

UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA
FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
Corso di Laurea in Informatica



**Costruzione di un thesaurus per gli algoritmi
di prossimità semantica**

DISCo – LET

Dipartimento Informatica, Sistemistica e COmunicazione –
Laboratorio Elaborazione Testi

Supervisori:

Prof. Huu LE VAN

Prof. Luca BERNARDINELLO

Relazione della prova finale di:

Roberto FERMI

Matricola n°048091

Anno Accademico 2004 – 2005

Ringraziamenti

Vorrei sinceramente ringraziare tutti coloro che hanno partecipato alla realizzazione di questo progetto: in particolar modo sono grato al Prof. Le Van Huu per essere sempre stato disponibile a fornirmi chiarimenti ed aiuti nella fase di progettazione e stesura del presente elaborato.

Un particolare ringraziamento va anche alla mia famiglia, in particolare a mia sorella Barbara, che mi ha supportato ed aiutato durante il corso di questi tre anni universitari.

Uno speciale ringraziamento anche a mio zio Gianpaolo, grazie al quale ho iniziato il mio percorso di studi in informatica.

Infine ringrazio i miei compagni di corso con i quali ho condiviso questa bellissima esperienza universitaria, in particolare Ignazio, Giovanni e Giuseppe

Roberto Fermi

Indice

Ringraziamenti	I
Indice.....	II
Introduzione	IV
1. L'information Retrieval.....	- 1 -
1.1 La nascita dell'IR	- 1 -
1.2 L'information Retrieval.....	- 1 -
1.3 Valutazione del processo di retrieval	- 3 -
2. Cenni sui metodi di ricerca sui testi	- 4 -
2.1 Modello booleano.....	- 4 -
2.2 Il modello Fuzzy.....	- 5 -
2.3 Il modello probabilistico	- 6 -
2.4 Il modello a spazio vettoriale	- 7 -
3. Analisi dei testi.....	- 9 -
3.1 StopList	- 9 -
3.2 Stemming	- 10 -
3.3 Weighting	- 11 -
3.4 Thesaurus	- 12 -
3.4.1 La costruzione di un thesaurus	- 12 -
3.5 La ricerca nei testi	- 13 -
3.5.1 Ricerca di stringhe.....	- 13 -
3.5.2 Clustering	- 14 -
3.5.3 Thesaurus	- 15 -

Parte 1: Thesaurus

1. Cenni teorici e analisi.....	- 16 -
1.1 Somiglianza lessicale	- 16 -
1.1.1 L'analisi tra stringhe.....	- 16 -
1.2 Somiglianza semantica	- 17 -
1.2.1 Coefficiente del coseno	- 18 -
1.2.2 Coefficiente di Dice	- 18 -
1.2.3 Coefficiente di Jaccard	- 19 -
2. Implementazione	- 20 -
2.1 Architettura generale	- 20 -
2.2 Avvio dell'algoritmo (CreaThesaurus.java).....	- 23 -
2.3 Il monitor (Monitor.java)	- 24 -
2.4 Il thread principale (ThreadPrincipale.java).....	- 24 -
2.5 La fase di normalizzazione (Normalize.java)	- 25 -
2.6 La costruzione del thesaurus (Thesaurus.java)	- 26 -
2.7 Il conteggio di parole (WordCount.java)	- 27 -
2.8 La scrittura dei risultati su database (ThesaurusDatabaseTerm.java)	- 28 -
3. Risultati della costruzione	- 29 -
3.1 Il tempo di calcolo.....	- 30 -
3.2 I dati raccolti.....	- 30 -
4. Conclusioni.....	- 32 -

Parte 2: Algoritmo di Paice-Ramirez

1.	Cenni teorici ed analisi.....	- 33 -
1.1	Il funzionamento	- 33 -
1.1.1	La funzione di mapping.....	- 34 -
1.1.2	La funzione di tidyness	- 34 -
2.	Implementazione dell'algoritmo	- 37 -
2.1	Architettura generale	- 37 -
2.2	Avvio dell'algoritmo (Calcola.java)	- 40 -
2.3	La prossimità semantica (SemanticProximity.java).....	- 40 -
2.4	L'algoritmo di Paice-Ramirez (PaiceRamirez.java)	- 40 -
2.4.1	Il peso dei risultati	- 43 -
2.4.2	Esempio di funzionamento.....	- 45 -
3.	Risultati dell'algoritmo	- 48 -
3.1	Valutazione dei sinonimi.....	- 48 -
3.2	Valutazione dell'ordinamento	- 49 -
3.3	Valutazione della lunghezza delle frasi.....	- 50 -
3.4	Valutazione globale dell'algoritmo	- 53 -
3.5	Valutazione dei parametri precision e recall	- 55 -
4.	Conclusioni.....	- 60 -
	Istruzioni per l'uso	- 61 -
	Codice.....	- 63 -
	Bibliografia.....	- 83 -

Introduzione

Il contenuto del mio stage riguarda l'Information Retrieval, ovvero quella disciplina che si occupa di studiare, progettare e realizzare sistemi informatici finalizzati al reperimento di documenti riguardanti le richieste degli utenti.

Questa tematica ha suscitato il mio interesse perché ritengo che nella società moderna l'Information Retrieval rappresenta e rappresenterà sempre di più il modo più veloce ed efficace per memorizzare e ricercare informazioni su qualsivoglia argomento.

L'opportunità fornita da questo stage di sviluppare un algoritmo di prossimità semantica mi consente di approfondire le mie conoscenze riguardo l'Information Retrieval, le sue fasi di sviluppo, i differenti metodi di analisi e ricerca sui testi, le migliori metodiche di implementazione di thesaurus e algoritmi di prossimità semantica.

Il principale obiettivo di questo stage è quello di sviluppare un algoritmo di prossimità semantica fra testi in linguaggio naturale, utile per la costruzione di un sistema per il reperimento di informazioni. Il sistema viene usato per il confronto tra il fabbisogno informativo dell'utente (query) e la rappresentazione logica dei documenti (surrogati di documenti).

L'esigenza di un buon sistema per il reperimento di informazioni nasce dal bisogno di gestire la ricerca ed il recupero di dati da una gran numero di documenti. Gli IRS (Information Retrieval System), infatti, possono essere applicati in moltissimi campi con finalità varie e diversificate; un possibile utilizzo, ad esempio, potrebbe essere la realizzazione di un motore di ricerca interno al sito del laboratorio LET (Laboratorio Elaborazione Testi) per permettere agli utenti di trovare informazioni sulle tesi, sui documenti e sugli articoli raccolti nel server WEB.

La realizzazione di un buon algoritmo per la ricerca di prossimità semantica necessita dell'individuazione di un'ottima metrica per la definizione della somiglianza tra un documento e il fabbisogno informativo dell'utente; infatti l'utilizzo di uno strumento di confronto non soddisfacente porterebbe ad una inconsistenza dei risultati trovati. Fra i vari algoritmi proposti dalla letteratura, si è scelto quello di Paice-Ramirez che permette di definire una buona metrica per la determinazione della prossimità semantica tra coppie di frasi ovvero, in questo caso, tra una query e le frasi del documento in cui effettuare la ricerca.

Per ottenere dei valori efficaci di prossimità semantica e garantire un buon funzionamento dell'algoritmo di Paice-Ramirez è anche necessario individuare una relazione tra le parole e per poter determinare quanto due termini sono simili (o vicini di significato) è importante avere uno strumento che permetta di valutarne il grado di affinità. Il mezzo che consente di ottenere queste informazioni è il thesaurus.

Un thesaurus è un dizionario dei sinonimi in cui sono memorizzate delle coppie di termini con il relativo valore di somiglianza. L'algoritmo di Paice-Ramirez funziona analizzando la prossimità semantica tra le frasi, quindi, anche in questo caso, sarà opportuno determinare una metrica per calcolare quanto due vocaboli siano correlati tra loro nel contesto e, successivamente, per individuare una sorta di sinonimo di ciascuna parola.

Per determinare la somiglianza semantica tra coppie di termini è necessario valutare in uno spazio multidimensionale la distanza che esiste tra le rappresentazioni delle parole. Il coefficiente del Coseno permette di calcolare l'angolo compreso tra i vocaboli considerati.

Al fine di ottimizzare la ricerca della affinità tra i termini è opportuno effettuare un'operazione di normalizzazione sui documenti originali; questa procedura permette, infatti, di ridurre considerevolmente il numero di parole del testo, eliminando quelle non significative nel contesto e riducendo ad una radice comune le derivazioni di uno stesso termine (stemming). Un buon algoritmo per lo stemming delle parole è quello realizzato da Porter; applicato su vocaboli della

lingua inglese, infatti, questa procedura è in grado di rimuovere successivamente i diversi suffissi, riportando, con una buona precisione, il termine alla sua forma originale.

La presente relazione è suddivisa in due parti distinte, precedute da una parte teorica riguardante le informazioni generali dell'information retrieval e dei sistemi per il reperimento di informazioni.

Nella prima sezione viene trattata la teoria, l'analisi, l'implementazione e la valutazione dell'algoritmo che permette la costruzione del thesaurus; vengono analizzati gli aspetti positivi e negativi inerenti la tecnica utilizzata per la sua realizzazione; si illustrano in dettaglio il funzionamento e la realizzazione del codice ed, infine, vengono presentati i test effettuati con le analisi dei risultati ottenuti.

Nella seconda parte viene trattata la teoria, l'analisi, l'implementazione e la valutazione dell'algoritmo di prossimità semantica di Paice-Ramirez. In particolare viene presentata la formula utilizzata, analizzandone in dettaglio le varie sezioni. Nel secondo capitolo vengono illustrate le fasi di implementazione del codice e le scelte fatte nella valutazione dei parametri di personalizzazione. Infine vengono riportati i test eseguiti sull'algoritmo illustrandone i risultati.

1. L'information Retrieval

Il termine *Information Retrieval* (Reperimento di Informazioni) è molto vago e, spesso, non stabilito correttamente. In molti casi si può definire adeguatamente semplicemente sostituendo il termine *Information* con *Document*.

Ciononostante il termine *Information Retrieval* è utilizzato per indicare quella disciplina che si occupa di studiare, progettare e realizzare sistemi informatici, denominati *Information Retrieval Systems* (Sistemi per il Reperimento di Informazioni).

Un sistema per il reperimento di informazioni ha il compito di informare l'utente sull'esistenza (o la non esistenza) ed sul dove reperire documenti riguardanti la sua richiesta.

1.1 La nascita dell'IR

I testi possono essere raccolti e preservati in archivi o biblioteche per diverse ragioni. Il motivo originario fu sicuramente la necessità di individuare un luogo dove riunire e catalogare dati amministrativi. A questo si aggiunse il bisogno di salvaguardare documenti di pubblica utilità o rilevanza (es. contratti, leggi...).

L'organizzazione utilizzata in antichità e nel Medio Evo era incentrata sullo scaffale su cui il libro era collocato e sull'autore dello stesso. L'invenzione del torchio tipografico e la rapida crescita della letteratura nel Rinascimento e nell'Illuminismo resero l'organizzazione delle biblioteche e il reperimento di informazioni sempre più complessi. Ciononostante non sembrò esserci una ragione urgente per ideare una nuova organizzazione delle librerie fino al tardo XIX secolo quando, finalmente, si sentì il bisogno di un sistema di catalogazione basato sul soggetto del libro piuttosto che sull'autore. Il maggior impulso che condusse a questo drastico cambiamento arrivò dalla scienza e dalla tecnologia.

Nel 1976 Dewey pubblicò un libro intolato "A classification and subject index for cataloguing and arranging the books and pamphlets of a library" che avrebbe avuto ampi effetti. Sebbene la sua classificazione sia ancora largamente utilizzata in tutto il mondo, il vero valore del lavoro di Dewey consiste nell'idea di:

- introdurre dislocazioni relative, anziché assolute.
- assegnare numeri decimali ai libri piuttosto che agli scaffali, rendendo la specifica degli argomenti più fattibile.
- introdurre la misura di un indice relativo sugli argomenti.

In sintesi la differenza sostanziale tra il sistema di Dewey e quelli precedenti è data dal fatto che l'argomento viene collegato al libro e non più allo scaffale e che i libri vengono posizionati utilizzando uno schema di numerazione lineare.

Con l'avvento dei computer si è pensato di utilizzare queste macchine per fornire un rapido ed intelligente sistema di reperimento delle informazioni. Nelle biblioteche, ad esempio, i *retrieval system* sono ampiamente utilizzati per la catalogazione e l'amministrazione dei volumi.

1.2 L'information Retrieval

Quando effettuiamo ricerche in sistemi come biblioteche, non ricerchiamo informazioni, ma puntatori a libri, articoli o documenti, eventualmente le informazioni di cui abbiamo bisogno

sono da qualche parte in quei libri o articoli. Prima che una ricerca possa essere eseguita, il sistema deve collezionare alcune informazioni riguardo il contenuto dei documenti.

Quando l'informazione rilevante del documento è collezionata deve anche essere tradotta in una rappresentazione comprensibile dall'*Information Retrieval System* in modo che sia accessibile da quella parte del sistema che si occupa di accettare e tradurre le interrogazioni dell'utente. Le problematiche più comuni riguardo ai sistemi di reperimento delle informazioni sono le seguenti:

- Quale parte del documento e quale semantica deve essere considerata per l'analisi?
- Quale deve essere l'unità metrica per la descrizione?
- Quali dispositivi per la manipolazione sintattica e semantica sono a disposizione durante la ricerca?

I parametri di ingresso di un *Information Retrieval System* sono il surrogato del testo, ovvero le informazioni rilevanti tratte dal testo originale, e la richiesta dell'utente, ovvero una query, che può essere espressa nel linguaggio naturale, oppure mediante altre espressioni (come, ad esempio, quelle booleane).

Il processo di retrieval, a questo punto, può essere sintetizzato nei seguenti passi:

- Creazione di una rappresentazione di ogni oggetto, basata sulla sua descrizione
- Creazione di una rappresentazione del fabbisogno informativo, query, dell'utente.
- Confronto tra le due rappresentazioni e scelta di quelle che meglio rappresentano il fabbisogno informativo dell'utente.

La parte critica di tutta la procedura è rappresentata dalla necessità di trovare rappresentazioni "funzionali" degli oggetti (che possono essere documenti, articoli, libri, etc.), in modo da ottimizzare il confronto tra la query dell'utente e gli oggetti di ricerca.

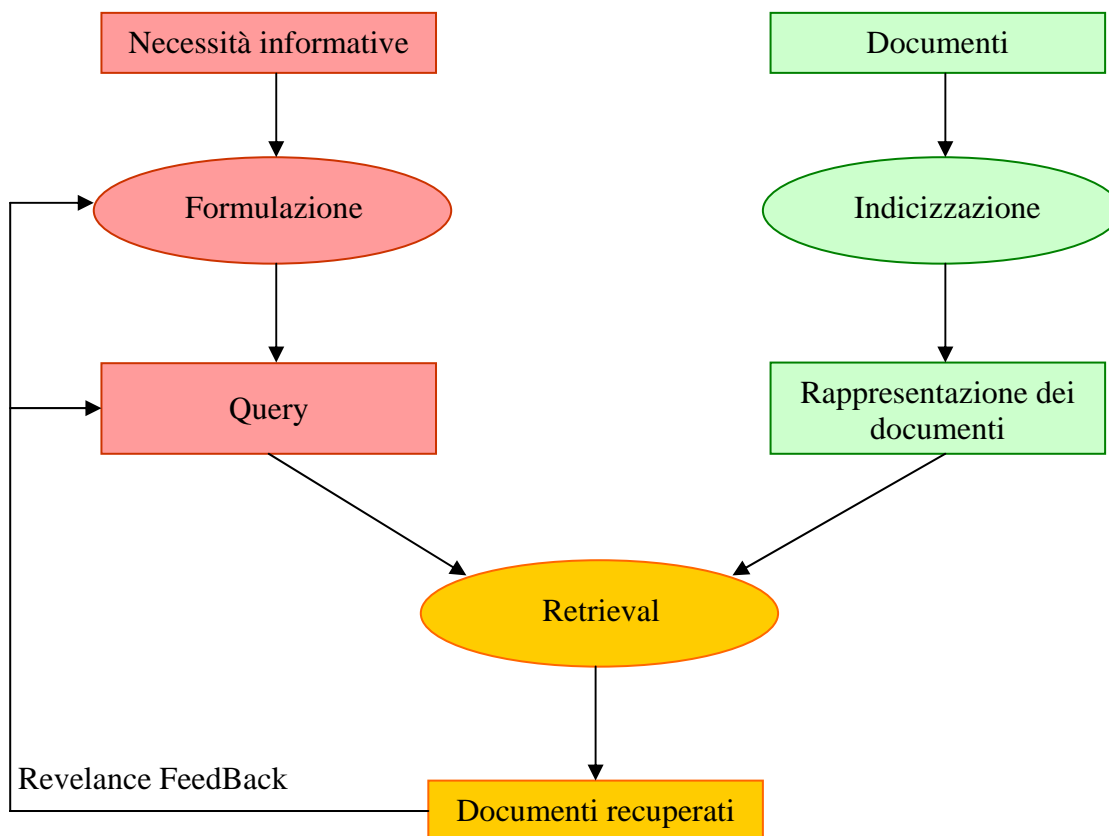


Fig 1.2 – 1 Modello classico delle attività di un sistema per il reperimento di informazioni

1.3 Valutazione del processo di retrieval

Un parametro importante da valutare per un *Information Retrieval System* è l'efficacia, ovvero la capacità di soddisfare l'utente fornendogli solamente informazioni rilevanti ed escludendo quelle superflue. L'obiettivo ottimale è quello di fornire all'utente solamente risultati totalmente pertinenti alla sua richiesta, escludendo tutto il resto.

In pratica questo traguardo non è raggiungibile e, quindi, si utilizzano due parametri per valutare l'efficacia di un processo di retrieval: *recall* (richiamo) e *precision* (precisione).

Questi due parametri di riferimento sono proporzionali al numero di documenti pertinenti individuati; quello che cambia è il peso assegnato a ciascun documento. Più in particolare, la precisione è data dal rapporto tra il numero di documenti pertinenti recuperati e il numero di documenti recuperati. Il richiamo, invece, è definito come il rapporto tra il numero di documenti recuperati pertinenti e il numero di documenti pertinenti.

$$precision = \frac{documenti_recuperati_pertinenti}{documenti_recuperati}$$

$$recall = \frac{documenti_recuperati_pertinenti}{totale_documenti_pertinenti}$$

La precisione, pertanto, misura l'abilità del sistema nel recuperare solamente documenti rilevanti per la richiesta dell'utente. Il richiamo misura l'abilità del sistema nel recuperare tutti i documenti rilevanti.

Un buon sistema per il reperimento delle informazioni dovrà essere strutturato in modo da massimizzare precisione e richiamo e nello stesso tempo diminuire il silenzio (ovvero l'assenza di informazioni) e il rumore (ovvero il deterioramento delle informazioni).

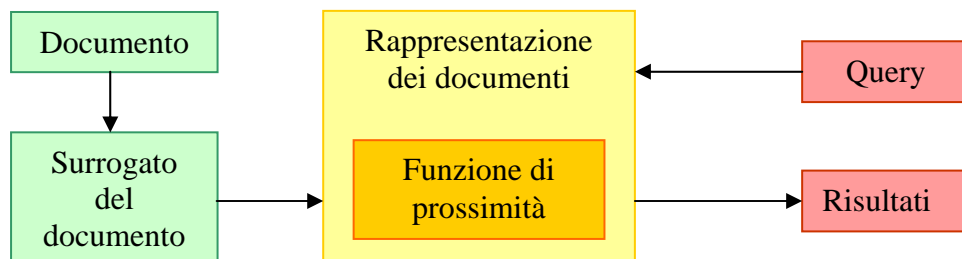


Fig 1.3 – 1 Modello classico di un sistema per il reperimento di informazioni

2. Cenni sui metodi di ricerca sui testi

Vi sono diversi metodi per effettuare ricerche su testi; le tecniche più efficaci sono quelle che analizzano la semantica del testo e ne ricavano delle parole chiave rappresentative dell'argomento del documento, utilizzate come se fossero un "riassunto" del testo. L'individuazione di queste *keyword* è molto complessa in quanto è necessario comprendere la semantica del documento che si sta analizzando.

L'idea che sta alla base di tutti i metodi di ricerca sui testi è quella di eseguire una ricerca preliminare su tutto il documento al fine di eliminare tutte le parole non rilevanti o comuni, basandosi su alcune regole sintattiche.

Nell'analisi lessicale di un documento si possono ritrovare questi punti principali di elaborazione:

- Eliminazione delle *stop-words*, ovvero quelle parole che non hanno significato dal punto di vista semantico del documento (es. articoli, congiunzioni, avverbi, etc.).
- Riduzione delle parole alla loro radice grammaticale. Questa procedura viene chiamata *stemming*.
- Sulla base del testo elaborato nei punti precedenti si procede alla costruzione di un thesaurus, ovvero un particolare dizionario dei sinonimi dove sono memorizzate coppie di parole insieme ad un valore numerico che ne indica la somiglianza semantica.

Mediante questi passi è possibile ridurre il documento originale ad un surrogato di documento (Vedi Fig. 1.3-1), cioè un riassunto significativo del documento originale. In questo modo si ottiene una rappresentazione logica dei testi più compatta che può essere utilizzata in maniera più efficiente dai sistemi di retrieval.

Una volta ottenuta un'elaborazione dei documenti è necessario un sistema di reperimento delle informazioni che stabilisca una metodologia per il paragone tra il fabbisogno informativo dell'utente (query) e la rappresentazione logica dei documenti (surrogati dei documenti). Vi sono molte metodologie di analisi dei testi; quelle più diffuse sono:

- Modello booleano.
- Modello Fuzzy.
- Modello probabilistico.
- Modello dello spazio vettoriale.

Ogni sistema di retrieval è caratterizzato da aspetti tipici e, soprattutto, da una corretta rappresentazione logica dei testi.

2.1 Modello booleano

Il modello booleano si basa sul confronto tra i termini che compongono l'indicizzazione del documento, ovvero il surrogato del documento, ed una query booleana dell'utente, che rappresenta il fabbisogno informativo.

Partendo da una collezione di documenti indicizzati attraverso dei vettori di termini è possibile formulare una tesi combinando i classici operatori logici ottenendo i seguenti risultati:

- **<termine1> AND <termine2>** comporta il recupero dei documenti che all'interno dei vettori di indicizzazione presentano contemporaneamente il **<termine1>** e il **<termine2>**.
- **<termine1> OR <termine2>** comporta il recupero dei documenti che all'interno dei vettori di indicizzazione presentano il **<termine1>** oppure il **<termine2>**.
- **NOT <termine1>** comporta il recupero dei documenti che all'interno dei vettori di indicizzazione non presentano il **<termine1>**.

Il modello booleano è molto veloce e permette di definire il fabbisogno informativo dell'utente in modo molto preciso. Uno dei principali difetti di questo metodo consiste nel fatto che il sistema di interrogazione basato su query booleane è poco comprensibile, soprattutto quando si ha la necessità di esprimere richieste strutturate. In questo caso, infatti, le query possono presentarsi difficili da implementare e da comprendere per gli utenti che hanno meno dimestichezza con questo argomento. Un secondo problema risiede nell'inadeguatezza dell'uso dell'algebra booleana per formalizzare l'incertezza che, inevitabilmente, si accompagna all'indicizzazione di documenti e testi. La distinzione tra i valori possibili, VERO/FALSO, infatti, non è adeguata nel momento in cui si tenta di esprimere nel proprio fabbisogno informativo un concetto.

2.2 Il modello Fuzzy

Il modello Fuzzy viene utilizzato per superare i limiti del modello booleano, mantenendone, nello stesso tempo, gran parte dei vantaggi di precisione, accuratezza e velocità.

I primi concetti della logica Fuzzy sono stati introdotti da Zadeh nel 1965, attraverso il suo scritto "fuzzy sets".

Un insieme Fuzzy è caratterizzato da una funzione che permette di identificare quanto verosimilmente possiamo associare un valore ad un concetto. La funzione di base ha di solito un andamento Gaussiano, detto anche "a campana".

Esempio

Vogliamo rendere conto del concetto di altezza. Consideriamo il valore di 1,75m come altezza normale. La percezione di altezza normale si può estendere in misura minore anche nel caso di individui di altezza compresa tra 1,70m e 1,80m. E' anche vero che nel caso di altezza pari a 1,65m, o 1,85m si può parlare con una certa sicurezza di una persona rispettivamente bassa o alta. Non risulta quindi possibile definire con precisione una soglia che identifichi l'idea di "altezza normale". Per questo motivo, se avessimo scelto un modello come quello booleano, esso non sarebbe risultato realistico nella definizione di un concetto come questo.

La logica Fuzzy, invece, è ideale per rappresentare l'incertezza correlata al processo di indicizzazione di un documento e al suo successivo reperimento poiché introduce il concetto di rilevanza di un documento. In questo modo un documento può essere considerato molto rilevante, poco rilevante, abbastanza rilevante e così via, rispetto ad un termine.

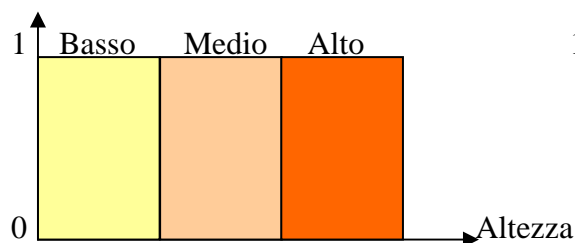


Fig 2.2 - 1 Sistema booleano

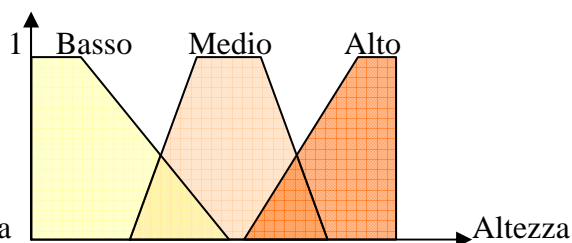


Fig 2.2 - 2 Sistema Fuzzy

2.3 Il modello probabilistico

Alla base del modello probabilistico si presentano due parametri fondamentali, la probabilità di rilevanza ($p_{(ril)}$) e la probabilità di non rilevanza ($p_{(non_ril)}$) di un determinato indice.

Se si assume che la rilevanza abbia proprietà binaria, ovvero che un indice possa essere solo rilevante o non rilevante, si avrà che:

$$P_{(non_ril)} = 1 - P_{(ril)}$$

Ora consideriamo altri due parametri:

$$a_1 \text{ e } a_2$$

che rappresentano rispettivamente la perdita associata al reperimento di un indice non rilevante ed al non reperimento di un indice rilevante.

Il reperimento di un indice non rilevante porta, quindi, ad una perdita pari a:

$$a_1 * (1 - P_{(ril)})$$

Diversamente, scartare un indice rilevante porta ad un fattore di perdita pari a:

$$a_2 * P_{(ril)}$$

La perdita totale è minimizzata se:

$$a_2 * P_{(ril)} \geq a_1 * (1 - P_{(ril)})$$

Allo stesso modo si introduce una funzione di reperimento g data da:

$$g = \frac{P_{(ril)}}{1 - P_{(ril)}} - \frac{a_1}{a_2}$$

Un documento viene reperito nel momento in cui g è non negativo.

Dimostrazione:

Consideriamo la definizione

$$a_2 * P_{(ril)} \geq a_1 * (1 - P_{(ril)})$$

essa diventa:

$$a_2 * P_{(ril)} - a_1 * (1 - P_{(ril)}) \geq 0.$$

Ora consideriamo la funzione di reperimento

$$g = \frac{P_{(ril)}}{1 - P_{(ril)}} - \frac{a_1}{a_2}.$$

Se calcoliamo il MCD tra $1 - p_{(ril)}$ e a_2 si avrà:

$$\frac{a_2 * p_{(ril)} - a_1 * (1 - p_{(ril)})}{1 - p_{(ril)} * a_2}$$

In precedenza abbiamo definito che la funzione g è ottimale nel momento in cui essa è maggiore di zero. Se poniamo la funzione $g \geq 0$, il denominatore viene eliminato e si giunge a

$$a_2 * p_{(ril)} - a_1 * (1 - p_{(ril)}) \geq 0.$$

Quindi, come si voleva dimostrare, se g è maggiore o uguale a zero, la perdita di informazione viene minimizzata.

Il modello probabilistico fornisce uno strumento teoricamente provato e potente per caratterizzare i processi di reperimento, ma non ha trovato un valido campo di applicazione pratica a causa della difficoltà incontrata nella stima della probabilità di occorrenza dei termini nella collezione. Senza una funzione di distribuzione abbastanza accurata i risultati sperimentali dell'approccio probabilistico sono scarsi.

2.4 Il modello a spazio vettoriale

Nel modello a spazio vettoriale, il fabbisogno informativo dell'utente e il surrogato di documento sono rappresentati mediante l'utilizzo di strutture dati vettoriali. L'analisi che il sistema andrà ad effettuare sarà eseguita su coppie di vettori, rappresentati da query e indicizzazione dei documenti.

Utilizzando questo metodo, si avrà che:

$$D_i = (d_{i1}, d_{i2}, d_{i3}, \dots, d_{in})$$

e

$$Q_j = (q_{j1}, q_{j2}, q_{j3}, \dots, q_{jm})$$

Dove i valori d_{in} e q_{jm} rappresentano i termini n e m dei documenti D_i e del fabbisogno informativo dell'utente Q_j .

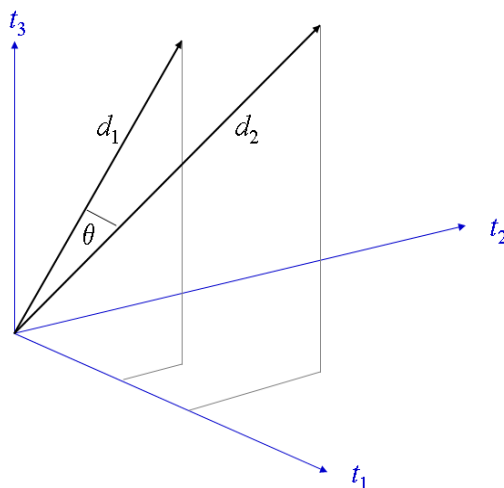


Fig 2.4 – 1 Rappresentazione del modello a spazio vettoriale. Lo spazio ha una dimensionalità pari al numero di termini del vocabolario e i termini sono rappresentati da vettori nello spazio. L'angolo θ rappresenta la distanza tra i termini.

Un primo modo di utilizzo del modello a spazio vettoriale è quello di porre a zero i coefficienti della query Q che non compaiono nel documento D, considerando uno i rimanenti coefficienti.

Oltre a questa primitiva applicazione, però, si può pensare di assegnare valori compresi in un determinato *range* (ad esempio [0,1]) in base all'importanza del termine all'interno del documento o del fabbisogno informativo dell'utente.

La possibilità di ottenere dei valori di similitudine tra i documenti e le query, compresi in un determinato intervallo chiuso e limitato, permette di effettuare un ordinamento dei documenti in base ad un valore di somiglianza con il fabbisogno informativo dell'utente. Si può, inoltre, prendere in considerazione una soglia di rilevanza al di sopra della quale i documenti sono considerati importanti rispetto alla query dell'utente, mentre al di sotto di tale valore vengono scartati in quanto ritenuti non significativi.

Esistono diversi metodi per "misurare" la somiglianza tra i termini di due vettori ed, in particolare, i principali sono:

Prodotto interno:
$$\sum_{i=1}^n x_i * y_i$$

Coefficiente di Dice:
$$\frac{2(\sum_{i=1}^n x_i * y_i)}{\sum_{i=1}^n (x_i)^2 + \sum_{i=1}^n (y_i)^2}$$

Coefficiente del Coseno:
$$\frac{\sum_{i=1}^n x_i * y_i}{\sqrt{\sum_{i=1}^n (x_i)^2 * \sum_{i=1}^n (y_i)^2}}$$

Coefficiente di Jacard:
$$\frac{\sum_{i=1}^n x_i * y_i}{\sum_{i=1}^n (x_i)^2 * (y_i)^2 - \sum_{i=1}^n x_i * y_i}$$

3. Analisi dei testi

Come visto in precedenza, un sistema per il reperimento di informazioni che agisce su testi scritti in linguaggio naturale deve eseguire alcune operazioni sui documenti in modo da recuperare da essi solamente i termini rilevanti. Più in particolare, prima che un documento possa essere utilizzato come base per il reperimento delle informazioni deve essere privato di tutte quelle parole comuni che non introducono significato al testo come, ad esempio, articoli, congiunzioni, preposizioni. Quindi è necessario ridurre tutti i verbi presenti nel documento ad un tempo infinito, eliminando, quindi, le eventuali coniugazioni. I termini devono essere ridotti alla minima radice, in modo da minimizzare il numero di operazioni da effettuare durante i confronti.

Le principali fasi sono:

- Stoplist
- Stemming
- Weighting
- Thesaurus

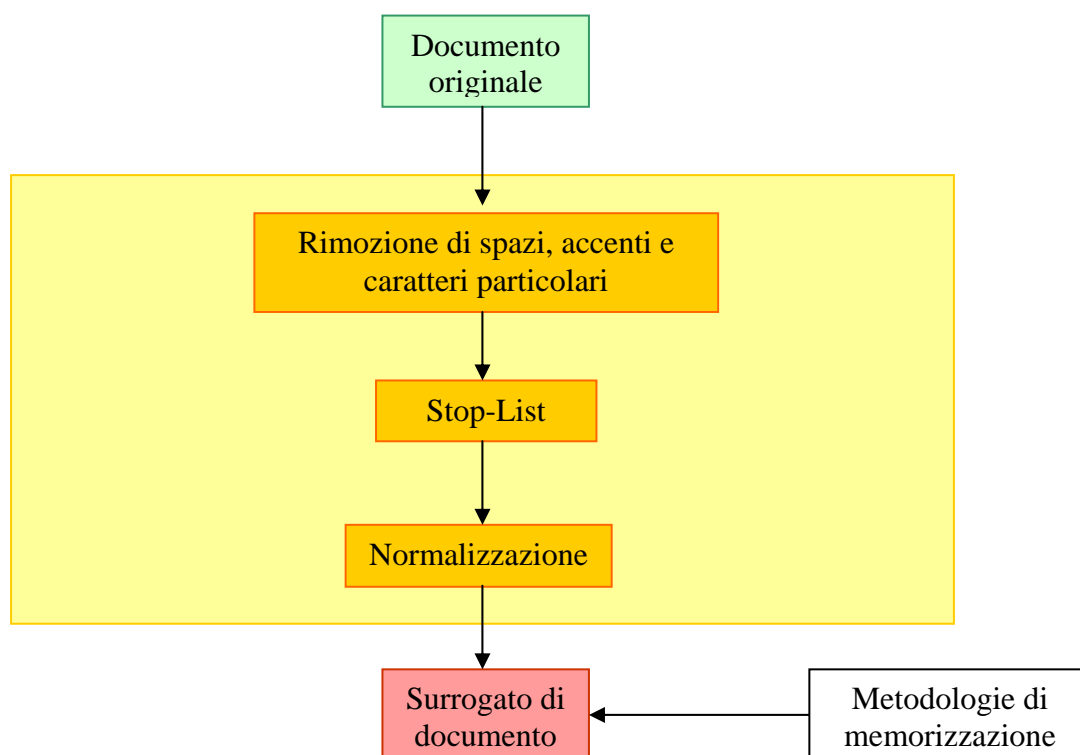


Fig 3 – 1 Schema logico delle operazioni eseguite su un testo per ottenere un documento utile all'analisi con algoritmi per la costruzione di thesaurus e prossimità semantica

3.1 StopList

Una stop-list è un elenco di parole che devono essere eliminate quando vengono incontrate nel documento che si sta analizzando.

Le parole contenute in una stop-list sono, generalmente, termini non significativi per la selettività del documento come, ad esempio, articoli, preposizioni, avverbi, congiunzioni, etc.

Nella stop-list, inoltre, possono essere inserite tutte quelle parole che, in base al contesto del fabbisogno informativo dell'utente e dell'applicazione del sistema per il reperimento delle informazioni, vengono ritenute non significative.

3.2 Stemming

Lo stemming è un processo che comporta la normalizzazione (o radicalizzazione) delle parole di un documento. Questo metodo permette di ridurre le varianti di una stessa parola ad una radice comune. Nella maggior parte dei casi, infatti, le variazioni morfologiche delle parole hanno interpretazioni semantiche simili e possono essere considerate come equivalenti allo scopo delle applicazioni dell'*Information Retrieval*.

Lo stemming di un documento permette di migliorare i valori del parametro "*recall*" in quanto riduce le parole ad una comune radice morfologica. Tuttavia, lo stemming tende a ridurre la precisione qualora non si ponga attenzione al rischio di rendere ad una radice comune termini che non sono correlati nel significato. Lo stemming riduce, inoltre, la dimensione degli indici dei termini dei documenti.

Nel tempo sono stati sviluppati diversi algoritmi di stemming che si differenziano per il modo in cui ricercano la radice comune di una parola.

L'idea del processo di stemming consiste nel ricercare la radice di una parola, applicando una serie successiva di regole che eliminano gli eventuali suffissi.

Ad esempio:

Collected, Collecting, Collection, Collections vengono tutte ridotte alla stessa radice *Collect*.

Se si considera la frase "*This means you can collect a library of images*" questa diventerà "*mean can collect library image*" (in questo esempio sono state eliminate alcune parole mediante una stop-list).

La valutazione di un processo di stemming può essere fatta analizzando le parole in *output* e confrontandole con un dizionario. Se la parola esiste nel dizionario, allora lo stemmer ha prodotto un troncamento di suffissi valido, altrimenti ha commesso un errore. Tuttavia, non è sufficiente verificare che la parola sia presente in un dizionario, ma è necessario controllare che il significato originale del termine sia mantenuto.

Negli algoritmi di stemming si possono valutare due tipi di errore:

- **Overstemming**: si verifica quando lo stemmer rende alla stessa radice parole che in realtà hanno significati diversi. Questo fa sì che il termine non sia correttamente interpretato e causa un peggioramento del parametro *Precision* del sistema per il reperimento di informazioni.
- **Understemming**: si verifica quando lo stemmer crea diverse radici da parole che, in realtà, hanno la stessa origine. Questo fa sì che un concetto sia suddiviso su diversi termini, influenzando in negativo il parametro *Recall* del sistema per il reperimento delle informazioni.

Un algoritmo di stemming che produce valori bassi di overstemming e understemming influenzerà positivamente la *precision* e la *recall* di un *Information Retrieval System*.

Il Porter Stemmer è tra i più conosciuti e utilizzati nella lingua inglese. Si basa sul fatto che i suffissi di tale lingua (circa 1200) sono combinazioni di suffissi più semplici. Questo algoritmo lineare analizza le parole in cinque passi successivi eliminando, di volta in volta, diverse tipologie di suffisso.

Il suo difetto è che non presta attenzione al significato dei termini che prende in considerazione perchè non ricerca una vera e propria radice linguistica, ma tenta di risalire ad una possibile radice nel minore tempo possibile. Al fine di minimizzare questo difetto, l'algoritmo di Porter

può essere utilizzato unitamente ad un dizionario di termini: se la radice trovata è presente nel dizionario della lingua corrente, allora lo stemmer ha prodotto un risultato accettabile, altrimenti si ignora la radice trovata e si utilizza il termine originale.

L'algoritmo di Porter presenta un basso numero di errori di overstemming, ma un più elevato numero di errori di undestemming.

Se, ad esempio, consideriamo il seguente gruppo di termini:

“Divide”, “Dividing”, “Divided”, “Division”, “Divisor”, “Divine”, “Divination”

questi possono essere raggruppati in due gruppi:

“Divide”, “Dividing”, “Divided”, “Division”, “Divisor” → gruppo numero 1

“Divine”, “Divination” → gruppo numero 2

In realtà, si può notare che effettuando uno stemming mediante il Porter stemmer questi gruppi sono 4.

Termine originale	Termine stemmato	
Divide	Divid	} Gruppo n°1
Dividine	Divid	
Divided	Divid	
Division	Divis	} Gruppo n°2
Divisor	Divisor	} Gruppo n°3
Divine	Divin	} Gruppo n°4
Divination	Divin	

Come si può notare dalla precedente tabella, l'algoritmo ha compiuto un errore di overstemming sulle parole in *input*.

3.3 Weighting

Un altro importante aspetto dell'*information retrieval* è il calcolo della frequenza dei termini in un documento.

Una prima regola da considerare è che se un termine compare molte volte in uno stesso documento, allora sarà presumibilmente correlato al contenuto. Questo è vero se viene fissata una soglia massima di rilevanza al di sopra della quale il termine non viene più considerato pertinente. Basti pensare, infatti, ad articoli e congiunzioni: queste parole sono ripetute molto spesso nel corso di un documento, ma, ciononostante, non lo caratterizzano.

Un'altra considerazione da fare è che le parole aventi un'alta frequenza tendono ad essere quelle che hanno numerosi significati.

Il metodo probabilistico per il peso dei termini prende in considerazione due componenti:

- TF (*Term Frequency*): specifica la frequenza di un termine del testo;
- IDF (*Inverse Document Frequency*): rappresenta un valore inversamente proporzionale alla frequenza di un termine nel documento.

Vi sono diverse funzioni utilizzate per calcolare il peso dei termini di un documento. Le principali sono:

- **Frequency Weighting**

$$g_{(x)} = \sum_{i \in query} t^* f_i * \log\left(\frac{p_i(1-q_i)}{(1-p_i)q_i}\right)$$

dove,

x è un vettore di parole che descrive il documento preso in esame;

$t^* f_i$ è la probabilità che il termine i sia usato per indicizzare il documento;

p_i è la probabilità che il termine i sia assegnato ad un documento qualsiasi tra quelli rilevanti (viene scelto casualmente).

q_i è la probabilità che il termine i sia assegnato ad un documento qualunque tra i documento non rilevanti

In questo caso, la componente TF è pari a $t^* f_i$, mentre la componente IDF è pari a

$$\log\left(\frac{p_i(1-q_i)}{(1-p_i)q_i}\right)$$

- **Sense weighting**

Anche in questo caso viene utilizzata la funzione $g(x)$ precedentemente esplicitata, ma la

componente IDF è posta pari a $\log\left(\frac{1}{w_i}\right)$, dove w_i è il numero di significati associati al

termine i -esimo (determinato in base ad un dizionario).

3.4 Thesaurus

Un thesaurus è un particolare dizionario dei sinonimi. In dettaglio, nell'*information retrieval* è uno strumento molto potente con cui si può determinare quanto due termini sono simili.

E' composto da una lista di coppie di termini cui è assegnato un valore di prossimità che specifica quanto i due vocaboli sono simili.

Ad esempio, supponendo che il *range* della prossimità possa variare da 0 ad 1, allora si può affermare che due termini che hanno somiglianza prossima a 0 non sono sinonimi, mentre due parole che hanno valore prossimo ad 1 saranno molto collegate tra loro.

Termine 1	Termine 2	Somiglianza
-----------	-----------	-------------

3.4.1 La costruzione di un thesaurus

Mediante l'unione delle operazioni precedentemente descritte, è possibile realizzare un thesaurus. In particolare, la prima trasformazione da applicare è la normalizzazione del testo preso in considerazione. In questa fase verranno eliminati i termini non necessari e non rilevanti. Quindi sarà necessario cancellare dal testo quei termini memorizzati nella stop-list. I termini

rimasti nel documento, a questo punto, vengono processati da un algoritmo di stemming per l'eliminazione di eventuali suffissi.

Una volta ottenuto un testo normalizzato, si può procedere la realizzazione del thesaurus vero e proprio. I termini vengono confrontati tra di loro a coppie e, applicando opportune formule per il calcolo della somiglianza (che verranno presentate nei successivi capitoli), sarà possibile determinare per ogni coppia di termini il grado di somiglianza.

3.5 La ricerca nei testi

Nella società moderna, la ricerca nei testi ricopre un ruolo fondamentale, basti pensare all'enorme utilizzo dei motori di ricerca sul WEB. Ma anche in situazioni più comuni si può riscontrare l'utilizzo di algoritmi per la ricerca di prossimità semantica tra testi: i sistemi di guida in linea dei software, integrano sempre un sistema per la ricerca delle informazioni necessarie all'utente; quasi tutte le biblioteche sono dotate di sistemi che permettono la ricerca dei libri mediante una query dell'utente.

Le principali caratteristiche di un buon algoritmo di prossimità semantica sono la capacità di interpretare correttamente il fabbisogno informativo dell'utente (*precision*) e la velocità con cui restituisce il risultato. E' anche essenziale che il sistema per il reperimento delle informazioni riduca al minimo l'assenza e il deterioramento di informazioni rispetto al fabbisogno informativo dell'utente.

Vi sono diversi metodi utilizzati per rilevare la prossimità semantica tra una query ed un insieme di documenti. Le tecniche sono molto differenti tra di loro e variano per il modo in cui i documenti sono analizzati. Le tecniche più diffuse sono:

- Ricerca di stringhe
- Clustering
- Thesaurus

Queste diverse tecniche, che verranno brevemente analizzate di seguito, possono anche essere combinate al fine di ottimizzare la ricerca dei pesi dei termini. Ad esempio, la tecnica di *clustering* può essere implementata unitamente all'utilizzo di un thesaurus permettendo di definire una metrica basata anche sul significato semantico dei termini dei documenti.

3.5.1 Ricerca di stringhe

Questo metodo di analisi della prossimità semantica si basa sulla ricerca nel documento, delle stringhe presenti nel fabbisogno informativo dell'utente. La tecnica prevede che tale ricerca possa essere esatta oppure approssimata mediante opportune operazioni, ma che venga svolta sull'intero documento.

Come si può facilmente intuire, questo metodo non è adatto all'applicazione nell'*information retrieval* in quanto:

- il tempo per il confronto diretto tra stringhe ha un costo computazionale molto alto
- la presenza di una determinata stringa in un documento non è una condizione necessaria nè sufficiente per garantire che il documento preso in considerazione sia effettivamente rilevante per il fabbisogno informativo dell'utente.

Come si evince dalle osservazioni precedenti, la metodologia di ricerca diretta di corrispondenze tra stringhe non tiene conto degli aspetti essenziali di un buon sistema per il reperimento di informazioni; il tempo di ricerca è del tutto inadeguato per un sistema che deve garantire l'interattività con l'utente ed i risultati non garantiscono valori ottimali di precisione e richiamo.

3.5.2 Clustering

Il *clustering* è una tecnica di analisi multivariata dei dati (ovvero un'analisi eseguita mediante procedimenti statistici che considerano un insieme di diversi elementi) volta alla selezione e al raggruppamento di elementi omogenei in un insieme di dati. Gli algoritmi di *clustering* si basano sul concetto di distanza tra due elementi e la qualità delle analisi ottenute dipende essenzialmente da quanto è significativa la metrica utilizzata. Le tecniche di *clustering* si possono distinguere in due principali filosofie:

- dal basso verso l'alto: prevede che tutti gli elementi siano considerati *cluster* separati e che, successivamente, l'algoritmo provveda a raggruppare i *cluster* più vicini.
- dall'alto verso il basso: prevede che all'inizio tutti gli elementi siano considerati un unico *cluster* e che in seguito l'algoritmo inizi a dividere i *cluster* in tanti sotto-*cluster* di dimensioni inferiori. Il criterio che guida la scomposizione è sempre quello di ricercare elementi che formino gruppi omogenei.

Applicata ai documenti, la tecnica di *clustering* è molto efficace in quanto permette di rilevare le parole significative nei documenti ed eseguire raggruppamenti sulle stesse, inoltre un suo grande vantaggio risiede nell'estrema velocità di ricerca effettuabile sulla foresta venutasi a creare. Il *clustering*, oltre a tutto, permette di eseguire le ricerche anche su documenti eterogenei in quanto sarà compito dell'algoritmo la costruzione di una struttura dati appropriata a rappresentare i testi analizzati.

Il principale svantaggio di questa tecnica è la complessità degli algoritmi utilizzati. Un buon algoritmo di *clustering* che utilizzi una metrica significativa, infatti, risulta di difficile implementazione e la gestione delle strutture dati che si vengono a formare risulta molto complicata.

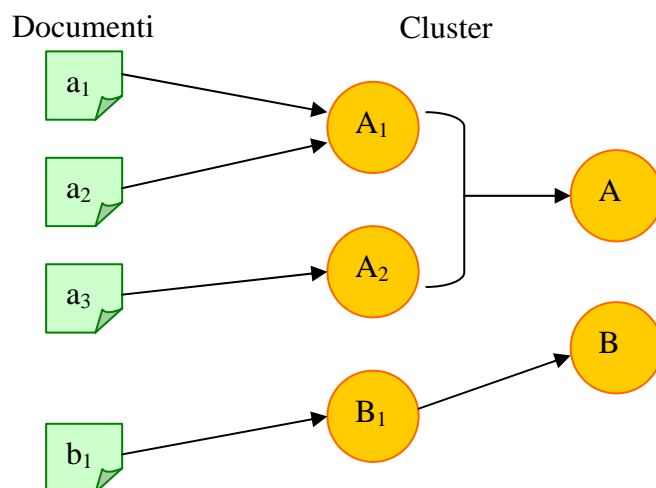


Fig 3.5.2 – Esempio di processo di clusterizzazione bottom-up (dal basso verso l'alto). I documenti vengono suddivisi in diversi cluster che, successivamente, vengono raggruppati in cluster meno rappresentativi. Si può notare la formazione di diversi alberi di cluster.

3.5.3 *Thesaurus*

Questo metodo di analisi per la ricerca di prossimità semantica si basa sulla ricerca di indici di significatività tra le parole dei testi e il fabbisogno informativo dell'utente. Questa tecnica, che viene analizzata in questa stage, prevede l'utilizzo di un dizionario dei sinonimi, creato analizzando la somiglianza lessicale o semantica tra i termini dei documenti, per determinare quanto due frasi possano essere vicine secondo una determinata metrica.

L'idea base è quella utilizzata nella ricerca tra stringhe, ma con la sostanziale differenza che la valutazione è ottenuta analizzando il documento nel suo contesto. La metrica tiene conto del valore di somiglianza lessicale o semantica precedentemente calcolato e dell'ordinamento delle parole nelle frasi analizzate.

Il vantaggio di questo tipo di approccio è la relativa facilità di implementazione degli algoritmi che governano le varie fasi di realizzazione. Le relazioni trovate sono calcolate mediante apposite formule che garantiscono un alto livello di precisione del risultato nella ricerca del fabbisogno informativo dell'utente. Per contro, i tempi di computazione di questa tecnica non sono ottimi, in quanto i documenti, se pur ridotti ad una forma normalizzata, sono considerati nella loro totalità.

Parte 1: Thesaurus

L'utilizzo dei thesaurus negli algoritmi di prossimità semantica è determinante. Questo strumento, infatti, permette di stabilire dei gradi di somiglianza tra i termini di un documento (o del fabbisogno informativo dell'utente) aumentando, quindi, la capacità di analisi dei testi.

Molti algoritmi di prossimità semantica, compreso quello analizzato in questa tesi, si basano totalmente sui valori contenuti nel thesaurus; la costruzione di un ottimo thesaurus, quindi, ricopre un ruolo essenziale, ma, d'altra parte, tale implementazione richiede un tempo di calcolo molto elevato e, spesso, risulta inapplicabile per i sistemi informatici comuni.

Sorge dunque il problema di ottimizzare il tempo di esecuzione dell'algoritmo, mantenendo allo stesso tempo un livello ottimale di ricerca della somiglianza dei termini.

Verrà trattata di seguito la teoria relativa alla realizzazione dei thesaurus e verrà analizzata dettagliatamente l'implementazione del thesaurus utilizzato in questa tesi.

1. Cenni teorici e analisi

Come già visto in precedenza, un thesaurus non è altro che un dizionario dei sinonimi. In particolare, in esso vengono memorizzati i valori di somiglianza tra coppie di parole. Esistono diverse tecniche per la costruzione automatizzata dei thesaurus che si distinguono in due gruppi principali:

- Tecniche per il calcolo della somiglianza lessicale: si basano sull'analisi della radice comune dei termini fornendo un indice che rappresenta quanto due parole derivino da una radice comune.
- Tecniche per il calcolo della somiglianza semantica: si basano sull'analisi delle relazioni concettuali dei termini fornendo un indice che rappresenta quanto due termini sono concettualmente legati.

1.1 Somiglianza lessicale

Esistono due tecniche per individuare le affinità semantiche tra coppie di termini di un documento: l'individuazione di una radice comune tra i termini (troncamento) e l'analisi tra stringhe.

Nell'utilizzo della prima metodologia si riscontrano ambiguità nella determinazione della radice comune e nell'impossibilità di analizzare correttamente le forme irregolari dei termini. Nel confronto tra stringhe, invece, la procedura è più semplice e consiste nel trovare una rappresentazione numerica al grado di somiglianza tra le parole

1.1.1 L'analisi tra stringhe

Esistono molti algoritmi per il calcolo della somiglianza lessicale basata sull'analisi delle stringhe. Questa tecnica esula dagli scopi della tesi e, quindi, verrà trattata in modo semplificato analizzando solamente una tecnica base.

Se si considerano due parole

P_1 e P_2

e si definiscono i seguenti parametri

- N_1 numero di lettere che compongono la parola P_1
 N_2 numero di lettere che compongono la parola P_2
 N_u numero di lettere comuni alla parola P_1 e alla parola P_2 , scandendo da sinistra a destra ed interrompendo la scansione alla prima lettera differente tra le due parole.

In questo caso la somiglianza lessicale tra le parole P_1 e P_2 sarà data dalla formula

$$M(P_1, P_2) = 2 \cdot \frac{N_u}{N_1 + N_2}$$

Il valore $M(P_1, P_2)$ indica la probabilità che le parole P_1 e P_2 provengano da una radice comune. Da analisi condotte risulta che per valori di $M(P_1, P_2) < 0,55$ i due termini sono difficilmente correlati tra loro, mentre per valori di $M(P_1, P_2) > 0,7$ le due parole sono quasi sicuramente correlate.

Utilizzando queste osservazioni si può stabilire la forza, V , nel seguente modo:

```
_____ <codice JDK1.5_06> _____  
if(M(P1, P2) > 0.7)  
    V = 1;  
else if(M(P1, P2) < 0.55)  
    V = 0;  
else  
    V = 5 * (M(P1, P2) - 0.525);  
_____ </codice> _____
```

In questo modo si avranno valori oscillanti tra 0 ed 1 esprimendo, così, la probabilità che i due termini P_1 e P_2 siano associati tra di loro.

1.2 Somiglianza semantica

La somiglianza semantica si basa sulla ricerca di termini nel documento che possono essere considerati vocaboli simili. Due parole sono sinonimi se il loro utilizzo in un determinato contesto è neutrale ai fini del significato.

Si possono analizzare diverse tipologie di correlazioni semantiche; i termini “gatto” e “micio”, ad esempio, sono sinonimi, ma anche i termini “gatto” e “felino” oppure “gatto” e “animale”, se pur in forma minore, sono dei sinonimi. In questi ultimi due casi si parla di iponimia della parola “gatto” in quanto i termini “felino” e “animale” sono concetti più generali rispetto al termine “gatto”. Oltre ai sinonimi vi sono anche altre correlazioni tra i termini, come esistono tra le parole “cane” e “gatto”. L’individuazione di questi legami di termini è una fase molto complessa che non può prescindere dall’intervento umano; la valutazione automatica di relazioni come quelle che esistono tra i termini “cane” e “gatto”, infatti, deve essere valutata manualmente in base al contesto di utilizzo.

Per valutare le dipendenze di somiglianza semantica che esistono tra i termini di un documento si può pensare di prendere in considerazione come unità fondamentale, una frase. E’ logico pensare che se due termini compaiono assieme in diverse frasi è plausibile che queste parole siano correlate tra loro. Su questa idea di base si sviluppano le principali tecniche di analisi automatica della somiglianza semantica di termini.

Definiamo i seguenti parametri che consentiranno di analizzare le formule utilizzate per il calcolo della somiglianza semantica tra termini.

N_f	Rappresenta il numero di frasi del campione
N_s	Rappresenta il numero di frasi significative presenti nel campione
P_1, P_2, \dots, P_n	Rappresenta le parole significative
$V_{pi}[N_f]$	Vettore che contiene nella posizione $V_{pi}[j]$ il numero di occorrenze della parola p_i nella frase j -esima

Le misure più diffuse per l'analisi della somiglianza lessicale dei termini contenuti in un documento sono tre. Si basano su un input di due vettori, V_{px} e V_{py} , che contengono le parole della frasi che devono essere analizzate.

1.2.1 Coefficiente del coseno

$$somiglianza_lessicale = \frac{\sum_{i=1}^{N_f} V_{px}[i] \cdot V_{py}[i]}{\sqrt{\sum_{i=1}^{N_f} V_{px}[i]^2 \cdot \sum_{i=1}^{N_f} V_{py}[i]^2}}$$

Il valore della somiglianza lessicale calcolato risulterà direttamente proporzionale al numero di volte che le parole P_x e P_y compaiono assieme nella frase ed inversamente proporzionale alla radice della somma dei quadrati delle singole occorrenze dei termini.

Il coefficiente del coseno è in grado di ottenere risultati ottimali di valori di somiglianza semantica rispetto alle altre formule presentate. Per questa tesi è stato utilizzato questo coefficiente in quanto garantisce un andamento costante, lungo l'intervallo del fattore di somiglianza semantica, di relazioni trovate.

1.2.2 Coefficiente di Dice

$$somiglianza_lessicale = \frac{2 \cdot \left(\sum_{i=1}^{N_f} V_{px}[i] \cdot V_{py}[i] \right)}{\sqrt{\sum_{i=1}^{N_f} V_{px}[i]^2 + \sum_{i=1}^{N_f} V_{py}[i]^2}}$$

Il valore della somiglianza lessicale calcolato risulterà direttamente proporzionale al doppio del numero di volte che le parole P_x e P_y compaiono assieme nella frase ed inversamente proporzionale alla radice della somma dei quadrati delle singole occorrenze dei termini.

1.2.3 Coefficiente di Jaccard

$$\text{somiglianza_lessicale} = \frac{\sum_{i=1}^{N_f} V_{p_x}[i] \cdot V_{p_y}[i]}{\sum_{i=1}^{N_f} V_{p_x}[i]^2 + \sum_{i=1}^{N_f} V_{p_y}[i]^2 - \sum_{i=1}^{N_f} V_{p_x}[i] \cdot V_{p_y}[i]}$$

Il valore della somiglianza lessicale calcolato risulterà direttamente proporzionale al numero di volte che le parole P_x e P_y compaiono assieme nella frase ed inversamente proporzionale alla differenza della sommatoria del quadrato delle occorrenze dei singoli termini e della sommatoria delle occorrenze delle singole parole.

2. Implementazione

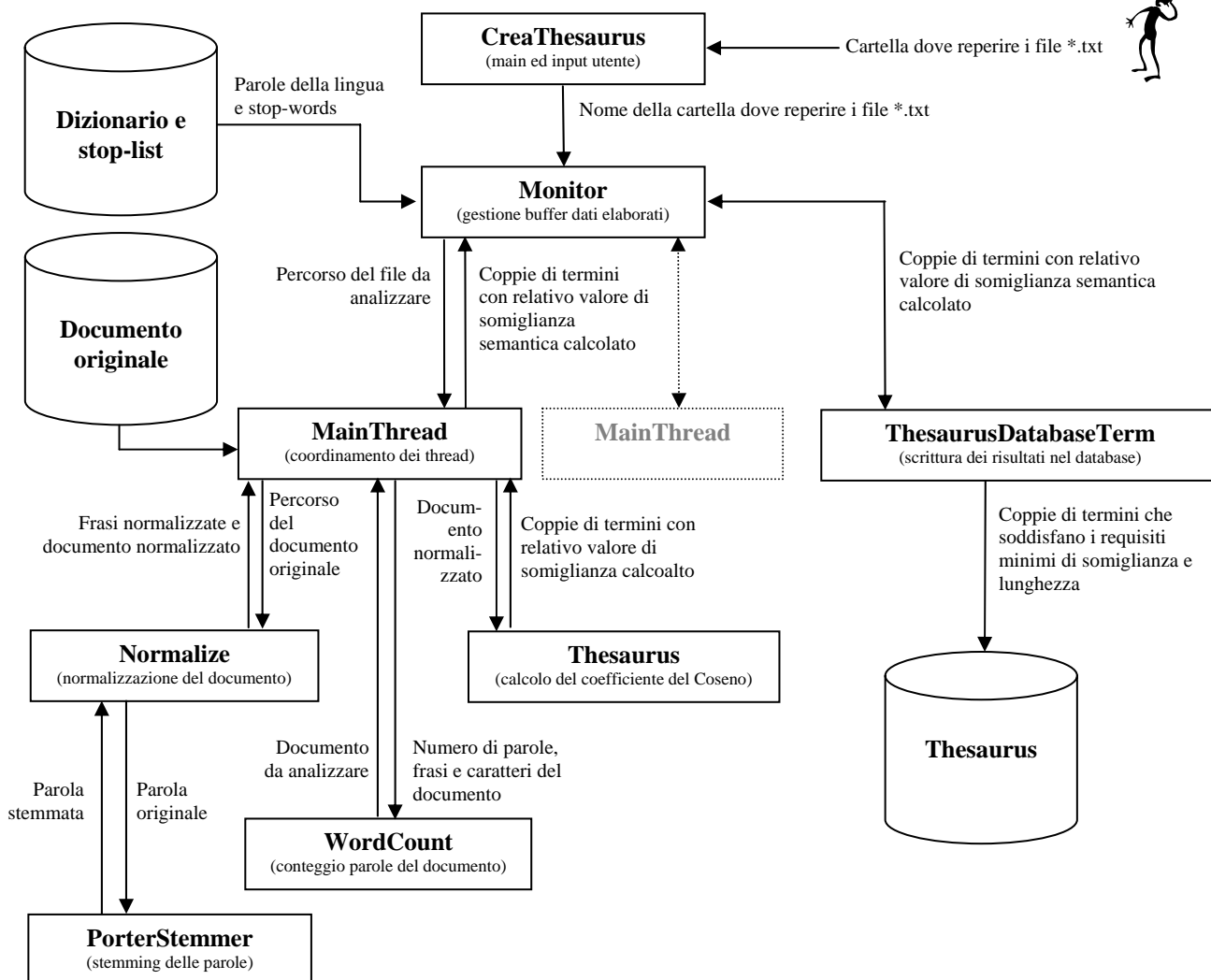
L'implementazione di un buon thesaurus è una fase critica, in quanto scelte sbagliate possono condizionare tempi di calcolo e qualità dei risultati trovati. Il thesaurus è realizzato utilizzando le tecniche precedentemente descritte ed in particolare:

- normalizzazione del testo in ingresso
- applicazione del coefficiente del Coseno per l'analisi della somiglianza semantica dei termini
- perfezionamento e memorizzazione dei dati in uscita

I diversi passi di elaborazione sopra descritti sono stati sviluppati in ambiente *multi-thread* principalmente per due motivi: in primo luogo, l'utilizzo di *thread* distinti velocizza l'esecuzione del codice sui più moderni calcolatori; in secondo luogo, in questo modo è possibile utilizzare sistemi multi processore per eseguire diverse istanze di analisi dei documenti simultaneamente, diminuendo i tempi di elaborazione.

2.1 Architettura generale

L'algoritmo sviluppato prevede un intenso scambio di informazioni tra le varie classi, mediante l'utilizzo di metodi sincronizzati. Lo schema base è il seguente:



Analizzando nel dettaglio lo schema riportato precedentemente, si può osservare che le principali strutture scambiate sono il documento originale, il documento modificato e le coppie di termini con associato il valore di somiglianza semantica. Il codice inizialmente viene lanciato senza alcun parametro, in modo da potere essere facilmente richiamato dall'utente. Quindi le informazioni riguardanti la cartella dove trovare i documenti da analizzare vengono inoltrate alla classe Monitor che si occuperà di richiamare tante istanze del codice MainThread quanti sono i testi da analizzare. Il documento viene quindi processato dalla classe Normalize che fornirà una lista di frasi normalizzate alla classe Thesaurus; quest'ultima provvederà a ricercare i valori di somiglianza semantica tra ogni coppia di termini che verranno a loro volta memorizzati nel database attraverso la classe ThesaurusDatabaseTerm.

Analizziamo nel dettaglio le proprietà ed i metodi delle singole classi:

```
CreaThesaurus.java  
public void main (String args[])
```

```
Monitor.java  
private java.util.List<String> vocabolario; //Struttura contenente il vocabolario  
private java.util.List<String> trasformazioniorigine; //Struttura contenente una modification list  
private java.util.List<String> trasformazionidestinazione; //Struttura contenente una modification list  
private String cartella; //Cartella dove sono contenuti i testi da analizzare  
private java.util.List<String> terminithesaurus; //Struttura per la memorizzazione dei dati calcolati  
private int processirimanenti; //Conteggio dei thread rimanenti  
private int numerodocumenti; //Specifica il numero di documenti da elaborare  
private final int maxmainthread = 6; //Numero di processi concorrenti massimo  
private int mainthreadattivi; //Numero di processi attivi  
public void run() //Istruzioni del thread  
public boolean searchWord(String parola) //ricerca una parola nel dizionario  
public String trasformazione(String parola) //ricerca una trasformazione dell'elenco di trasformazioni  
private void preloadData() //effettua il pre-caricamento dei dati nella memoria RAM  
private Connection openDatabaseConnection(String database) //apre la connessione alla base di dati  
private void closeDatabaseConnection(Connection connessione) //chiude la connessione alla base dei dati  
synchronized public void aggiungiTermine(String termine, String nomeprocesso) //aggiunge una coppia di termini da memorizzare  
synchronized public void mainThreadEnd(String nomeprocesso) //evento richiamato al termine di una elaborazione  
synchronized public void mainThreadStart(String nomeprocesso) //evento richiamato all'inizio di una elaborazione  
synchronized public String prelevaTermine(String nomeprocesso) //legge un termine dalla struttura dati  
synchronized public boolean terminiFiniti(boolean ... terminifiniti) //controlla se vi sono altri termini da elaborare
```

```
WordCount.java  
private String testo; //memorizza il documento da analizzare  
public void run() //Istruzioni del thread
```

```

MainThread.java
private int frasi; //numero di frasi
private int parole; //numero di parole
private int caratterisenzaspazi; //numero di caratteri senza spazio
private int caratteri; //numero di caratteri
private boolean contatorisettaggi; //specifica se i contatori sono stati impostati
private List<String> frasinormalizzate; //struttura per la memorizzazione delle frasi già normalizzate
private boolean frasiterminate; //specifica se le frasi normalizzate sono terminate
private final int maxbuffer = 100; //buffer di memorizzazione delle frasi normalizzate
private String filedaleggere; //memorizza il file da cui leggere il testo
private String testodocumento; //memorizza il testo del documento

public void run() //istruzioni del thread
private void leggiTesto() //legge il documento da analizzare e lo memorizza in una stringa
synchronized public void aggiungiFrase(String frase, String nomeprocesso) //Aggiunge alla struttura dati una frase normalizzata
synchronized public String prelevaFrase(String nomeprocesso) //Legge dalla struttura dati una frase da elaborare
synchronized public boolean frasiTerminate(boolean ... finito) //Controlla se le frasi sono terminate
synchronized public String prelevaTesto(String nomeprocesso, boolean preserva) //Legge il testo
synchronized public void aggiungiTesto(String testo, String nomeprocesso) //Scrive il testo
synchronized public void setContatori(int f,int p,int c,int css) //Imposta i contatori di frasi,parole e caratteri
synchronized public void setContatoriSettaggi(boolean valore) //imposta il frag dei contatori
synchronized public int getFrase(String nomeprocesso) //preleva il numero di frasi
synchronized public int getParole(String nomeprocesso) //preleva il numero di parole
synchronized public int getCaratteri(String nomeprocesso) //preleva il numero di caratteri
synchronized public int getCaratteriSenzaSpazi(String nomeprocesso) //preleva il numero di caratteri senza spazio
    
```

```

Normalize.java
private String testo; //memorizza il documento da analizzare
private String[] frase; //memorizza le frasi analizzate

public void run() //istruzioni del thread
private void findSentences() //suddivide il documento da analizzare in frasi
public void replaceWords() //si occupa dell'applicazione della procedura di normalizzazione
    
```

```

Thesaurus.java
private List<String> tabellahash; //contiene l'elenco delle parole singole presenti nel documento
private byte[][] vettoreoccorrenze; //memorizza le occorrenze di ogni termine per ciascuna frase
private String testo; //memorizza il documento da analizzare
private String[] frase; //memorizza le frasi analizzate
private final int minimaoccorrenza = 4; //Numero minimo di occorrenze di una parola per essere considerata

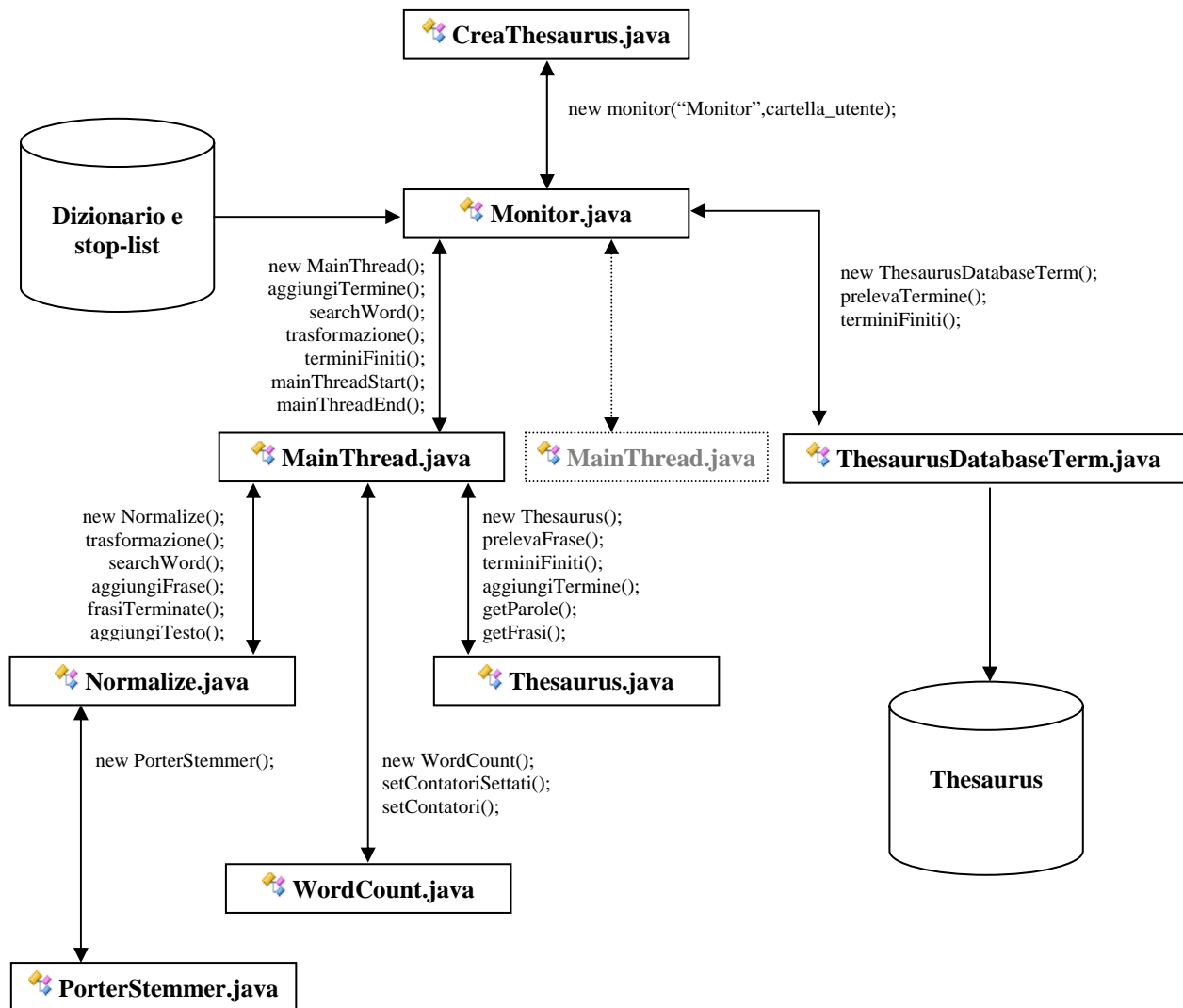
public void run() //istruzioni del thread
private void createHash() //Costruisce la lista di parole uniche del documento
private void cosineMethod() //applica il metodo del coseno per la ricerca di somiglianza semantica tra coppie di termini unici
    
```

```

ThesaurusDatabaseTerm.java
private final Double sogliaminima = 0.3; //Soglia minima sopra cui memorizzare i dati nel database
private final int minimalunghezza = 2; //Soglia minima di lunghezza parola, sotto la quale viene ignorata

public void run() //istruzioni del thread
private Connection openDatabaseConnection(String database) //apre la connessione alla base di dati
private void closeDatabaseConnection(Connection connessione) //chiude la connessione alla base dei dati
    
```

Si può ora esaminare come le classi riportate precedentemente interagiscano tra di loro e mediante quali metodi sia possibile questa comunicazione. Per semplificare lo schema, la struttura delle classi verrà assunta come quella precedentemente illustrata, ma verranno riportati solamente i nomi delle classi interessate. Accanto alle frecce di collegamento verranno indicati i metodi che permettono lo scambio di informazioni tra le due classi collegate.



L’algoritmo crea diverse istanze della classe MainThread in relazione al valore specificato nell’apposita variabile. Ad ogni istanza di MainThread viene creato un identico sottoalbero come quello rappresentato nello schema qui sopra.

La classe ThesaurusDatabaseTerm è l’ultima procedura a terminare. Una volta completata, fornisce il consenso al thread Monitor per chiudersi e quindi la procedura di creazione del Thesaurus finisce.

2.2 Avvio dell’algoritmo (CreaThesaurus.java)

L’algoritmo viene eseguito lanciando il file CreaThesaurus.class tramite esecuzione del comando “java”. A causa della grande quantità di memoria utilizzata dalle strutture dati in uso, se si prevede di utilizzare il programma con documenti di grandi dimensioni, sarà necessario specificare una dimensione personalizzata, iniziale e massima, dello heap java. Per l’esecuzione del codice su un insieme di 40 documenti (massimo 3 thread paralleli) con una media di 30 pagine ciascuno, è stato riscontrato un valore ottimale per la dimensione iniziale dello heap di

256MB e una dimensione massima di 512MB. Questa personalizzazione si può impostare mediante la forma estesa del comando “java”:

```
java -Xms256m -Xmx512m CreaThesaurus
```

La classe CreaThesaurus richiede all’utente di inserire il percorso di una cartella in cui sono contenuti i documenti da analizzare. All’input dell’utente, viene lanciato il thread “Monitor”, che accetta come parametro il percorso della cartella specificato dall’utente, e il main della classe CreaThesaurus termina.

2.3 Il monitor (Monitor.java)

Il thread Monitor si occupa di eseguire le procedure comuni ad ogni istanza di normalizzazione. Per prima cosa gestisce il pre-caricamento delle regole grammaticali e del dizionario della lingua utilizzata, in modo da velocizzare l’accesso a tali informazioni. Il grande numero di accessi a tali informazioni situate in una base di dati, rallenta notevolmente l’esecuzione del codice. Si è scelto, pertanto, di caricare i dati nella memoria centrale del calcolatore (circa 3MB di dati) per ottimizzare il tempo di calcolo dell’algoritmo.

Successivamente la classe Monitor si occupa di ricercare i file contenuti nella cartella specificata dall’utente e creare una lista dei solo file “TXT”. Utilizzando questa lista, vengono create tante istanze del thread “ThreadPrincipale” quanti sono i file presenti nella cartella dell’utente.

Mediante metodi sincronizzati, il processo Monitor esegue solamente un numero limitato, specificato dall’utente, di istanze di normalizzazione.

La scelta di creare differenti thread paralleli nasce dall’esigenza di sfruttare al massimo le potenzialità dei nuovi calcolatori che, spesso, sono in grado di eseguire più operazioni contemporaneamente.

Questo thread si occupa anche di fornire i metodi di accesso alla scrittura dei termini elaborati in un database. Tali termini, assieme al valore di somiglianza semantica calcolata dagli altri thread, vengono memorizzati in una struttura dati che verrà svuotata dal thread “ThesaurusDatabaseTerm”.

2.4 Il thread principale (ThreadPrincipale.java)

Questo thread fornisce i metodi per l’interscambio di informazioni nell’ambito di un solo documento.

Si occupa di leggere il contenuto del documento da normalizzare e di memorizzarlo in una variabile locale. Crea, quindi, i thread necessari alla procedura di normalizzazione del documento: “Normalize”, che esegue una normalizzazione dei dati in ingresso, e “Thesaurus”, che crea le strutture dati necessarie e calcola il coefficiente del Coseno per ogni coppia di termini.

Per poter comunicare tra loro, questi due thread hanno bisogno di una serie di metodi sincronizzati per lo scambio di frasi e termini già elaborati.

2.5 La fase di normalizzazione (Normalize.java)

La normalizzazione del documento avviene grazie al thread “Normalize”. Questo codice utilizza l’algoritmo di Porter per lo stemming di parole; da ricerche effettuate in merito è emerso che tale algoritmo garantisce il migliore rapporto tra velocità di esecuzione ed errori di Under/Overstemming commessi.

La procedura di questo algoritmo è semplice.

Per prima cosa il documento viene processato per eliminare tutti i caratteri non convenzionali: tutti i segni di punteggiatura interni alla frase, come la virgola, il punto e virgola, i due punti, etc., vengono sostituiti da un carattere vuoto. Vengono anche tolte le parentesi, le virgolette e altri segni che delimitano in qualunque modo una porzione di testo. I segni di punteggiatura forte, come il punto interrogativo o esclamativo, vengono sostituiti da punti semplici. Vengono ricercati anche spazi multipli e ridotti a spaziature singole.

Una volta terminata questa prima fase di normalizzazione, il testo viene diviso in frasi, spezzando la stringa principale dove si presentano i punti. Quindi vengono eliminati eventuali spaziature all’inizio o alla fine della frase.

Le frasi così ottenute vengono scandite da un ciclo ed ogni termine viene processato dall’algoritmo di Porter Stemmer. La parola stemmata viene confrontata con le parole di un dizionario, se la parola è presente, allora viene mantenuta, altrimenti viene ripristinato il termine originale.

Ogni frase processata, viene inserita in una struttura dati del “ThreadPrincipale” in modo che possa essere analizzata immediatamente dal thread “Thesaurus”.

Esaminiamo l’esecuzione del thread “Normalize” sul documento campione:

```
Since audio is by nature totally serial, arbitrary random
access to audio information may be of limited use. To
facilitate access to useful segments of audio information
within an audio recording deemed relevant by a user, we need
to identify entry points/jump locations. Further, multiple
contiguous segments may form a relevant and useful news
item.
```

Dopo il primo passo si avrà l’eliminazione dei segni di punteggiatura superflui, arrivando ad avere il seguente documento:

```
Since audio is by nature totally serial arbitrary random
access to audio information may be of limited use. To
facilitate access to useful segments of audio information
within an audio recording deemed relevant by a user we need
to identify entry points jump locations. Further multiple
contiguous segments may form a relevant and useful news
item.
```

e le seguenti frasi:

- 1) Since audio is by nature totally serial arbitrary random access to audio information may be of limited use.
- 2) To facilitate access to useful segments of audio information within an audio recording deemed relevant by a user we need to identify entry points jump locations.
- 3) Further multiple contiguous segments may form a relevant and useful news item.

I termini delle frasi vengono analizzati dall'algoritmo di stemming, ottenendo:

- 1) audio be nature total serial arbitrary random access
audio inform be limit us.
- 2) facilitate access us segment audio inform audio record
deem relevant user need identify entry point jump
locations.
- 3) further multiple contiguous segment form relevant us new
item.

Come si può notare, l'algoritmo ha permesso di ottenere un documento di 42 parole, contro i 55 termini del documento originale. Tutte le parole non selettive sono state eliminate e le rimanenti sono state riportate alla loro forma normalizzata.

Ognuna delle frasi verrà poi analizzata dal thread "Thesaurus" che si occuperà dell'applicazione della formula più opportuna.

2.6 La costruzione del thesaurus (Thesaurus.java)

Una volta che la fase di normalizzazione si è conclusa, il thread thesaurus procede con l'analisi dei termini presenti nelle frasi e nel calcolo del coefficiente del Coseno.

Per via delle grandi strutture dati richieste dal coefficiente del Coseno, l'occupazione di memoria di questo thread è elevata. E' infatti necessario creare una matrice con dimensioni *numero_parole* × *numero_fras*i che conterrà le occorrenze di ogni singola parola in ogni frase del documento.

Una volta che la matrice verrà riempita con i dati, il programma provvederà a scandire ogni termine del testo e all'applicazione su di esso della formula del coseno.

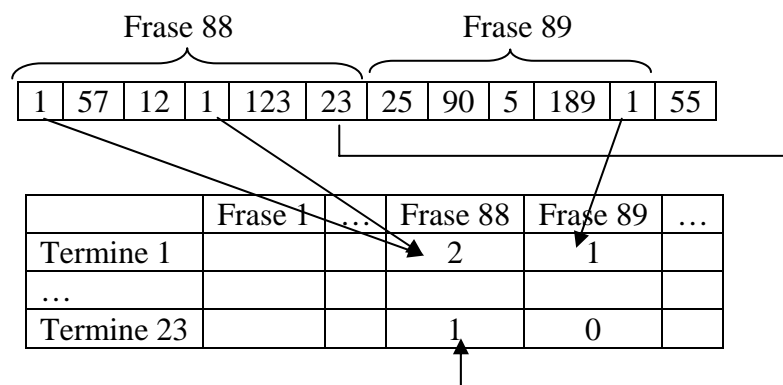
In particolare, per risparmiare spazio nel database e tempo di calcolo, si è scelto di calcolare solamente le dipendenze dirette e non quelle inverse. Questo non influenzerà il risultato finale in quanto, se si cambia l'ordine dei termini analizzati, il risultato del calcolo del coefficiente del Coseno rimane invariato; è logico, quindi, eseguire questo calcolo solamente una volta.

La formula utilizzata è quella presentata in precedenza:

$$\text{somiglianza_lessicale} = \frac{\sum_{i=1}^{N_f} V_{px}[i] \cdot V_{py}[i]}{\sqrt{\sum_{i=1}^{N_f} V_{px}[i]^2 \cdot \sum_{i=1}^{N_f} V_{py}[i]^2}}$$

L'algoritmo esegue un ciclo che scandisce ogni termine del documento e lo confronta con ogni altra parola dello stesso documento, applicando le operazioni indicate dalla formula del coefficiente del Coseno.

Se si pensa ad ogni termine del documento come identificato da un numero progressivo univoco, si può vedere la frase come una successione di numeri:



Le occorrenze presenti nella tabella vengono utilizzate da un altro ciclo, che questa volta scandisce le frasi significative del testo, in cui verrà calcolato numeratore e denominatore della formula precedentemente indicata.

Se si suppone di volere calcolare il coefficiente del coseno limitatamente alla frase 88 e 89, per le parole 1 e 23, si avranno i seguenti valori:

- il numeratore sarà dato dalla somma dei prodotti delle occorrenze dei termini in ogni frase, quindi, in questo caso: $2 \cdot 1 + 1 \cdot 0$ il che corrisponde a moltiplicare le occorrenze nella frase 88 dei termini 1 e 23 e sommare il risultato alla moltiplicazione delle occorrenze nella frase 89 dei termini 1 e 23.
- il denominatore sarà dato dal prodotto al quadrato delle somme delle occorrenze dei termini in ogni frase, quindi, in questo caso: $\sqrt{(2^2 + 1^2) * (1^2 + 0^2)}$

Avendo calcolato numeratore e denominatore, il valore del coefficiente del Coseno per i termini 1 e 23 sarà dato da:

$$somiglianza_semantica = \frac{2 \cdot 1 + 1 \cdot 0}{\sqrt{(2^2 + 1^2) * (1^2 + 0^2)}} = \frac{2}{\sqrt{5}} = 0,894$$

Nell'esempio riportato, dunque, le parole 1 e 23 avranno un valore di somiglianza semantica pari a 0,894. Questo significa che i due termini sono strettamente correlati tra di loro.

Ogni nuova coppia di termini per cui è stato calcolato il valore di somiglianza viene passata come parametro al metodo sincronizzato del processo padre, in modo che possa essere successivamente scritta su database.

2.7 Il conteggio di parole (WordCount.java)

Questo thread effettua il conteggio degli elementi presenti nel documento corrente. Mediante opportune operazioni sulla stringa che rappresenta il testo, è in grado di fornire informazioni riguardo al numero di frasi, parole, caratteri (con e senza spazi) che sono presenti nella stringa. Completata l'operazione di conteggio, il thread restituisce, mediante metodi sincronizzati, al padre le informazioni.

2.8 La scrittura dei risultati su database (ThesaurusDatabaseTerm.java)

Questo thread si occupa della scrittura nell'apposito database dei termini elaborati dagli altri processi. La scelta di utilizzare un apposito thread per questa funzione nasce dalla necessità di non aprire più di una connessione al database per volta. L'apertura di numerose sessioni di scrittura simultanee, infatti, degrada notevolmente le prestazioni del DBMS.

Questo thread si occupa inoltre di eliminare i dati che sono sotto una determinata soglia di importanza: dopo alcune analisi si è rilevato che termini con somiglianza semantica inferiore al 30% difficilmente sono realmente correlati tra di loro, il codice, quindi, memorizza nel database solamente quelle parole che hanno il coefficiente del Coseno al di sopra di tale soglia.

Se si opera su diversi documenti, può accadere che la stessa coppia di termini venga analizzata più di una volta ottenendo, magari, anche valori di somiglianza semantica differenti. In questo caso, però, non si saprebbe quale dei valori mantenere memorizzato nella base di dati, perciò il thread effettua una media tra i valori precedentemente rilevati ed il valore attuale, per mantenere traccia di ogni singolo risultato trovato.

La struttura del database in cui i dati vengono memorizzati ha la seguente forma:

Termine1	Termine2	Somiglianza	Occorrenze
----------	----------	-------------	------------

Nei campi "termine1" e "termine2" vengono memorizzati le due parole confrontate. Nel campo "somiglianza" viene memorizzata la media del valore del coefficiente del Coseno tra i due termini ed, infine, nel campo "Occorrenze" viene memorizzato il numero di volte che la coppia "termine1" e "termine2" è stata analizzata.

3. Risultati della costruzione

Il thesaurus è un elemento di importanza fondamentale per gli algoritmi di prossimità semantica. E' essenziale, quindi, valutare l'efficacia del codice nella ricerca della somiglianza semantica dei termini.

Da precedenti studi è già stata evidenziata la superiorità del coefficiente del Coseno nella rilevazione di relazioni nella metà superiore dell'intervallo.

Uno degli obiettivi di questo stage è quello di realizzare un thesaurus utilizzabile dagli algoritmi di prossimità semantica per ricerche in differenti ambiti. Il numero di termini presente nel database, quindi, deve essere sufficientemente consistente da permettere di ottenere buoni risultati per qualsiasi ricerca. In questi casi, la velocità di computazione è fondamentale per garantire la fattibilità della costruzione del thesaurus.

document	collect	0,4250184533	4
document	complete	0,4008918629	1
document	compos	0,3383577712	1
document	construct	0,3535533906	1
document	contain	0,3488761884	2
document	content	0,4564354646	1
document	database	0,5328004013	3
document	databases	0,3372170176	1
document	defined	0,3406046172	2
document	estimate	0,3154063474	1
document	expect	0,3144854510	1
document	find	0,4008918629	1
document	full	0,4806580133	1
document	generated	0,5050762723	1
document	hypothesis	0,4	1
document	improvements	0,3603749851	1
document	inverse	0,3863090652	1
document	judged	0,3721936842	1
document	length	0,4030997850	3
document	link	0,3585685828	1
document	list	0,4537479863	1
document	long	0,4834937784	1
document	medium	0,3741657387	1
document	method	0,4160251472	1
document	movies	0,4029114820	1
document	nonrelevant	0,3204617977	1
document	number	0,4441717816	4

Fig 3 – 1 Esempio di thesaurus. La tabella è un estratto del thesaurus che è stato realizzato per questo stage..

3.1 Il tempo di calcolo

L'algoritmo è stato eseguito su due calcolatori completamente differenti proprio per poterne valutare l'effettiva efficacia.

Computer 1:

CPU: Intel Mobile Centrino 1,4GHz – 2MB L2 cache – 400MHz FSB

CHIPSET: Intel 855

RAM: 768 MB

DISCO: IDE 40GB (ATA 133) – 8MB cache – 5400giri/min

Computer 2:

CPU: 2x Intel XEON 2GHz – 512KB L2 cache – 533MHz FSB – HT

CHIPSET: Intel 915

RAM: 1GB

DISCO: SCSI 9,2GB (UltraWide SCSI2) – 8MB cache – 7200giri/min

Nel dettaglio, il computer n°2 è in grado di supportare l'esecuzione di 4 thread simultanei, mentre il computer n°1 solamente 1.

Analizzando il tempo di calcolo impiegato (il tempo di calcolo viene conteggiato direttamente dal ThreadPrincipale) è stato rilevato che effettivamente il codice eseguito sul computer n°2 impiega circa 1/3 del tempo impiegato dal computer n°1.

Il raffronto è stato fatto su un campione di 40 documenti con una media di 30 pagine ciascuno. Ovviamente, il tempo di calcolo analizzato su un unico documento risulterebbe pressoché uguale per la presenza di un solo thread. La peculiarità dell'algoritmo risiede proprio nella capacità di potere eseguire diverse istanze di analisi contemporaneamente, sfruttando a pieno le potenzialità di ogni calcolatore.

3.2 I dati raccolti

Poiché il thesaurus viene costruito solamente in base alla frequenza dei termini è necessario che ogni documento preso in analisi abbia un contesto ben definito. In questo modo si evitano ambiguità tra i valori dei dati raccolti.

Testo n°1

Frase totali	1.305
Parole totali	14.738
Parole totali dopo normalizzazione	8.596
Parole singole	1.467
Confronti significativi (>0.3)	297
Confronti imporanti (>0.75)	18
Tempo impiegato (secondi)	166

Testo n°2

Frase totali	2.758
Parole totali	24.426
Parole totali dopo normalizzazione	15.922
Parole singole	2.371
Confronti significativi (>0.3)	359
Confronti importanti (>0.75)	20
Tempo impiegato (secondi)	520

Con le voci “frasi totali” e “parole totali” si indicano, rispettivamente, il numero di frasi ed il numero di parole presenti nel documento originale, prima che su di esso venga fatta alcuna elaborazione. Con la voce “parole totali dopo normalizzazione” si intende il numero di termini presente nel testo del documento una volta che su di esso è stato eseguito l’algoritmo di normalizzazione (realizzato da Normalize.java). Con “parole singole” si intende il numero di parole uniche presenti nel documento dopo che è stata effettuata la normalizzazione. Per “confronti significativi” si intendono quelle relazioni che hanno un coefficiente del Coseno superiore al valore di 0,3. Infine, la voce “confronti importanti” si riferisce ai termini che hanno una somiglianza semantica superiore al valore 0,75. Tali termini si possono considerare fortemente collegati tra di loro ed influenzano in modo significativo i risultati degli algoritmi di prossimità semantica.

Il tempo impiegato riportato nella tabella, si riferisce al tempo effettivo di calcolo di ogni coefficiente di somiglianza del testo ed esclude, quindi, il tempo di memorizzazione su database. Come si può notare, l’aumento di grandezza dei documenti non comporta necessariamente un equivalente aumento delle relazioni significative individuate. Per questo motivo, per realizzare un buon thesaurus non è importante la dimensione di ogni singolo documento, ma è rilevante il numero totale di documenti analizzati.

Per questa tesi sono stati analizzati 42 documenti con una media di circa 10.000 parole totali, ottenendo un thesaurus di 11.405 relazioni significative e circa 408 relazioni importanti. Il tempo impiegato per l’intero calcolo del thesaurus, effettuato mediante il computer n°2, è di approssimativamente 2 ore.

4. Conclusioni

L'obiettivo di questa fase dello stage si limitava a trovare un'implementazione di un algoritmo per la realizzazione di thesaurus, che si adattasse all'utilizzo con algoritmi di prossimità semantica. Uno dei principali problemi affrontati è stato quello di ridurre al minimo i tempi di computazione per rendere possibile il calcolo di un grande numero di relazioni significative tra i termini. Infatti è di cruciale importanza per un algoritmo di prossimità semantica, avere un thesaurus con un grande numero di voci su cui effettuare confronti. Da questo punto di vista il risultato ottenuto è più che soddisfacente: il tempo di calcolo è relativamente ridotto e l'algoritmo è in grado di funzionare con thread paralleli ottimizzando le caratteristiche dei computer moderni.

Un altro punto cruciale è stato l'individuazione della migliore combinazione di parametri che consentisse di ottenere dei valori di somiglianza veramente significativi e, soprattutto, utilizzabili da un algoritmo di prossimità semantica. Anche in questo caso, le relazioni che l'algoritmo ha trovato nei documenti si possono ritenere soddisfacenti e il loro numero, se pur non particolarmente esteso, è sufficiente per l'implementazione e la valutazione di un algoritmo di prossimità semantica; la configurazione di parametri scelta (ad esempio l'eliminazione delle relazioni con valore inferiore a 0,3) rende molto significativi i confronti trovati.

Parte 2: Algoritmo di Paice-Ramirez

Già in precedenza è stata rilevata la fondamentale importanza del processo di ricerca e reperimento di informazioni. La necessità di utilizzare i sistemi per il reperimento di informazioni con il linguaggio naturale ha spinto alla ricerca di algoritmi che permettessero di soddisfare questa esigenza.

L'algoritmo analizzato in questa tesi si basa sull'utilizzo di un thesaurus, un dizionario dei sinonimi, per riuscire ad interpretare correttamente il fabbisogno informativo dell'utente. L'intero processo di reperimento dipende dal confronto tra frasi e dalla corretta individuazione di un legame tra esse. E' evidente, quindi, l'importanza che riveste il meccanismo preposto a calcolare il valore di prossimità esistente tra due frasi.

Dalla qualità dell'algoritmo per il calcolo della prossimità semantica dipendono in gran parte la consistenza dei risultati restituiti dal processo di ricerca.

1. Cenni teorici ed analisi

Dopo attente ricerche è stato rilevato che il migliore algoritmo di prossimità semantica basato sull'utilizzo di thesaurus è quello sviluppato da Paice e Ramirez. Il funzionamento si basa sulla quantificazione della vicinanza tra due frasi in relazione al grado di somiglianza esistente tra le parole della prima e della seconda frase.

1.1 Il funzionamento

Come detto precedentemente il funzionamento di basa sull'analisi del grado di somiglianza che sussiste tra coppie di parole delle frasi prese in esame. Per questo motivo è essenziale avere a disposizione un thesaurus contenente coppie di parole che presentano tra loro un'affinità lessicale, come spiegato nella parte precedente.

Forniamo alcune definizioni che si utilizzeranno per definire opportunamente il funzionamento dell'algoritmo:

- **Parola:** si intende come parola qualsiasi sequenza finita e non nulla di caratteri alfabetici avente lunghezza uguale o maggiore ad un valore minimo predefinito
- **Termine:** qualsiasi parola che non compare all'interno di una lista predefinita di parole non significative (Vedi *stop-list*).
- **Frase:** si definisce come frase una sequenza finita e non nulla di termini.

Possiamo considerare:

$$S=(S_1, S_2, \dots, S_m)$$

e

$$T=(T_1, T_2, \dots, T_n)$$

due frasi composte, rispettivamente, da m ed n termini. Si suppone, inoltre, che il numero di parole che compone la frase S sia inferiore a quello che compone la frase T ($m \leq n$).

La funzione che calcola la prossimità semantica tra le due frasi S e T ha la seguente forma:

$$R(S, T) = f(S_x, T_y)$$

1.1.1 La funzione di mapping

E' necessario definire un *mapping*, ovvero un'associazione tra i termini della prima frase ed i termini della seconda frase, in modo tale che i due termini associati presentino una somiglianza non nulla. Si può assumere che: ogni termine S_x di S può essere associato ad un termine distinto T_y di T , oppure non essere mappato. Allo stesso modo, uno o più termini di T possono non essere associati ad alcun termine di S . Quindi, un termine di S non può essere associato a più di un termine di T e viceversa.

Possiamo definire $J(x)$ come una funzione che associa il termine $T_{J(x)}$ di T con il termine S_x di S . Per ogni S_x di S che non presenta *mapping* con termini di T , vale che $J(x) = 0$.

Per ogni coppia di termini associati S_x e $T_{J(x)}$ si intende determinare un valore, compreso nel *range* $[0,0 - 1,0]$, che ne definisca l'affinità. Una prima definizione del valore di prossimità semantica tra S e T può essere:

$$R(S,T) = \frac{1}{F} \sum_{x=1}^m W_x \cdot V \cdot (S_x \cdot T_{J(x)})$$

dove:

- W_x è il peso associato alla parola S_x
 - F è un fattore di normalizzazione
 - $V \cdot (S_x \cdot T_{J(x)})$ denota la prossimità semantica o lessicale tra i due termini S_x e $T_{J(x)}$.
- V viene calcolato sulla base dei valori contenuti nel thesaurus.

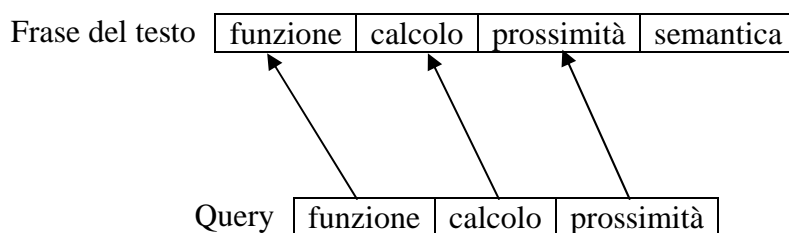
1.1.2 La funzione di tidyness

Un fattore di cui è importante tenere conto è l'ordine dei termini associati nelle due frasi esaminate. Infatti, se i termini di S e T sono associati a $J(x)$ in modo tale che non ci siano discrepanze nell'ordine dei termini, ovvero se:

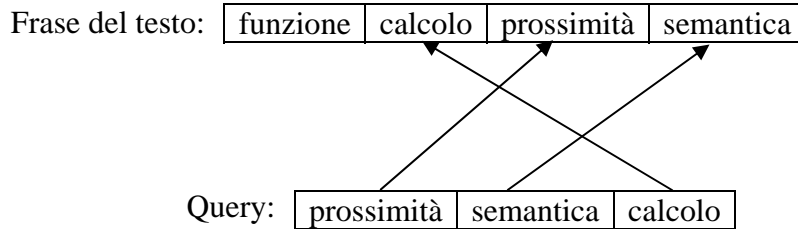
$$J(x+1) = J(x) + 1 \quad \text{per ogni } x = 1, 2, \dots, m-1$$

e quindi ogni valore di $J(x)$ vale uno in più del suo predecessore, allora ci troviamo in un caso ottimale in cui le parole associate compaiono nelle due frasi nel medesimo ordine. Se, d'altra parte, esiste una discrepanza tra l'ordine dei termini associati da $J(x)$, sembra opportuno applicare una penalizzazione. E' vero, però, che anche il caso peggiore di discrepanza nell'ordine dei termini non dovrebbe portare a zero un valore non nullo di prossimità semantica.

Analizziamo un esempio per comprendere meglio il significato di questa necessità:

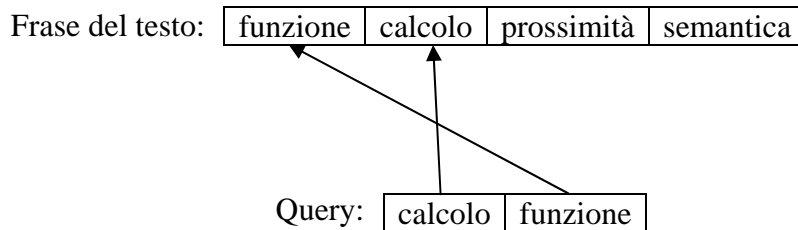


In questo caso l'ordinamento tra le parole della query e della frase del testo (che è rappresentata già normalizzata) è esatto, ovvero ogni parola del fabbisogno informativo dell'utente si trova nel medesimo ordinamento rispetto alla frase del documento considerata. Se, diversamente, si considera la query "prossimità semantica calcolo", l'ordinamento sarà il seguente:



In questo caso sarà logico pensare di applicare una penalizzazione al valore di prossimità tra le due frasi in quanto i termini non della query non si trovano nel medesimo ordine della frase del documento.

Se, infine, si considera l'esempio della query "calcolo funzione", la corrispondenza sarà:



In questo particolare caso, l'ordinamento tra i termini della query e del documento è opposto. Anche in questo caso sarà necessario applicare una penalizzazione al valore di prossimità semantica calcolato, ma nonostante non vi sia ordinamento tra le parole il valore di prossimità semantica restituito non deve essere nullo.

Occorre, quindi, introdurre una nuova funzione

$$\Psi(J) = \psi_{\min} + \Phi(J) \cdot (1 - \psi_{\min})$$

che rappresenta la *tidyness* (accuratezza) dell'ordinamento tra termini S e T fornito da $J(x)$. $\Psi(J)$ può assumere valori in $[\psi_{\min} \dots 1, 0]$, dove ψ_{\min} è una costante che specifica il valore minimo che la funzione $\Psi(J)$ può assumere.

La funzione $\Phi(J)$ rappresenta la frequenza delle associazioni ordinate nel mappaggio tra S e T , ovvero:

$$\Phi(J) = \frac{1}{m-1} \sum_{x=1}^{m-1} K_x$$

dove

$$K_x = \begin{cases} 1 & \text{se } J(x+1) = J(x) + 1 \\ 0 & \text{se } J(x+1) \neq J(x) + 1 \end{cases}$$

La funzione $\Psi(J)$ vale 1 se $\Phi(J) = 1$, ovvero se tutte le parole di S sono mappate nello stesso ordine in T . Viceversa $\Psi(J)$ vale ψ_{\min} nel caso in cui tutte le associazioni tra S e T siano non ordinate tra loro.

La funzione per il calcolo della prossimità semantica, considerando il vincolo di *tidyness*, diventa:

$$R(S, T) = \frac{1}{F} \sum_{x=1}^m W_x \cdot V \cdot (S_x \cdot T_{J(x)}) \cdot \Psi(J)$$

dove

- $V \cdot (S_x \cdot T_{J(x)})$ denota la prossimità semantica o lessicale tra i due termini S_x e $T_{J(x)}$.
 V viene calcolato sulla base dei valori contenuti nel thesaurus.
- $J(x)$ denota la funzione di *mapping*
- $\Psi(J)$ denota la funzione di *tidyness*

L'introduzione di $\Psi(J)$ si basa sul presupposto che l'informazione portata da coppie di parole contigue in un testo è molto meno significativa qualora le due parole siano separate tra loro nel testo e vuole valutare proprio questa perdita di informazione. In realtà questo non è vero sempre ed in generale il vincolo di *tidyness* può risultare, in determinati contesti linguistici, del tutto ininfluenza.

2. Implementazione dell'algoritmo

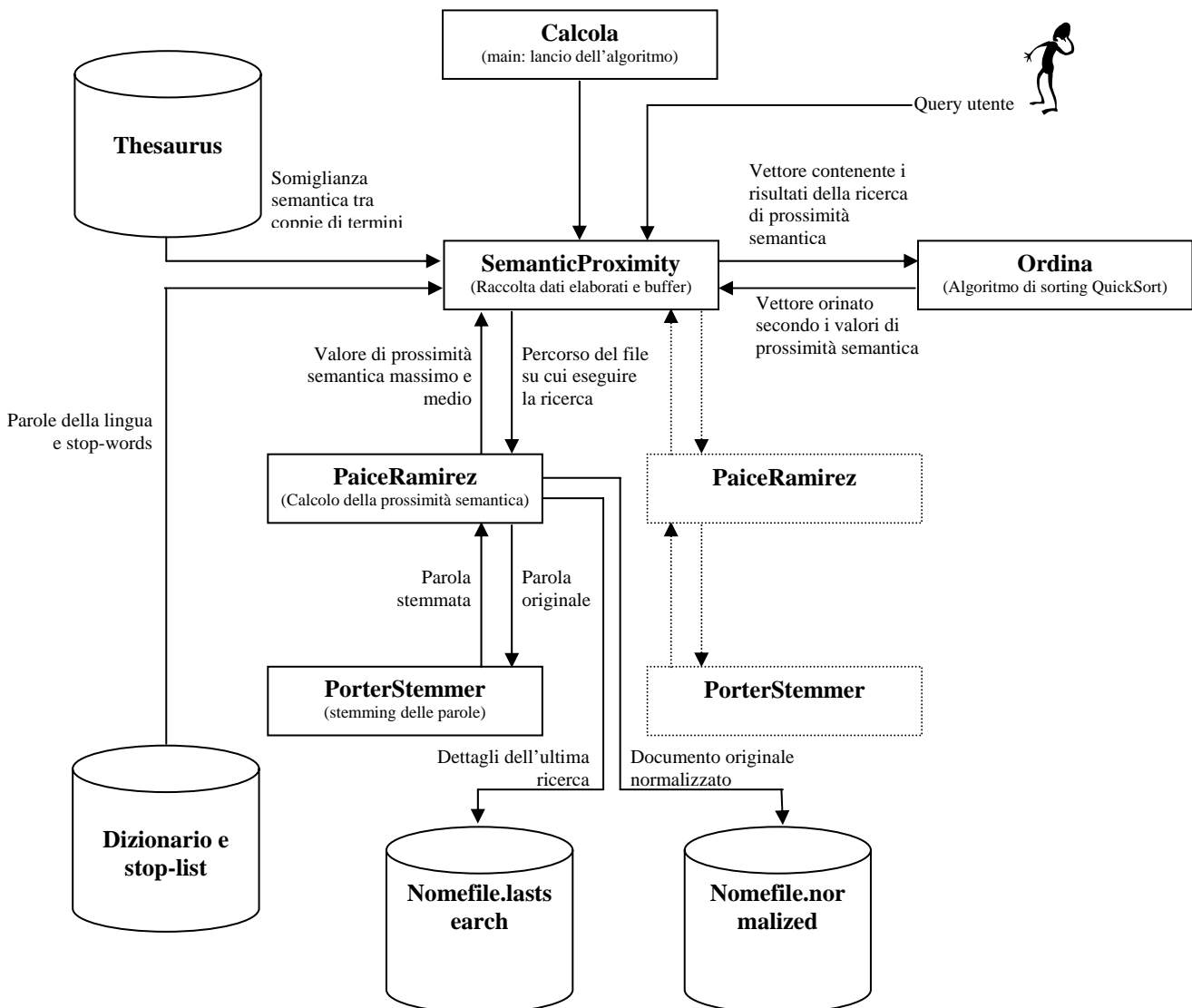
La qualità dell'algoritmo per la prossimità semantica utilizzato determina la precisione dei risultati forniti e l'efficienza del processo di ricerca.

L'algoritmo si occupa dell'applicazione delle formule precedentemente analizzate e dell'interpretazione dei valori ottenuti per classificare i documenti in relazione al fabbisogno informativo dell'utente.

I diversi passi di elaborazione sopra descritti sono stati sviluppati in ambiente *multi-thread* principalmente per due motivi: in primo luogo, l'utilizzo di *thread* distinti velocizza l'esecuzione del codice sui più moderni calcolatori; in secondo luogo, in questo modo è possibile utilizzare sistemi multi processore per eseguire diverse istanze di analisi dei documenti simultaneamente, diminuendo i tempi di elaborazione.

2.1 Architettura generale

Lo schema generale di funzionamento dell'algoritmo per il calcolo della prossimità semantica è il seguente:




Analizzando lo schema riportato precedentemente si può osservare che il codice produce un grande numero di dati in output. Vengono infatti creati due file contenenti i risultati, frase per frase, i valori dei diversi parametri calcolati dall'algoritmo nell'ultima ricerca effettuata dall'utente su ogni documento, e il documento normalizzato. Il codice fa uso del thesaurus creato in precedenza per ricercare i valori di somiglianza semantica tra le coppie di parole e del database grammaticale per normalizzare i documenti su cui le ricerche vengono eseguite.

Il codice viene lanciato senza alcun parametro, in modo da potere essere facilmente richiamato dall'utente. La classe SemanticProximity, successivamente, scansiona una directory predefinita alla ricerca dei documenti su cui effettuare la ricerca dell'utente. In relazione ai testi trovati, vengono create diverse istanze della classe PaiceRamirez che accettano come parametro il percorso del documento che si vuole analizzare. Dopo avere eseguito le operazioni necessarie per determinare il valore di prossimità semantica del documento, la classe restituisce una coppia di valori numerici che rappresentano il massimo valore di prossimità trovato e la media dell'intero documento.


Quando tutti i *thread* di ricerca sono terminati, la classe SemanticProximity chiama il metodo per ordinare il vettore dei risultati così ottenuto. Infine, l'elenco dei valori di prossimità semantica viene stampato sullo schermo.

Analizziamo nel dettaglio le proprietà ed i metodi delle singole classi:

```


 Calcola.java
public void main (String args[])
    
```

```

 SemanticProximity.java
private java.util.List<String> vocabolario; //Struttura contenente il vocabolario
private java.util.List<String> trasformazioniorigine; //Struttura contenente una modification list
private java.util.List<String> trasformazionidestinazione; //Struttura contenente una modification list
private java.util.List<String> parole; //Struttura contenente le parole del thesaurus
private java.util.List<Double> somiglianza; //Struttura contenente i valori di somiglianza del thesaurus
private String[] risultatifile; //Elenco dei file in cui è stata fatta la ricerca
private double[] risultatiosomiglianza; //Elenco dei risultati di prossimità semantica calcolati
private final int maxthread = 1; //Numero massimo di thread concorrenti per la ricerca
private int threadattivi; //Numero di thread attualmente attivi
private int processirimanenti; //Numero di thread ancora da eseguire
private String cartella = ".\\Testi"; //Cartella dove sono contenuti i testi da analizzare

public void run() //Istruzioni del thread
public boolean searchWord(String parola) //ricerca una parola nel dizionario
public String trasformazione(String parola) //ricerca una trasformazione dell'elenco di trasformazioni
private void preloadData() //effettua il pre-caricamento dei dati nella memoria RAM
private Connection openDatabaseConnection(String database) //apre la connessione alla base di dati
private void closeDatabaseConnection(Connection connessione) //chiude la connessione alla base dei dati
synchronized public void paiceRamirezEnd(String nomeprocesso) //gestisce l'avvio di un processo di ricerca
synchronized public void paiceRamirezStart(String nomeprocesso) //gestisce il termine di un processo di ricerca
private void stampaVettore() //stampa il vettore contenente i valori di prossimità semantica per ogni documento
synchronized public void memorizzaRisultato(String file,double somiglianza,int numeroprocesso) //memorizza il valore di una ricerca
public Double confrontaParole(String word1, String word2) //interroga il thesaurus per recuperare la somiglianza semantica tra termini
    
```

```

 Ordina.java
private double[] somiglianza; //Elenco dei risultati di prossimità semantica calcolati
private String[] file; //Elenco dei file in cui è stata fatta la ricerca

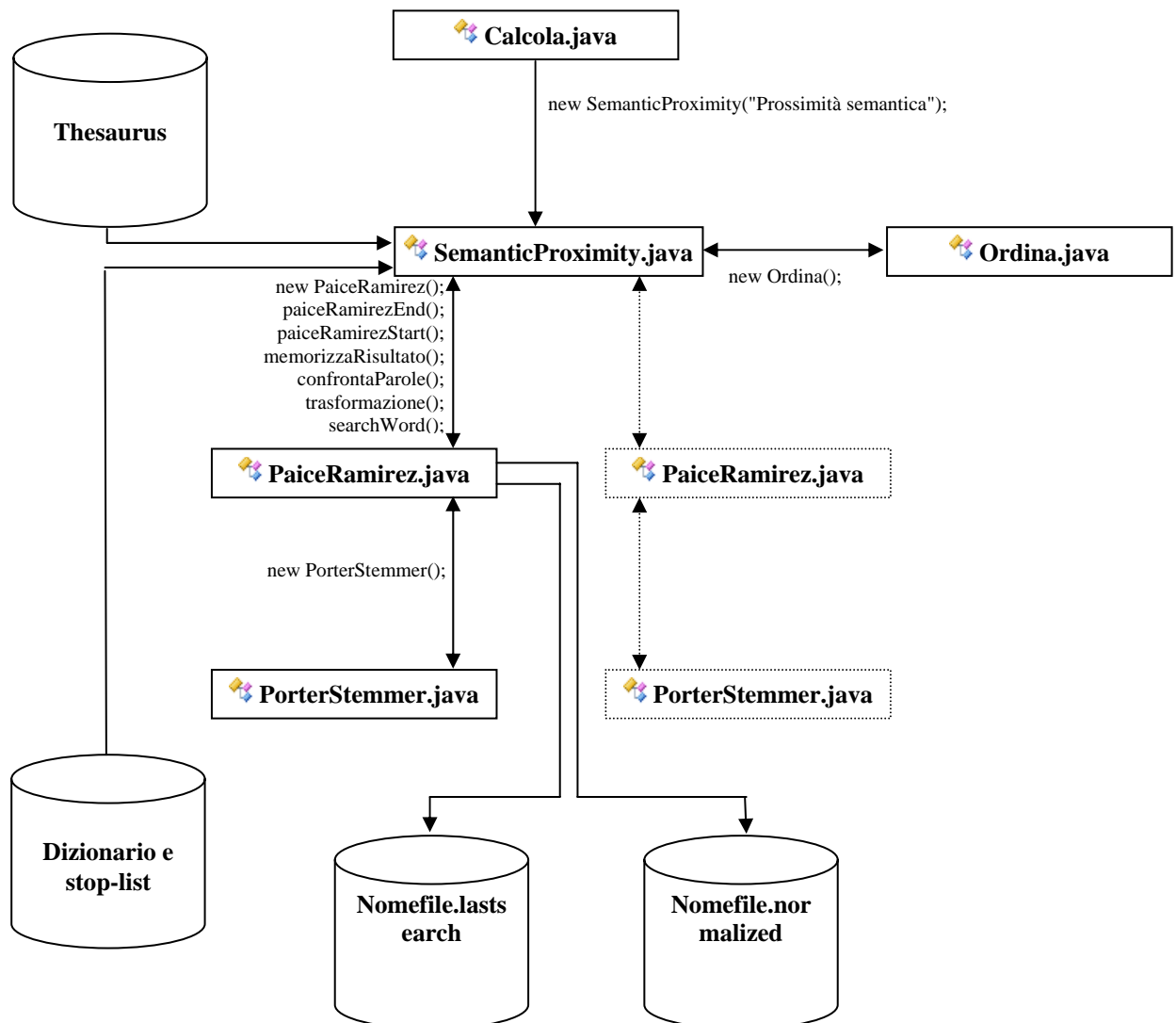
public double[] vettoreSomiglianzaOrdinato() //recupera il vettore delle somiglianze ordinato
public String[] vettoreFileOrdinato() //recupera il vettore dell'elenco dei file ordinato secondo il vettore delle somiglianze
public void sort(int low, int high) //ordinamento mediante QuickSort
private int partition(int low, int high) //partizione del vettore
private void swap(int i, int j) //scambio dei valori
    
```

```

PaiceRamirez.java
private String file; //nome del file che viene analizzato
private String percorso; //percorso del file che viene analizzato
private String testodocumento; //testo del documento (originale e normalizzato)
private String frase; //query dell'utente
private double coefficientepeso; //coefficiente per il calcolo dei pesi delle frasi
private double prossimita; //valore massimo di prossimità semantica trovato
private String[] frasi; //elenco delle frasi del documento
private long denominatoremedia; //denominatore per il calcolo del valore medio di prossimità semantica
private double numeratoremedia; //numeratore per il calcolo del valore medio di prossimità semantica

public void run() //istruzioni del thread
private void semanticProximity() //ricerca il valore di prossimità semantica tra ogni frase del documento e la query utente
private double calcolaPeso(double valore) //determina il peso della frase corrente, in base al suo valore di prossimità semantica
private void stampaUltimaRicerca() //stampa su file i dettagli dell'ultima ricerca effettuata
private void stampaUltimaRicerca(...) //overload del metodo precedente con tutti i parametri
private String normalize(String testo) //gestisce la suddivisione in frasi del documento
private String replaceWords() //gestisce lo stemming del documento
private String leggiTesto(String file) //legge da file il testo da analizzare
private boolean scriviTesto(String file,String testo,boolean append) //scrive su file i risultati del calcolo e i dettgli della ricerca
    
```

Si può ora esaminare come le classi riportate precedentemente interagiscano tra di loro e mediante quali metodi sia possibile questa comunicazione. Per semplificare lo schema, la struttura delle classi verrà assunta come quella precedentemente illustrata, ma verranno riportati solamente i nomi delle classi interessate. Accanto alle frecce di collegamento verranno indicati i metodi che permettono lo scambio di informazioni tra le due classi collegate



L'algoritmo crea diverse istanze della classe PaiceRamirez in relazione al valore specificato nell'apposita variabile. Ad ogni istanza viene creato un identico sottoalbero come quello rappresentato nello schema qui sopra.

2.2 Avvio dell'algoritmo (Calcola.java)

L'algoritmo per il calcolo della prossimità semantica viene lanciato eseguendo il file Calcola.class mediante il comando "java". Questa classe si occupa semplicemente del richiamo del *thread* principale che si occuperà di gestire l'esecuzione della ricerca nei documenti.

Non vi sono particolari requisiti di memoria in quanto le strutture dati utilizzate sono relativamente piccole.

Come verrà illustrato successivamente, l'algoritmo tenta di ottimizzare i tempi di ricerca memorizzando in appositi file le normalizzazioni dei singoli documenti: questo consente di risparmiare il tempo di normalizzazione per le successive ricerche sugli stessi documenti.

2.3 La prossimità semantica (SemanticProximity.java)

Questo *thread* si occupa del coordinamento dei processi di ricerca. Svolge l'importante funzione di pre-caricamento dei dati contenuti nei database. Il grande numero di accessi a tali informazioni, situate in una base di dati, rallenta notevolmente l'esecuzione del codice. Si è scelto, pertanto, di caricare i dati nella memoria centrale del calcolatore (circa 3MB di dati) per ottimizzare il tempo di calcolo dell'algoritmo.

Un altro ruolo rilevante di questo *thread* è quello dell'analisi dei dati elaborati dai singoli processi di ricerca; la memorizzazione degli stessi in una idonea struttura dati e il successivo ordinamento in base all'importanza calcolata dall'algoritmo di Paice-Ramirez, infatti, è a carico di questo *thread*.

Il numero di ricerche simultanee è definito dall'utente mediante un apposito parametro.

L'ordinamento dei risultati forniti dalle varie istanze di ricerca è effettuato mediante l'algoritmo QuickSort sul valore di prossimità semantica calcolati. L'utilizzo di QuickSort è particolarmente indicato nei casi in cui si abbia una grande quantità di voci da ordinare e gli elementi siano dei numeri a virgola mobile. L'ordinamento con questo algoritmo, infatti, garantisce un mantenimento delle prestazioni del codice senza introdurre ulteriori tempi di attesa.

2.4 L'algoritmo di Paice-Ramirez (PaiceRamirez.java)

Questo *thread* rappresenta il cuore dell'algoritmo di prossimità semantica. Le sue funzioni sono quelle di normalizzare la query dell'utente e i documenti su cui vengono fatte le ricerche. Per ottimizzare il tempo di calcolo, la normalizzazione effettuata sui documenti di ricerca viene memorizzata in un apposito file. Questo file viene chiamato con lo stesso nome del file originale, ma con estensione ".normalized". Quando una nuova ricerca viene avviata, per prima cosa viene controllato se esiste il suddetto file: in caso negativo, il documento viene normalizzato e il testo così ottenuto viene memorizzato, altrimenti viene saltato il processo di normalizzazione e viene caricato in memoria direttamente il documento già processato da normalizzazione.

Una volta normalizzata la query dell'utente, espressa in linguaggio naturale, e il documento di ricerca, viene calcolato il valore di prossimità semantica tra il fabbisogno informativo dell'utente ed ogni frase del documento in esame.

L'implementazione di una buona strategia per valutare il *mapping* tra le due frasi S e T (dove S rappresenta il fabbisogno informativo dell'utente e T rappresenta la frase generica del documento preso in esame) influenza notevolmente le prestazioni dell'algoritmo sia dal punto di vista dei risultati, sia per quello che riguarda il tempo di elaborazione.

Se si suppone di analizzare le frasi

$$S=(S_1, S_2, \dots, S_m)$$

e

$$T=(T_1, T_2, \dots, T_n)$$

I valori di thesaurus corrispondenti possono essere convenientemente rappresentati da una matrice $M[m \times n]$:

$$M = \begin{pmatrix} v_{11} & v_{12} & \dots & v_{1n} \\ v_{21} & v_{22} & \dots & v_{2n} \\ \dots & \dots & \dots & \dots \\ v_{m1} & v_{m2} & \dots & v_{mn} \end{pmatrix}$$

dove v_{ij} è il valore di thesaurus corrispondente alla coppia di termini (S_i, T_j) .

Il mappaggio più semplice a cui si può pensare viene calcolato con un algoritmo molto elementare che esamina una ad una le righe della matrice individuando il valore più elevato ed eliminando la colonna corrispondente:

```

_____ <codice JDK1.5_06> _____
For(int i=1;i<m;i++)
{
  Leggi la i-esima riga della matrice M;
  M[i][j] = Math.max(M[i][:]);
  somma += M[i][j];
  elimina la colonna j in modo che non venga più scelta;
}
_____ </codice> _____

```

I pregi di questa implementazione sono la semplicità e la velocità di esecuzione. Si tratta però di un algoritmo cosiddetto miope, ovvero la scelta dei valori non dipende in alcun modo dalla forma della matrice M . Inoltre, al crescere di i la scelta del valore sulla riga diventa forzata e molto spesso porta ad accoppiamenti a valore nullo.

Un metodo che implementi l'ottimo globale, ovvero che esamini tutte le possibili combinazioni, avrebbe un costo computazionale notevole, ovvero dell'ordine di: $\frac{n!}{(n-m)!}$.

L'utilizzo di un simile algoritmo renderebbe impossibile completare le ricerche in tempi ragionevolmente brevi.

Per questo motivo, è stato realizzato un algoritmo che tenga conto in qualche modo della distribuzione dei valori sulla matrice M senza per questo dovere calcolare tutte le possibili combinazioni di valori possibili. In questo modo è stato possibile ottenere un mapping tra le frasi più significativo senza che le prestazioni in termini di tempo di elaborazione ne risentissero in maniera sensibile:

```

<codice descrittivo>
For(int i=1;i<m;i++)
{
  Crea una matrice W(mxn) di pesi tale che wij contenga la somma di tutti gli elementi non nulli
  della riga i e della colonna j, escluso vij stesso.
  Somma ad ogni wij il valore 1- vij;
  Seleziona l'elemento (x,y) avente peso wij minimo ed incrementa la somma totale di un valore
  pari a vij;
  elimina la riga x e la colonna y della matrice M;
}
</codice>

```

I valori contenuti nella matrice W , w_{ij} , sono pesi che penalizzano una eventuale scelta di m_{ij} come prossimo elemento da aggiungere alla somma; infatti il valore di w_{ij} è direttamente proporzionale alla somma dei valori che si trovano sulla riga i o sulla colonna j , mentre un alto valore di m_{ij} stesso abbassa il valore del peso:

$$w_{ij} = \sum_{k=1, k \neq i}^n m_{ik} + \sum_{k=1, k \neq j}^m m_{ik} + (1 - m_{ij})$$

In altre parole se un valore è alto sarebbe auspicabile sceglierlo, ma se la sua scelta comportasse il dover scartare molti altri valori, allora la sua selezione risulterebbe meno conveniente.

Vediamo un esempio:

$$M = \begin{pmatrix} 0,0 & 1,0 & 0,8 & 0,0 \\ 0,3 & 0,9 & 0,0 & 0,4 \\ 0,3 & 0,2 & 0,0 & 0,5 \end{pmatrix}$$

e

$$W = \begin{pmatrix} 3,4 & 1,9 & 1,2 & 3,7 \\ 2,3 & 2,0 & 3,4 & 2,3 \\ 1,7 & 3,5 & 2,8 & 1,4 \end{pmatrix}$$

Dove il valore

$$w_{11} = (m_{12} + m_{13} + m_{14}) + (m_{21} + m_{31}) + (1 - m_{11}) = (1,0 + 0,8 + 0,0) + (0,3 + 0,3) + (1 - 0,0) = 3,4$$

Il valore di peso più basso è il valore 1,2 corrispondente alla scelta dell'elemento in (1 3), con valore 0,8. Scegliendolo si scarta un elemento con valore 1, ma tutti gli altri elementi sulla riga 1 e sulla colonna 3 sono nulli.

Le matrici M e W al secondo passo diventeranno:

$$M = \begin{pmatrix} 0,3 & 0,9 & 0,4 \\ 0,3 & 0,2 & 0,5 \end{pmatrix}$$

e

$$W = \begin{pmatrix} 2,3 & 1,0 & 2,3 \\ 1,7 & 2,5 & 1,4 \end{pmatrix}$$

Questa volta il peso più basso (valore 1,0) corrisponde all'elemento in (1 2), con valore 0,9. Le matrici si riducono a:

$$M = (0,3 \quad 0,5)$$

e

$$W = (1,2 \quad 0,8)$$

In quest'ultimo passo, viene scelto l'elemento che sta in (1 2) in quanto ha peso più basso. La somma totale risulta dunque:

$$(0,9 + 0,9 + 0,5) = 2,2$$

mentre, utilizzando l'algoritmo più semplice citato in precedenza, si avrebbe avuto:

$$(1,0 + 0,4 + 0,3) = 1,7$$

2.4.1 Il peso dei risultati

Quanto precedentemente detto trova la sua applicazione nell'analisi di due frasi separate, ma nell'ambito dei documenti, le frasi di confronto da prendere in esame sono numerose. Per questo motivo è necessario introdurre una valutazione generale sull'effettiva somiglianza semantica del documento.

Può capitare, infatti, che un documento presenti una sola frase con una elevata somiglianza semantica, mentre le altre producano un valore nullo. In questo caso è logico pensare che il documento abbia poca importanza per la ricerca dell'utente. Diversamente, se un documento presenta numerose frasi con somiglianza semantica diversa da 0, sarà logico supporre che questo testo abbia rilevanza per l'utente.

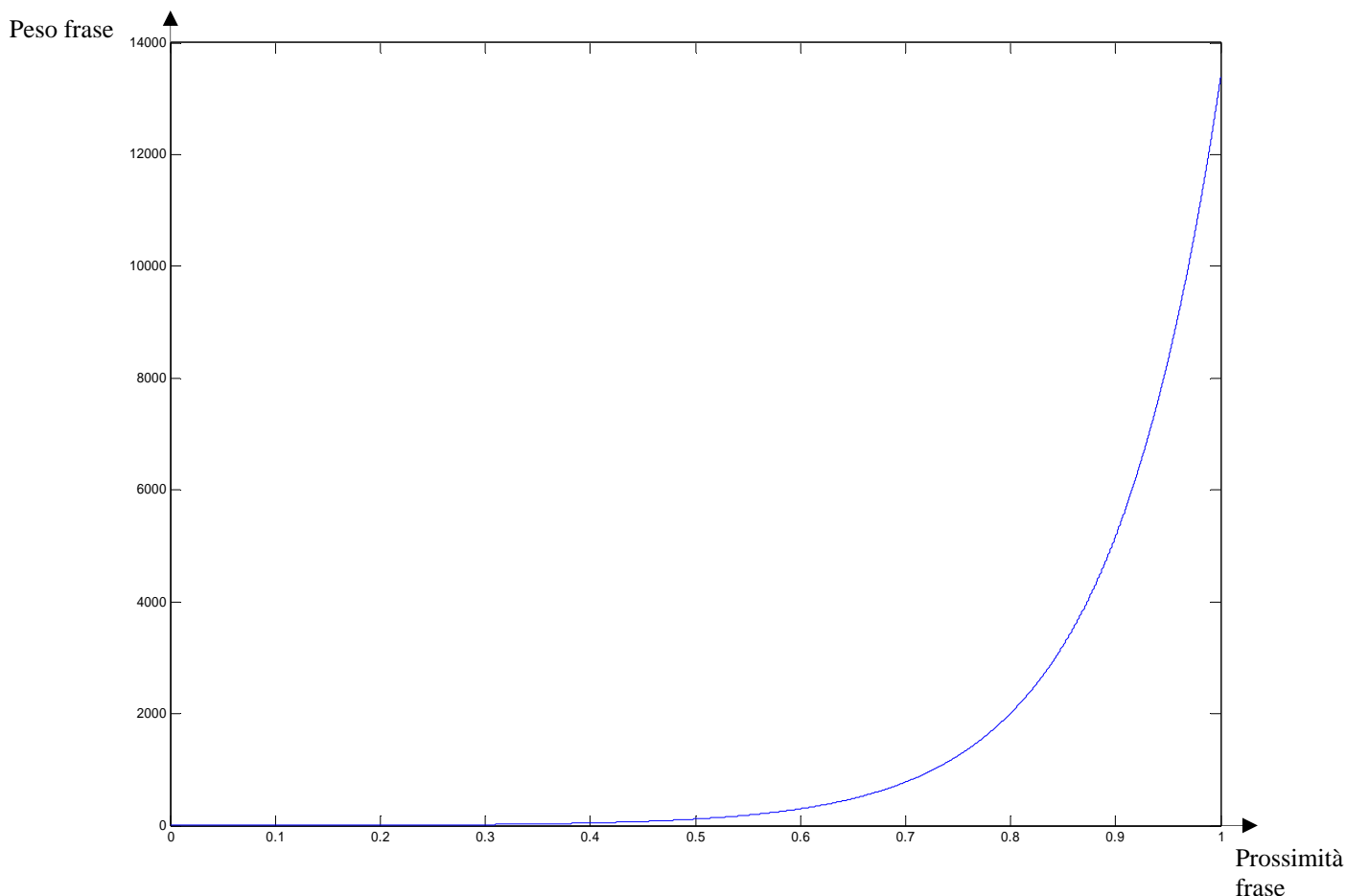
Per ovviare a questo problema, ogni frase dei documenti viene pesata in base al valore di prossimità semantica trovato. In questo modo, frasi con somiglianza nulla influiranno sul risultato permettendo di discriminare documenti non rilevanti.

Il valore di prossimità semantica tra il documento ed il fabbisogno informativo dell'utente sarà quindi data dalla media pesata dei singoli valori di prossimità semantica di ogni frase del documento.

In particolare è stata scelta una funzione esponenziale per il calcolo del peso delle singole frasi:

$$peso = e^{(prossimità * coefficiente)}$$

Questa funzione restituisce un andamento come quello rappresentato nel diagramma seguente:



In questo caso, in corrispondenza di un valore di prossimità 0, si avrà un peso unitario, mentre nel caso in cui si abbia valore di prossimità 1, si avrà un peso proporzionale al numero di frasi. I valori intermedi, riassunti nella seguente tabella riferita ad un documento con circa 2000 frasi, avranno un andamento esponenziale:

Prossimità frase	Peso frase
0	1
0,1	3
0,2	7
0,3	17
0,4	45
0,5	116
0,6	299
0,7	773
0,8	2000
0,9	5172
1,0	13374

I valori sono proporzionali al numero di frasi del documento. L'idea di base è che deve essere possibile annullare la media nel caso in cui nel documento sia presente solamente una frase con prossimità inferiore all'80%. E' ragionevole pensare che:

- se nel documento è presente una frase con prossimità semantica rispetto al fabbisogno informativo dell'utente superiore all'80%, molto probabilmente questo documento sarà significativo per l'utente

- se nel documento è presente una sola frase con prossimità semantica dell'80%, mentre le altre hanno significatività nulla, allora probabilmente il documento sarà poco rilevante per l'utente. Nel caso estremo in cui nel documento sia presente una frase con prossimità 0.8 e tutte le altre frasi 0.0, allora la media sarà molto prossima a 0
- se nel documento sono distribuiti valori di prossimità variabili, allora la media rispecchierà la prossimità semantica generale del documento.

E' importante osservare che, nel caso in cui la ricerca di prossimità semantica individui una frase con valore 1.0, anche se tutte le altre presentassero valore nullo la media assumerebbe un valore superiore al 50%. Per la struttura della formula di Paice-Ramirez, infatti, si può supporre che, in questo caso, il documento sia molto probabilmente quello ricercato dall'utente.

Il coefficiente per il calcolo del peso della frase in relazione al numero di frasi presenti nel documento è strutturato nel seguente modo:

$$Coefficiente = \ln(\text{numero_parole}) \cdot \frac{100}{80}$$

2.4.2 Esempio di funzionamento

Per esemplificare meglio il funzionamento di questo algoritmo, proponiamo un esempio di computazione.

Data la seguente query dell'utente:

S'="Funzione che converte i caratteri in minuscolo"

e il seguente documento:

T'="Metodo per trasformare in spazi i caratteri TAB"

Le due frasi prese in considerazione vengono, quindi, normalizzate, ottenendo:

S="funzione convertire carattere minuscolo"

e

T="metodo trasformare spazio carattere tab"

Si supponga, inoltre, di avere un thesaurus in cui sono memorizzati i seguenti valori di somiglianza semantica tra le coppie di termini:

	<i>metodo</i>	<i>trasformare</i>	<i>spazio</i>	<i>carattere</i>	<i>tab</i>
<i>funzione</i>	0,83	0,21	0	0	0
<i>convertire</i>	0	0,91	0	0,42	0,24
<i>carattere</i>	0	0	0,29	1	0,22
<i>maiuscolo</i>	0	0,23	0	0,37	0

Ora l'algoritmo calcola la matrice W del primo passo:

$$W = \begin{pmatrix} 0,38 & 2,76 & 2,33 & 3,83 & 2,5 \\ 3,4 & 1,19 & 2,86 & 3,1 & 2,31 \\ 3,34 & 3,86 & 1,93 & 1,3 & 2,31 \\ 2,43 & 2,26 & 1,89 & 2,28 & 2,06 \end{pmatrix}$$

Il valore minimo della matrice è 0,38. Il valore da scegliere, quindi è quello nella posizione (1 1), ovvero la coppia (*funzione metodo*).

Le matrici diventano, eliminando la riga e la colonna n°1:

$$M = \begin{pmatrix} 0,91 & 0 & 0,42 & 0,24 \\ 0 & 0,29 & 1 & 0,22 \\ 0,23 & 0 & 0,37 & 0 \end{pmatrix}$$

e

$$W = \begin{pmatrix} 1,19 & 2,86 & 3,1 & 2,31 \\ 3,86 & 1,93 & 1,3 & 2,31 \\ 2,26 & 1,89 & 2,28 & 2,06 \end{pmatrix}$$

Il valore minimo della matrice è 1,19. Il valore da scegliere, quindi è quello nella posizione (2 2), ovvero la coppia (*convertire trasformare*).

Le matrici diventano, eliminando la riga e la colonna n°2:

$$M = \begin{pmatrix} 0,29 & 1 & 0,22 \\ 0 & 0,37 & 0 \end{pmatrix}$$

e

$$W = \begin{pmatrix} 1,93 & 1,3 & 2,31 \\ 1,89 & 2,28 & 2,06 \end{pmatrix}$$

Il valore minimo della matrice è 1,3. Il valore da scegliere, quindi è quello nella posizione (4 3), ovvero la coppia (*carattere carattere*).

Le matrici diventano, eliminando la riga n° 3 e la colonna n°4:

$$M = (0 \ 0)$$

e

$$W = (1 \ 1)$$

Il questo caso la scelta tra le coppie di termini è irrilevante, in quanto presentano lo stesso valore. Inoltre, il valore “m” scelto, è in ogni caso 0, quindi la scelta non influisce sul risultato.

Al termine della fase di mapping, quindi, si sono rilevati i seguenti accoppiamenti:

$$J(1) = 1 \quad v_{11} = 0,83$$

$$J(2) = 2 \quad v_{22} = 0,91$$

$$j(3) = 4 \quad v_{34} = 1,0$$

e, dunque, il valore della funzione di prossimità risulterà essere:

$$R(S,T) = \frac{1}{5}(0,83 + 0,91 + 1,00) \cdot \Phi(J) = 0,548 \cdot \Phi(J)$$

Rimane ora da valutare la funzione di *tidyness* per poter quantificare il risultato del confronto tra S e T . Esaminando se esiste un ordinamento tra gli accoppiamenti si nota che:

$$K_1 = 1 \text{ in quanto } J(2) = 2 = J(1) + 1$$
$$K_2 = 0 \text{ in quanto } J(3) = 4 > J(2) + 1$$

Quindi si avrà che:

$$\Phi(J) = \frac{1}{3}(1+0)$$

e la funzione di *tidyness* ritorna il valore:

$$\Psi(J) = 0,8 + \frac{1}{3} \cdot (1 - 0,8)$$

Per cui, il valore complessivo dato dall'algoritmo di prossimità semantica è:

$$R(S,T) = \frac{1}{5}(2,74) \cdot \Phi(J) = 0,475$$

3. Risultati dell'algoritmo

La complessità delle formule implicate in questo algoritmo rende difficile una valutazione complessiva della qualità del codice. Il grande numero di parametri che entrano in gioco, infatti, richiederebbe l'utilizzo di uno spazio multidimensionale per avere una visione complessiva del funzionamento dell'algoritmo.

Per ovviare a questo problema si è scelto di eseguire dei test separati che valutino le singole componenti dell'algoritmo.

Il testo preso in esame per le valutazioni è il seguente:

In XML retrieval, two distinct approaches have been established and pursued without much cross-fertilization taking place so far. On the one hand, native XML databases tailored to the semistructured data model have received considerable attention, and a wealth of index structures, join algorithms, tree encodings and query rewriting techniques for XML have been proposed. On the other hand, the question how to make XML .t the relational data model has been studied in great detail, giving rise to a multitude of storage schemes for XML in relational database systems (RDBSs). In this paper we examine how native XML indexing techniques can boost the retrieval of XML stored in an RDBS. We present the Relational CADG (RCADG), an adaptation of several native indexing approaches to the relational model, and show how it supports the evaluation of a clean formal language of conjunctive XML queries. Unlike relational storage schemes for XML, the RCADG largely preserves the underlying tree structure of the data in the RDBS, thus addressing several open problems known from the literature. Experiments show that the RCADG accelerates retrieval by up to two or even three orders of magnitude compared to both native and relational approaches.

3.1 Valutazione dei sinonimi

E' importante che l'algoritmo utilizzi in modo appropriato i dati contenuti nel thesaurus per effettuare le ricerche. Il test condotto consiste nell'eseguire differenti ricerche sullo stesso testo inserendo come query dei sinonimi. Sarà quindi necessario verificare che il risultato finale di prossimità del testo sia proporzionale al valore di somiglianza semantica presente nel thesaurus.

	Prossimità massima	Prossimità media	Somiglianza tra XML	Rapporto tra prossimità massima XML
XML	25,0 %	13,1 %	1,000	1,000
Document	11,33 %	4,64 %	0,453	0,453
Database	14,72 %	7,09 %	0,589	0,588
Schema	8,02 %	2,47 %	0,321	0,311

Tab. 3.1 – 1 Nella colonna "prossimità semantica" è riportato il valore di prossimità semantica massimo ritrovato nel documento, mentre la voce "prossimità media" rappresenta il valore medio dell'intero documento. Nella colonna "somiglianza tra XML" viene riportato il valore di somiglianza semantica, contenuto nel thesaurus, tra il termine "XML" e il termine della riga corrispondente. Nel'ultima colonna viene riportato il rapporto tra il valore di somiglianza massimo rilevato nella query "XML" rispetto alla query della corrispondente riga.

Il test è stato condotto eseguendo la ricerca con la parola “XML”. E’ stato analizzato, quindi, il thesaurus alla ricerca dei sinonimi della parola e ne sono stati scelti alcuni a caso. E’ stata eseguita nuovamente la ricerca con i sinonimi individuati in precedenza e sono stati registrati i valori ottenuti.

La tabella mostra che l’algoritmo utilizza correttamente lo strumento thesaurus in quanto restituisce dei risultati paragonabili con l’effettiva somiglianza semantica tra le parole analizzate.

3.2 Valutazione dell’ordinamento

Un altro importante aspetto dell’algoritmo che è necessario verificare è la capacità di valutare l’ordinamento delle parole della query rispetto alla frase analizzata del documento. Per valutare questo aspetto è stata introdotta nel documento un’apposita frase contenente tre parole chiave; i test sono stati condotti modificando l’ordine dei termini della query ed analizzando le variazioni della prossimità semantica rilevata.

XML document database. In XML retrieval, two distinct approaches have been established and pursued without much cross-fertilization taking place so far. On the one hand, native XML databases tailored to the semistructured data model have received considerable attention, and a wealth of index structures, join algorithms, tree encodings and query rewriting techniques for XML have been proposed. On the other hand, the question how to make XML fit the relational data model has been studied in great detail, giving rise to a multitude of storage schemes for XML in relational database systems (RDBSs). In this paper we examine how native XML indexing techniques can boost the retrieval of XML stored in an RDBS. We present the Relational CADG (RCADG), an adaptation of several native indexing approaches to the relational model, and show how it supports the evaluation of a clean formal language of conjunctive XML queries. Unlike relational storage schemes for XML, the RCADG largely preserves the underlying tree structure of the data in the RDBS, thus addressing several open problems known from the literature. Experiments show that the RCADG accelerates retrieval by up to two or even three orders of magnitude compared to both native and relational approaches.

Le ricerche sono state fatte con le seguenti combinazioni:

Query	Prossimità semantica
XML document database	100%
Database XML document	86,66%
Database document XML	80%
XML database document	86,66%
Database score relational	53,01%
Structured database schema	53,46%
Database structured schema	49,34%
Schema database structured	53,46%
Schema structured database	61,68%

Tab. 3.2 – 1 Nella colonna “prossimità semantica” è riportato il valore di prossimità semantica massimo ritrovato nel documento, mentre la voce “query” contiene il fabbisogno informativo cui è associato il valore di prossimità semantica.

Dalla tabella precedente si può osservare che:

- l'algoritmo tiene effettivamente conto dell'ordinamento delle parole del fabbisogno informativo dell'utente rispetto alle frasi analizzate nel documento. In particolare, tenendo conto dell'osservazione fatta nei capitoli precedenti in cui si considera che anche nel caso in cui i termini fossero invertiti la rilevanza non dovrebbe essere pari a zero, si può notare che, se i termini analizzati sono in ordine invertito rispetto a quelli presenti nel testo, la prossimità semantica è dell'80%.
- l'algoritmo tiene conto unicamente dell'ordinamento di due o più parole all'interno della frase del documento. Questo comportamento si può osservare nella combinazione "Database document XML" dove la posizione centrale del termine "document" non influisce sul peso dell'ordinamento dei termini. Nella combinazione "Database XML document" l'ordinamento dei termini "XML document" comporta un peso maggiore della ricerca.
- l'ordinamento che comprende l'analisi dei sinonimi è molto più complesso in quanto la sua scelta dipende dai pesi di ogni termine delle frasi. Per questo motivo test diretti sulla valutazione del peso dell'ordinamento con l'utilizzo di sinonimi risultano molto difficili.

L'algoritmo è quindi in grado di analizzare l'ordinamento dei termini del fabbisogno informativo dell'utente rispetto alla frase del documento, per ottimizzare il valore di prossimità semantica individuato.

3.3 Valutazione della lunghezza delle frasi

Un altro aspetto da analizzare è l'influenza che la lunghezza della frase ha sul valore di prossimità semantica indicato dall'algoritmo. Per eseguire questa valutazione si può modificare la lunghezza delle frasi di un testo e analizzare la variazione dei risultati. Il testo originale è quello riportato all'inizio di questo capitolo, mentre le variazioni sono le seguenti:

MODIFICA 1: In XML retrieval, two distinct approaches have been established and pursued without much cross-fertilization taking place so far. On the one hand, native XML databases tailored to the semistructured data model have received considerable attention, and a wealth of index structures, join algorithms, tree encodings and query rewriting techniques for XML have been proposed. On the other hand, the question how to make XML .t the relational data model has been studied in great detail, giving rise to a multitude of storage schemes for XML in relational database systems (RDBSs). In this paper we examine how native XML indexing techniques can boost the retrieval of XML stored in an RDBS. We present the Relational CADG (RCADG), an adaptation of several native indexing approaches to the relational model, and show how it supports the evaluation of a clean formal language of conjunctive XML queries. Unlike relational storage schemes for XML, the RCADG largely preserves the underlying tree structure of the data in the RDBS, thus addressing several open problems known from the literature. Experiments show that the RCADG accelerates retrieval by up to two or even three orders of magnitude compared to both native and relational approaches.

MODIFICA 2: In XML retrieval, two distinct approaches have been established and pursued without much cross-fertilization taking place so far. On the one hand, native XML databases tailored to the semistructured data model have received considerable attention, and a wealth of index structures, join algorithms, tree encodings and query rewriting techniques for XML have been proposed. On the other hand, the question how to make XML fit the relational data model has been studied in great detail, giving rise to a multitude of storage schemes for XML in relational database systems (RDBSs). In this paper we examine how native XML indexing techniques can boost the retrieval of XML stored in an RDBS. We present the Relational CADG (RCADG), an adaptation of several native indexing approaches to the relational model, and show how it supports the evaluation of a clean formal language of conjunctive XML queries. Unlike relational storage schemes for XML, the RCADG largely preserves the underlying tree structure of the data in the RDBS, thus addressing several open problems known from the literature. Experiments show that the RCADG accelerates retrieval by up to two or even three orders of magnitude compared to both native and relational approaches.

Nella seguente tabella sono riportati i dettagli dell'esecuzione dell'algoritmo sul testo campione. Sono indicati i dati significativi come i valori delle funzioni phi e psi, i valori di somiglianza e di prossimità rilevati, nonché il peso della frase nella media.

n° Frase	Frase	PHI	PSI	Somiglianza	Prossimità	Peso
0	xml retrieval two distinct approach have be establish pursued cross fertilization take place far	1	1	2	0,142857	1
1	on hand native xml databases tailor semistructured datum model have received consider attention wealth index structures join algorithm tree encodings query rewriting techniques xml have be proposed	0,5	0,9	1,383879	0,046129	1
2	hand question make xml	0	0,8	1,308697	0,261739	1
3	relational datum model have be studied great detail give rise multitude storage scheme xml relational database system rdbss	0	0,8	1,40161	0,062294	1
4	paper examine native xml index techniques can boost retrieval xml store rdbs	0,5	0,9	2	0,15	1
5	present relational cadg rcadg adapt sever native index approach relational model show support evaluation clean formal language conjunct xml queries	0,5	0,9	1,392474	0,062661	1
6	unlike relational storage scheme xml rcadg largely preserves underlying tree structure datum rdbs thus address sever open problem know literature	0,5	0,9	1,40161	0,063072	1
7	experiments show rcadg accelerates retrieval up two three order magnitude compared native relational approach	0,5	0,9	1,628746	0,104705	1

In questa tabella sono riportati i valori dell'esecuzione dell'algoritmo sul testo modificato n°1. Come si può notare, la frase n°3 (modificata) risulta troncata della parte iniziale. Questo è dovuto al fatto che nella normalizzazione la prima parte consisteva nel solo termine "hand". Le frasi che hanno un numero di termini inferiore a quelli del fabbisogno informativo dell'utente vengono scartate. La frase n°3 modificata è costituita da 3 termini anziché da 4 come quella originale. Nella formula generale dell'algoritmo di Paice-Ramirez si può notare che il risultato finale è diviso per il numero di parole della frase analizzata. Infatti, essendo diminuiti i termini, il valore di prossimità calcolato è aumentato, nonostante i valori delle altre funzioni (phi, psi e somiglianza) risultino invariati.

n° Frase	Frase	PHI	PSI	Somiglianza	Prossimità	Peso
0	xml retrieval two distinct approach have be establish pursued cross fertilization take place far	1	1	2	0,142857143	1
1	on hand native xml databases tailor semistructured datum model have received consider attention wealth index structures join algorithm tree encodings query rewriting techniques xml have be proposed	0,5	0,9	1,383878864	0,046129295	1
3	question make xml	0	0,8	1,308697453	0,348985988	2
4	relational datum model have be studied great detail give rise multitude storage scheme xml relational database system rdbss	0	0,8	1,401609664	0,062293763	1
5	paper examine native xml index techniques can boost retrieval xml store rdbss	0,5	0,9	2	0,15	1
6	present relational cadg rcadg adapt sever native index approach relational model show support evaluation clean formal language conjunct xml queries	0,5	0,9	1,392473953	0,062661328	1
7	unlike relational storage scheme xml rcadg largely preserves underlying tree structure datum rdbss thus address sever open problem know literature	0,5	0,9	1,401609664	0,063072435	1
8	experiments show rcadg accelerates retrieval up two three order magnitude compared native relational approach	0,5	0,9	1,628746083	0,104705105	1

In questa tabella sono riportati i valori dell'esecuzione dell'algoritmo sul testo modificato n°2. La frase originale n°6 è stata suddivisa in due frasi: n°6 e n°7. Come si può notare, delle due parti, quella con minore lunghezza ha un valore di prossimità superiore per i motivi riportati in precedenza.

n° Frase	Frase	PHI	PSI	Somiglianza	Prossimità	Peso
0	xml retrieval two distinct approach have be establish pursued cross fertilization take place far	1	1	2	0,142857	1
1	on hand native xml databases tailor semistructured datum model have received consider attention wealth index structures join algorithm tree encodings query rewriting techniques xml have be proposed	0,5	0,9	1,383879	0,046129	1
2	hand question make xml	0	0,8	1,308697	0,261739	2
3	relational datum model have be studied great detail give rise multitude storage scheme xml relational database system rdbss	0	0,8	1,40161	0,062294	1
4	paper examine native xml index techniques can boost retrieval xml store rdbss	0,5	0,9	2	0,15	1
5	present relational cadg rcadg adapt sever native index approach relational model show support evaluation clean formal language conjunct xml queries	0,5	0,9	1,392474	0,062661	1
6	unlike relational storage scheme xml	0,5	0,9	1,40161	0,25229	1
7	rcadg largely preserves underlying tree structure datum rdbss thus address sever open problem know literature	0,5	0,9	0,632604	0,037956	1
8	experiments show rcadg accelerates retrieval up two three order magnitude compared native relational approach	0,5	0,9	1,628746	0,104705	1

Dai precedenti risultati si può evincere che l'algoritmo viene influenzato correttamente dalla lunghezza della frase considerata.

3.4 Valutazione globale dell'algoritmo

E' importante analizzare, come ultima valutazione, l'efficacia globale dell'algoritmo. Per fare questo verrà preso in analisi lo stesso gruppo di documenti utilizzati per la costruzione del thesaurus. I testi hanno argomentazione inerente l'informatica ed, in particolare, trattano di *information retrieval*, metodi per la memorizzazione dei testi e XML. Sarà importante in questo caso notare il tempo impiegato per portare a termine la ricerca.

Analizziamo le seguenti query:

- information retrieval
- xml document retrieval
- Client-Server computing in mobile environments

information retrieval	xml document retrieval	Client-Server computing in mobile environments
345 secondi	384 secondi	594 secondi
p144-kobayashi.txt - 24.41% C86-1090.txt - 24.41% p178-hiemstra.txt - 13.93% p79-xu.txt - 12.25% p35-kherfi.txt - 12.01% p23-weigel.txt - 11.02% p183-ogilvie.txt - 10.02% p668-ko.txt - 8.27% p965-chor.txt - 7.63% p27-berger.txt - 6.85% p337-wong.txt - 6.59% p375-liu.txt - 6.45% p313-fuhr.txt - 6.33% p73-downie.txt - 6.29% p10-roennau.txt - 6.29% p19-si.txt - 6.09% p133-riloff.txt - 5.94% p11-huibers.txt - 5.85% p252-greiff.txt - 5.78% p331-westermann.txt - 5.69% p32-si.txt - 5.45% p48-chen.txt - 5.01% p538-montague.txt - 4.95% p83-calvanese.txt - 4.71% p357-amati.txt - 4.71% p132-n_robinson.txt - 4.65% p599-hall.txt - 4.44% a27-wilkinson.txt - 4.23% a32-bieber.txt - 4.13% p1-dedrick.txt - 3.98% p87-malone.txt - 3.92% p117-jing.txt - 3.84% p11-yu.txt - 3.32% p415-frakes.txt - 2.97% p119-dietrich.txt - 2.86% p653-doyle.txt - 2.63%	p10-roennau.txt - 25.19% p331-westermann.txt - 24.07% p178-hiemstra.txt - 14.55% p313-fuhr.txt - 14.41% p183-ogilvie.txt - 12.26% p23-weigel.txt - 10.26% p375-liu.txt - 10.2% p79-xu.txt - 10.11% p668-ko.txt - 9.55% p32-si.txt - 9.42% p35-kherfi.txt - 9.19% p357-amati.txt - 8.85% p19-si.txt - 8.56% p337-wong.txt - 8.18% p144-kobayashi.txt - 7.62% C86-1090.txt - 7.62% p133-riloff.txt - 7.58% p252-greiff.txt - 7.53% p27-berger.txt - 6.57% p965-chor.txt - 5.74% p83-calvanese.txt - 5.66% p538-montague.txt - 5.63% p11-huibers.txt - 5.53% p48-chen.txt - 5.5% a32-bieber.txt - 5.43% p11-yu.txt - 5.42% p73-downie.txt - 4.94% p599-hall.txt - 4.66% a27-wilkinson.txt - 3.56% p132-n_robinson.txt - 3.54% p87-malone.txt - 3.46% p415-frakes.txt - 3.42% p117-jing.txt - 3.32% p119-dietrich.txt - 3.16% p1-dedrick.txt - 2.99% p653-doyle.txt - 2.45%	p117-jing.txt - 25.27% p337-wong.txt - 6.62% p313-fuhr.txt - 6.59% p965-chor.txt - 6.42% p357-amati.txt - 6.11% p79-xu.txt - 4.69% p415-frakes.txt - 4.25% p331-westermann.txt - 3.88% p87-malone.txt - 3.74% p35-kherfi.txt - 3.71% p178-hiemstra.txt - 3.43% p132-n_robinson.txt - 3.36% p599-hall.txt - 3.23% p538-montague.txt - 3.0% p653-doyle.txt - 2.85% a32-bieber.txt - 2.76% p1-dedrick.txt - 2.66% p119-dietrich.txt - 2.66% p144-kobayashi.txt - 2.56% C86-1090.txt - 2.55% p19-si.txt - 2.23% p133-riloff.txt - 2.13% p73-downie.txt - 1.88% p83-calvanese.txt - 1.79% p32-si.txt - 1.79% p183-ogilvie.txt - 1.7% p252-greiff.txt - 1.58% p11-huibers.txt - 1.56% a27-wilkinson.txt - 1.51% p23-weigel.txt - 1.47% p27-berger.txt - 1.43% p668-ko.txt - 1.32% p10-roennau.txt - 1.27% p375-liu.txt - 0.96% p48-chen.txt - 0.95% p11-yu.txt - 0.77%

Tab. 3.4 – I Nella tabella sono riportati i risultati dell’esecuzione del codice realizzato in questo stage utilizzando le query contenute nella prima riga. La lista dei documenti risulta ordinata secondo il valore di prossimità semantica. Nella seconda riga è riportato il tempo, in secondi, necessario ad ottenere il risultato (computer n°2)

Osservando la tabella precedente si può subito notare che i tempi necessari all’esecuzione dell’algoritmo su un discreto numero di documenti è molto alto (dell’ordine dei 5 minuti). Si tenga anche conto che i documenti sono già normalizzati e, quindi, il tempo esclude la fase di normalizzazione. Questo tempo alto è dovuto al fatto che il tempo di calcolo è proporzionale a:

$$\text{numero_frasi} * \text{numero_parole_documento}^2 * \text{numero_parole_query}$$

Dai risultati ottenuti si può osservare che i valori di prossimità semantica rilevati non sono elevati, ma bisogna precisare che questi sono i valori medi per il documento e non massimi. Come si può notare, la seconda ricerca “xml document retrieval” fornisce risultati mediamente più alti: questo è dovuto al fatto che il fabbisogno informativo dell’utente comprende la maggior parte degli argomenti dei documenti. Nell’esecuzione con query “Client-Server computing in mobile environments”, invece, i risultati sono mediamente più bassi perché l’argomento indicato dal fabbisogno informativo dell’utente è trattato esaurientemente solo nel documento “p117-jing.txt” che, infatti, ha un valore di prossimità semantica nettamente superiore agli altri.

3.5 Valutazione dei parametri precision e recall

Al fine di verificare l'efficienza dell'algoritmo sviluppato è necessario analizzare i parametri di precisione e richiamo, che caratterizzano i sistemi per il reperimento di informazioni. Per via della complessità di questo tipo di analisi è utile usare dei documenti semplificati che possano essere valutati correttamente anche manualmente.

Sono stati considerati solo i titoli dei documenti già citati in precedenza al fine di semplificare notevolmente l'analisi dei risultati pur mantenendo il significato generale. Esaminando una sola frase per documento, infatti, la verifica dell'effettiva pertinenza dei risultati diventa immediata ed è perciò possibile applicare correttamente e facilmente le formule di richiamo e precisione.

Sono state condotte tre ricerche:

- ricerca 1: information retrieval
- ricerca 2: xml document
- ricerca 3: link generation

I risultati ottenuti sono stati raccolti nella seguente tabella:

	Ricerca 1	Ricerca 2	Ricerca 3
a27wilkinson.txt	9,70%	11,60%	60,00%
a32bieber.txt	19,80%	0,00%	15,32%
C861090.txt	66,67%	13,33%	12,12%
p10roennau.txt	7,20%	32,00%	5,57%
p117jing.txt	5,33%	0,00%	0,00%
p119dietrich.txt	0,00%	0,00%	0,00%
p11huibers.txt	50,00%	10,00%	9,09%
p11yu.txt	16,36%	3,64%	5,94%
p132n_robinson.txt	12,14%	0,00%	0,00%
p133riloff.txt	50,00%	10,00%	7,94%
p144kobayashi.txt	66,67%	13,33%	12,12%
p178hiemstra.txt	36,00%	8,00%	5,82%
p183ogilvie.txt	33,33%	7,38%	4,85%
p19si.txt	9,47%	4,50%	0,00%
p1dedrick.txt	17,41%	3,84%	4,55%
p23weigel.txt	13,85%	12,60%	3,21%
p252greiff.txt	28,57%	8,64%	5,70%
p27berger.txt	25,71%	5,71%	11,00%
p313fuhr.txt	28,57%	20,61%	9,88%
p32si.txt	8,97%	5,39%	4,32%
p331westermann.txt	4,55%	15,46%	4,01%
p337wong.txt	30,00%	6,00%	4,85%
p357amati.txt	22,50%	5,00%	8,64%
p35kherfi.txt	18,74%	8,59%	3,15%
p375liu.txt	22,18%	6,40%	8,08%
p415frakes.txt	8,64%	7,48%	0,00%
p48chen.txt	13,75%	10,02%	13,33%
p538montague.txt	22,50%	9,00%	0,00%
p599hall.txt	27,17%	6,95%	7,13%
p653doyle.txt	0,00%	0,00%	0,00%
p73downie.txt	28,57%	5,94%	4,16%
p79xu.txt	28,57%	5,14%	4,16%
p83calvanese.txt	14,14%	0,00%	0,00%
p87malone.txt	0,00%	0,00%	0,00%
p965chor.txt	60,00%	12,00%	9,70%

Successivamente, i documenti sono stati analizzati manualmente, valutandone la pertinenza con il fabbisogno informativo dell'utente. E' stata rappresentata con il valore "1" la rilevanza del testo per la query, mentre è stato assegnato il valore "0" in caso di non attinenza. I dati sono raccolti nella seguente tabella:

	Ricerca 1	Ricerca 2	Ricerca 3	Pertinenza ricerca 1	Pertinenza ricerca 2	Pertinenza ricerca 3
a27wilkinson.txt	9,70%	11,60%	60,00%	0	0	1
a32bieber.txt	19,80%	0,00%	15,32%	0	1	1
C861090.txt	66,67%	13,33%	12,12%	1	0	1
p10roennau.txt	7,20%	32,00%	5,57%	0	1	0
p117jing.txt	5,33%	0,00%	0,00%	1	0	0
p119dietrich.txt	0,00%	0,00%	0,00%	0	0	0
p11huibers.txt	50,00%	10,00%	9,09%	1	0	0
p11yu.txt	16,36%	3,64%	5,94%	1	0	1
p132n_robinson.txt	12,14%	0,00%	0,00%	0	0	0
p133riloff.txt	50,00%	10,00%	7,94%	1	1	0
p144kobayashi.txt	66,67%	13,33%	12,12%	1	1	1
p178hiemstra.txt	36,00%	8,00%	5,82%	1	0	0
p183ogilvie.txt	33,33%	7,38%	4,85%	1	0	0
p19si.txt	9,47%	4,50%	0,00%	1	1	0
p1dedrick.txt	17,41%	3,84%	4,55%	1	0	0
p23weigel.txt	13,85%	12,60%	3,21%	1	1	0
p252greiff.txt	28,57%	8,64%	5,70%	1	0	0
p27berger.txt	25,71%	5,71%	11,00%	1	0	0
p313fuhr.txt	28,57%	20,61%	9,88%	1	1	0
p32si.txt	8,97%	5,39%	4,32%	1	0	0
p331westermann.txt	4,55%	15,46%	4,01%	0	1	0
p337wong.txt	30,00%	6,00%	4,85%	1	0	0
p357amati.txt	22,50%	5,00%	8,64%	1	0	0
p35kherfi.txt	18,74%	8,59%	3,15%	1	1	1
p375liu.txt	22,18%	6,40%	8,08%	1	0	0
p415frakes.txt	8,64%	7,48%	0,00%	0	0	0
p48chen.txt	13,75%	10,02%	13,33%	1	0	1
p538montague.txt	22,50%	9,00%	0,00%	1	0	0
p599hall.txt	27,17%	6,95%	7,13%	1	0	1
p653doyle.txt	0,00%	0,00%	0,00%	0	0	0
p73downie.txt	28,57%	5,94%	4,16%	1	0	0
p79xu.txt	28,57%	5,14%	4,16%	1	0	0
p83calvanese.txt	14,14%	0,00%	0,00%	1	0	0
p87malone.txt	0,00%	0,00%	0,00%	0	0	0
p965chor.txt	60,00%	12,00%	9,70%	1	0	0
Totale documenti rilevanti:				26	9	8

Tab. 3.5 – 1 Nella prima colonna sono riportati i documenti utilizzati per completare il test. Nelle colonne "ricerca 1", "ricerca 2" e "ricerca 3" sono riportati i valori di prossimità semantica calcolati per il documento riportato nella riga corrispondente. Nelle colonne "pertinenza ricerca 1", "pertinenza ricerca 2" e "pertinenza ricerca 3" sono, invece, riportati i valori di pertinenza del documento, secondo la relativa query, calcolati manualmente. L'analisi è stata fatta manualmente per ogni documento: la pertinenza è stata individuata con il valore "1", mentre la non pertinenza con il valore "0". La voce "totale documenti rilevanti" raccoglie il numero di documenti rilevanti per ogni query di riferimento

Al fine di applicare correttamente le formule per il calcolo dei due coefficienti, è necessario introdurre una conversione dalla percentuale al valore booleano. E' stato scelto di considerare effettivamente rilevanti solamente i documenti che presentino un valore di prossimità semantica superiore al 10% assegnando valore "1". La precedente tabella, quindi, può essere riscritta nel seguente modo, eliminando le percentuali:

	Ricerca 1	Ricerca 2	Ricerca 3	Pertinenza ricerca 1	Pertinenza ricerca 2	Pertinenza ricerca 3
a27wilkinson.txt	0	1	1	0	0	1
a32bieber.txt	1	0	1	0	1	1
C861090.txt	1	1	1	1	0	1
p10roennau.txt	0	1	0	0	1	0
p117jing.txt	0	0	0	1	0	0
p119dietrich.txt	0	0	0	0	0	0
p11huibers.txt	1	0	0	1	0	0
p11yu.txt	1	0	0	1	0	1
p132n_robinson.txt	1	0	0	0	0	0
p133riloff.txt	1	0	0	1	1	0
p144kobayashi.txt	1	1	1	1	1	1
p178hiemstra.txt	1	0	0	1	0	0
p183ogilvie.txt	1	0	0	1	0	0
p19si.txt	0	0	0	1	1	0
p1dedrick.txt	1	0	0	1	0	0
p23weigel.txt	1	1	0	1	1	0
p252greiff.txt	1	0	0	1	0	0
p27berger.txt	1	0	1	1	0	0
p313fuhr.txt	1	1	0	1	1	0
p32si.txt	0	0	0	1	0	0
p331westermann.txt	0	1	0	0	1	0
p337wong.txt	1	0	0	1	0	0
p357amati.txt	1	0	0	1	0	0
p35kherfi.txt	1	0	0	1	1	1
p375liu.txt	1	0	0	1	0	0
p415frakes.txt	0	0	0	0	0	0
p48chen.txt	1	1	1	1	0	1
p538montague.txt	1	0	0	1	0	0
p599hall.txt	1	0	0	1	0	1
p653doyle.txt	0	0	0	0	0	0
p73downie.txt	1	0	0	1	0	0
p79xu.txt	1	0	0	1	0	0
p83calvanese.txt	1	0	0	1	0	0
p87malone.txt	0	0	0	0	0	0
p965chor.txt	1	1	0	1	0	0
Tot documenti rilevanti:	25	9	6	26	9	8

Tab. 3.5 – 2 Le colonne rappresentate sono quelle illustrate nella tabella precedente, ma al posto del valore di prossimità semantica è stato inserito un valore booleano per indicare la pertinenza del documento. Valore di prossimità semantica superiori al 10% sono stati convertiti in "1", altrimenti è stato inserito il valore "0". La voce "totale documenti rilevanti" è stata espansa anche ai valori calcolati automaticamente dall'algoritmo.

Per calcolare quali tra i documenti recuperati sono effettivamente rilevanti è sufficiente applicare l'operazione logica di AND tra le colonne corrispondenti di "ricerca" e "pertinenza ricerca". In questo modo si otterrà il valore "1" solamente in corrispondenza di quei documenti che sono pertinenti e che sono stati effettivamente recuperati.

La tabella precedentemente riportata, quindi, viene espansa nel seguente modo:

	Ricerca 1	Ricerca 2	Ricerca 3	Pert. ricerca 1	Pert. ricerca 2	Per. ricerca 3	Doc recuperato 1	Doc. recuperato 2	Doc. recuperato 3
a27wilkinson.txt	0	1	1	0	0	1	no	no	si
a32bieber.txt	1	0	1	0	1	1	no	no	si
C861090.txt	1	1	1	1	0	1	si	no	si
p10roennau.txt	0	1	0	0	1	0	no	si	no
p117jing.txt	0	0	0	1	0	0	no	no	no
p119dietrich.txt	0	0	0	0	0	0	no	no	no
p11huibers.txt	1	0	0	1	0	0	si	no	no
p11yu.txt	1	0	0	1	0	1	si	no	no
p132n_robinson.txt	1	0	0	0	0	0	no	no	no
p133riloff.txt	1	0	0	1	1	0	si	no	no
p144kobayashi.txt	1	1	1	1	1	1	si	si	si
p178hiemstra.txt	1	0	0	1	0	0	si	no	no
p183ogilvie.txt	1	0	0	1	0	0	si	no	no
p19si.txt	0	0	0	1	1	0	no	no	no
p1dedrick.txt	1	0	0	1	0	0	si	no	no
p23weigel.txt	1	1	0	1	1	0	si	si	no
p252greiff.txt	1	0	0	1	0	0	si	no	no
p27berger.txt	1	0	1	1	0	0	si	no	no
p313fuhr.txt	1	1	0	1	1	0	si	si	no
p32si.txt	0	0	0	1	0	0	no	no	no
p331westermann.txt	0	1	0	0	1	0	no	si	no
p337wong.txt	1	0	0	1	0	0	si	no	no
p357amati.txt	1	0	0	1	0	0	si	no	no
p35kherfi.txt	1	0	0	1	1	1	si	no	no
p375liu.txt	1	0	0	1	0	0	si	no	no
p415frakes.txt	0	0	0	0	0	0	no	no	no
p48chen.txt	1	1	1	1	0	1	si	no	si
p538montague.txt	1	0	0	1	0	0	si	no	no
p599hall.txt	1	0	0	1	0	1	si	no	no
p653doyle.txt	0	0	0	0	0	0	no	no	no
p73downie.txt	1	0	0	1	0	0	si	no	no
p79xu.txt	1	0	0	1	0	0	si	no	no
p83calvanese.txt	1	0	0	1	0	0	si	no	no
p87malone.txt	0	0	0	0	0	0	no	no	no
p965chor.txt	1	1	0	1	0	0	si	no	no
Tot documenti rilevanti:	25	9	6	26	9	8	23	5	5

Tab. 3.5 – 3 Le colonne rappresentate sono quelle illustrate nella tabella precedente con l'aggiunta delle tre colonne "Documento recuperato 1", "Documento recuperato 2" e "Documento recuperato 3". In queste colonne è indicato se i documenti che sono stati reputati rilevanti sono stati effettivamente recuperati dall'algoritmo. In particolare, è stata applicata la funzione AND tra le relative colonne "Ricerca n" e "Pertinenza ricerca n".

Ora è possibile calcolare i parametri di precisione e richiamo ed effettuare delle valutazioni sull'algoritmo sviluppato.

Per puntualizzare si ricorda che la precisione è data dal rapporto tra il numero di documenti pertinenti recuperati e il numero di documenti recuperati, mentre il richiamo è definito come il rapporto tra il numero di documenti recuperati pertinenti e il numero di documenti pertinenti.

Si ha, quindi:

$$precision = \frac{\text{documenti_recuperati_pertinenti}}{\text{documenti_recuperati}} = \frac{\text{documento_recuperato}}{\text{ricerca}}$$

$$recall = \frac{\text{documenti_recuperati_pertinenti}}{\text{totale_documenti_pertinenti}} = \frac{\text{documento_recuperato}}{\text{pertinenza_ricerca}}$$

Ogni ricerca sarà caratterizzata dai propri parametri:

- **information retrieval**

$$precision = \frac{\text{documento_recuperato_1}}{\text{ricerca_1}} = \frac{23}{25} = 0,92$$

$$recall = \frac{\text{documento_recuperato_1}}{\text{pertinenza_ricerca_1}} = \frac{23}{26} = 0,88$$

- **XML document**

$$precision = \frac{\text{documento_recuperato_2}}{\text{ricerca_2}} = \frac{5}{9} = 0,56$$

$$recall = \frac{\text{documento_recuperato_2}}{\text{pertinenza_ricerca_2}} = \frac{5}{9} = 0,56$$

- **Link generation**

$$precision = \frac{\text{documento_recuperato_3}}{\text{ricerca_3}} = \frac{5}{6} = 0,83$$

$$recall = \frac{\text{documento_recuperato_3}}{\text{pertinenza_ricerca_3}} = \frac{5}{8} = 0,63$$

Come si può evincere dai risultati precedenti, l'algoritmo ha una buona abilità nel recuperare solamente i documenti rilevanti per l'utente, mentre la sua capacità di recuperare tutti i documenti effettivamente importanti per la ricerca dell'utente è inferiore. In ogni caso, si può affermare che questo algoritmo risponde bene al fabbisogno informativo dell'utente e che la persona che utilizzerà questo sistema per il reperimento di informazioni riuscirà con un considerevole margine di successo a trovare i documenti desiderati.

4. Conclusioni

L'obiettivo di questa seconda parte dello stage era l'utilizzo del thesaurus costruito precedentemente al fine di implementare l'algoritmo di prossimità semantica di Paice-Ramirez. Una delle difficoltà più importanti riscontrate è stata quella di dare una corretta interpretazione alla formula generale. La formula presentata da Paice-Ramirez, infatti, lascia un notevole spazio all'interpretazione personale delle varie parti e questo può creare ambiguità nelle implementazioni.

Un altro aspetto cruciale è stato quello dell'ottimizzazione del codice per ottenere le massime prestazioni computazionali possibili. In questo senso è stata di notevole aiuto la memorizzazione dei testi normalizzati in cui viene effettuata la ricerca in un apposito file su disco, eliminando, così, il tempo necessario alla fase di normalizzazione.

Dai risultati raccolti si può affermare che l'obiettivo generale di questo stage è stato raggiunto in modo più che soddisfacente. I tempi di calcolo sono limitatamente contenuti, l'algoritmo si adatta ai più moderni calcolatori sfruttandone a pieno le potenzialità e i dati di output sono coerenti con le ricerche dell'utente.

L'esperienza di questo stage mi ha permesso di approfondire le tematiche trattate durante la mia carriera accademica che più mi interessano, come la programmazione Java e l'ambiente *multithread*. Ma più di ogni altra cosa ho avuto modo di analizzare gli aspetti cruciali dell'*information retrieval*. Le numerose problematiche emerse durante la scrittura del codice per la realizzazione del thesaurus e per l'algoritmo di Paice-Ramirez mi hanno anche permesso di studiare aspetti nuovi della programmazione, dell'*information retrieval*, dell'ottimizzazione del codice e della configurazione hardware dei calcolatori.

Il codice può, comunque, essere migliorato tentando un'integrazione con qualche sistema di indicizzazione delle ricerche per ridurre drasticamente i tempi di calcolo necessari per le ricerche. Per ottimizzare l'utilizzo dei database, invece, è pensabile l'utilizzo di DBMS più sofisticati, come MySQL oppure SQLserver. In questo modo si ridurrebbero i tempi di accesso in lettura e scrittura sulla base di dati. Un'ulteriore possibile miglioria apportabile sarebbe quella di inserire tutti i parametri variabili degli algoritmi (come il numero di processi concorrenti, i livelli di significatività minimi e massimi, etc.) in appositi file di configurazione XML, in modo da poterli variare senza dovere ricompilare le classi Java.

Istruzioni per l'uso

Il codice è stato realizzato utilizzando la piattaforma Java JDK1.5.0_06 della Sun. Si possono individuare tre categorie di appartenenza per le classi che costituiscono questo stage:

Thesaurus:

- CreaThesaurus.java
- Monitor.java
- ThreadPrincipale.java
- Normalize.java
- Thesaurus.java
- WordCount.java
- ThesaurusDatabaseTerm.java

Paice-Ramirez:

- Calcola.java
- SematichProximity.java
- PaiceRamirez.java
- Ordina.java

Comuni:

- PorterStemmer.java

I programmi per la costruzione del thesaurus e per il calcolo della prossimità semantica utilizzano due database principali: in uno sono contenuti i dati della grammatica della lingua utilizzata dal processo di normalizzazione, mentre nel secondo sono memorizzati i dati del thesaurus. Il codice per l'analisi della somiglianza semantica si occuperà della scrittura del database, mentre il codice dell'algoritmo di Paice-Ramirez lo utilizzerà per ricavare i suddetti valori.

Il codice ha dei parametri personalizzabili in base alle esigenze dell'utente:

- classe Monitor:
private final int maxmainthread → Rappresenta il numero di processi concorrenti per la costruzione del thesaurus. Il numero di *thread* non dovrebbe essere superiore al doppio del numero di operazioni parallele computabili dal calcolatore.
- classe MainThread
private final int maxbuffer → Specifica il numero di frasi da memorizzare per l'interscambio tra i processi di normalizzazione e costruzione del thesaurus. Ogni frase normalizzata viene inserita in questo buffer e prelevata dal processo di costruzione del thesaurus che la analizza per estrapolare le parole.
- classe Thesaurus
private final int minimaoccorrenza → Specifica il minimo numero di occorrenze che una parola deve avere in un documento perchè sia presa in considerazione nell'analisi.
- classe ThesaurusDatabaseTerm
private final Double sogliaminima → Specifica la soglia minima di somiglianza, sotto la quale la coppia considerata non viene inserita nel database
private final int minimalunghezza → Specifica la lunghezza minima delle parole, al di sotto della quale la coppia considerata viene scartata.

- classe SemanticProximity
private final int maxthread → Rappresenta il numero di processi concorrenti per la costruzione del thesaurus. Il numero di *thread* dovrebbe essere non superiore al doppio del numero di operazioni parallele computabili dal calcolatore.
private String cartella → specifica la cartella in cui il codice ricercherà i documenti su cui effettuare la ricerca di prossimità semantica.

Il codice ricerca i database e i testi in determinate cartelle. Per quanto riguarda la costruzione del database, la cartella contenente i documenti campione viene richiesta all'utente durante la fase di inizializzazione. Per il codice di Paice-Ramirez, invece, i testi su cui viene effettuata la ricerca devono trovarsi nella sottocartella "testi". I database devono essere collocati nella sottocartella "database" e devono avere privilegi di lettura e scrittura.

Per via della grande quantità di memoria necessaria nella realizzazione del thesaurus si raccomanda, al fine di evitare errori di out of heap memory, di eseguire il codice con i seguenti comandi:

```
java -Xms128m -Xmx512m CreaThesaurus
e
java Calcola
```

Il codice precarica i dati della grammatica della lingua corrente prima di proseguire con l'esecuzione. Questa procedura potrebbe richiedere alcuni minuti e l'avanzamento è mostrato in percentuale.

```
#### B U I L D I N G   T H E S A U R U S ####
```

```
Inserire il percorso completo della cartella che contiene i documenti
(es. c:\nomecartella\nomesottocartella\):
```

```
.\testi
```

```
Monitor is preloading data... (this may take few minutes)
```

```
Monitor is preloading data: 0%
```

```
Monitor is preloading data: 10%
```

```
Monitor is preloading data: 20%
```

```
Monitor is preloading data: 30%
```

```
Monitor is preloading data: 40%
```

```
Monitor is preloading data: 50%
```

```
Monitor is preloading data: 60%
```

L'algoritmo per il calcolo della prossimità semantica provvede anche alla scrittura di alcuni file nella medesima cartella in cui sono collocati i documenti campione per il confronto. Per ogni documento ".txt" presente, vengono salvati due file:

- ".normalized" → contiene la versione già normalizzata del documento. Questa scelta è stata fatta per ottimizzare i tempi di computazione, eliminando l'attesa derivante dalle procedura di normalizzazione
- ".lastsearch" → contiene i dati dettagliati dell'ultima ricerca effettuata sul documento nominato prima dell'estensione. I dati sono salvati in formato importabile da Excel. Ogni colonna è separata mediante un punto e virgola.

Codice

>> CreaThesaurus.java

```
public class CreaThesaurus
{
    public static void main(String[] args)
    {
        System.out.println("\n\t\t\t\t\t#### B U I L D I N G   T H E S A U R U S   ####\n");
        String cartella = "";
        System.out.println("\n\n");
        System.out.println("Inserire il percorso completo della cartella che contiene i documenti\n(es.
c:\\\nomecartella\\nomesottocartella\\):");
        cartella = SavitchIn.readLine();
        Monitor monitor = new Monitor("Monitor",cartella);
    }
}
```

>> Monitor.java

```
import java.io.*;
import java.util.*;
import java.util.List.*;
import java.sql.*;

public class Monitor implements Runnable
{
    public static Thread thread;

    private java.util.List<String> vocabolario;
    private java.util.List<String> trasformazioniorigine;
    private java.util.List<String> trasformazionidestinazione;
    private String cartella;

    private java.util.List<String> terminithesaurus;
    private int processirimanenti;
    private int numerodocumenti;

    private final int maxmainthread = 6;
    private int mainthreadattivi;

    public Monitor(String nome, String cartella)
    {
        thread = new Thread(this,nome);
        vocabolario = new ArrayList<String>();
        trasformazioniorigine = new ArrayList<String>();
        trasformazionidestinazione = new ArrayList<String>();
        processirimanenti = 0;
        terminithesaurus = new ArrayList<String>();
        this.cartella = cartella;
        this.mainthreadattivi = 0;
        thread.start();
    }

    public void run()
    {
        //Controllo se è stata specificata una cartella
        if(cartella=="")
        {
            System.out.println("ERROR: no folder has been entered.");
            return;
        }
        //Controllo se la cartella specificata esiste.
        File dir = new File(cartella);
        if(!dir.isDirectory())
        {
            System.out.println("ERROR: not a valid folder.");
            return;
        }
        //Imposto un filtro per la lettura dei file che il programma analizzerà. Per motivi di compatibilità
        //si possono leggere solamente file con estensione TXT, ovvero testo semplice
        FilenameFilter filter = new FilenameFilter()
        {
            public boolean accept(File dir, String name) {
                return name.endsWith(".txt");
            }
        };
        //Prelevo la lista dei file, applicando il filtraggio.
        String[] listafile = dir.list(filter);
        if(listafile.length==0)
        {
            System.out.println("ERROR: no file found.");
            return;
        }
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
}
//Lancio il precaricamento dei dati prima della creazione dei thread
preloadData();
for(int i=0;i<listafile.length;i++)
{
    //System.out.println(dir.getAbsolutePath() + "\\\" + listafile[i]);
    try{
        new MainThread(this,"Main Thread", i, dir.getAbsolutePath() + "\\\" + listafile[i]);
        processirimanenti++;
    }catch(Exception e){
        mainThreadEnd("Main Thread"+ i);
        processirimanenti--;
    }
}
//Lancio il processo che si occupa di scrivere nel database le coppie di termini che sono stati trovati
ThesaurusDatabaseTerm tdt = new ThesaurusDatabaseTerm("ThesaurusDatabaseTerm",this);
//Attendo che tutti i processi terminino per visualizzare un messaggio di attesa per la memorizzazione
//dei valori calcolati nel database.
while(processirimanenti>0)
    try{
        wait();
    }catch(Exception e){
    }
}
System.out.println("\n\nWait a few minutes...the program is writing results in database...");
//createWindow();

//System.out.println("Termino il main e lascio il controllo completo al thread principale.");
}

synchronized public void mainThreadEnd(String nomeprocesso)
{
    //Decremento il numero di processi in esecuzione
    mainthreadattivi --;
    notifyAll();
}
synchronized public void mainThreadStart(String nomeprocesso)
{
    //Se il numero massimo di processi in esecuzione è stato raggiunto, allora pongo in WAIT il processo
    //chiamante, fino a quando non termina.
    while(mainthreadattivi>=maxmainthread)
        try
        {
            wait();
        }catch(Exception e){}

    //Aggiungo il processo attuale allo stato di "esecuzione" incrementando il contatore di processi
    //attivi. Risveglio gli altri processi in coda nel caso qualche condizione sia verificata
    mainthreadattivi++;
    notifyAll();
}

public boolean searchWord(String parola)
{
    return vocabolario.contains(parola.trim());
}
public String trasformazione(String parola)
{
    if(trasformazioniorigine.indexOf(parola.trim())==-1)
        return parola;
    return trasformazioneidestinazione.get(trasformazioniorigine.indexOf(parola.trim()));
}

//Precarico i dati da database (più veloce in lettura). Li carico in memoria, altrimenti incorro in
//problemi di sovraccarico di database. Inoltre la lettura da memoria risulta essere molto più veloce.
private void preloadData()
{
    System.out.println(thread.getName() + " is preloading data... (this may take a few minutes)");
    //Apro la connessione al database.
    Connection connessione = openDatabaseConnection("GrammaticaInglese.mdb");
    try{
        //Creo uno Statement per eseguire l'operazione sul database e imposto la stringa
        //SQL per l'interrogazione al database. Impongo l'apertura della tabella alla tipologia scrollabile
        //in entrambe le direzioni e di sola lettura.
        Statement operazione =
connessione.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
        String SQL = "SELECT COUNT(Parola) FROM Vocabolario";
        ResultSet tabella = operazione.executeQuery(SQL);
        tabella.first();
        int numeroparole = tabella.getInt(1);
        SQL = "SELECT Parola FROM Vocabolario";
        //Eseguo la query SQL e memorizzo il risultato
        tabella = operazione.executeQuery(SQL);
        //Controllo se la prima riga della tabella esiste oppure no, per controllare se la parola
        //specificata è presente o meno nel database
        int avanzamento = -100;
        while(tabella.next())
        {

```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
vocabolario.add(tabella.getString("Parola"));
    if(((int)(100*((double)tabella.getRow()/((double)numeroparole)))>=(avanzamento+10) &&
avanzamento%10==0)
    {
        System.out.println(thread.getName() + " is preloading data: "+
((int)(100*((double)tabella.getRow()/((double)numeroparole)))+"%");
        avanzamento = (int)(100*((double)tabella.getRow()/((double)numeroparole)));
    }
}

SQL = "SELECT * FROM Trasformazioni";
//Eseguo la query SQL e memorizzo il risultato
tabella = operazione.executeQuery(SQL);
//Controllo se la prima riga della tabella esiste oppure no, per controllare se la parola
//specificata è presente o meno nel database
String destinazione = "";
while(tabella.next())
{
    trasformazioniorigine.add(tabella.getString("Parola"));
    destinazione = tabella.getString("Trasformazione");
    if(destinazione==null)
        trasformazionidestinazione.add("");
    else
        trasformazionidestinazione.add(tabella.getString("Trasformazione"));
}
} catch (Exception e){
    System.out.println("Si è verificato un errore durante l'esecuzione di una interrogazione: " + e);
}
//Chiudo la connessione al database
closeDatabaseConnection(connessione);
System.out.println("\n"+ thread.getName() + " preload complete!");
}


synchronized void stampaVideo(String stringa)
{
    System.out.println(stringa);
}

//Manipolazione database
private Connection openDatabaseConnection(String database)
{
    Connection connessione = null;
    try{
        //Carico i driver JDBC della sun
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        //Carico il database della semantica della lingua inglese
        File file = new File(".\\database\\"+ database);
        //Creo la stringa di connessione al database Access sopra indicato. Il nome del database
        //è recuperato dal percorso completo del file
        String stringaconnessione = "jdbc:odbc:driver={Microsoft Access Driver (*.mdb)};DBQ="+
file.getAbsolutePath().trim();
        //Instauro la connessione con il database usando la stringa di connessione
        connessione = DriverManager.getConnection(stringaconnessione);
    } catch (Exception e){
        System.out.println("Si è verificato un errore durante il tentativo di aprire il database: " + e);
    }
    //Controllo se la connessione è stata effettivamente creata correttamente.
    if(connessione==null)
        System.out.println("Si è verificato un errore irreversibile durante l'instaurazione della
connessione con il database.");
    //Ritorno la connessione creata
    return connessione;
}
private void closeDatabaseConnection(Connection connessione)
{
    try{
        //Tengo la chiusura della connessione al database instaurata dal metodo openDatabaseConnection()
        connessione.close();
    } catch (Exception e){
        System.out.println("Si è verificato un errore durante il tentativo di chiudere il database: " + e);
    }
}

//Metodi per la scrittura di dati su database
synchronized public void aggiungiTermine(String termine, String nomeprocesso)
{
    //System.out.println("--> MultiThread Warning: "+ nomeprocesso + " starting add term to buffer.");
    terminithesaurus.add(termine);
    //System.out.println("--> MultiThread Warning: "+ nomeprocesso + " added term to buffer.");
    notifyAll();
}
synchronized public String prelevaTermine(String nomeprocesso)
{
    //System.out.println("--> MultiThread Warning: "+ nomeprocesso + " starting remove term from buffer.");
    while(terminithesaurus.size()==0)
        try{
            //System.out.println("--> MultiThread Warning: "+ nomeprocesso + " no data in buffer.");
            wait();
        }
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
        }catch(InterruptedException e){
            System.out.println("--> MuliThread Warning: wait interrupted");
        }
        String termine = terminithesaurus.remove(0);
        //System.out.println("--> MuliThread Warning: "+ nomeprocesso +" removed term from buffer.");
        notifyAll();
        return termine;
    }
    synchronized public boolean terminiFiniti(boolean ... terminifiniti)
    {
        if(terminifiniti.length>0)
        {
            processirimanenti --;
        }
        notifyAll();
        return processirimanenti==0 && (terminithesaurus.size()==0);
    }
}
```

>>  MainThread.java

```
import java.io.*;
import java.util.*;
import java.util.List.*;
import java.sql.*;

public class MainThread implements Runnable
{
    public Thread thread;
    private int numerothread;
    public Monitor mt;

    private int frasi;
    private int parole;
    private int caratterisenzaspazi;
    private int caratteri;
    private boolean contatorisettati;

    private List <String> frasinormalizzate;
    private boolean frasiterminate;
    private final int maxbuffer = 100;

    private String filedaleggere;
    private String testodocumento;

    private List<String> vocabolario;
    private List<String> trasformazioniorigine;
    private List<String> trasformazionidestinazione;

    public MainThread(Monitor mt, String nome, int numerothread,String filedaleggere)
    {
        thread = new Thread(this,nome+"_"+numerothread);
        this.mt = mt;
        this.numerothread = numerothread;
        frasiterminate = false;
        contatorisettati = false;
        testodocumento = "";
        frasinormalizzate = new ArrayList<String>();
        this.filedaleggere = filedaleggere;
        thread.start();
    }

    public void run()
    {
        //Richiedo la messa in esecuzione del processo. Se ci sono troppi processi in esecuzione
        //viene messo in WAIT
        mt.mainThreadStart(thread.getName());
        //Tempo di inizio
        long tempo = System.nanoTime();
        //Leggo il testo da analizzare da file. La procedura effettua anche una serie di operazioni
        //preliminari sull'eliminazione di caratteri non ammessi
        leggiTesto();
        //Lancio i thread principali del programma come il normalizzatore ed il thesaurus
        WordCount contatoreparole;
        Normalize normalizzatore;
        Thesaurus thesaurus;
        try{
            contatoreparole = new WordCount("WordCounter_"+ numerothread,this);
        }catch(Exception e){
            mt.mainThreadEnd(thread.getName());
            return;
        }
        try{
            normalizzatore = new Normalize("Normalizer_"+ numerothread,this);
        }catch (Exception e){
            mt.mainThreadEnd(thread.getName());
        }
    }
}
```



```
        return;
    }
    try{
        thesaurus = new Thesaurus("ThesaurusBuilder_"+ numerothread, this);
    } catch (Exception e){
        mt.mainThreadEnd(thread.getName());
        return;
    }
    //Attendo il completamento del contaparole per mostrare su schermo le statistiche documento
    try{
        contatoreparole.thread.join();
    } catch (InterruptedException e){
        System.out.println("--> MultiThread Warning: "+ thread.getName() +"Interruzione");
    }
    //Rilancio il contaparole...questa volta deve attendere il testo del normalizzatore!

    contatoreparole = new WordCount("WordCounter_"+ numerothread, this);
    try{
        contatoreparole.thread.join();
    } catch (InterruptedException e){
        System.out.println("--> MultiThread Warning: "+ thread.getName() +"Interruzione");
    }
    //Attendo il termine degli altri thread
    try{
        normalizzatore.thread.join();
    } catch (InterruptedException e){
        System.out.println("--> MultiThread Warning: "+ thread.getName() +"Interruzione");
    }
    //Attendo il termine di thesaurus
    try{
        thesaurus.thread.join();
    } catch (InterruptedException e){
        System.out.println("--> MultiThread Warning: "+ thread.getName() +"Interruzione");
    }

    //Calcolo il tempo di esecuzione del programma e lo stampo a monitor
    tempo = System.nanoTime()-tempo;
    String mostratempo = tempo/(1000*1000) +" millisecondi --> "+ tempo/(1000*1000*1000) +" secondi";
    if((tempo/(1000*1000*1000))>60)
        mostratempo += " --> "+ ((long)tempo/(long)(1000*1000))/(long)(1000*60) +" minuti";
    System.out.println(""+ thread.getName() +" on "+ filedaleggere +" compute time: "+ mostratempo +".");
    mt.mainThreadEnd(thread.getName());
}

//Metodo per la lettura da file del documento e ottimizzazione dei caratteri in lettura
private void leggiTesto()
{
    String testo = "";
    try{
        //Apro il file
        BufferedReader in = new BufferedReader(new FileReader(filedaleggere));
        //Leggo una riga del file.
        String riga = in.readLine();
        //Continuo a leggere nuove righe dal file fino a quando il documento non è temrinato
        while(riga != null)
        {
            testo += riga;
            riga = in.readLine();
        }
        //Chiudo il file
        in.close();
    } catch (IOException e){
        System.out.println("Error reading file: "+ e);
    }
    //Riduco tutto minuscolo per poterlo paragonare con i dati nel database.
    testo = testo.toLowerCase();
    //Rimuovo gli spazi doppi.
    testo = testo.replaceAll("\\s+", " ");
    //Sostituisco il ? e ! con un . normale
    testo = testo.replace("?", ".").replace("!", ".");
    //Aggiungo il testo per gli altri processi
    aggiungiTesto(testo, thread.getName());
}

//Metodi di comunicazione tra Normalizzatore e Thesaurus
//Si riferiscono allo scambio di frasi analizzate/da analizzare
synchronized public void aggiungiFrase(String frase, String nomeprocesso)
{
    frasinormalizzate.add(frase);
    notifyAll();
}
synchronized public String prelevaFrase(String nomeprocesso)
{
    while(frasinormalizzate.size()==0)
        try{
            if(!frasiterminate)
                wait();
            else

```

```
        {
            notifyAll();
            return null;
        }
    } catch (InterruptedException e) {
        System.out.println("--> MuliThread Warning: wait interrupted");
    }
}
String frase = frasinormalizzate.remove(0);
notifyAll();
return frase;
}
synchronized public boolean frasiTerminate(boolean ... finito)
{
    if(finito.length>0)
    {
        frasiTerminate = finito[0];
        notifyAll();
        return false;
    }
    notifyAll();
    return frasiTerminate && (frasinormalizzate.size()==0);
}

//Metodi per la lettura/scrittura del testo del documento
synchronized public String prelevaTesto(String nomeprocesso, boolean preserva)
{
    while(testodocumento=="")
    {
        try{
            wait();
        } catch (InterruptedException e){
            System.out.println("--> MuliThread Warning: wait interrupted");
        }
    }
    String ritorno = testodocumento;
    if(!preserva)
        testodocumento = "";
    notifyAll();
    return ritorno;
}
synchronized public void aggiungiTesto(String testo, String nomeprocesso)
{
    while(testodocumento!="")
    {
        try{
            wait();
        } catch (InterruptedException e){
            System.out.println("--> MuliThread Warning: wait interrupted");
        }
    }
    testodocumento = testo;
    notifyAll();
}

//Metodi per le statistiche del documento
synchronized public void setContatori(int f,int p,int c,int css)
{
    frasi = f;
    parole = p;
    caratteri = c;
    caratterisenzaspazi = css;
    contatorisettati = true;
    notifyAll();
}
synchronized public void setContatoriSettati(boolean valore)
{
    contatorisettati = valore;
    notifyAll();
}
synchronized public int getFrase(String nomeprocesso)
{
    while(!contatorisettati)
    {
        try{
            wait();
        } catch (InterruptedException e){
            System.out.println("--> MuliThread Warning: wait interrupted");
        }
    }
    notifyAll();
    return frasi;
}
synchronized public int getParole(String nomeprocesso)
{
    while(!contatorisettati)
    {
        try{
            wait();
        } catch (InterruptedException e){
            System.out.println("--> MuliThread Warning: wait interrupted");
        }
    }
    notifyAll();
    return parole;
}
synchronized public int getCaratteri(String nomeprocesso)
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
{
    while(!contatorisettati)
        try{
            wait();
        }catch(InterruptedException e){
            System.out.println("--> MutiThread Warning: wait interrupted");
        }
        notifyAll();
        return caratteri;
}
synchronized public int getCaratteriSenzaSpazi(String nomeprocesso)
{
    while(!contatorisettati)
        try{
            wait();
        }catch(InterruptedException e){
            System.out.println("--> MutiThread Warning: wait interrupted");
        }
        notifyAll();
        return caratterisenzaspazi;
}
}
```

>>  **Normalize.java**

```
import java.sql.*;
import java.util.*;
import java.io.*;
import java.util.regex.*;

public class Normalize implements Runnable
{
    public Thread thread;
    private Mainthread mt;
    private String testo;
    private String[] frase;

    public Normalize(String nome, Mainthread mt)
    {
        this.mt = mt;
        thread = new Thread(this,nome);
        this.testo = "";
        thread.start();
    }

    public void run()
    {
        testo = mt.prelevaTesto(thread.getName(),false);
        findSentences();
        replaceWords();
    }

    private void findSentences()
    {
        //Se testo è vuoto esco dalla funzione.
        if(testo==null)
            return;
        //Sostituisco i punti interrogativi ed esclamativi con i punti semplici, così da potere più facilmente
        //dividere il documento in frasi.
        //Predispongo il pattern per la separazione della Stringa
        Pattern pat = Pattern.compile("[. ]");
        //Scorro tutto il testo per analizzare eventuali caratteri non alfabetici da eliminare.
        Character carattere = null;
        char[] vettoretesto = testo.toCharArray();
        for(int contatore=0;contatore<vettoretesto.length;contatore++)
        {
            carattere = new Character(vettoretesto[contatore]);
            if(!Character.isLetter(carattere) && carattere!='.')
                vettoretesto[contatore] = ' ';
        }
        //Ricompongo il vettore di caratteri in una Stringa
        testo = new String(vettoretesto);
        //Utilizzo una regular expression per rimuovere gli spazi dal doppi dal documento.
        testo = testo.replaceAll("\\s+", " ");
        //Divido le frasi basandomi sul pattern delle regular String precedentemente dichiarato
        frase = new String[pat.split(testo).length];
        frase = pat.split(testo);
        //Elimino gli spazi in eccesso all'inizio e alla fine di una frase.
        for(int contatore=0;contatore<frase.length;contatore++)
            frase[contatore] = frase[contatore].trim();
    }

    public void replaceWords()
    {
        //Inizializzo il contatore di avanzamento
        int avanzamento = -100;
        for(int contatore=0;contatore<frase.length;contatore++)
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
{
    //Mostro un indicato di avanzamento dell'elaborazione.
    if(((int)(100*((double)(contatore+1)/(double)frase.length))>=(avanzamento+10) &&
avanzamento%10==0)
    {
        avanzamento = (int)(100*((double)(contatore+1)/(double)frase.length));
    }
    //Suddivido la frase corrente in parole. In questo modo posso analizzarle singolarmente
    //e mantengo traccia della frase.
    String[] parola = new String[frase[contatore].split(" ").length];
    parola = frase[contatore].split(" ");
    //Resetto la frase corrente. La ricostruisco con le parole sostituite.
    frase[contatore] = "";
    String parolatemp = null;
    //Se la frase è composta da una sola parola non la analizzo neanche, in quanto non è rilevante dal
punto di vista
    //del thesaurus (non è possibile calcolare significativamente la ricorrenza della parola nelle
frasi)
    for(int contatore2=0;contatore2<parola.length;contatore2++)
    {
        //Ignoro le parole con lunghezza minore di 2
        if(parola[contatore2].length()<2)
            continue;
        //Controllo se è presente qualche trasformazione della parola in questione nel database
        //delle trasformazioni
        parolatemp = mt.mt.trasformazione(parola[contatore2]);
        if(parolatemp=="")
        {
            parola[contatore2] = "";
            continue;
        }
        //Per prima cosa sulla parola elimino eventuali suffissi per renderla il più generico
possibile.
        String parolastemmata = "";
        try{
            PorterStemmer ps= new PorterStemmer();
            ps.add(parolatemp.trim().toCharArray(),parolatemp.trim().length());
            ps.stem();
            parolastemmata = ps.toString();
        }catch(Exception e){}
        //Controllo se la parola è presente nel dizionario
        if(mt.mt.searchWord(parolastemmata.trim()))
            parolatemp = parolastemmata.trim();
        //Aggiungo la parola alla lista delle parole
        frase[contatore] += parolatemp.trim() + " ";
        parola[contatore2] = parolatemp.trim();
    }
    mt.aggiungifrase(frase[contatore],thread.getName());
}
mt.fraseTerminate(true);

String testonuovo = "";
for(int i=0;i<frase.length;i++)
{
    testonuovo += frase[i].trim() + ".";
}
mt.aggiungiTesto(testonuovo,thread.getName());
}
}
```

>>  Thesaurus.java

```
import java.sql.*;
import java.util.*;
import java.io.*;
import java.util.regex.*;

public class Thesaurus implements Runnable
{
    public Thread thread;
    private MainThread mt;
    private List<String> tabellahash;
    private byte[][] vettoreoccorrenze;
    private String testo;
    private String[] frase;
    private final int minimaoccorrenza = 4;

    public Thesaurus(String nome, MainThread mt)
    {
        this.mt = mt;
        thread = new Thread(this, nome);
        this.testo = "";
        thread.start();
    }

    public void run()
    {
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
//Dimensiono vettore delle occorrenze come il numero di parole totali del documento non normalizzato e
del
//numero di frasi del documento originale. Quando sarà completata la creazione della tabella hash
troncheo
//gli elementi inutilizzati del vettore.
vettoreoccorrenze = new byte[mt.getParole(thread.getName())[mt.getFrase(thread.getName())];
frase = new String[mt.getFrase(thread.getName())];
tabellahash = new ArrayList<String>(mt.getParole(thread.getName()));
//Creo la tabella hash delle parole significative del documento.
createHash();
//Richiamo il metodo del coseno per la similitudine tra le parole
cosineMethod();
}

private void createHash()
{
//Dimensiono un vettore che conterrà tutte le parole della frase corrente.
String[] parola = new String[0];
int numerofrase = 0;
//Fino a che le frasi analizzate dal normalizzatore non sono terminate continuo ad
//eseguire il codice per riempire la tabella hash.
String frasecorrente = mt.prelevaFrase(thread.getName());
while(frasecorrente!=null)
{
//Prelevo la prima frase disponibile già normalizzata'
try{
frase[numerofrase] = frasecorrente;
}catch(Exception e){
frasecorrente = mt.prelevaFrase(thread.getName());
continue;
}
//Separo la frase in parole.
parola = frase[numerofrase].split(" ");
//Per ogni parola controllo se è già presente nella tabella hash, altrimenti
//la aggiungo. Se è già presente aggiorno il vettore delle occorrenze!
for(int contatore=0;contatore<parola.length;contatore++)
if(!tabellahash.contains(parola[contatore]))
{
tabellahash.add(parola[contatore]);
vettoreoccorrenze[tabellahash.indexOf(parola[contatore])][numerofrase] = 1;
}else{
vettoreoccorrenze[tabellahash.indexOf(parola[contatore])][numerofrase] += 1;
}
numerofrase++;
frasecorrente = mt.prelevaFrase(thread.getName());
}
//Rimpicciolisco il vettore in base agli effettivi elementi che ha.
byte[][] nuovovettore = new byte[tabellahash.size()][numerofrase+1];
for(int contatore=0;contatore<tabellahash.size();contatore++)
System.arraycopy(vettoreoccorrenze[contatore],0,nuovovettore[contatore],0,numerofrase);

vettoreoccorrenze = nuovovettore;
}

private void cosineMethod()
{
//Variabili per il metodo del coseno
double numeratore = 0.0;
double denominatore1 = 0.0;
double denominatore2 = 0.0;
int occorrenzetotali = 0;
int avanzamento = -100;
//Scandisco le parole significative
for(int contatore1=0;contatore1<tabellahash.size()-1;contatore1++)
{
//Mostro un indicato di avanzamento dell'elaborazione.
//Verifico che la parola presa in considerazione sia presente più di n volte nel
//documento, controllando il numero totale di occorrenze!
occorrenzetotali = 0;
for(int contatore3=0;contatore3<frase.length;contatore3++)
occorrenzetotali += vettoreoccorrenze[contatore1][contatore3];
//Se è presente meno di n volte, allora passo alla parola successiva!
if(occorrenzetotali<minimaoccorrenza)
continue;
for(int contatore2=contatore1+1;contatore2<tabellahash.size();contatore2++)
{
//Non considero il confronto tra parole uguali!
if(contatore2==contatore1)
continue;
//Verifico che la parola presa in considerazione sia presente più di n volte nel
//documento, controllando il numero totale di occorrenze!
occorrenzetotali = 0;
for(int contatore3=0;contatore3<frase.length;contatore3++)
occorrenzetotali += vettoreoccorrenze[contatore2][contatore3];
//Se è presente meno di n volte, allora passo alla parola successiva!
if(occorrenzetotali<minimaoccorrenza)
continue;
//Resetto le variabili per il metodo del coseno!
}
}
}
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
numeratore = 0.0;
denominatore1 = 0.0;
denominatore2 = 0.0;
//Calcolo il metodo del coseno
for(int contatore3=0;contatore3<frase.length;contatore3++)
{
    numeratore +=
vettoreoccorrenze[contatore1][contatore3]*vettoreoccorrenze[contatore2][contatore3];
    denominatore1 += Math.pow(vettoreoccorrenze[contatore1][contatore3],2.0);
    denominatore2 += Math.pow(vettoreoccorrenze[contatore2][contatore3],2.0);
}
mt.mt.aggiungiTermine(tabellahash.get(contatore1) + "#" + tabellahash.get(contatore2) + "#" +
(numeratore/Math.sqrt(denominatore1*denominatore2)),thread.getName());
}
}
mt.mt.terminiFiniti(true);
}
}
```

>>  WordCount.java

```
import java.util.regex.*;
```

```
public class WordCount implements Runnable
```

```
{
    Thread thread;
    private MainThread mt;
    private String testo;

    public WordCount (String nome, MainThread mt)
    {
        this.testo = "";
        this.mt = mt;
        thread = new Thread(this, nome);
        thread.start();
    }

    public void run()
    {
        //Prelevo il testo su cui effettuare i conteggi senza eliminarlo.
        testo = mt.prelevaTesto(thread.getName(),true);
        //Imposto i contatori a false, in modo che le statistiche non risultino aggiornate.
        mt.setContatoriSettati(false);
        //Imposto le variabili utilizzate
        int frasi = 0;
        int parole = 0;
        int caratteri = 0;
        int caratterisenzaspazi = 0;
        //Conto le frasi del documento spezzandolo sui caratteri !.?
        Pattern pat = Pattern.compile("[.!?\n]");
        frasi = pat.split(testo).length;
        //Conto le parole spezzando il documento sui tipici caratteri separato delle parole: spazio ,; etc...
        testo = testo.replace(" ", " ").replace(";", " ").replace(":", " ").replace("!", " ")
        ).replace("?", " ").replace("\n", " ");
        //Elimino gli spazi doppi!
        testo = testo.replaceAll("\\s+", " ");

        pat = Pattern.compile(" ");
        parole = pat.split(testo).length;
        //Conto i caratteri totali (esclusi gli spazi doppi)
        caratteri = testo.length();
        //Elimino gli spazi ed effettuo nuovamente il conteggio dei caratteri (senza spazi)
        testo = testo.replace(" ", "");
        caratterisenzaspazi = testo.length();
        //Imposto le statistiche e le rendo disponibili per gli altri processi.
        mt.setContatori(frasi,parole,caratteri,caracterisenzaspazi);
    }
}
```

>>  ThesaurusDatabaseTerm.java

```
import java.util.regex.*;
```

```
import java.sql.*;
```

```
import java.util.*;
```

```
import java.io.*;
```

```
public class ThesaurusDatabaseTerm implements Runnable
```

```
{
    Thread thread;
    private Monitor mt;

    //Soglia minima sopra cui memorizzare i dati nel database.
    private final Double sogliaminima = 0.3;
    //Soglia minima di lunghezza parola, sotto la quale viene ignorata
    private final int minimalunghezza = 2;
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
public ThesaurusDatabaseTerm (String nome, Monitor mt)
{
    this.mt = mt;
    thread = new Thread(this, nome);
    thread.start();
}

public void run()
{
    //La variabile voci contiene i dati da inserire nel database.
    String[] voci = new String[3];
    //Pattern è predisposto per spezzare la stringa al carattere #
    Pattern pat = Pattern.compile("#");
    Connection connessione = openDatabaseConnection("Thesaurus.mdb");
    Statement operazione = null;
    try{
        operazione =
connessione.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
    }catch(Exception e){
        System.out.println("Si è verificato un errore durante l'interrogazione alla tabella Trasformazioni:
"+e);
    }
    while(!mt.terminiFiniti())
    {
        //Spezzo la stringa passata come parametro per ottenere le tre voci da memorizzare nel database
        voci = pat.split(mt.prelevaTermine(thread.getName()));
        //Se il valore di somiglianza è minore di 0.2 non memorizzo i valori nel database in quanto sono
poco simili
        if(Double.parseDouble(voci[2])<=sogliaminima)
            continue;
        if((voci[0].length()<=minimalunghezza) || (voci[1].length()<=minimalunghezza))
            continue;
        //Tento la connessione con il database
        try{
no. Nel primo
            //Eseguo una SELECT per verificare se la coppia di termini è già presente nel database oppure
            //caso devo fare la media dei valori considerati, altrimenti devo inserirli ex-novo.
            String SQL = "SELECT * FROM SomiglianzaSemantica WHERE Parola1='"+
voci[0].trim().replace("'", "'') +"' AND Parola2='"+ voci[1].trim().replace("'", "'') +"'";
            ResultSet tabella = operazione.executeQuery(SQL);
            String SQLupdate = "";
            //Controllo se la tabella restituita dall'interrogazione presenta una riga. (può esserci al
massimo 1 riga
            //in quanto le due parole sono chiave primaria!)
            if(tabella.first())
            {
                //Prelevo il valore di somiglianza precedente
                double valoreprecedente = tabella.getDouble("Somiglianza");
                //Prelevo il numero di occorrenze trovate precedentemnte
                int numeromodifiche = tabella.getInt("NumeroModifiche");
                //Effettuo la media, aggiungendo al totale precedente il valore attuale e dividendo per il
numero di occorrenze
                //più 1, che è quella attuale
                valoreprecedente *= numeromodifiche;
                valoreprecedente += Double.parseDouble(voci[2]);
                valoreprecedente /= numeromodifiche+1;
                //Scrivo i valori nuovamente nel database, aggiornando i campi di somiglianza e del numero
di modifiche.
                SQLupdate = "UPDATE SomiglianzaSemantica SET Somiglianza="+ valoreprecedente +",
NumeroModifiche="+ (numeromodifiche+1);
                SQLupdate += " WHERE Parola1='"+ voci[0].trim().replace("'", "'') +"' AND Parola2='"+
voci[1].trim().replace("'", "'') + "'";
                operazione.executeUpdate(SQLupdate);
            }else{
                //Inserisco i nuovi dati nel database.
                SQLupdate = "INSERT INTO SomiglianzaSemantica (Parola1,Parola2,Somiglianza,NumeroModifiche)
VALUES (";
                SQLupdate += "'"+ voci[0].trim().replace("'", "'') + "','"+
voci[1].trim().replace("'", "'') + "','"+ voci[2].trim() + "','1)";
                operazione.executeUpdate(SQLupdate);
            }
            //Ritardo introdotto per evitare sovraccarichi al database! Altrimenti si rischia di ricevere
l'errore
        }catch(Exception e){
            //Catturo l'eccezione, se si verifica
            System.out.println("Si è verificato un errore durante l'interrogazione alla tabella
Trasformazioni: "+e);
        }
    }
    closeDatabaseConnection(connessione);
}

private Connection openDatabaseConnection(String database)
{
    Connection connessione = null;
    try{
        //Carico i driver JDBC della sun
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
//Carico il database della semantica della lingua inglese
File file = new File(".\\database\\"+ database);
//Creo la stringa di connessione al database Access sopra indicato. Il nome del database
//è recuperato dal percorso completo del file
String stringaconnessione = "jdbc:odbc:driver={Microsoft Access Driver (*.mdb)};DBQ="+
file.getAbsolutePath().trim();
//Instauro la connessione con il database usando la stringa di connessione
connessione = DriverManager.getConnection(stringaconnessione);
} catch (Exception e){
    System.out.println("Si è verificato un errore durante il tentativo di aprire il database: " + e);
}
//Controllo se la connessione è stata effettivamente creata correttamente.
if(connessione==null)
    System.out.println("Si è verificato un errore irreversibile durante l'instaurazione della
connessione con il database.");
//Ritorno la connessione creata
return connessione;
}
private void closeDatabaseConnection(Connection connessione)
{
    try{
        //Tengo la chiusura della connessione al database instaurata dal metodo openDatabaseConnection()
        connessione.close();
    } catch (Exception e){
        System.out.println("Si è verificato un errore durante il tentativo di chiudere il database: " + e);
    }
}
}
```

>>  **Calcola.java**

```
public class Calcola
{
    public static void main(String[] args)
    {
        new SemanticProximity("Prossimità semantica");
    }
}
```

>>  **SemanticProximity.java**

```
import java.io.*;
import java.util.*;
import java.util.List.*;
import java.sql.*;

public class SemanticProximity implements Runnable
{
    public static Thread thread;

    //Struttura del database
    private java.util.List<String> vocabolario;
    private java.util.List<String> trasformazioneiorigine;
    private java.util.List<String> trasformazioneidestinazione;
    //Struttura del database thesaurus con la somiglianza tra coppie di parole
    private java.util.List<String> parole;
    private java.util.List<Double> somiglianza;

    private String[] risultatifile;
    private double[] risultatisomiglianza;
    private final int maxthread = 1;
    private int threadattivi;
    private int processirimanenti;
    private String cartella = ".\\Testi";

    public SemanticProximity(String nome)
    {
        this.thread = new Thread(this, nome);
        vocabolario = new ArrayList<String>();
        trasformazioneiorigine = new ArrayList<String>();
        trasformazioneidestinazione = new ArrayList<String>();
        parole = new ArrayList<String>();
        somiglianza = new ArrayList<Double>();
        threadattivi = 0;
        processirimanenti = 0;
        risultatifile = new String[0];
        risultatisomiglianza = new double[0];
        this.thread.start();
    }

    public void run()
    {
        //Precarico i dati grammaticali e del thesaurus per rendere più efficienti le ricerche su database
        preloadData();

        File dir = new File(cartella);
```


>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
if(!dir.isDirectory())
{
    System.out.println("ERROR: not a valid folder.");
    return;
}
//Imposto un filtro per la lettura dei file che il programma analizzerà. Per motivi di compatibilità
//si possono leggere solamente file con estensione TXT, ovvero testo semplice
FilenameFilter filter = new FilenameFilter()
{
    public boolean accept(File dir, String name) {
        return name.endsWith(".txt");
    }
};
//Prelevo la lista dei file, applicando il filtraggio.
String[] listafile = dir.list(filter);
if(listafile.length==0)
{
    System.out.println("ERROR: no file found.");
    return;
}
//Lancio i vari thread per l'esecuzione dell'algoritmo di Paice-Ramirez
System.out.print("Inserire il testo da ricercare: ");
String ricercautente = SavitchIn.readLine();
System.out.println();
while(!ricercautente.equals(""))
{
    //Tempo di inizio
    long tempo = System.nanoTime();
    //Inizializzo i vettori che conterranno i dati dell'elaborazione
    risultatifile = new String[listafile.length];
    risultatisomiglianza = new double[listafile.length];
    for(int i=0;i<listafile.length;i++)
    {
        try{
            new PaiceRamirez(this, "PaiceRamirez", i,ricercautente,dir.getAbsolutePath() +"\\"+
listafile[i]);
            processirimanenti++;
        }catch(Exception e){
            //Se una sessione di ricerca termina in modo anomalo, provvedo a segnalarlo e a
decrementare il contatore
            //dei processi attivi, in modo da non immobilizzare il thread principale
            System.out.println("Anomalous termination of PaiceRamirez"+ i);
            paiceRamirezEnd("PaiceRamirez"+ i);
            processirimanenti--;
        }
    }
    //Attendo finchè i processi non hanno terminato l'esecuzione.
    while(processirimanenti>0)
        try{
            wait();
        }catch(Exception e){}
    //Eseguo l'ordinamento dei risultati mediante una chiamata di procedura.
    Ordina ordinamento = new Ordina(risultatisomiglianza,risultatifile);
    risultatifile = ordinamento.vettoreFileOrdinato();
    risultatisomiglianza = ordinamento.vettoreSomiglianzaOrdinato();
    System.out.println("--- RISULTATI RICERCA per: "+ ricercautente);
    stampaVettore();
    System.out.println();
    //Calcolo il tempo di esecuzione del programma e lo stampo a monitor
    tempo = System.nanoTime()-tempo;
    String mostratempo = tempo/(1000*1000) + " millisecondi --> "+ tempo/(1000*1000*1000) + " secondi";
    if((tempo/(1000*1000*1000))>60)
        mostratempo += " --> "+ ((long)tempo/(long)(1000*1000))/(long)(1000*60) + " minuti";
    System.out.println(""+ thread.getName() + " compute time: " + mostratempo + ".\n");

    //Mostro all'utente il prompt per inserire il testo da ricercare
    System.out.print(">>>> Inserire il testo da ricercare: ");
    ricercautente = SavitchIn.readLine();
    System.out.println();
}
}

public boolean searchWord(String parola)
{
    //Restituisco un valore booleano che indica se la parola è presente o meno nel dizionario
    return vocabolario.contains(parola.trim());
}
public String trasformazione(String parola)
{
    if(trasformazioniorigine.indexOf(parola.trim())==-1)
        return parola;
    return trasformazioneidestinazione.get(trasformazioniorigine.indexOf(parola.trim()));
}
public Double confrontaParole(String word1, String word2)
{
    //Ricerco le possibili combinazioni di confronto tra le parole passate come parametro.
    //Se le due parole sono uguali, allora restituisco un valore di prossimità di 1
    //Se non trovo corrispondenze restituisco il valore 0
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
//Altrimenti restituisco il valore presente nel database
if(word1.equals(word2))
    return 1.0;
if(parole.indexOf(word1.trim() + ";" + word2.trim())>-1 && parole.indexOf(word2.trim() + ";" +
word1.trim())>-1)
    return (somiglianza.get(parole.indexOf(word1.trim() + ";" +
word2.trim()))+somiglianza.get(parole.indexOf(word2.trim() + ";" + word1.trim())))/2;
if(parole.indexOf(word1.trim() + ";" + word2.trim())>-1)
    return somiglianza.get(parole.indexOf(word1.trim() + ";" + word2.trim()));
if(parole.indexOf(word2.trim() + ";" + word1.trim())>-1)
    return somiglianza.get(parole.indexOf(word2.trim() + ";" + word1.trim()));
else
    return 0.0;
}

synchronized public void memorizzaRisultato(String file,double somiglianza,int numeroprocesso)
{
    risultatofile[numeroprocesso] = file;
    risultatisomiglianza[numeroprocesso] = somiglianza;
    processirimanenti --;
    notifyAll();
}

private void stampaVettore()
{
    for(int i=r risultatisomiglianza.length-1;i>=0;i--)
        System.out.println(risultatofile[i] + " >> " + risultatisomiglianza[i] + "%");
}

synchronized public void paiceRamirezEnd(String nomeprocesso)
{
    //Decremento il numero di processi in esecuzione
    threadattivi --;
    notifyAll();
}

synchronized public void paiceRamirezStart(String nomeprocesso)
{
    //Se il numero massimo di processi in esecuzione è stato raggiunto, allora pongo in WAIT il processo
    //chiamante, fino a quando uno non termina.
    while(threadattivi>=maxthread)
        try
        {
            wait();
        }catch(Exception e){}

    //Aggiungo il processo attuale allo stato di "esecuzione" incrementando il contatore di processi
    //attivi. Risveglio gli altri processi in coda nel caso qualche condizione sia verificata
    threadattivi++;
    notifyAll();
}

//Precarico i dati da database (più veloce in lettura). Li carico in memoria, altrimenti incorro in
//problemi di sovraccarico di database. Inoltre la lettura da memoria risulta essere molto più veloce.
private void preloadData()
{
    System.out.println(thread.getName() + " is preloading grammatical data... (this may take a few
minutes)");
    //Apro la connessione al database.
    Connection connessione = openDatabaseConnection("GrammaticaInglese.mdb");
    try{
        //Creo uno Statement per eseguire l'operazione sul database e imposto la stringa
        //SQL per l'interrogazione al database. Impongo l'apertura della tabella alla tipologia scrollabile
        //in entrambe le direzioni e di sola lettura.
        Statement operazione =
connessione.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
        String SQL = "SELECT COUNT(Parola) FROM Vocabolario";
        ResultSet tabella = operazione.executeQuery(SQL);
        tabella.first();
        int numeroparole = tabella.getInt(1);
        SQL = "SELECT Parola FROM Vocabolario";
        //Eseguo la query SQL e memorizzo il risultato
        tabella = operazione.executeQuery(SQL);
        //Controllo se la prima riga della tabella esiste oppure no, per controllare se la parola
        //specificata è presente o meno nel database
        int avanzamento = -100;
        while(tabella.next())
        {
            vocabolario.add(tabella.getString("Parola"));
            if(((int)(100*((double)tabella.getRow()/((double)numeroparole)))>=(avanzamento+10) &&
avanzamento%10==0)
            {
                System.out.println(thread.getName() + " is preloading grammatical data: "+
((int)(100*((double)tabella.getRow()/((double)numeroparole)))+"%");
                avanzamento = (int)(100*((double)tabella.getRow()/((double)numeroparole));
            }
        }

        SQL = "SELECT * FROM Trasformazioni";
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
//Eseguo la query SQL e memorizzo il risultato
tabella = operazione.executeQuery(SQL);
//Controllo se la prima riga della tabella esiste oppure no, per controllare se la parola
//specificata è presente o meno nel database
String destinazione = "";
while(tabella.next())
{
    trasformazioneorigine.add(tabella.getString("Parola"));
    destinazione = tabella.getString("Trasformazione");
    if(destinazione==null)
        trasformazionidestinazione.add("");
    else
        trasformazionidestinazione.add(tabella.getString("Trasformazione"));
}
} catch(Exception e){
    System.out.println("Si è verificato un errore durante l'esecuzione di una interrogazione: " + e);
}
}
//Chiudo la connessione al database
closeDatabaseConnection(connessione);
System.out.println(thread.getName() + " is preloading thesaurus data... (this may take a few minutes)");
//Apro la connessione al database.
connessione = openDatabaseConnection("Thesaurus.mdb");
try{
    //Creo uno Statement per eseguire l'operazione sul database e imposto la stringa
    //SQL per l'interrogazione al database. Imposto l'apertura della tabella alla tipologia scrollabile
    //in entrambe le direzioni e di sola lettura.
    Statement operazione =
connessione.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
    String SQL = "SELECT COUNT(Parola1) FROM SomiglianzaSemantica";
    ResultSet tabella = operazione.executeQuery(SQL);
    tabella.first();
    int numeroparole = tabella.getInt(1);
    SQL = "SELECT Parola1,Parola2,Somiglianza FROM SomiglianzaSemantica";
    //Eseguo la query SQL e memorizzo il risultato
    tabella = operazione.executeQuery(SQL);
    //Controllo se la prima riga della tabella esiste oppure no, per controllare se la parola
    //specificata è presente o meno nel database
    int avanzamento = -100;
    while(tabella.next())
    {
        parole.add(tabella.getString("Parola1") + ";" + tabella.getString("Parola2"));
        somiglianza.add(tabella.getDouble("Somiglianza"));
        if(((int)(100*((double)tabella.getRow()/((double)numeroparole)))>=(avanzamento+10) &&
avanzamento%10==0)
        {
            System.out.println(thread.getName() + " is preloading thesaurus data: " +
((int)(100*((double)tabella.getRow()/((double)numeroparole)))+"%");
            avanzamento = (int)(100*((double)tabella.getRow()/((double)numeroparole)));
        }
    }
} catch(Exception e){
    System.out.println("Si è verificato un errore durante l'esecuzione di una interrogazione: " + e);
}
}
//Chiudo la connessione al database
closeDatabaseConnection(connessione);
System.out.println("\n"+ thread.getName() + " preload complete!");
}

//Manipolazione database
private Connection openDatabaseConnection(String database)
{
    Connection connessione = null;
    try{
        //Carico i driver JDBC della sun
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        //Carico il database della semantica della lingua inglese
        File file = new File(".\\database\\"+ database);
        //Creo la stringa di connessione al database Access sopra indicato. Il nome del database
        //è recuperato dal percorso completo del file
        String stringaconnessione = "jdbc:odbc:driver={Microsoft Access Driver (*.mdb)};DBQ="+
file.getAbsolutePath().trim();
        //Instauro la connessione con il database usando la stringa di connessione
        connessione = DriverManager.getConnection(stringaconnessione);
    } catch (Exception e){
        System.out.println("Si è verificato un errore durante il tentativo di aprire il database: " + e);
    }
    //Controllo se la connessione è stata effettivamente creata correttamente.
    if(connessione==null)
        System.out.println("Si è verificato un errore irreversibile durante l'instaurazione della
connessione con il database.");
    //Ritorno la connessione creata
    return connessione;
}
private void closeDatabaseConnection(Connection connessione)
{
    try{
        //Tengo la chiusura della connessione al database instaurata dal metodo openDatabaseConnection()
        connessione.close();
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
    } catch (Exception e){
        System.out.println("Si è verificato un errore durante il tentativo di chiudere il database: " + e);
    }
}
}
```

>>  PaiceRamirez.java

```
import java.util.regex.*;
import java.sql.*;
import java.util.*;
import java.io.*;
import java.text.*;

public class PaiceRamirez implements Runnable
{
    Thread thread;
    SemanticProximity mt;
    int numeroprocesso;

    private String file;
    private String percorso;
    private String testodocumento;
    private String frase;
    private double coefficientepeso;

    //Variabili per la memorizzazione delle informazioni relative al valore di prossimità semantica
    private double prossimita;
    private String[] frasi;

    //Variabili per la media del della prossimità del testo totale
    private long denominatoremedia;
    private double numeratoremedia;

    public PaiceRamirez(SemanticProximity mt, String nome,int numero, String frase, String file)
    {
        this.mt = mt;
        this.prossimita = 0;
        this.thread = new Thread(this, nome+numero);
        this.numeroprocesso = numero;
        this.testodocumento = file;
        this.file = new File(file).getName();
        this.percorso = new File(file).getAbsolutePath();
        this.frase = frase;
        this.coefficientepeso = 1;
        this.thread.start();
    }

    public void run()
    {
        //Richiedo il permesso di esecuzione al processo padre.
        mt.paiceRamirezStart(thread.getName());
        //Leggo il testo dal file
        //Normalizzo la frase di confronto e tutto il documento su cui verrà effettuato il confronto
        frase = normalize(frase);
        frase = frase.substring(0,frase.length()-1);
        if(new File(percorso.substring(0,percorso.length()-file.length()) + (file.substring(0,file.length()-3)
+"normalized")).exists())
        {
            testodocumento = leggiTesto(percorso.substring(0,percorso.length()-file.length()) +
file.substring(0,file.length()-3) + "normalized");
            //Divido le frasi basandomi sul pattern delle regular string precedentemente dichiarato. Saltando
la procedura di normalizzazione, infatti
            //ho la necessità di identificare le frasi del documnto (procedura che faceva la normalizzazione)
            Pattern pat = Pattern.compile("[.]");
            frasi = new String[pat.split(testodocumento).length];
            frasi = pat.split(testodocumento);
        }else{
            testodocumento = leggiTesto(testodocumento);
            testodocumento = normalize(testodocumento);
            scriviTesto(percorso.substring(0,percorso.length()-file.length()) + file.substring(0,file.length()-
3) + "normalized", testodocumento,false);
        }
        //Se la frase di confronto o il documento sono vuoti, allora termino l'esecuzione
        if(frase==" " || testodocumento==" " || testodocumento==null)
            return;

        coefficientepeso = Math.log(frasi.length)*100/80;
        semanticProximity();
        //Segnalo al processo padre che il processo termina.
        mt.paiceRamirezEnd(thread.getName());
    }

    private void semanticProximity()
    {
        //Suddivido in parole la frase campione ed eseguo la procedura per ogni frase contenuta nel documento
        //che sto analizzando.
    }
}
```

```
String[] paroleS = frase.split(" ");
String[] paroleT = null;
//Matrice della somiglianza tra le varie coppie di termini
double[][] V = null;
//Somma riga-colonna per ogni cella
double[][] W = null;
//Ordinamento delle parole tra le S e T
int[] J = null;
double phi = 0.0;
double somiglianza = 0.0;
//Inizializzo le variabili per il calcolo della media del peso del documento
numeratoremedia = 0.0;
denominatoremedia = 0;
//Inizializzo il file su cui verranno salvati i risultati della ricerca.
stampaUltimaRicerca();
//Imposto la formattazione dei numeri a due cifre decimali dopo la virgola.
NumberFormat formatonumero = NumberFormat.getInstance();
formatonumero.setMaximumFractionDigits(2);
//Scorro tutte le frasi nel documento. Ogni frase deve essere analizzata e valutata.
for(int contatore=0;contatore<frasi.length;contatore++)
{
    //Resetto le variabili che utilizzo durante il programma.
    phi = 0.0;
    somiglianza = 0.0;
    J = null;
    V = null;
    W = null;
    paroleT = null;
    //Se la frase è vuota allora salto al controllo successivo. Questo controllo è necessario in quanto
    //può sfuggire al normalizzatore un controllo di un carattere vuoto che provocherebbe
    l'inutilizzazione
    //dei parametri successivi
    if(frasi[contatore].trim()=="")
        continue;
    //Carico la frase corrente in memoria
    paroleT = frasi[contatore].split(" ");
    //Controllo che il numero di parole della frase del documento sia superiore al numero di parole
    della
    //frase di ricerca dell'utente.
    if(paroleT.length<paroleS.length)
        continue;
    //Dimensiono le matrici che utilizzerò
    V = new double[paroleS.length][paroleT.length];
    W = new double[paroleS.length][paroleT.length];
    J = new int[paroleT.length];
    //Inizializzo i valori del vettore J
    for(int contatore1=0;contatore1<J.length;contatore1++)
        J[contatore1] = -1;
    //Riempio la matrice V
    for(int contatore1=0;contatore1<paroleS.length;contatore1++)
        for(int contatore2=0;contatore2<paroleT.length;contatore2++)
            V[contatore1][contatore2] = mt.confrontaParole(paroleS[contatore1],paroleT[contatore2]);

    //Calcolo la matrice W di partenza
    for(int contatore1=0;contatore1<V.length;contatore1++)
        for(int contatore2=0;contatore2<V[contatore1].length;contatore2++)
        {
            //Calcolo il valore w(ij) della matrice W
            double somma = 0.0;
            for(int contatore3=0;contatore3<V[contatore1].length;contatore3++)
                if(contatore3!=contatore2)
                    somma += V[contatore1][contatore3];
            for(int contatore3=0;contatore3<V.length;contatore3++)
                if(contatore3!=contatore1)
                    somma += V[contatore3][contatore2];
            somma += (1-V[contatore1][contatore2]);
            W[contatore1][contatore2] = somma;
        }
    //Eseguo n volte la ricerca del minimo sulla matrice W
    for(int contatore1=0;contatore1<paroleS.length;contatore1++)
    {
        //Il valore 3.4e+038 corrisponde al valore infinito del tipo float. Questo valore è sufficiente
        //a garantirmi che è maggiore di qualsiasi valore presente nella tabella W.
        double minimo = 3.4e+038;
        //Questa variabile memorizza le coordinate della tabella W in cui è situato il valore minimo
        //che viene man mano trovato.
        String coordinateminimo = "";
        //Trovo il minimo nella matrice W. Scandisco l'intera matrice memorizzando, di volta in volta,
        //il valore minimo che trovo.
        for(int contatore2=0;contatore2<paroleS.length;contatore2++)
            for(int contatore3=0;contatore3<paroleT.length;contatore3++)
                if(W[contatore2][contatore3]>-1.0)
                    if(W[contatore2][contatore3]<minimo)
                    {
                        //Memorizzo il nuovo minimo trovato
                        minimo = W[contatore2][contatore3];
                        //Memorizzo le coordinate della posizione del nuovo valore minimo
                        coordinateminimo = contatore2 + "+" + contatore3;
                    }
    }
}
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```

        J[contatore3] = contatore2;
    }
    //Se per qualche motivo non è stato trovato il minimo, passo alla frase successiva
    if(coordinateminimo.equals(""))
        continue;
    //Sommo il valore minimo che ho trovato agli altri valori precedenti.
    somiglianza +=
V[Integer.parseInt(coordinateminimo.split(";")[0])[Integer.parseInt(coordinateminimo.split(";")[1])];
    //Elimino riga e colonna appartenente alle coordinate del minimo e memorizzo
    //le coordinate del minimo nel vettore J
    for(int contatore2=0;contatore2<paroleS.length;contatore2++)
        W[contatore2][Integer.parseInt(coordinateminimo.split(";")[1])] = -1.0;
    for(int contatore2=0;contatore2<paroleT.length;contatore2++)
        W[Integer.parseInt(coordinateminimo.split(";")[0])[contatore2] = -1.0;
    //Ricalcolo i valori della matrice W
    for(int contatore2=0;contatore2<V.length;contatore2++)
        for(int contatore3=0;contatore3<V[contatore2].length;contatore3++)
        {
            if(W[contatore2][contatore3]==-1)
                continue;
            //Calcolo il valore w(ij) della matrice W
            double somma = 0.0;
            for(int contatore4=0;contatore4<V[contatore1].length;contatore4++)
                if(contatore4!=contatore3)
                    somma += V[contatore2][contatore4];
            for(int contatore4=0;contatore4<V.length;contatore4++)
                if(contatore4!=contatore2)
                    somma += V[contatore4][contatore3];
            somma += (1-V[contatore2][contatore3]);
            W[contatore2][contatore3] = somma;
        }
    }

    //La funzione tidyness calcolata in un confronto 1:1 vale uno solamente se
    //il peso nella matrice 1x1 è diverso da 0
    if(J[0]==0)
        phi += 1;
    //Calcolo la funzione tidyness
    for(int contatore1=0;contatore1<J.length-1;contatore1++)
        if(J[contatore1]!=-1 && J[contatore1+1]!=-1)
            if(J[contatore1+1]==(J[contatore1]+1))
                phi += 1;

    phi = phi*(1.0/(paroleS.length));
    double psi = 0.8 + phi*(1-0.8);

    //Se il valore di prossimità di questa frase è superiore a quello che è stato trovato nelle frasi
precedenti
    //allo provvedo a sostituirlo con quello attuale.
    if(((1.0/paroleT.length)*somiglianza*psi)>prossimita)
        prossimita = (1.0/Math.max(paroleT.length,paroleS.length))*somiglianza*psi;

    //Calcolo la media della prossimità del documento, appoggiandomi ad una funzione che assegna il
peso
    //secondo la somiglianza della frase.
    numeratoremedia +=
(1.0/Math.max(paroleT.length,paroleS.length))*somiglianza*psi*(long)calcolaPeso((1.0/Math.max(paroleT.length,pa
roleS.length))*somiglianza*psi);
    denominatoremedia +=
(long)calcolaPeso((1.0/Math.max(paroleT.length,paroleS.length))*somiglianza*psi);

    //Scrivo su file i dettagli dei calcoli effettuati per il documento corrente, frase per frase. Il
documento è nominato nello stesso
    //modo del documento originale, ma con estensione "lastsearch"
    stampaUltimaRicerca(frasi[contatore], contatore, phi, psi, somiglianza,
(1.0/Math.max(paroleT.length,paroleS.length))*somiglianza*psi,
(int)calcolaPeso((1.0/Math.max(paroleT.length,paroleS.length))*somiglianza*psi), false);
    }
    //Rimando il risultato calcolato al processo padre, in modo che possa raccogliere tutti i risultati
disponibili.

mt.memorizzaRisultato(file,Double.parseDouble(formatonumero.format((numeratoremedia/denominatoremedia)*100)).rep
lace(",","."),numeroprocesso);
    System.out.println(file + " - Massimo: "+ formatonumero.format(prossimita*100));
    System.out.println(file + " - Media: "+ formatonumero.format((numeratoremedia/denominatoremedia)*100));
    }

    private double calcolaPeso(double valore)
    {
        return Math.exp(valore*coefficientepeso);
    }

    private void stampaUltimaRicerca()
    {
        String testo = "";
        testo += "n° Frase;";
        testo += "Frase;";
        testo += "PHI;";
    }

```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
        testo += "PSI;";
        testo += "Somiglianza;";
        testo += "Prossimità;";
        testo += "Peso;\n";
        scriviTesto(percorso.substring(0,percorso.length()-file.length()) + file.substring(0,file.length()-3)
+"lastsearch", testo,false);
    }
    private void stampaUltimaRicerca(String frase,int numerofrase,double phi,double psi, double
somiglianza,double prossimita,int peso,boolean inzializza)
    {
        String testo = "";

        testo += numerofrase +";";
        testo += frase +";";
        testo += phi +";";
        testo += psi +";";
        testo += somiglianza +";";
        testo += prossimita +";";
        testo += peso;
        testo += "\n";

        scriviTesto(percorso.substring(0,percorso.length()-file.length()) + file.substring(0,file.length()-3)
+"lastsearch", testo,true);
    }

    private String normalize(String testo)
    {
        //Se testo è vuoto esco dalla funzione.
        if(testo==null)
            return "";
        testo = testo.toLowerCase();
        //Sostituisco i punti interrogativi ed esclamativi con i punti semplici, così da potere più facilmente
        //dividere il documento in frasi.
        //Predispongo il pattern per la separazione della stringa
        Pattern pat = Pattern.compile("[.]");
        //Scorro tutto il testo per analizzare eventuali caratteri non alfabetici da eliminare.
        Character carattere = null;
        char[] vettoretesto = testo.toCharArray();
        for(int contatore=0;contatore<vettoretesto.length;contatore++)
        {
            carattere = new Character(vettoretesto[contatore]);
            if(!Character.isLetterOrDigit(carattere) && carattere!='.')
                vettoretesto[contatore] = ' ';
        }

        //Ricompongo il vettore di caratteri in una stringa
        testo = new String(vettoretesto);
        //Utilizzo una regular expression per rimuovere gli spazi dal doppi dal documento.
        testo = testo.replaceAll("\\s+"," ");
        //Divido le frasi basandomi sul pattern delle regular string precedentemente dichiarato
        frasi = new String[pat.split(testo).length];
        frasi = pat.split(testo);
        //Elimino gli spazi in eccesso all'inizio e alla fine di una frase.
        for(int contatore=0;contatore<frasi.length;contatore++)
            frasi[contatore] = frasi[contatore].trim();

        return replaceWords();
    }

    private String replaceWords()
    {
        int avanzamento = -100;
        for(int contatore=0;contatore<frasi.length;contatore++)
        {
            //Suddivido la frase corrente in parole. In questo modo posso analizzarle singolarmente
            //e mantengo traccia della frase.
            String[] parola = new String[frasi[contatore].split(" ").length];
            parola = frasi[contatore].split(" ");
            //Resetto la frase corrente. La ricostruisco con le parole sostituite.
            frasi[contatore] = "";
            String parolatem = null;
            for(int contatore2=0;contatore2<parola.length;contatore2++)
            {
                //Ignoro le parole con lunghezza minore di 2
                if(parola[contatore2].length()<2)
                    continue;
                //Controllo se è presente qualche trasformazione della parola in questione nel database
                //delle trasformazioni
                parolatem = mt.trasformazione(parola[contatore2]);
                if(parolatem=="")
                {
                    parola[contatore2] = "";
                    continue;
                }
                //Per prima cosa sulla parola elimino eventuali suffissi per renderla il più generico
                possibile.
                String parolastemmata = "";
                try{
```

>>Costruzione di un thesaurus per gli algoritmi di prossimità semantica >>Codice

```
        PorterStemmer ps= new PorterStemmer();
        ps.add(parolatemp.trim().toCharArray(),parolatemp.trim().length());
        ps.stem();
        parolastemmata = ps.toString();
    }catch(Exception e){ }
    //Controllo se la parola è presente nel dizionario
    if(mt.searchWord(parolastemmata.trim()))
        parolatemp = parolastemmata.trim();
    //Aggiungo la parola alla lista delle parole
    frasi[contatore] += parolatemp.trim() + " ";
    parola[contatore2] = parolatemp.trim();
}
}

String testonuovo = "";
for(int i=0;i<frasi.length;i++)
{
    testonuovo += frasi[i].trim() + ".";
}
//Ritorno il nuovo testo
return testonuovo;
}

private String leggiTesto(String file)
{
    String testo = "";
    try{
        //Apro il file
        BufferedReader in = new BufferedReader(new FileReader(file));
        //Leggo una riga del file.
        String riga = in.readLine();
        //Continuo a leggere nuove righe dal file fino a quando il documento non è temrinato
        while(riga != null)
        {
            testo += riga;
            riga = in.readLine();
        }
        //Chiudo il file
        in.close();
    } catch (IOException e){
        System.out.println("Error reading file: "+ e);
        return null;
    }
    //Riduco tutto minuscolo per poterlo paragonare con i dati nel database.
    testo = testo.toLowerCase();
    //Rimuovo gli spazi doppioni.
    testo = testo.replaceAll("\\s+", " ");
    //Sostituisco il ? e ! con un . normale
    testo = testo.replace("?", ".").replace("!", ".");

    return testo;
}

private boolean scriviTesto(String file,String testo,boolean append)
{
    try{
        //Creo un flusso verso il file
        BufferedWriter out = new BufferedWriter(new FileWriter(file,append));
        //Continuo a scrivere nuove righe dal file fino a quando il documento non è temrinato
        out.write(testo);
        //Chiudo il file
        out.close();
        return true;
    } catch (IOException e){
        System.out.println("Error writing file: "+ e);
        return false;
    }
}
}
```


Bibliografia

Libri

- *Java J2SE 5 - La guida completa* – Herbert Schildt – Mc Graw Hill, 2005
- *Java – An introduction to computer science & programming* – Walter Savitch – Prentice Hall, 2001
- *Automatic text processing* – Gerard Salton – Addison Wesley, 1989

Articoli

- *Approximate String Matching* - Patrick A. V. Hall , Geoff R. Dowling - Computing Surveys, Vol 12, No. 4, December 1980, pp.381 – 401
- *The calculation of similarities between multi-word strings using a thesaurus* - C. D. Paice, V. Aragon-Ramirez - Proc. RIAO, 1985, pp. 293 – 319
- *Lexical Ambiguity and Information Retrieval* - Robert Krovetz, W. Bruce Croft - ACM Transaction on Information Systems, Vol 10, No. 2, April 1992, pp. 115 – 141
- *Automatic Thesaurus Costruction Based on Term Centroids* - Robert G. Crawford - Tecnical Report 78-80, Department of Computing and Information Science, May 1979, pp. 1 – 29
- *Building a large thesaurus for information retrieval* – E. Fox, J. Nutter, T. Ahlswede, J. Markwitz – A proceedings of te second conference of Applied Natural Language Processing, Austing, Texas, 1988
- *Relational thesauru in Information Retrieval* – Y. Wang, J. Vandendorpe, M. Evans – American society for information scienze, 1991
- *Exploiting Native XML Indexing Techniques for XML Retrieval in Relational Database Systems* - Felix Weigel* Klaus U. Schulz* Holger Meuss - Centre for Information and Language Processing European Southern Observatory University of Munich – Germany
- *Information Technology and Economic Performance: A Critical Review of the Empirical Evidence* – Jason Derick, Vijay Gurbaxani, Kenneth L. Kramer - University of California, Irvine
- *Requirements Interaction Management* – William N. Robinson, Suzanne D. Pawlowski, Vecheslav Volkov - Georgia State University

Web

- <http://java.sun.com>
Informazioni riguardo librerie Java, classi per connessioni a database e forum di supporto tecnico per problemi incontrati durante la scrittura del codice.
- <http://www.wikipedia.org>
Informazioni di carattere generale su information retrieval e le tecniche più diffuse per creazione di thesaurus e analisi di prossimità semantica
- <http://www-db.deis.unibo.it/~srizzi/PDF/IR.pdf>
Testo che introduce le tematiche dell'information retrieval
- <http://web.tiscali.it/mmariotti/irewww>
Documenti relativi all'information retrieval

- <http://www.dcs.gla.ac.uk/Keith>
Articolo che tratta dell'information retrieval
- <http://www-db.deis.unibo.it/courses/SI2/slides/IRbasics.pdf>
Documento che tratta concetti di information retrieval, stemming e stop-words
- <http://www.comp.lancs.ac.uk/computing/research/stemming/general/performance.htm>
Testo che tratta degli errori che si possono commettere durante la fase di stemming
- <http://www.math.unipd.it/~fabseb60/SI-2005-06.html>
Insegnamento di sistemi informativi – Laurea specialistica in informatica – Università di Padova – Prof. Fabrizio Sebastiani – Dispense e lucidi del corso
- [http://www.unibg.it/struttura/struttura.asp?corso=9007&nomecorso=Sistemi%20informativi%20III%20\(5%20crediti\)](http://www.unibg.it/struttura/struttura.asp?corso=9007&nomecorso=Sistemi%20informativi%20III%20(5%20crediti))
Sistemi informativi III – Università degli studi di Bergamo – Prof.ssa Gloria Bordogna – Dispense e lucidi del corso