali sarà `numParole - windowSize + 1`: quindi per ogni finestra richiamiamo il metodo `augmentWindows`, dopodiché passiamo alle operazioni da effettuare su ogni singola parola nella finestra. L'arraylist `alreadySeen` verrà utilizato per due scopi:

- Dato che la frequenza verrà calcolata in base al numero di finestre all'interno delle quali il termine compare e *non in base al numero di occorrenze*, prima di aumentare il numero di occorrenze verificheremo se il termine è già stato incontrato in precedenza;
- Se è la prima volta che si vede un termine, bisogna aumentare il numero di occorrenze in congiunzione con tutti i termini già visti, contenuti appunto in `alreadySeen`.

```
1  for(int i = 0; i < filteredSentence.size() - windowSize + 1; i++ ){
2
3            ArrayList<String> alreadySeen = new ArrayList<String>();
4            freq.augmentWindows();
5            for(int j = i; j < i + windowSize; j++){
```

Bisogna controllare se le `searchPattern.length` parole che seguono corrispondono al pattern di ricerca: prima controlliamo se "c'è ancora spazio" per questo controllo, verificando che la posizione attuale più la lunghezza del pattern non superino la lunghezza della frase. La variabile `term`, inizializzata come stringa vuota, conterrà il termine costruito seguendo il pattern, se il pattern viene rispettato. Se il controllo fallisce e non ci sono abbastanza termini per rispettare il pattern, il termine j-esimo viene direttamente considerato come singolo.

```
1  if( (j + searchPattern.length) < filteredSentence.size()){
2
3    term = "";
4
```

```
5     ...
6   } else {
7       term = filteredSentence.get(j).word();
8   }
```

Ora controlliamo in ordine il tag di ogni termine per verificare se coincide con il corrispettivo del pattern. Se la condizione alla riga 5 del seguente codice è verificata, allora c'è discrepanza fra tag attuale e tag cercato, quindi il termine j-esimo va considerato singolarmente e non come inizio del pattern. Pertanto alla variabile `term` viene assegnata la parola j-esima e si esce dal ciclo di controllo del pattern.

```
1  ...
2  for(int k=0; k < searchPattern.length; k++){
3
4                  curr = filteredSentence.get(j+k);
5
6                  if(!curr.tag().equalsIgnoreCase(searchPattern[k])){
7
8                    term = filteredSentence.get(j).word();
9                    break;
10                 }
```

Ottenuto il termine da prendere in considerazione controlliamo se è già stato visto: in caso negativo, aumentiamo il numero di occorrenze del termine e aumentiamo la sua frequenza congiunta ad ogni termine già visto. Quindi lo aggiungiamo all'arraylist `alreadySeen`.

```
1  if(!alreadySeen.contains(term)){
2
3      freq.addOccurrence(term);
4      for(String w: alreadySeen){
5        freq.addOccurrence(term, w);
6      }
7
8      alreadySeen.add(term);
9  }
```

Qui si chiude anche il ciclo principale, quindi invochiamo il metodo `computeFrequencies` sull'oggetto `freq` in modo da avere le frequenze relative e restituiamo tale oggetto in output.

```
1  freq.computeFrequencies();
2  return freq;
```

3.3.2 Metodo: `createContextVectors`

Questo metodo contiene l'implementazione dell'algoritmo 2. Il metodo prende in input i seguenti parametri:

`TermFrequencies<String> frequencies:` Un oggetto contenente i termini e le loro le frequenze:

`CorrelationFunction corrFunc:` Un'istanza di una classe che implementi l'interfaccia CorrelationFunction;

`double alpha:` Il valore della soglia di taglio per scartare i concetti.

```
public static ContextVectors<String> createContextVectors(
     TermFrequencies<String> frequencies, CorrelationFunction
         corrFunc, double alpha){
   int count;
   double fc, ft, fct, membership;
   ArrayList<String> terms = frequencies.getTerms();
   ContextVectors<String> contextVectors = new ContextVectors<
       String>(terms);
```

Ogni termine viene considerato come un concetto candidato, quindi per ognuno di essi costruiamo il vettore di contesto. Notiamo che non viene calcolata la membership di un termine con se stesso, in quanto è già stata settata pari a 1 dal costruttore della classe `ContextVectors`. Una volta calcolate e salvate tutte le membership, viene invocato il metodo `normalizeMemberships` in modo da portare tutti i valori nell'intervallo unitario [0, 1].

```
for(String concept: terms){
     fc = frequencies.getFrequency(concept);

     for(String term: terms){
       if(concept.equals(term)) continue;

       ft = frequencies.getFrequency(term);
       fct = frequencies.getFrequency(concept, term);

       membership = corrFunc.calculateCorrelation(fc, ft, fct);

       contextVectors.setMembership(term, concept, membership);
     }
   }
   contextVectors.normalizeMemberships();
```

Ottenuti i vettori di contesto a scala comune, andiamo a verificare quali di essi possono essere scartati, controllando quali di essi hanno tutte le componenti del vettore di contesto inferiori alla soglia `alpha`: i candidati concetti che non superano questo controllo vengono cancellati invocando il metodo `deleteConcept`, quindi restituiamo in output i vettori di contesto risultanti.

```
for(String concept: terms){
```

```
      count = 0;
      for(String term: terms){
        if ( contextVectors.getMembership(concept, term) < alpha)
            count++;
      }

      if(count == terms.size()-1)
        contextVectors.deleteConcept(concept);
    }

    return contextVectors;
```

3.3.3 Metodo: `createTaxonomy`

Questo metodo implementa l'algoritmo 3. Il metodo prende in input i seguenti parametri:

`ContextVectors<String> contextVectors`**:** vettori di contesto;

`double lambda`**:** soglia di accettazione delle relazioni.

```
public static Taxonomy<String> createTaxonomy(ContextVectors<String>
     contextVectors, double lambda){

    ArrayList<String> concepts = contextVectors.getConcepts();

    ArrayList<String> terms = contextVectors.getTerms();

    Taxonomy<String> taxonomy = new Taxonomy<String>(concepts);

    double num = 0, denom = 0;

    double membership_1, membership_2;

    double spec;
```

Il calcolo dei gradi di specificità viene fatto utilizzando la (2.3): vengono calcolati separatamente il numeratore e il denominatore, divisi, se il risultato è maggiore della soglia `lambda` allora la relazione viene salvata invocando il metodo `setSpecificity` dell'oggetto `taxonomy`. Una volta ottenuti tutti i valori viene lanciato il metodo `taxonomyPruning` in modo da potare la tassonomia.

```
for(String c1: concepts){
      for(String c2: concepts){

        if (c1.equals(c2))
          taxonomy.setSpecificity(c1, c2, 1);
```

```
6         else {
7
8           num = 0;
9           denom = 0;
10
11          for(String term: terms){
12
13            membership_1 = contextVectors.getMembership(c1, term);
14            membership_2 = contextVectors.getMembership(c2, term);
15
16            num += Math.min(membership_1, membership_2);
17
18            denom += membership_1;
19          }
20
21          spec = num / denom;
22
23          if(spec > lambda)
24            taxonomy.setSpecificity(c1, c2, spec);
25        }
26      }
27    }
28    taxonomy.taxonomyPruning();
29
30    return taxonomy;
```

3.3.4 Metodo: `saveOWL2FuzzyOntology`

Il metodo `saveOWL2FuzzyOntology`, a partire dal vettore dei concetti e dalle tassonomie, crea l'ontologia in formato OWL 2. Per poter far ciò è stata utilizzata la libreria Java open-source **OWL API** [Horridge and Bechhofer, 2011], la quale fornisce metodi per creare, manipolare e salvare file e modelli OWL . Il primo passo consiste nel creare un oggetto OWLOntologyManager e un oggetto OWLDataFactory, i quali permetteranno di gestire diversi metodi per la creazione e la manipolazione, e nel creare l'IRI per riferirsi all'ontologia.

```
1
2   OWLOntologyManager m = OWLManager.createOWLOntologyManager();
3
4   IRI ontIRI = IRI.create("http://www.semanticweb.org/ontologies/
        concetti");
5
6   OWLOntology ont = m.createOntology(ontIRI);
7
8   OWLDataFactory factory = m.getOWLDataFactory();
```

Il secondo passo è la creazione di una classe OWL, a cui sarà associata un'IRI relativa, per ogni concetto presente nel vettore dei concetti, salvandone i riferimenti in una mappa.

```
1
2  Map<String,OWLClass> classi=new HashMap<String,OWLClass>();
3  for(String con: concetti){
4
5    IRI iriClass = IRI.create(ontIRI + "#" + con);
6    OWLClass cl=factory.getOWLClass(iriClass);
7    classi.put(con, cl);
8  }
```

Successivamente per ogni coppia di concetti per cui è stata trovata una relazione si crea la proprietà OWL **subClassOf** aggiungendo, attraverso l'**AnnotationProperty** ed il metodo `createFuzzyAnnotation`, il valore della relazione stessa.

```
1
2  for(i=0; i<concetti.size(); i++){
3      for(j=0; j<concetti.size(); j++){
4        String sub=concetti.get(j);
5        String cls=concetti.get(i);
6        double spec=tax.getSpecificity(sub, cls);
7        if(spec > 0){
8          OWLAnnotation fuzzyAnnotation = createFuzzyAnnotation(
               factory, spec);
9
10         Set<OWLAnnotation> annotazioni = new HashSet<OWLAnnotation
               >();
11         annotazioni.add(fuzzyAnnotation);
12
13         s=classi.get(sub);
14         c=classi.get(cls);
15         OWLAxiom axiom = factory.getOWLSubClassOfAxiom(s, c,
               annotazioni);
16
17         AddAxiom add = new AddAxiom(ont, axiom);
18         m.applyChange(add);
19       }
20     }
21   }
```

Il metodo `createFuzzyAnnotation` si occupa semplicemente della costruzione della stringa che successivamente diventerà un'AnnotationProperty.

```
1  public static OWLAnnotation createFuzzyAnnotation(OWLDataFactory f
       , double value){
2    OWLAnnotationProperty fuzzyTag = f.getOWLAnnotationProperty(IRI.
       create("#fuzzyLabel"));
```

```
3     String fuzzyowl2 = "<fuzzyOwl2 fuzzyType=\"axiom\">\n\t<Degree
          value=\"" + value + "\"/></fuzzyOwl2>";
4     OWLLiteral fuzzyLiteral = f.getOWLLiteral(fuzzyowl2);
5     return f.getOWLAnnotation(fuzzyTag, fuzzyLiteral);
6   }
```

Infine il metodo `saveOntology` si occupa della scrittura dell'ontologia creata su un file.

```
1   m.saveOntology(ont, new FileOutputStream(fileName));
```

4 Test

Verranno qui presentati i test effettuati. A causa della limitata potenza hardware a nostra disposizione, siamo stati costretti ad usare file di test di dimensione relativamente piccola, con un limitato numero di termini. Tuttavia ciò costutisce anche una sorta di stress-test per l'algoritmo, che ci consentirà di osservare la sua capacità di estrazione di concetti a fronte di un limitato numero di dati. Essendo gli output troppo grandi per essere inclusi all'interno di questo documento, verranno mostrati parte dei risultati che a nostro avviso mostrano, nel bene e nel male, la qualità dei risultati. Per effettuare i test sono stati utilizzati due dataset, che indicheremo con i nomi *Hackers* e *Cryptography*, i quali verrano presentati ed analizzati nelle sezioni che seguono.

4.1 Hackers

Il primo caso di test, *hackers*, contiene il saggio "Hackers Heroes of the Computer Revolution" di Steven Levy, riguardante la cultura hacker e l'impatto che essi hanno avuto sul mondo dell'informatica a partire dai pionieri dell'MIT degli anni cinquanta fino ad arrivare agli ottanta ed alle vicende che vedono protagonista Richard Stallman, considerato da Levy come "l'ultimo grande hacker". La tabella 1 contiene i valori dei parametri utilizzati per il test *hackers*. Tali parametri sono stati scelti dopo prove successive, focalizzando l'attenzione sopratutto sulla grandezza della finestra di testo e sulle frequenze di taglio.

Terminato il primo passo dell'algoritmo, sono state esaminate 3506 finestre di testo, che hanno portato all'estrazione dei termini e delle frequenze mostrate nella tabella 2.

I termini "machine", "hacker", "mit", "program" e "samson" riassumono in effetti la trama del libro utilizzato come test. Purtroppo l'algoritmo estrae anche termini non desiderati come "like" o "use", la cui presenza può essere giustificata dal loro frequente uso da parte dello scrittore. Non è possibile eliminare tali falsi positivi mediante una semplice manipolazione dei

Parametro	Valore
windowSize	7
lower	0.036
upper	0.1
alpha	0.04
beta	0.55
lambda	0.022

Tabella 1: Parametri utilizzati per il dataset *hackers*

Termine	Frequenza
machine	0.07273245864232744
people	0.03793496862521392
hacker	0.09868796349115802
like	0.061038220193953226
work	0.045636052481460354
mit	0.043069024529378205
write	0.03793496862521392
use	0.044780376497432973
instruction	0.038505419281232175
program	0.09127210496292071
samson	0.05219623502567028

Tabella 2: Concetti candidati per il test *hackers* estratti dal primo passo dell'algoritmo

parametri, in quanto questo comporterebbe l'eliminazione di veri positivi: ad esempio abbassando il parametro upper a 0.60 oltre al concetto *like* anche il termine *hacker* verebbe scartato. Siamo convinti che l'aggiunta di altri testi il cui tema principale riguardi l'hacking a questo dataset, permetterebbe all'algoritmo di isolare meglio i concetti più significativi.

La creazione dei vettori di contesto porta con se buoni risultati, come mostrato nella tabella 3. Molti termini vengono correttamente associati ad un concetto che dovrebbero descrivere, anche se con grado discutibile: ad esempio siamo soddisfatti della comparsa di "people" come termine descrivente i concetti "samson" e "hackers", sebbene il loro grado sia relativamente basso. Buono anche l'uso di "write" come descrizione dei concetti "instruction" e "program", anche se il grado 1.0 forse è troppo alto. Alcuni invece non sono molto convincenti: tuttavia, eliminare tali relazioni aumentando il valore della soglia alpha, causerebbe l'eliminazione di vettori di contesto che a nostro avviso sono più significativi per questo contesto, ma che contengono valori relativamente bassi.

La tabella 4 mostra alcuni dei risultati ottenuti dalla creazione della tassonomia. Alcuni di essi sono molto buoni: l'individuazione di "hacker"

Termine	**Contesto**	**Grado**
instruction	program	0.5837699884732048
people	hacker	0.2265422256707387
people	samson	0.2875998756954409
mit	work	0.3088347053253056
program	work	0.5288165735854224
hacker	mit	0.6583099312503513
write	instruction	0.4392046432371759
write	program	1.0
use	samson	0.3874648806326899
machine	samson	0.30019857495802
machine	people	0.33378840566551393
like	people	0.6487899778658063

Tabella 3: Estratto dei vettori di contesto del test *Hackers*

come sottoconcetto di "people", "mit" e "samson" è molto convincente, così come l'individuazione di "instruction" come sottoconcetto di "program". Altre relazioni invece, come quella fra "instruction" e "people", o "use" e "like" andrebbero in qualche modo rimosse. Il problema è stato risolto aggiungendo i termini rumorosi alla lista delle stop words. Ciò tuttavia ha determinato una modifica delle frequenze dei termini rimanenti (leggermente più alte), cosa che a sua volta richiederebbe di cercare nuovi valori per i parametri, cosa non possibile entro i termini di consegna.

Sottoconcetto	**Concetto**	**Grado**
hacker	people	0.6282689039308634
hacker	mit	0.8323313346137051
hacker	samson	0.5272839410940927
work	mit	0.6248301841988633
mit	people	0.7117197443495283
samson	people	0.7409321280500031
instruction	program	0.6680015258610522
program	write	0.9315043010894793
use	people	0.6481634114184861
instruction	people	0.6379460758799467
like	people	0.7619477421105741
use	like	0.5581473444077601

Tabella 4: Estratto della tassonomia del test *Hackers*

4.2 Cryptography

Il dataset *cryptography* è formato da una serie di brevi articoli il cui tema centrale è la crittografia, spaziando da articoli generici su tale argomento ad articoli riguardanti alcuni aspetti specifici della crittografia. I dati sono stati estratti dall'enciclopedia libera *wikipedia*. Differentemente dal test *Hackers*, questo test è composto da articoli scritti da autori diversi, che trattano l'argomento da punti di vista lievemente differenti: ciò dovrebbe portare a risultati in generale migliori, nonostante in termini di numero di parole i due dataset siano pressoché uguali. La tabella 5 mostra i valori utilizzati per i parametri in questo test.

Parametro	**Valore**
windowSize	10
lower	0.03
upper	0.55
alpha	0.02
beta	0.55
lambda	0.022

Tabella 5: Parametri utilizzati per il dataset *Cryptography*

L'algoritmo ha analizzato 2281 finestre di testo in totale, estrapolando tra gli altri i termini elencati nella tabella 6 con relative frequenze.

Termine	**Frequenza**
cryptosystem	0.03463393248575186
keystream	0.03024989039894783
decryption	0.04866286716352477
secret	0.04997807978956598
secure	0.05392371766768961
problem	0.030688294607628234
rsa	0.031126698816308637
encrypt	0.03682595352915388
know	0.04164839982463832
scheme	0.04603244191144235
information	0.051731696624287594
cryptanalysis	0.04603244191144235
attacker	0.030688294607628234
example	0.03857957036387549
cryptographic	0.049539675580885574
generally	0.033318719859710653

Tabella 6: Alcuni concetti candidati del test *Cryptography*

Si nota che la maggior parte dei termini estratti sono effettivamente rappresentativi del contesto, come ad esempio “cryptosystem”, “keystream”, “rsa”, “attacker”. Tuttavia sono presenti alcuni termini rumorosi come “generally”, “make” ed “example”. Anche qui come in precedenza, la modifica dei parametri non consente di migliorare tali risultati: portando ad esempio la soglia *lower* a 0.36 si eliminerebbero i termini “include” e “generally”, ma anche “attacker”, “rsa” e “cryptosystem”. La tabella 7 mostra un estratto dei vettori di contesto ottenuti dopo l’esecuzione del secondo passo. Anche qui buona parte dei risultati sono buoni, si veda ad esempio l’individuazione dei termini “problem”, “rsa” e “information” come termini per descrivere il contesto “cryptosystem”. Non siamo riusciti tuttavia a trovare un valore del parametro *alpha* tale da eliminare “senza danni” concetti superflui come “generally” o “example”.

Termine	**Contesto**	**Grado**
problem	cryptosystem	0.0823620757628883
rsa	cryptosystem	0.39109735301762344
information	cryptosystem	0.047666649148074336
cryptanalysis	cryptosystem	0.08952609974510639
cryptanalysis	cryptosystem	0.08952609974510639
generally	cryptosystem	0.17344906001089036
problem	decryption	0.06187562621733617
computer	decryption	0.2308051037943146
secure	secret	0.07493091007248055
problem	secret	0.20851833714943493
know	secret	0.04276222610507922
choose	secret	0.05578125036999302
cryptanalysis	secret	0.1255967061089433
generally	secure	0.07809514159459775
secret	information	0.10770146394549972

Tabella 7: Estratto dei vettori di contesto del test *Cryptography*

A partire da questi dati è stata generata la tassonomia: la tabella 8 ne mostra un estratto.

Sottoconcetto	Concetto	Grado
cryptosystem	secure	0.5560688353174652
cryptosystem	rsa	0.6068825223282452
cryptosystem	make	0.4483878380875385
keystream	scheme	0.6069045996415785
keystream	technique	0.4989224357143318
decryption	problem	0.4280887242677086
decryption	attacker	0.4216906756383301
secret	secure	0.4678331917325567
secret	know	0.6271712087323913
secret	example	0.4468515339093231
encrypt	scheme	0.5335464258437376
cryptanalysis	problem	0.4813377559794309
computer	attacker	0.4972076120613539
generally	scheme	0.6289752386961022

Tabella 8: Estratto della tassonomia del test *Cryptography*

Nonostante i risultati siano molto buoni in generale, continuiamo a vedere relazioni superflue generate a partire dai termini rumorosi che sono stati estratti dopo il primo passo dell'algoritmo. Tali termini presentano valori che rendono impossibile il loro filtraggio attraverso la modifica dei parametri, in quanto si finisce sempre con il perdere informazioni che, anche se più importanti, presentano valori di membership minori. Anche qui il problema è stato risolto aggiungendo i termini "generally" ed "example" alla lista delle stop words, A parità di valori dei parametri, l'algoritmo con questa modifica ha estratto nuovi termini oltre a quelli della tabella 6, alcuni di essi sono mostrati nella tabella 9.

Termine	Frequenza
algorithm	0.1015018125323666
cipher	0.2739513205592957
ciphertext	0.0709476954945624

Tabella 9: Nuovi termini estratti dopo l'aggiunta dei termini rumorosi alle stop words

Ovviamente le modifiche si estendono anche ai vettori di contesto e alla tassonomia: di particolare interesse l'estrazione di nuove relazioni, come ad esempio la relazione `rsa is algorithm`, con grado circa 0.87.

5 Conclusioni e sviluppi futuri

Abbiamo studiato un algoritmo di data mining per l'estrazione automatica di ontologie fuzzy, presentandone una nostra implementazione in linguaggio Java. I test, nonostante siano stati effettuati su file di piccole dimensioni, hanno dato risultati abbastanza convincenti dato che l'algoritmo è stato in grado di estrarre buona parte dei concetti principali dei singoli contesti, sebbene con qualche falso positivo. L'algoritmo si presenta fortemente sensibile alla variazione dei parametri, i cui valori purtroppo vanno scelti tramite esperimenti empirici, magari su un sottoinsieme dei dati da analizzare. A questo proposito, crediamo che uno dei principali sviluppi di questo progetto possa consistere nell'individuazione di metodi automatici per la scelta dei valori di tali parametri, valori che anche se non ottimi potrebbero dare un buon punto di partenza. Questo ragionamento vale in particolar modo per i parametri relativi all'estrazione delle frequenze: grandezza della finestra di testo, frequenza di taglio inferiore e frequenza di taglio superiore, dato che a nostro avviso, questi sono i parametri a cui l'algoritmo è più sensibile. Di seguito alcuni spunti:

- La grandezza della finestra di testo influisce sulla media e sulla varianza delle frequenze ottenute, quindi le frequenze di taglio potrebbero dipendere in qualche modo direttamente da questo valore;
- Si potrebbero rendere le frequenze di taglio indipendenti dal dataset applicando una funzione di smoothing (ad esempio una gaussiana) all'intervallo compreso tra la frequenza minima e massima.

Altri possibili sviluppi possono riguardare l'uso di metodi alternativi ai singoli passi dell'algoritmo, ad esempio usare una formula alternativa alla (2.3) per il calcolo della tassonomia, e lo sviluppo di un plug-in per il software *Protegé*, che consenta di importare ontologie direttamente da file contenenti solo testo.

Indice

Riferimenti bibliografici

[Bobillo and Straccia, 2011] Bobillo, F. and Straccia, U. (2011). Fuzzy ontology representation using owl 2. *International Journal of Approximate Reasoning*, 52(7):1073 – 1094. Selected Papers - Uncertain Reasoning Track - FLAIRS 2009.

[Horridge and Bechhofer, 2011] Horridge, M. and Bechhofer, S. (2011). The owl api: A java api for owl ontologies. *Semant. web*, 2(1):11–21.

[Lau et al., 2007] Lau, R., Li, Y., and Xu, Y. (2007). Mining fuzzy domain ontology from textual databases. In *Web Intelligence, IEEE/WIC/ACM International Conference on*, pages 156–162.

[Toutanova et al.,] Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. Stanford log-linear part-of-speech tagger. `"http://nlp.stanford.edu/software/tagger.shtml"`.

[Toutanova et al., 2003] Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL 2003*, pages 252–259.