ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

DISAMBIGUAZIONE DI DATI DA FONTI ETEROGENEE IN LINKED OPEN DATA: UN'ESPERIENZA

Relatore:
Chiar.mo Prof.
FABIO VITALI

Presentata da:
MAURIZIO
MASTRIA

Co-relatore
SILVIO PERONI

Sessione I
Anno Accademico 2014/2015

Indice

Introduzione

Con l'avanzare degli anni, il mondo della letteratura ha visto un vertiginoso incremento nel numero di libri, pubblicazioni, articoli e ogni forma di media. Considerando poi, che il World Wide Web è sicuramente la migliore “biblioteca” dove deporre tali opere, godendo di una immediata accessibilità e di una capacità praticamente infinita, possiamo farci un'idea di quante pubblicazioni si possano trovare in esso.

Volendo quindi ricercare un testo, basta inserire il titolo dell'opera e subito possiamo ottenere tutte le informazioni che ci servono riguardanti essa, come l'autore, l'anno di pubblicazione, i coautori, nonché il testo della pubblicazione stessa.

Purtroppo però ritroviamo anche molti casi di autori omonimi che rendono difficile riconoscere con sicurezza l'associazione di un'opera ad uno specifico proprietario, senza ambiguità; e ritroviamo anche autori i cui nomi sono indicati più volte ma in modi diversi.

I casi specifici sono molti, a volte facilmente distinguibili, a volte quasi impossibili, anche per l'occhio umano. A tale scopo, quindi, bisogna avvalersi di tutte le informazioni possibili che ruotano intorno ad una determinata opera, per stabilirne l'effettivo proprietario, e la nascita di modelli, procedure e software è una necessaria conseguenza in risposta a come risolvere tale rompicapo.

In quest'articolo descriverò la mia esperienza nel risolvere e disambiguare questi Linked Open Data tramite un'applicazione alla quale ho dato il nome di CALID, evidenziando i problemi incontrati, i metodi creati per risolverli e le scelte progettuali che hanno consentito di migliorarlo, descrivendolo prima in forma più ampia e discorsiva, per una lettura leggera e comprensibile, e poi nei dettagli, illustrando anche alcuni algoritmi utilizzati che potrebbero essere chiarificatori durante i futuri sviluppi da parte di altri programmatori.

Infine, dimostrerò come i test da me effettuati abbiano prodotto un risultato soddisfacente, rendendo CALID un tool valido per risolvere il nostro problema.

Capitolo 1

Il problema della Disambiguazione degli Autori

Se stessimo trattando dati all'interno di un database relazionale, avremmo la possibilità di godere della “unicità” della chiave primaria, in quanto ogni record all'interno di un database, anche con chiave primaria identica, sarebbe comunque distinto e a se stante.

Infatti se andassimo ad inserire un secondo record multiplo del primo, e se entrambi identificassero lo stesso autore, avremmo soltanto il problema di accorpare i due autori in uno unico; se invece aggiungessimo un omonimo non si creerebbe il problema di dividere i due, visto che la chiave primaria gode della sua unicità e quindi i due record sarebbero ancora considerati distinti. [VP14]

Purtroppo, trattare tali dati con un DBMS porterebbe ad avere numerosi problemi.

- bisognerebbe creare un database globale la cui dimensione sarebbe realmente notevole e la sua gestione sarebbe complicata, in quanto una modifica, relazionata ad un determinato autore, andrebbe fatta su tutti i record presenti che lo interessano.
- poiché ad ogni autore dovrebbe essere garantita la possibilità di inserire, aggiornare, modificare i propri record, ci si troverebbe con “molte mani” su un singolo database, con conseguenti problemi di sicurezza, dovuti anche a pecche di conoscenza sull'utilizzo dei database relazionali.
- si avrebbe, inoltre, un inutile spreco di banda e numerosi “colli di bottiglia” sui server.

Proprio per ovviare a questi problemi, il Web 3.0 mette a disposizione le tecnologie del Semantic Web, grazie alle quali ci si troverebbe non più davanti a record memorizzati in database, ma a Linked Data, cioè un grafo di risorse collegate tra loro tramite uno schema <soggetto→predicato→oggetto> dove il soggetto e l'oggetto rappresentano dei concetti, e il predicato è la relazione che li lega.

Quindi, nello specifico di questo trattato, avremmo a che fare con concetti aventi relazione del tipo <Persona→AutoreDi→Pubblicazione>, le cui risorse sono identificate nel vasto World Wide Web tramite URI, facilmente reperibili con queries SPARQL tramite end-points pubblici.

Ma purtroppo, la leggerezza di questa tecnologia si scontra con il problema di identificare la vera identità di una risorsa, proprio perché esse sono slegate da una supervisione che potrebbe garantirne l'affidabilità.

Per questo motivo, l'utilizzo di queste tecnologie deve essere necessariamente accompagnato da un software in grado di “capire” come distinguere gli omonimi e accorpare gli autori duplicati.

1.1 Una precisa descrizione del problema

Insegnare ad un computer come distinguere o accorpare due autori, rappresentati come risorse, comporterebbe innanzitutto renderlo capace di avviare un processo di riconoscimento di cognomi e nomi. Questo è almeno quello che farebbe qualsiasi essere umano per iniziare a risolvere tale problema.

Prendiamo come esempio 4 autori, ognuno avente identificatore, uri, nome e cognome, e che abbiano pubblicato almeno un documento scientifico. Ad esempio:

Ids	Uri	Nome	Cognome	Titolo Pubblicazione
aa1	/wen-gao/	Wen	Gao	A Reconfigurable High Availability Infrastructure in Cluster for Grid
aa2	/wen-wu/	Wen	Wu	How many users should be turned on in a multi-antenna broadcast channel?
aa3	/wen-chen/	Wen	Chen	Efficient Relay Beamforming Design with SIC Detection for Dual-Hop MIMO Relay Networks
aa4	/guanju-gao/	Guanjun	Gao	Relevancy based semantic interoperation of reuse repositories

Come possiamo notare, abbiamo una lista che presenta delle caratteristiche che ci permettono di identificare i 4 autori come unici, dove nessuno è duplicato dell'altro.

Abbiamo casi di omonimia nei cognomi o nei nomi, ma l'unicità delle coppie <nome-

cognome> non rendono ambiguo il sistema.

Vediamo un altro esempio:

Ids	Uri	Nome	Cognome	Titolo Pubblicazione
xx1	/thom-hermann/	Thom	Hermann	Grenzen der Software-Ergonomie bei betrieblichen ISDN-Anlagen
xx2	/tommy-hermann/	Tommy	Hermann	Grenzen der Software-Ergonomie bei betrieblichen ISDN-Anlagen
xx3	/thom-hermann/	Thom	Hermann	Improving and Extending the Lim/Lee Exponentiation Algorithm

In questo caso, i primi 3 autori della lista hanno cognome identico e nomi similari, fatti che potrebbero farci supporre che questi elementi siano uno duplicato dell'altro: infatti “Tommy” e “Thom” sono i diminutivi di “Thomas”.

Ma in realtà, l'autore di “Improving and Extending the Lim/Lee Exponentiation Algorithm” non è lo stesso di “Grenzen der Software-Ergonomie bei betrieblichen ISDN-Anlagen”, ma un omonimo, e in questo caso, solo ponendo attenzione al titolo della pubblicazione avremmo potuto accorgercene.

Invece gli elementi con id xx1 e xx2 sono multipli e rappresentano lo stesso autore.

Ma continuiamo con altri esempi per spiegare meglio il fenomeno:

Ids	Uri	Nome	Cognome	Titolo Pubblicazione
yy1	/thomas-meyer/	Thomas	Meyer	Robustness to Code and Data Deletion in Autocatalytic Quines
yy2	/thomas-meyer/	Thomas	Meyer	A self-healing multipath routing protocol

Notiamo una situazione similare a quella precedente, dove i cognomi e i nomi coincidono; però guardando il titolo e ricordando il caso “Hermann”, potremmo anche essere portati a supporre che i due autori non siano uno duplicato dell'altro, ma che siano invece distinti, anche se in questo caso gli id yy1 e yy2 rappresentano lo stesso “Thomas Meyer” autore di entrambe le pubblicazioni.

Iniziamo dunque a comprendere la difficoltà della ricerca della soluzione al problema della disambiguazione degli autori, anche se i casi elencati fin qui sono relativamente difficili da disambiguare, in quanto l'utilizzo di dati di supporto, come il titolo della pubblicazione, ci ha fornito il modo per risolvere il problema.

Guardiamo un altro esempio:

Ids	Nome	Cognome	Titolo Pubblicazione

zz1	J.	Wang	Interference Aware and Delay Bounded Routing in Hybrid Wireless-Optical Access Network
zz2	Jiamping	Wang	Interference Aware and Delay Bounded Routing in Hybrid Wireless-Optical Access Network
zz3	Jin	Wang	A Study of Network Throughput Gain in Optical-Wireless (FiWi) Networks Subject to Peer-to-Peer Communications

Qui viene rappresentato un caso in cui saremmo portati a identificare l'autore con id zz1 con l'autore di id zz2.

Ma in realtà, "Interference Aware and Delay Bounded Routing in Hybrid Wireless-Optical Access Network" è una pubblicazione che ha come autori sia Jianping Wang sia Jin Wang, e quindi non possiamo effettivamente sapere quale tra gli autori di id zz2 e zz3 sia il duplicato di quello con id zz1, a meno di avere ulteriori dati a disposizione; e comunque, anche avendoli, di questa ambiguità rimarrà sempre una percentuale rilevante che potrebbe invalidare il lavoro di disambiguazione, almeno per questa coppia.

Fortunatamente casi come questo sono molto rari in quanto è difficile che due autori, aventi cognome e iniziale del nome uguale, scrivano insieme lo stesso libro.

Questi sono dei casi generali standard, ma nel seguito di questo articolo introdurremo altre particolarità da considerare per la risoluzione del problema.

1.2 Soluzioni esistenti

La maggior parte delle esistenti soluzioni al problema concentrano i loro algoritmi in due fasi:

- quella in cui si cercano tutte le coppie i cui nomi sono similari
- quella in cui si cerca di distinguere i falsi duplicati dai veri, tramite l'acquisizione e il confronto di informazioni legate agli autori in questione.

Tra le soluzioni esistenti in letteratura per il calcolo della similarità tra due nomi troviamo:

- la "Levenshtein Distance", che si preoccupa di calcolare il minimo numero di modifiche necessarie per trasformare una stringa nell'altra. [LEV66]

- la “Soundex Distance”, algoritmo che identifica come “uguali” due parole che hanno una simile pronuncia in inglese. [HM02]

Queste due funzioni di string matching però, hanno dei punti deboli:

- la Soundex Distance funziona soltanto per i nomi inglesi, e questo è un serio problema in quanto le comparazioni vengono fatte per gli autori di tutto il mondo; si potrebbe utilizzare in aggiunta ad un altro algoritmo di comparazione e soltanto per i nomi inglesi, visto che all'interno di un dataset ottenuto da una sparql query, è prevista anche la keyword “lang” indicante la “lingua” del dato; il problema è che questo tipo di dato è opzionale, quindi non sempre presente, e a volte il dataset da esaminare, non è esattamente l'originale ma ottenuto da precedenti trasformazioni che abbiano potuto causare la perdita dell'informazione.
- la Levenshtein Distance invece è un'ottima funzione per il calcolo della distanza da percorrere per raggiungere il nome di arrivo da quello di partenza, ma è effettivamente funzionale solo nel caso in cui uno dei due nomi sia stato trascritto con errori. Ad esempio, se dovessimo considerare la “distance” tra “Jiamping” e “Jianping”, noteremmo che essa ha valore 1 in quanto i due autori sono diversamente espressi a causa di un errore; ma se consideriamo la distance, sempre di valore 1, tra “Lei” e “Lee”, avremmo l'accorpamento di due nomi in realtà diversi, ma che l'algoritmo vede similari.

Troviamo poi metodi che considerano i casi in cui i nomi siano scritti in forma abbreviata [NEW01] :

- Il “First Initial Method” che consiste nel considerare come similari due nomi aventi in comune la prima lettera iniziale. Ad esempio “Thom Hermann” ,“T. Hermann” e ”Tommy Hermann”, secondo questo metodo, sarebbero nomi similari.
- L' “All Initial Method” che invece prevede la considerazione di tutte le sue iniziali. Secondo questa tecnica, “Thom Hermann” risulterà diverso da “Tommy Hermann” in quanto già la seconda iniziale tra i due è diversa, ma uguale a “Thomas Hermann”, in quanto “Thom” è contenuto in “Thomas”.

È stato anche ideato un terzo metodo ibrido:

- L' “Hybrid Method” [MIL13] che decide se una coppia di autori debba essere comparata tramite il primo o il secondo metodo, basandosi su dati statistici ottenuti durante l'esecuzione del programma. Ad esempio, se in un dataset ritroviamo frequentemente il cognome “Hermann”, l'Hybrid Method sceglierà di confrontare due autori aventi questo stesso cognome secondo l'All Initial Method, in quanto, ipotizzando che all'aumentare della frequenza del cognome aumentano anche il numero degli omonimi, occorre una disambiguazione di precisione maggiore;

  al contrario, per una coppia di autori il cui cognome si incontra raramente, verrà utilizzato il “First Initial Method”, in quanto si ipotizzerà che la bassa frequenza del cognome concordi con una bassa frequenza di autori omonimi e distinti, e quindi due nomi aventi la prima iniziale in comune produrranno un match positivo.

L'“Hybrid Method” può avere risultati interessanti, ma la definizione di un valore limite di frequenza per il quale usare o il primo o il secondo metodo dipende fortemente dal dataset, dalla sua lunghezza e dalla sua eterogeneità. Tra l'altro, i test effettuati tramite questo metodo ibrido hanno visto variazioni di precisione “anomale” a seconda che i dati trattati siano relativi ad autori che lavorano nell'ambito dell'astronomia, economia, matematica, robotica, ecologia, ecc, come si può vedere dalla seguente tabella:

Table 1. Properties of empirical datasets. All data refer to 2006-2010 publication period.

Field	Number of articles	Number of authors	Average productivity (in five years)	Percent of authors with middle initials	Reporting rate of the middle initial of authors who have it
Astronomy	31,473	30,605	6.93	50%	74%
Mathematics	3,244	4,396	1.43	29%	100%
Robotics	2,630	5,734	1.54	31%	76%
Ecology	5,420	11,308	1.69	67%	83%
Economics	2,352	2,836	1.64	32%	97%

Ulteriori scelte per decidere quando un nome sia similare ad un altro sono:

- considerare i diminutivi o i “nicknames”, i quali però non godono della proprietà transitiva: ad esempio, il diminutivo “Chris” verrà considerato similare a

“Christopher” e “Christian”, ma quest'ultimi non lo saranno tra loro.

- “Noisy First” o “Last Names” : secondo questa metodologia, vengono prima calcolate delle statistiche su un singolo nome. In seguito, i nomi che compaiono raramente vengono confrontati con tutte le possibili sotto stringhe scartando i caratteri all'inizio o alla fine della singola parola che compongono il nome. Se queste sotto stringhe appaiono frequentemente nel dataset, il nome che le ha generate viene sostituito con esse. [LLL13]

  Ad esempio, se all'interno del dataset il nome “Modianoy” appare raramente, si attua il processo scartando o la prima o l'ultima lettera. Scartando l'ultima, se il nome risultante “Modiano” apparirà frequentemente nel dataset, allora “Modianoy” verrà sostituito da quest'ultimo.

- “Mistakenly Separated or Merged Name Units”:

  Metodo secondo il quale due nomi sono similari se sono stati divisi o uniti erroneamente. In questo caso l'algoritmo si preoccupa di unire i divisi e confrontarli tra loro. [LLL13]

Questi metodi sono efficaci ma il loro difetto è che troppe coppie vengono definite similari, e la fase successiva per dividere gli omonimi, risulta pesante e con una bassa precisione in caso di dati non abbastanza eterogenei.

La fase successiva processa gli autori omonimi e gli eventuali accorpati nella precedente fase analizzando i dati relativi essi, come le loro pubblicazioni che includono l'anno e la conferenza in cui sono state presentate, la lista dei coautori con i quali generalmente collaborano, con le sedi delle affiliazioni per le quali lavorano, ecc.

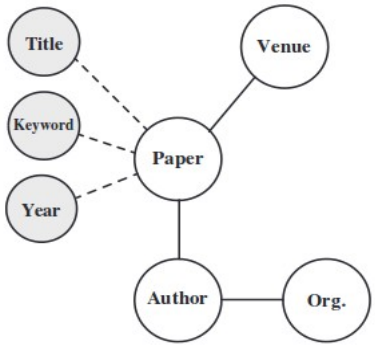
Un metodo a mio parere molto valido è il “Metapath Based Similarity”

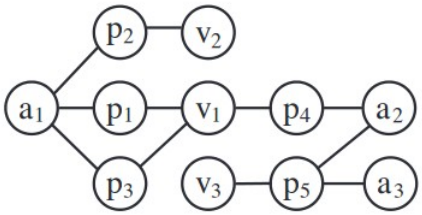
- **Metapath Based Similarity**

  Un “metapath” è una relazione che lega due autori tramite un percorso avente una certa lunghezza. Il calcolo della similarità tramite metapath è una procedura che assegna un punteggio compreso tra 0 e 1 al “path” da percorrere per raggiungere un autore a partire da un altro, utilizzando i dati relativi al contesto in comune tra i

due. [SHY11]

Consideriamo un grafo descrivente il contesto che ruota intorno ad un autore secondo lo schema della “Microsoft Academic Search” [RDM13] e un grafo come esempio di possibili “path” tra autori.





Volendo confrontare l'autore a1 con l'autore a2, il percorso da percorrere sarebbe “a1 → p1 → v1 → p4 → a2”. I due autori, secondo questo criterio, avrebbero un percorso con punteggio sicuramente positivo, in quanto, oltre ad essere omonimi, esso descrive la partecipazione dei due alla stessa conferenza (venue).

Gli autori a1 e a3, invece, non essendo connessi direttamente, avrebbero in questo caso similarità pari a 0

Il risultato di un metapath, è ottenuto a partire dalla costruzione delle matrici di adiacenza delle sotto relazioni; esse vengono moltiplicate e normalizzate secondo la lunghezza del percorso da esaminare.

Le matrici di adiacenza $M_{A,P}$ e $M_{P,V}$ in riferimento al grafo precedente sono:

$$M_{A,P} = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \end{array} \begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \qquad M_{P,V} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

I valori 1 nella $M_{A,P}$ indicano le pubblicazioni connesse agli autori, mentre nella $M_{P,V}$ indicano le connessioni “pubblications ↔ venues”.

La matrice risultante $M_{A,A}$ basata sul metapath APVPA sarà quindi:

$$\overline{M_{A,A}} = \overline{M_{A,V}} \times \overline{M_{A,V}}^T = \begin{matrix} \\ a_1 \\ a_2 \\ a_3 \end{matrix} \begin{matrix} \begin{matrix} a_1 & a_2 & a_3 \end{matrix} \\ \begin{bmatrix} 1.0000 & 0.6325 & 0 \\ 0.6325 & 1.0000 & 0.7071 \\ 0 & 0.7071 & 1.0000 \end{bmatrix} \end{matrix}$$

dove:

$$\overline{M}_{A,V= \text{Normalize}}(M_{A,P} \text{ x } M_{P,V})$$

Metapath multipli possono poi essere costruiti per aumentare la precisione dell'algoritmo.

Ritengo che questa sia un ottima soluzione al problema, ma purtroppo l'algoritmo viene applicato su un numero troppo esteso di coppie di autori, poiché la fase di “recall-improving”, in cui vengono recuperate tutte le coppie con nome similare, non scarta quelli in conflitto tra loro, e di conseguenza il tempo di esecuzione aumenta considerevolmente visto la grande quantità di dati da gestire.

Solo nella procedura finale di “precision-improving” i nomi vengono processati in modo più aggressivo, definendo quali sono in conflitto e quindi da scartare da quelli che invece superano il test.

La tabella seguente mostra infatti i tempi impiegati affinché il programma termini la sua esecuzione. [LLL13]

Performance	Gain	New Module(s)	Days
95.376	-	Same Author Name Benchmark	-
95.786	0.410	+ Meta-path: Coauthor	19
96.623	0.837	+ Name Initials, Omitted Middle Name	21
97.427	0.804	+ Meta-path: Covenue	27
97.770	0.343	+ Nicknames + Asian names handling	33
98.729	0.959	+ Accepting name-compatible author pair even with zero meta-path-based similarity	36
99.020	0.291	+ Name reordering + noisy last/first name pre-processing	37
99.036	0.016	+ Rough post-processing	42
99.075	0.039	+ SoundEx distance	45
99.130	0.055	+ Name units breaking/merging pre-process, + name expansion	49
99.157	0.027	+ Iterative framework + more aggressive post-processing	54

La soluzione da me proposta al problema, in modo da ottenere un valido risultato in termini di precision, recall e costo computazionale, è racchiusa in un programma da me chiamato CALID.

CALID è la sigla di ***“Customizable Application for Literal and Iri's Disambiguation”.***

Esso è un programma valido non solo per affrontare il problema di disambiguazione degli autori, ma è più generico, in quanto parametrico e fortemente configurabile.

Come abbiamo visto in precedenza, la maggior parte degli algoritmi produce risultati più o meno soddisfacenti a seconda del dataset che si sta trattando. Anche CALID funziona in questo modo, ma è possibile configurarlo per renderlo efficiente per ogni tipo di dato.

In particolare, il programma divide la sua esecuzione in 3 fasi:

- **preprocessing:** fase di ottimizzazione del dataset per una migliore velocità ed utilizzo nelle fasi successive
- **processing:** fase in cui avvengono i confronti tra gli autori, con conseguente calcolo del punteggio di similarità
- **postprocessing,** nella quale si può scegliere tra:
    - **fase di filtering:** fase in cui viene prodotto un file di filtraggio per essere riutilizzato in caso di esecuzioni multiple del programma sullo stesso dataset
    - **fase di finalizzazione:** fase in cui il programma restituisce un file contente i dati disambiguati

La fase di *preprocessing* permette di ottimizzare il dataset, in modo da renderlo più leggero e direttamente utilizzabile dalle funzioni che costituiscono il nucleo del programma; grazie a questa fase, molte delle operazioni che sarebbe necessario applicare durante la fase di processing, vengono attuate prima, con un risparmio, in termini di costo computazionale, davvero notevole.

Nella fase di *processing* avviene il calcolo delle similarità utilizzando le varie funzioni di confronto secondo le modalità scelte nel file di configurazione, quindi personalizzate per il tipo di dati trattati. In particolare CALID permette di confrontare più insiemi di dati contemporaneamente, fornendo all'utente uno strumento per combinarli e gestirli.

Infine nella fase di *postprocessing*, CALID consente di scegliere se l'elaborato risultante dovrà essere riutilizzato da una successiva riesecuzione del programma sullo stesso dataset *(fase di filtering)* o se generare i risultati finali *(fase di finalizzazione).*

- La **fase di filtering** è molto utile perché riduce notevolmente il tempo di

esecuzione totale del programma, ma, purtroppo, non è sempre applicabile, in quanto dipende fortemente dal tipo di dati trattati.

- La **fase di finalizzazione** recupera le coppie che durante la fase di processing sono state scartate, restituendo l'elaborato finale formato dagli insiemi degli autori calcolati come identici.

Inoltre, se viene fornito un file di soluzioni, specifico in ambiente di testing, CALID calcola l'esattezza e la completezza dei risultati in termini di ***precision*** e ***recall*** [WC15].

Capitolo 2

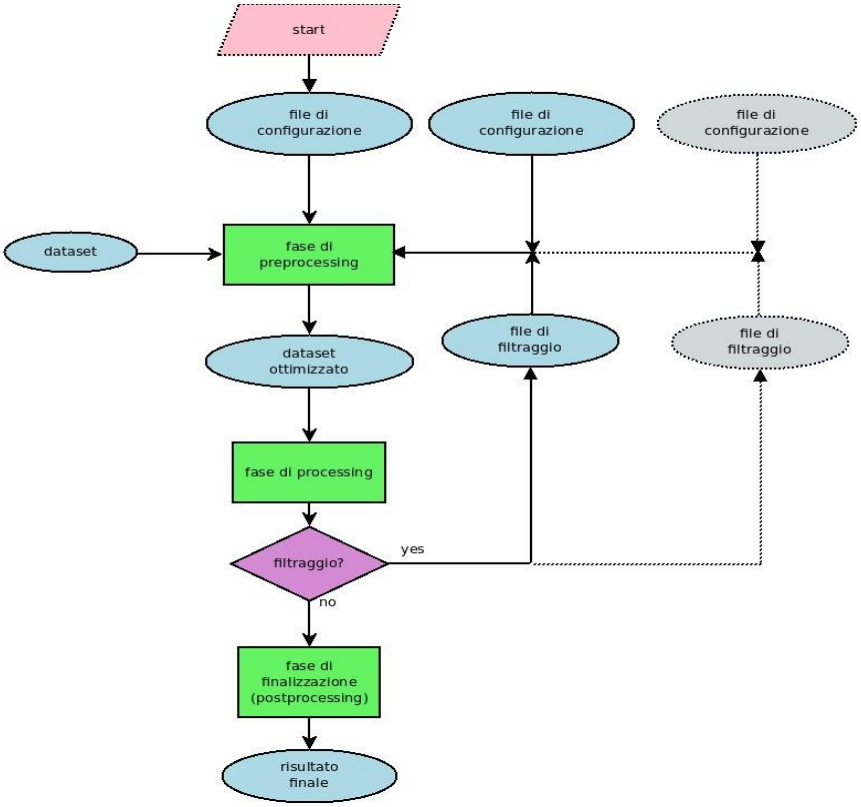
In questo capitolo sarà argomentata la soluzione “CALID”, spiegandone il funzionamento, le scelte progettuali, le soluzioni ereditate e, per alcune di queste, le modifiche effettuate per rendere più efficiente il programma.

Inoltre verrà spiegato come il programma affronti il problema della disambiguazione degli autori, specificandone la sua configurazione e le modalità di utilizzo.

CALID – La descrizione del programma

Come accennato nel capitolo precedente, CALID è un programma che può essere utilizzato in diverse modalità e che si compone di 3 fasi principali: ***preprocessing, processing e postprocessing***.

Il seguente diagramma dà una prima indicazione su come esse sono legate tra di loro e illustra il flusso di esecuzione del programma.



2.1 Pre-Processing

Come vediamo, la prima fase di preprocessing accetta in input un file di configurazione e il dataset da analizzare, determina come esso deve essere ottimizzato e ne restituisce una copia modificata e più leggera.

La scelta del contenuto del file di configurazione, strettamente legata al dataset, dipende dal *tipo di dati*, da quello che *rappresentano*, dal modo in cui essi sono *strutturati* e dal modo in cui essi sono *relazionati*.

- A livello micro, il tipo di dati che si incontrano sono stringhe composte da numeri, caratteri, simboli e combinazione di essi. ***(tipo)***
- A livello macro, le stringhe possono essere composte e combinate per indicare parole, date, risorse, nomi, cognomi e queste combinate per creare frasi, testi, codici e intervalli. ***(rappresentazione)***
- Essi possono essere ***strutturati*** in liste di dati omogenei, o in insiemi di dati eterogenei, considerando anche liste di insiemi o insiemi di liste dove il concetto di insieme implica una ***relazione*** stretta dei suoi componenti, inscindibili durante la fase di processing.

Nel caso specifico della disambiguazione degli autori, il dataset da me analizzato conteneva questo tipo di dati: *identificatore*, *nome* e *cognome* dell'autore, le sue *pubblicazioni* sotto forma di identificatore, *titolo*, *conferenza* in cui sono state presentate, *anno* di pubblicazione, eventuale *affiliazione* dell'autore in relazione alla pubblicazione, *lista di autori* con i quali ha collaborato, ecc.

Un simile esempio può essere questo:

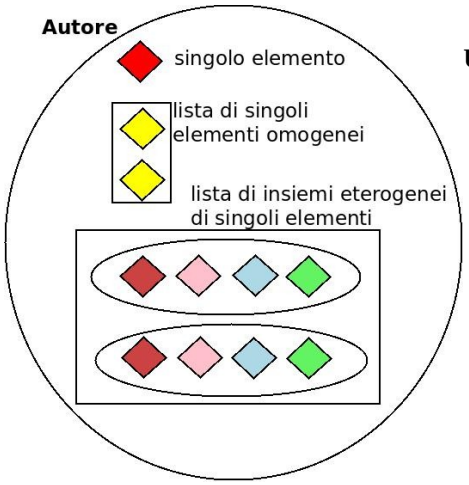
AuthorId	**Name**	**Family name**	**Publication Id**	**Title Publication**	**Venue**	**Year**	**Affiliation**	**Co Authors**
x9d02q	Zhiwen	Mo	29323	“Web Generation”	GECCO	1995	Xidian University	23r0aw, 3d9jfo2
23r0aw	Kwei	Lin	20972	“Ninux”	APSEC	2002	Sichuan University	e238du, 22932r, 298r9e
23r0aw	Kwei-Jay	Lin	98374	“Security Guide”	ACM	2009	M.I.T.	rih49d, cj9f90

Come possiamo notare, tutte queste informazioni possono essere utilizzate ai fini della soluzione, ma è necessaria una loro comprensione sui 4 punti prima citati.

Tralasciando, per ovvie e semplicistiche ragioni, le caratteristiche di tipo, notiamo che i

nomi, i cognomi, titoli delle pubblicazioni, le venues, e le affiliazioni possono essere identificati come liste di stringhe, gli identificatori come numeri o come codici, gli anni come date e i coautori come liste di identificatori.

Inoltre vediamo che il secondo e il terzo autore sono identificati con lo stesso codice, quindi, essendo la stessa persona, in questa fase viene creato un solo oggetto autore che abbia identificatore univoco, come nome una lista di nomi ("Kwei" e "Kwei-Jay"), e che abbia scritto due articoli, che verranno considerati come una lista di due insiemi composti ognuno da titolo, anno, conferenza e affiliazione.



Un esempio di oggetto autore.

- *Singolo elemento:*
  - *id autore*
  - *cognome*
- *Lista di singoli elementi:*
  - *lista dei nomi di un autore*
- *Lista di insiemi di elementi eterogenei:*
  - *lista di insiemi composti da*
    - *id pubblicazione*
    - *titolo*
    - *anno*
    - *venue*

Questo primo passo riduce il dataset ad una lunghezza uguale a quella del numero degli identificatori di un autore, fattore importante visto che nella fase di processing il numero di confronti sarà del tipo $O(n^2)$, dove n rappresenta appunto la lunghezza del set di dati.

Un secondo passo per ridurre il costo computazionale del programma, oltre all'organizzazione degli elementi, è quello di preparare e trasformare i singoli elementi in valori direttamente utilizzabili dalle funzioni di confronto.

Le stringhe possono essere:

- pulite dai caratteri simbolici:

  nel caso in cui si debbano analizzare insiemi di parole, la cui punteggiatura e caratteri simbolici non dovrebbe essere presa in considerazione durante il confronto.

  Nel nostro caso, titoli, venue e affiliazioni

- *inserite in liste:*

  in questo modo abbiamo dei dati pronti sui quali calcolare le intersezioni, unioni e differenze

- *trattate come nomi:*

  in questo particolare caso sono disponibili due opzioni, diverse in fatto di precisione e velocità.

  - La prima opzione, molto veloce durante il confronto, analizza e crea, per un singolo nome, una lista di nomi ipotetici derivanti da abbreviazioni, sostituzioni con i rispettivi diminutivi, permutazioni delle parole che lo costituiscono, permutazione delle combinazioni di possibili unioni.

    A tal proposito, in entrambi i casi che ho analizzato, ho potuto notare che nello scrivere un nome, di esso ne viene rispettato l'ordine. Non esistono autori il cui nome sia scritto in ordine diverso.

    Tra gli autori troviamo *"Li-Chun" e "Chun-Li"* che apparentemente potrebbero apparire come nomi similari in ordine diverso; ma in Cina l'ordine determina quale sia il nome della persona e a quale famiglia appartenga, quindi effettivamente questi due nomi sono diversi.

    Però, "Chun-Li" può essere scritto anche "Chunli" e in questo caso, l'unione delle due parole è fondamentale per recuperare la similarità tra le due coppie.

    In riferimento al caso in cui un autore possa essere rappresentato dal suo diminutivo, come ad esempio "Bob" per "Robert" o "Dave" per "David", ho inserito nel programma un metodo per il calcolo dei diminutivi della maggior parte dei nomi inglesi.

  - La seconda opzione, più lenta in fase di processing, distingue le parole che compongono un nome tra "short-name" e "long-name" dove i primi rappresentano le parole abbreviate o appuntate e i secondi, invece, parole estese, il cui significato risulta più chiaro. Questo perché in fase di confronto, i match tra gli short-name avranno un punteggio minore rispetto a quello tra long-name; infatti il nome "Robert Johnson" è uguale a "Robert Johnson", ma "Robert B. Johnson" confrontato con "Robert B. Johnson", implica l'uguaglianza tra stringhe ma non quella tra nomi.

    Questa seconda opzione è più precisa, in quanto scarta numerosi nomi che invece venivano recuperati precedentemente.

  Il motivo per cui ho creato queste due opzioni dipende da due fattori:

- velocità: in una possibile fase di filtering, dobbiamo scartare quanti più nomi possibili, ma vista la quantità di combinazioni da processare, abbiamo bisogno di una funzione leggera

- correttezza: se il dataset contiene troppi errori di trascrizione, la seconda opzione deve cedere il posto alla prima, che è in grado di recuperarli

Finita la fase di preparazione del dataset, il cuore del programma effettua i confronti e determina quali coppie di autori scartare e quali invece giudicare identici.

2.2 Processing

La fase di processing, è costituita da un motore che confronta i dati generando un punteggio e decide se tale punteggio passi il test di convalida, secondo i valori inseriti dall'utente nel file di configurazione.

Elencherò, dunque, quali metodi CALID mette a disposizione per effettuare i confronti e descriverò, anche con degli esempi, come estrarre le coppie di duplicati grazie al *Multi-Condition Method.*

2.2.1 Metodi di confronto

Una volta che i dati sono stati preparati, essi devono essere processati per essere confrontati.

CALID mette a disposizione 10 tipi di metodi di confronto.

- ***word***: secondo la *Levenshtein distance*, due stringhe vengono confrontate e viene restituita la percentuale di uguaglianza, intesa come il reciproco della distanza.

  Il metodo *word* è indicato per confrontare i cognomi.

- ***string***: due stringhe possono essere intese come frasi, testi, sigle, e nell'ambito del problema della disambiguazione sono indicate per confrontare titoli, venue e affiliazioni.

  La percentuale di match restituita tra le due è il risultato della funzione "***coseno di similitudine***", [HUA08]. Esso è una tecnica per la misurazione della similitudine tra due vettori, calcolando il coseno secondo la formula:

  $$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}.$$

  più chiaramente definita come:

  $$\frac{\sum_{k=1}^{n} A(k)B(k)}{\sqrt{\sum_{k=1}^{n} A(k)^2}\sqrt{\sum_{k=1}^{n} B(k)^2}}$$

  Il contenuto dei due vettori è il numero di volte in cui una certa parola compare

all'interno di una delle due stringhe.

Il risultato di tale formula è un numero compreso tra 0 e 1, dove 1 indica che le stringhe sono uguali tranne che per l'ordine.

- ***number***: normale confronto tra numeri. È previsto anche un confronto tra intervalli di numeri, se specificato.

  Considerando due intervalli $[x_0,y_0]$ e $[x_1,y_1]$ il match sarà positivo se uno dei due è contenuto nell'altro; semi-positivo se i due intervalli si accavallano, ma non si contengono; negativo se non esistono punti di intersezione.

- ***date:*** un confronto tra date e intervalli di date, espresse come anni oppure mese e anno oppure giorno, mese, anno. Come per i numeri, gli intervalli restituiranno tre tipi di punteggio a seconda se si contengono, abbiano intersezioni o che siano completamente separati.

  Le date delle conferenze *(venues)* vengono confrontate con questa funzione.

- ***resource:*** nata per confrontare "uri", confronta se due stringhe sono identiche. I dati che vengono analizzati da questa funzione, non subiscono alcun tipo di modifica.

- ***code:*** ridondante rispetto a *resource*, in quanto il dato non subisce modifiche e il confronto restituisce l'uguaglianza o la diversità.

- ***intersection:*** è il calcolo del numero degli elementi in comune tra due liste di risorse, in rapporto al minimo degli elementi tra i due insiemi di confronto.

  Questa funzione è utile per calcolare il punteggio di match tra liste di co-autori.

- ***string intersection:*** per calcolare le intersezioni tra stringhe.

  Nel file di configurazione, per essa bisogna definire un parametro che indica quando due stringhe possono essere giudicate simili e quindi far parte dell'intersezione. Il confronto tra le singole stringhe delle liste viene fatto dalla "*string*", sempre utilizzando il *coseno di similitudine.*

  Viene restituito un punteggio in percentuale dove 0 indica che l'insieme intersezione è vuoto e 100 indica l'uguaglianza con l'insieme con numero di elementi minore.

  Questa funzione è utile per confrontare liste di "*venue*" o di "*affiliation*".

- ***name (filtro):*** confronta due liste di stringhe di nomi e in caso di match tra almeno un elemento di queste restituisce un risultato positivo. Altrimenti restituisce 0.

- ***name:*** è la funzione più complicata, ma la più adatta per confrontare nomi, se correttamente scritti.

  Essa riceve due strutture contenenti i nomi distinti come abbreviati ed estesi.

  Due nomi, anche se identici, ma che abbiano almeno una parte scritta in forma abbreviata, non possono restituire match massimo, altrimenti non verrebbe rispettato il concetto di uguaglianza del nome.

Questi 10 metodi di confronto sono la base del programma per il calcolo delle similarità.

Un dettaglio del file di configurazione mostra la keyword ***"type"*** che indica i metodi di confronto da utilizzare:

```
"KEYS": {
      "family_name": {
         "group_concat": null,
         "kg": 1,
         "lock": null,
         "type": "word",
         "accuracy": 100
      },
"given_name": {
         "group_concat": null,
         "kg": 2,
         "lock": "pubs",
         "type": "name",
         "accuracy": null
      },
"title": {
         "group_concat": null,
         "kg": 1,
         "lock": "pubs",
         "type": "string",
         "accuracy": 90
      }
}
```

In questo esempio il campo ***"family_name"*** utilizza la Levenshtein Distance (word) per il confronto, la ***"name"*** per il confronto tra nomi di persona, e il "coseno di similitudine",

indicato dalla keyword ***“string”***, per confrontare i titoli delle pubblicazioni.

La keyword ***“lock”*** indica invece l'insieme di appartenenza di un elemento, in modo che esso non sia considerato separatamente dagli elementi che compongono il suo insieme.

Consideriamo il seguente esempio:

AuthorId	Name	Family name	Publication Id	Title Publication	Venue	Year	Affiliation	Co Authors
wrfq	Kwei	Lin	29323	“Web Generation”	GECCO	1995	Xidian University	23r0aw, 3d9jfo2
wrfq	Kwei	Lin	20972	“Ninux”	APSEC	2002	Sichuan University	e238du, 22932r, 298r9e
wrfq	Kwei-Jay	Lin	98374	“Security Guide”	ACM	2009	M.I.T.	rih49d, cj9f90

In fase di preprocessing, l'oggetto autore che viene creato è strutturato in modo da minimizzare il numero di confronti possibili in fase di processing.

Se nessun “lock” fosse impostato, l'oggetto autore risultante avrebbe questo tipo di struttura:

AuthorId	Name	Family name	Publication Id	Title Publication	Venue	Year	Affiliation	Co Authors
wrfq	Kwei Kwei-Jay	Lin	29323, 20972, 98374	“Web Generation”, “Ninux”, “Security Guide”,	GECCO, APSEC, ACM	1995, 2002, 2009	Xidian University, Sichuan University, M.I.T.	23r0aw, 3d9jfo2, rih49d, cj9f90, e238du, 22932r, 298r9e

Come si può notare, non si potrebbe più sapere se la pubblicazione numero 29323 sia del 1995, 2002, o del 2009, in quanto inserite in liste aventi una relazione stretta soltanto con l'id dell'autore.

Se invece impostiamo la variabile *lock* per le pubblicazioni, titoli, venue, year e affiliazioni otteniamo un oggetto del tipo:

AuthorId	Name	Family name	Publication Id	Title Publication	Venue	Year	Affiliation	Co Authors
wrfq	Kwei Kwei-Jay	Lin	29323	“Web Generation”	GECCO	1995	Xidian University	23r0aw, 3d9jfo2, rih49d, cj9f90, e238du, 22932r, 298r9e
			20972	“Ninux”	APSEC	2002	Sichuan University	
			98374	“Security Guide”	ACM	2009	M.I.T.	

In questo modo riusciamo a scegliere quali relazioni tra gli elementi conservare.

Questa modalità è stata inserita per permettere all'utente di poter modellare il dataset, in quanto a modelli diversi corrispondono risultati diversi.

Inoltre CALID permette di poter dare un "*peso*", definito dalla keyword ***"kg"***, al confronto di un singolo elemento, rendendolo ancor più modellabile e incline alle nostre aspettative di utilizzo; questo perché il raggiungimento di un buon risultato dipende da una giusta configurazione del programma e da una precedente e attenta analisi dei dati, in quanto dataset diversi, anche se formati da contenuti simili, differiscono per i motivi descritti precedentemente alla fine del primo capitolo.

Avendo a disposizione le tecnologie di comparazione e i dati da confrontare, bisogna capire come utilizzare logicamente i valori che esse restituiscono.

A tal proposito, la sezione successiva spiega come riuscire ad ottenere dei risultati validi, grazie alla configurazione di ***"Conditions"*** multiple.

2.2.2 Multi Conditions

Nei dataset analizzati disponevo delle informazioni di *nome e cognome* dell'autore, *titolo* della pubblicazione, *anno* e nome della conferenza (*venue*) in cui essa veniva presentata, la *sede dell'affiliazione* dell'autore durante tale conferenza per quella pubblicazione e *lista degli autori* con i quali aveva scritto l'articolo. Inoltre ho utilizzato anche i dati delle *co-venue*, cioè le conferenze alle quali i co-autori avevano partecipato.

Un problema però nasce dal fatto che a volte i dati erano scritti in modo errato, a volte incompleti e altre ancora totalmente mancanti, come nel caso delle affiliazioni.

Per alcuni autori erano disponibili le informazioni di molte pubblicazioni, per altri invece una o due, e per trovare un possibile match, bisognava quindi capire quale tipo di confronti effettuare.

Per questo infatti ho preso spunto dall'idea del metodo *"Metapath-Based Similarity"* per creare le ***"Multi Conditions"***, cioè condizioni per le quali ritenere che una coppia di autori sia un duplicato o meno; appena una di queste risulterà soddisfatta, la coppia di autori sarà dichiarata duplicata.

Una ***Condition*** è sostanzialmente composta da due sezioni:

- ***la sezione IF_SUM:*** composta da un elenco di campi e da un valore minimo tale che se questo viene superato dalla somma dei confronti degli elementi appartenenti ai campi inseriti, allora la coppia di autori verrà accorpata;

- ***la sezione AND:*** opzionale, composta sempre da un elenco di campi e da un valore minimo che determinano le condizioni per le quali la sezione IF_SUM sia valida.

Consideriamo la seguente tabella:

(dati fittizi)

Author	**Name**	**Surname**	**Title (publication)**	**Venue**	**Affiliation**	**CoAuthors**
x001	R.	Johnson	"Web Generation"	GECCO		
x002	Richard	Johnson	"Web Generation"	GECCO	Xidian University	Y004,y006,y007
x003	R.	Johnson			Xidian University	Y004,y002,y005
x004	Richard	McLean			Xidian University	y004,y006,y007

A prima vista, notiamo che la coppia **<x001,x002>** è un potenziale caso di duplicati in quanto i confronti sono validi per *Name*,*Surname*,T*itle e Venue*.

Una *Condition* che può convalidare la coppia è:

```
"IF_SUM":{"name,surname,title,venue":360}
```

Infatti abbiamo un match completo (100) per *Surname*,*Title e Venue*; e un match di 66 per il *Name*.

Per recuperare anche la coppia **<x002,x003>** dobbiamo inserire la *Condition*:

```
"IF_SUM":{"name,surname,affiliation,coAuthors":289}
```

in quanto abbiamo un punteggio di 66 per il *Name*, di 100 per il *Surname* e *Affiliation* e di 33 per *CoAuthors*.

Quindi l'utilità di poter utilizzare "*Condition*" multiple ci aiuta nel superare le difficoltà relative ad errori e/o a mancanze dei dati.

Ma quest'ultima *Condition* creerebbe un legame anche tra la coppia **<x004,x002>** in quanto si avrebbe un punteggio di 100 per *Name*, *Affiliation* e *CoAuthors*. Bisogna quindi inserire una condizione **AND** che permetta che la coppia sia convalidata solo se il cognome superi il punteggio indicato; e cioè:

```
"IF_SUM":{"name,surname,affiliation,coAuthors":283},
"AND":{"surname":100}
```

In questo modo siamo sicuri di escludere la coppia **<x002,x004>** dalla lista dei risultati validi.

Le condizioni AND, anche se a volte potrebbero essere ridondanti, semplificano l'utilizzo

delle *Conditions*, e velocizzano la comprensione del dataset che si sta trattando.

Quindi, in fase di convalida, se la coppia di autori processata rispetta i parametri di almeno una *Condition*, essa verrà memorizzata tra le soluzioni, altrimenti verrà scartata.

Una volta che la fase di processing abbia finito tutti i confronti e determinato i punteggi di similarità, la fase di post-processing, permetterà di salvare le coppie ottenute in modo che siano ulteriormente processate da una successiva riesecuzione del programma (***filtering***), o se concludere recuperando le coppie similari sfuggite al controllo precedente (***finalizzazione***)

2.3 Post-Processing

2.3.1 Filtering

Le coppie giudicate similari secondo la fase precedente, possono essere salvate su file, il quale può essere fornito se si voglia riprocessare il dataset e calcolare i nuovi confronti solo tra le coppie presenti nel file.

L'idea per cui ho creato questa modalità nasce per due motivi: ottenere una maggiore velocità di esecuzione totale e poter salvare lo stato di analisi del dataset nel caso in cui lo voglia sottoporre ad un nuovo confronto.

- ***velocità:*** l'idea di fondo è che il dataset è un insieme di dati, liste e insiemi di liste, e che esse devono essere confrontate a due a due.

  Se volessimo confrontare soltanto due liste di nomi e cognomi, si impiegherebbe poco tempo; ma nel caso in cui volessimo confrontare anche liste di titoli, di anni, di coautori, di covenue o liste di insiemi, in una volta sola, pena la correttezza del risultato, può accadere che ci vogliano anche giorni per dei computer desktop o laptot standard.

  Se però avessimo la possibilità di disporre di un tipo di dato corretto e sempre presente, e il risultato del confronto di questo dato, tra tutte le coppie di autori che in realtà rappresentano la stessa persona, superi un certo valore calcolabile, allora potremmo generare un filtro.

  Nei 2 dataset utilizzati, il dato “cognome” è sempre risultato corretto (a meno della codifica) e presente: quindi preparando il file di configurazione in modo da confrontare le coppie di autori soltanto secondo il dato “cognome”, possiamo ottenere in poco tempo una lista di coppie di elementi con cognome uguale.

Dopo questa procedura, con un dataset di circa 11'000 autori e di circa 60'000'000 di coppie da analizzare, ho ottenuto un file da riutilizzare come filtro che mi ha permesso di poter rianalizzare il dataset soltanto per circa 100'000 coppie riducendo drasticamente il tempo di esecuzione nelle fasi successive.

- ***"salvataggio dello stato":*** nel caso in cui il programma sia stato eseguito e, giunti alla fine, vorremmo ricalcolare le soluzioni sotto altre *Conditions*, sarebbe stressante dover aspettare nuovamente che il programma riesegua tutti i confronti.

  Quindi, se ad ogni esecuzione creiamo un file di filtro, possiamo "salvare lo stato" e, successivamente, CALID effettuerà i confronti a partire dall'ultimo risultato salvato.

2.3.2 Finalizzazione

Le coppie di autori, risultato della fase di processing, devono ora essere analizzate per eliminare del tutto la disambiguazione, dovuta a casi in cui la convalida di alcune sia stata erroneamente o giustamente saltata.

Se la fase di processing ha fornito risultati secondo i quali le seguenti coppie identificano lo stesso autore:

A(x03,y83) A(x03,y29) A(y49,r83) A(y29,g23)

possiamo notare che l'autore *x03* è lo stesso di *y83* e di *y29*, ma la coppia *A(y83,y29)* non compare nelle soluzioni. Anche l'autore *g23* crea un'ambiguità.

Quindi, per risolvere il problema, bisogna aggiungere gli id *y29* e g23 alla prima coppia ed eliminare quelle in cui compaiono. Otteniamo quindi gli insiemi:

A(x03,y83,y29,g23) e **A(y49,r83)**

risolvendo l'ambiguità.

Questa soluzione ha l'effetto di aumentare la *recall* ma anche quello di diminuire la *precision*.

Un modello valido che ci permetta di scegliere quali coppie unificare e quali no, è incluso nello sviluppo futuro del programma, ma comporterebbe sicuramente mantenere in memoria molte più informazioni, comportando un utilizzo improprio di risorse (se si utilizza la memoria volatile) o lentezza (se si opta per il salvataggio su file).

Infine il file dei risultati viene salvato e se presente un file di soluzioni per verificare l'efficienza del programma, viene confrontato con esso e vengono calcolati i valori per ottenere i dati di precision e recall.

Essi comprendono:

- ***TruePositive***: Sono il numero di coppie di autori che CALID reputa valide ed esse lo sono effettivamente, in quanto presenti nel file di confronto.
- ***FalsePositive***: numero di coppie di autori che CALID reputa valide, mentre in realtà non lo sono.
- ***FalseNegative***: numero di coppie che CALID non ha considerato come valide, ma che invece lo sono perché presenti nel file delle soluzioni.

Una volta ottenuti questi dati, la misura di precisione viene calcolata come:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

e quella di recupero come:

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

Nel capitolo successivo, spiegherò alcune sezioni di codice del programma, utile per gli sviluppi futuri, e per successive modifiche da parte di altri programmatori.

Capitolo 3

In questo capitolo descriverò nel dettaglio la struttura concettuale e l'implementazione del programma, con annessi le tecnologie e gli algoritmi utilizzati.

Implementazione, tecnologie e algoritmi

CALID è un programma scritto in Python 2.7 attualmente soltanto per Linux.

Esso riceve in input un file di configurazione e un dataset, descritto in tale file, entrambi in linguaggio JSON, l'ultimo dei quali corrisponde allo “SPARQL 1.1 Query Result Json Format” (http://www.w3.org/TR/sparql11-results-json/) .

Esso è composto da 6 files principali:

- ***calid.py*** **–** file contenente la funzione *main* che permette l'iniziale lettura del file di configurazione e del dataset, passando poi il comando alla funzione dei confronti.
- ***config.py*** **–** all'interno vi sono le funzioni che leggono e verificano la correttezza del file di configurazione, pena la terminazione del programma con messaggi di aiuto per l'utente.
- ***reader.py*** **–** permette la lettura e la preparazione del dataset ai confronti: è il *core* della fase di pre-processing.
- ***runner.py*** **–** è il core della fase di *processing* e *post-processing*; organizza il lavoro richiamando la funzione per i confronti.
- ***engine.py*** **–** è il file che contiene tutti i metodi di confronto: ritorna un dizionario con i valori di similarità associati.
- ***utilities.py*** **–** contiene una lista di funzioni generiche utili e comuni per il funzionamento di tutto il programma.
- ***conf.help.json*** **–** file di configurazione richiamabile tramite ***“python calid.py –help”***

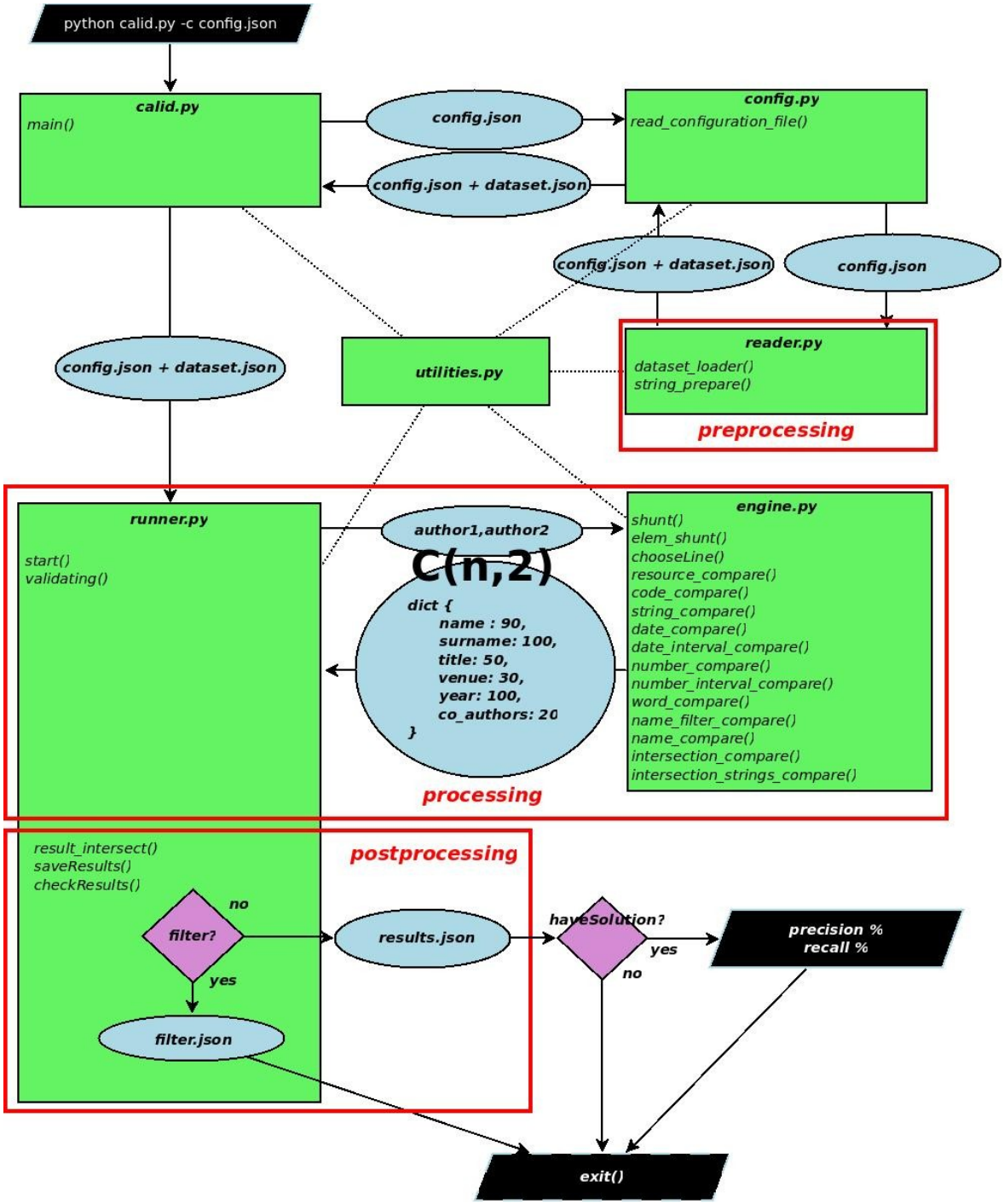
Librerie particolari utilizzate sono:

- ***itertools*** **–** contenente strumenti semplici per creare iteratori di combinazioni di oggetti
- ***json*** **–** permette la traduzione di file “*json*” in semplici *dizionari* python *(dict)*
- ***Levenshtein*** **–** contenente la funzione “*distance*” per il calcolo della distanza di

Levenshtein tra due stringhe

La struttura interna e il funzionamento del programma è esemplificato dal seguente modello grafico:

struttura, files e funzioni di CALID



In questo modello viene mostrata la struttura di CALID, i suoi 6 files, il loro legame e le funzioni significative che essi contengono.

Tutti i passaggi di dati tra files non influiscono sul costo computazionale del programma, in quanto Python è un linguaggio che utilizza il *passaggio di variabili per riferimento*; solo in alcuni casi è stato necessario effettuare delle copie di dati, per preservare gli originali da modifiche che sarebbe stato necessario attuare.

Inoltre, l'utilizzo dei dizionari è stato sempre prioritario rispetto alle liste, perchè molto più veloci, visto che la ricerca di un elemento avviene tramite hash-map al contrario di una ricerca iterativa, molto più lenta.

Visto che nella *start()* il numero di confronti da fare è del tipo $O(n^2)$ ho preferito anticipare, durante la preparazione del dataset, anche le più piccole operazioni (tipo *splitting*, *replace*) riducendo significativamente il tempo totale di esecuzione.

3.1 Componenti del programma

Analizziamo ora i singoli files e le operazioni principali che essi compiono, rispettando il flusso di esecuzione del programma:

3.1.1 calid.py

E' il file di avvio, contenente la funzione *main()* che effettua il controllo dei parametri in ingresso e richiama le funzioni per il controllo del file di configurazione e del dataset in esso indicato; una volta effettuati i controlli e finita la preparazione, chiama la funzione *start()* in *runner.py*, cuore del programma, destinando ad essa la configurazione e i dati da processare.

Esso viene richiamato tramite terminale con il comando:

```
CALID$ python -c <configuration_file>
```

Durante il controllo dei parametri passati da riga di comando, durante la chiamata del programma da terminale, la *main()* effettua un controllo sull'effettiva presenza del file di configurazione, restituendo in alternativa un messaggio di errore.

Inoltre, per aiutare l'utente ad utilizzare il programma, ho inserito un'opzione richiamabile tramite "***python calid.py –help***" che stampa sul terminale un esempio di file di configurazione, che l'utente può modificare e riutilizzare:

```
CALID$ python calid.py --help
Esempio di un possibile file di configurazione:
{
   "COMMENT": "Configuration file for CALID - Customizable Application
for Literals and Iri's Disambiguation",
```

```
"PRIMARY_KEY": "person",
"DATASET": "dataset/dataset.json",
"KEYS": {
     "family_name":{
         "group_concat": null,
         "kg": 9,
         "lock": null,
         "type": "word",
         "accuracy": null
     },
     "document_name":{
         "group_concat": "|",
         "kg": 2,
         "lock": "pub",
         "type": "intersection_string",
         "accuracy": null,
         "accuracy_string": 80
     },
     "given_name":{
         "group_concat": null,
         "kg": 2,
         "lock": null,
         "type": "name",
         "accuracy": null
     },
     "affiliation":{
         "group_concat": "|",
         "kg": 2,
         "lock": "pub",
         "type": "string",
         "accuracy": null
     }
 },
 "CONDITION" : [
             {
                 "IF_SUM" : {
                     "family_name,given_name": 166,
                 },
                 "AND" : {
                     "given_name": 66,
                     "affiliation" : 95,
                     "document_name" : 20
                 }
             }
         ],
"MODE" : "CONDITION",
"TEST": false,
"FILTER" : true,
"OUTPUT_FILTER" : false,
"FILTER_FILE" : "filter/semantic_filter.json",
"OUTPUT_FILTER_FILE" : "filter/phase3_precision.json",
"WRITE" : true,
"OUT_DIRECTORY": "testing",
"OUTPUT_FILE": "testing/semantic_test.json",
"ANSWERS" : true,
"ANSWERS_FILE": "dataset/semanticLancet/sameauthors.json",
```

```
   "CHECK_ANSWERS" : true,
   "INTERSECT" : true,
   "VERBOSE": true
}
```

In questo file di configurazione sono presenti le impostazioni generali, quelle relative alle chiavi di confronto, e quelle relative alle *conditions*.

Tra le impostazioni generali sono presenti le seguenti chiavi:

- ***PRIMARY_KEY :*** indica il nome del campo utilizzato per identificare un autore
- ***DATASET :*** indica il percorso dove è stato memorizzato il file json contenente il dataset
- ***MODE :*** se impostato a ***CONDITION*** permette l'utilizzo delle *condition multiple* definite tramite la chiave omonima, altrimenti, se impostato a ***SUM***, permette un utilizzo facilitato per cui la similarità tra due autori è ottenuta tramite semplice somma dei valori ottenuti. (SUM è obsoleto)
- ***FILTER, FILTER_FILE, OUTPUT_FILTER, OUTPUT_FILTER_FILE :*** indicano l'abilitazione/disabilitazione dei filtri in input e in output con i rispettivi file di ingresso e uscita.
- ***WRITE, OUT_DIRECTORY, OUTPUT_FILE :*** abilita/disabilita il salvataggio dei risultati nella directory e file indicati (non riguarda i filtri).
- ***ANSWERS, ANSWERS_FILE, CHECK_ANSWERS :*** abilita/disabilita la lettura del file delle soluzioni indicato, e combinato con ***VERBOSE***, stampa sullo schermo i risultati istantanei di *precision* e *recall*
- ***VERBOSE :*** se ***true*** stampa il tempo rimanente al termine del programma; combinato con ***ANSWERS*** visualizza le informazioni qualitative/quantitative istantanee approssimate.
- ***INTERSECT :*** se abilitato, la fase di post-processing recupera le coppie di autori sfuggite nella fase di processing; è utile disabilitarla nel caso di fasi intermedie che porterebbero a calcolare intersezioni inutili di numerose coppie di autori, con conseguente aumento di tempo e memoria.
- ***KEYS :*** contiene l'elenco dei campi da considerare per il confronto.

All'interno della direttiva ***KEYS***, in aggiunta a quanto detto nel capitolo precedente, la ***"group_concat"*** permette di utilizzare i dati che sono stati concatenati tramite l'omonima direttiva nella *query sparql* scritta per ottenere il dataset, inserendo il carattere separatore

usato.

“kg” indica il valore intero o decimale da moltiplicare per alterare il risultato ottenuto dal confronto.

A tal proposito, ricordo che il risultato ottenuto da ogni confronto è in forma percentuale, quindi bisogna prestare attenzione ad utilizzi impropri della chiave in questione.

La direttiva ***“lock”*** indica l'appartenenza del campo ad un insieme. Tutte le chiavi, aventi uguale valore secondo questa direttiva, sono esplicitamente relazionate tra loro.

La ***“type”*** indica il tipo di confronto che utilizzeremo per quel campo (vedi precedente capitolo).

La keyword ***“accuracy”***, se utilizzata, permette di definire il minimo valore percentuale secondo il quale il confronto vale o non vale. E' opzionale, ma comodo durante l'analisi dei dati per determinare le *condition* da inserire.

Nel caso in cui il valore del campo “type” è ***“intersection_strings”***, è necessario inserire la direttiva ***“accuracy_string”*** che determina per quale minima percentuale il valore determinato dal confronto tra stringhe produce un'intersezione.

Una volta impostate le chiavi generali e quelle relative ai campi per il confronto, bisogna inserire le *conditions*; la keyword ***“CONDITION”*** corrisponde ad una lista contenente più strutture con campi ***IF_SUM*** e ***AND*** all'interno (a tal proposito si veda il capitolo precedente). Se un campo utilizza la direttiva *lock*, esso deve essere indicato nella forma: ***valoreLock#nomeCampo***; più campi possono essere inseriti, separati con il carattere ***“,”***.

3.1.2 config.py

Il file di configurazione indicato all'avvio del programma, viene passato alla funzione “*read_configuration_file()”* che ne verifica la correttezza, terminando, in caso contrario, il programma con un messaggio indicante l'errore commesso. Questo facilita notevolmente l'utilizzo del programma.

3.1.3 reader.py

Una volta che il file di configurazione viene letto e convalidato, la funzione *dataset_loader()* si preoccupa di analizzare e preparare i dati contenuti nel dataset per essere direttamente confrontati.

Essa identifica la fase di preprocessing di CALID, durante la quale i dati, oltre ad essere processati, sono ristrutturati in dipendenza alle direttive inserite nella *KEYS*, in modo da ottenere un oggetto struttura “autore” quanto più compatto e semplice possibile, per

minimizzare il numero di confronti mantenendo la correttezza delle relazioni tra i suoi valori.

Tra i metodi di confronto permessi, è interessante conoscere l'algoritmo dei seguenti:

- ***string,intersection_strings :*** poiché questi due metodi utilizzano il coseno di similitudine per calcolare la similarità tra due stringhe, e poiché per ogni stringa interessata nel confronto bisogna calcolarne un vettore contenente per ogni parola il numero di volte in cui questa compare all'interno della stringa, ho utilizzato la classe ***Counter*** della libreria ***"collection"*** che appunto permette di creare un oggetto vettore, associato alla stringa da processare, contenente tali valori.

```
x=str()
v=[]
v = collection.Counter(x)
```

- ***name :*** secondo questo metodo di confronto, in fase preparativa viene creata una struttura dizionario di 2 liste, contenenti da un lato le parole del nome estese, e dall'altro le abbreviazioni, riconoscibili dalla loro lunghezza (inferiore o uguale a 2) oppure dalla presenza del carattere "." alla fine della parola (a meno di errori di scrittura)

  La struttura di esempio è:

```
name={}
name["long"]=["J"]
name["short"]=["Fiztgerald","Kennedy"]
```

- ***name_filter :*** questa modalità, usata principalmente nella fase di *filtering*, permette di processare un nome, riscrivendolo in più forme, abbreviandone le parole, unendole e permutandole, per riuscire a ricostruire eventuali errori commessi in fase di inserimento e recuperare possibili coppie di autori i cui nomi, durante il confronto successivo, porterebbe invece a scartarle.

  Interessanti sono gli algoritmi di unione e permutazione riportati qui di seguito, definiti internamente alla ***name_filter().***

```
# Funzione di supporto per combinare le unioni dei nomi che le vengono
passati
    def xjoin(v,index,length):
        if(index<0 or length==0 or index+length>=len(v)):
            return False
        s=str()
        n_v=list()
        if(index>0):
            n_v=v[:index]
```

```
        for i in range(index,index+length+1):
            s+=v[i]
        n_v.append(s)
        n_v+=v[index+length+1:]
        return n_v


# Funzione di supporto che invia i nomi da processare alla x_join
    def iterjoin(v):
        j_v=list()
        i=0
        while(i<len(v)):
            j_v.append(v[i])
            j=i
            if(j==0):
                j+=1
            while(j+i<len(v)):
                j_v.append(xjoin(v,i,j))
                j+=1
            i+=1
        return j_v


# Funzione che crea una lista di nomi permutati e chiama la iterjoin.
    def create_permutaded_list(list_name):
        tmp_list=list()
        names_permutations=permutations(list_name)
        for np in names_permutations:
            tmp_list.append(list(np))
        out_list=list()
        out_list+=tmp_list
        for np in tmp_list:
            out_list+=iterjoin(np)
        return out_list
```

In questo file sono definite anche due strutture contenenti le liste dei diminutivi dei nomi inglesi, e un elenco di parole che in fase di confronto tra stringhe possono essere scartati in quanto potrebbero portare a confronti inesatti, come articoli, avverbi, pronomi, ecc.

3.1.4 runner.py

All'interno di questo file si svolgono le fasi di processing e quelle di post-processing.

Le funzioni principali sono 3: ***start()***, ***validating()***, e ***result_intersect()***, rispettivamente riguardanti la procedura per il confronto, per convalidare il risultato tramite le *conditions* e per recuperare le coppie perse durante il processing.

- ***start() :*** Questa funzione permette al programma di selezionare le coppie di autori che devono essere confrontate. A tal proposito, viene usata la funzione ***combinations()*** della libreria ***itertool***, capace di generare un iteratore delle

combinazioni degli oggetti autore in classe 2. In questo modo si ha la certezza di confrontare una coppia una singola volta.

Durante l'iterazione, se nel file di configurazione si è indicato un file di filtro, la funzione permetterà di eseguire il confronto soltanto tra le coppie indicate in esso, chiamando la funzione ***shunt()*** definita in **engine.py**, e convalidando la coppia tramite la ***validating()***.

Se un file di soluzioni è stato inserito in fase di configurazione, la coppia verrà controllata confrontandola con quelle in esso presenti, stampando un messaggio in tempo reale se la modalità ***VERBOSE*** è attiva.

- ***validating() :*** convalida la coppia ricevendo in input una struttura con i valori dei confronti corrispondenti ai nomi di campo indicati, analizza le *conditions* e se una di esse risulta vera, restituisce il booleano ***True***, altrimenti ***False.***
- ***result_intersect() :*** secondo quanto detto nel capitolo precedente riguardo la fase di postprocessing, la *result_intersect()* calcola le intersezioni degli insiemi di tutte le coppie convalidate e se trova degli identificatori in comune le unifica, costruendo un insieme unico contenente gli identificatori processati.

Infine i risultati vengono salvati su file in formato json, come una lista di liste di autori duplicati e nel caso in cui si ha un file si soluzioni, queste liste verranno confrontate con tale file, determinando i valori finali di *precision* e *recall*.

L'ultimo file da analizzare è quello contenente la funzione *shunt()* in *engine.py*, “core” dei confronti e della disambiguazione che CALID permette di ottenere.

3.1.5 engine.py

Il file contiene due funzioni principali, la *shunt()* e la *elem_shunt()* che permettono lo “smistamento” degli elementi appartenenti agli oggetti autori, verso le funzioni indicate per il confronto.

- ***shunt() :*** essa riceve in input due oggetti autore, le chiavi da confrontare e il file di configurazione, e per ogni coppia di elementi controlla il nome e il tipo del campo a cui appartengono e decide se trattarli come due liste, o insiemi o elementi singoli.

  Nel caso siano due elementi singoli, verrà chiamata direttamente la *elem_shunt()* che si preoccuperà di invocare il giusto metodo di confronto; nel caso in cui gli elementi siano liste, per tutti i confronti ottenibili dalle combinazioni delle coppie degli elementi appartenenti ad esse, verrà chiamata la *elem_shunt()* e calcolate le loro similarità, prediligendo, come risultato finale, quella di valore maggiore.

  Nel caso siano elementi appartenenti ad insiemi indicati nel campo *lock* nel file di

configurazione, per ogni coppia di ugual genere, sia essa coppia di elementi singoli o coppia di liste, la funzione verrà richiamata ricorsivamente e per ogni chiamata verranno ricontrollati i valori delle coppie restituendo infine i risultati richiesti.

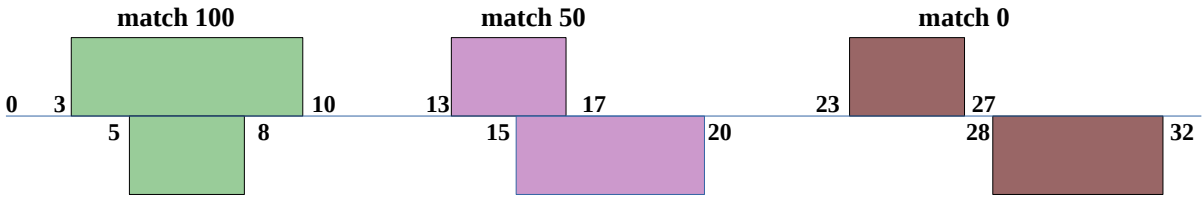
- ***elem_shunt() :*** si preoccupa di associare gli elementi ricevuti in input dalla funzione *shunt()* e di inviarli alle giuste funzioni di confronto, che si preoccuperanno di calcolarne il punteggio di similarità. Nel caso di valori nulli, essa restituirà direttamente il valore 0, senza proseguire al confronto.

Tutte le altre funzioni presenti in questo file contengono i metodi di confronto.

Esse sono:

- ***resource_compare() :*** è una semplice comparazione tra stringhe non modificate in fase di preparazione. Restituisce **100** in caso di uguaglianza o **0** altrimenti.
- ***code_compare() :*** identica a *resource_compare()*
- ***string_compare() :*** riceve in input due oggetti ***Collection*** contenenti parole prive di spazi e/o simboli, componenti le due stringhe da confrontare. Esse verranno unite e per ognuna calcolato il vettore contenente il numero delle volte in cui, una parola del vettore unione, è presente in essa.

  Infine verrà calcolata il valore di similarità secondo la formula del coseno di similitudine, descritta nel capitolo precedente, e poi convertito in percentuale.
- ***number_compare() :*** è un normale confronto tra numeri, e restituisce **0** in caso di match negativo e **100** in caso positivo.
- ***number_interval_compare() :*** è una funzione che viene utilizzata se la chiave *interval*, nel file di configurazione, contiene un carattere speciale utilizzato per separare l'intervallo. La funzione utilizza i semplici operatori di confronto “<”,”>”,”==”,”<=”,”>=” per calcolare le intersezioni tra gli intervalli, restituendo 100 se un insieme contiene l'altro, 50 se vi è intersezione ma senza contenimento, 0 se non esistono intersezioni tra gli insiemi.



- ***date_compare() :*** la funzione riceve in input due oggetti ***date***, controllate in fase di preparazione tramite la funzione ***“strptime”*** della libreria ***“time”*** per verificarne la

validità, e restituisce un valore percentuale di uguaglianza o disuguaglianza, quindi **0** o **100**.

- ***date_interval_compare() :*** la funzione riceve in input due coppie di oggetti date, controllati sempre in fase di preparazione, e tramite i normali operatori di confronto, come per la funzione *number_interval_compare()*, viene calcolata la similarità nel caso di contenimento, intersezione positiva, o intersezione nulla.
- ***word_compare() :*** importa la liberia ***"Levenshtein"*** per utilizzare il metodo "***distance"***, che calcola la distanza di Levenshtein tra le due stringhe in input.

  Esse vengono processate come stringhe generiche, senza tener conto se siano composte o meno da più parole. Infatti la funzione è indicata per confrontare singole "word". Il risultato restituito è definito, in percentuale, come il reciproco della distanza, "ammorbidito" dal fattore numerico **0.6** per poter avere un valore che diminuisce più lentamente all'aumentare della distanza.

  La formula utilizzata è:

$$\begin{cases} 0 & \text{per} \quad distance = 0 \\ 100 \cdot \dfrac{1}{0.6 * distance} & \text{per} \quad distance > 0 \end{cases}$$

- ***name_filter_compare() :*** riceve in input due liste di stringhe e restituisce **100** se almeno una delle coppie confrontate risulta identica, altrimenti **0**.
- ***name_compare() :*** riceve in input due dizionari, ognuno strutturato in due campi come indicato precedentemente, durante la descrizione della fase preprocessing del file ***reader.py***.

```
{
   "short" : [ "n", "n2", "n."] ,
   "long" : ["name1", "name2"]
}
```

La funzione è costituita da 6 cicli *while*, ognuno utilizzato per verificare l'esistenza di match tra i nomi estesi (*long-long*) e i diminutivi degli stessi (*long-diminutive* e *diminutive-long*), tra i nomi in forma estesa e quelli in forma abbreviata (*short-long e long-short*), e tra i nomi entrambi in forma abbreviata (*short-short*). Vengono anche calcolati i possibili match tra i nomi in forma abbreviata e i possibili diminutivi dei nomi in forma estesa (*short-diminutive* e *diminutive-short*).

Una volta avuti i risultati di questi confronti, viene restituito il risultato finale, in termini percentuali, secondo la formula:

$$0 \qquad n_x = n_y = 0$$

$$100 * \frac{long + 0.8 \cdot short}{max(n_x, n_y)} \qquad max(n_x, n_y) > 0$$

con:

- long = numero di match tra parole estese
- short = numero di match tra parole abbreviate o tra parole estese con le abbreviate
- n_x, n_y = numero di elementi totali dei nomi da confrontare

- ***intersection_compare() :*** riceve in input due liste di risorse, numeri, codici o stringhe e ne controlla l'esatta intersezione, restituendo un valore compreso tra **0** e **100** secondo la formula:

$$0 \qquad n_x = n_y = 0$$

$$\frac{n_{intersezioni}}{min(n_x, n_y)} \qquad min(n_x, n_y) > 0$$

  - $n_{intersezioni}$ = numero di intersezioni trovate tra le due liste
  - n_x, n_y = lunghezza delle due liste

- ***intersection_strings_compare() :*** riceve in input due liste di stringhe e un intero che ne stabilisce il livello di accuratezza che bisogna rispettare affinché due stringhe possano far parte dell'intersezione.

  Durante il calcolo dell'intersezione, per ogni coppia di stringhe, calcolata come combinazione delle due liste, viene restituito un valore risultante dal confronto delle stesse tramite il coseno di similitudine, e se questo è maggiore o uguale al livello di accuratezza definito, il numero che definisce l'intersezione viene incrementato di 1; dopo l'intero processo, tale numero viene diviso per il valore minimo delle lunghezze delle due liste e restituito alla funzione chiamante.

3.1.6 utilities.py

Infine, il file **utilities.py** contiene poche ma utili funzioni:

- per caricare un file ***"json"***, o salvarlo, dove è stata aggiunta la funzionalità di poter decidere se sovrascrivere o cambiare nome al file, nel caso in cui tale file sia già presente
- data una ***"date"***, ne viene controllata la sua validità temporale, tramite il metodo ***"strptime"*** contenuto nella libreria ***time***.

Capitolo 4

Valutazione e Test

CALID è un programma costruito in base alle esigenze di dover risolvere il problema della disambiguazione degli autori, basandosi su due tipi di dataset, di grandezza e “difficoltà” diverse.

Inizialmente CALID permetteva di effettuare confronti di campi prestabiliti, come nome, cognome, titolo pubblicazione, anno, lista di coautori utilizzando dei metodi comparativi prestabiliti e non proponeva nessuna pre-elaborazione del dataset in input. L'utente poteva “agire” sul programma soltanto utilizzando i parametri di “peso” per riuscire ad ottenere un risultato valido, e di conseguenza, questa configurazione produceva valori molto bassi sia di precision sia di recall, in quanto bisognava, per ogni tipo di dato, utilizzare un confronto appositamente studiato ed aumentare i tipi di dati da confrontare; inoltre l'algoritmo risultava lento, perché non conteneva nessuna fase preparativa del dataset che ristrutturasse e comprimesse al minimo possibile gli elementi “autore” aventi lo stesso identificatore; infatti nel momento in cui ho utilizzato un dataset di lunghezza maggiore, il tempo che CALID impiegava per poter risolvere la disambiguazione, passava dalle 2/3 ore alle “settimane”, improponibile poi se il programma in futuro avesse dovuto trattare dataset dell'ordine di gigabyte.

La priorità di dover raggiungere un risultato valido in termini di precision e recall di almeno 70%/70% mi ha portato ad immaginare i tipi di dati che si potessero incontrare nell'ambiente della letteratura e di inserire funzioni mirate alla gestione degli stessi, dando la possibilità all'utente di poter scegliere quando e quali di questi metodi di confronto utilizzare.

Questa scelta implementativa mi ha portato ad ottenere una precision e una recall di 82% / 83% sui dati temporaneamente testati, ma ha decisamente aumentato il tempo di esecuzione, in quanto metodi di confronto come la distanza di Levenshtein e la funzione coseno di similitudine portavano il programma ad essere molto lento: CALID non era ancora una soluzione proponibile.

Il passo successivo mi ha dunque portato a ristrutturare e comprimere il dataset a run-time, pre-processando il dataset e facendo in modo che anche le più piccole e semplici operazioni siano fatte nella fase preparativa, lì dove il costo computazionale sarebbe stato di O(n).

In questo modo i tempi sono diminuiti di molto, passando dalle 6/7 ore alle 2/3 ore: CALID iniziava a definirsi un programma valido. Era arrivato il momento di testarlo su un dataset “diverso”.

Notai, però, che i risultati ottenuti con questo nuovo dataset erano non soddisfacenti, infatti riuscivo ad ottenere dei valori che non superavano il 50% in entrambi i termini, e, vista la maggiore quantità di dati, il tempo di esecuzione mi portava a poter rieffettuare nuovi test ogni 6/7 ore, soprattutto nel momento in cui aumentavo la lista dei campi del dataset da processare.

Dopo numerose, ma inconcludenti prove, ho potuto notare che i dati riguardanti i cognomi degli autori potevano aiutarmi ad eliminare almeno le coppie di autori sicuramente diversi; infatti applicando CALID soltanto su questo tipo di dato, ho ottenuto dei risultati che mi portavano ad avere una precision infinitesimale, ma una recall del 100%: tutte le coppie degli autori duplicati erano state salvate nel file delle soluzioni, senza nessuna perdita.

Visto che il programma, quando ha effettuato questo confronto, ha impiegato circa 40 minuti, e visto che il numero delle coppie con cognome diverso, scartate da CALID, erano di circa 64'500'000 su 65'000'000, ho rieseguito l'applicazione soltanto su quelle coppie rimaste, circa 500'000, applicando i confronti dei restanti campi disponibili.

Il risultato è stato sorprendente: il programma ha impiegato 1 minuto e 20 secondi per analizzare le coppie e restituire un risultato.

Ora bisognava soltanto concludere e migliorare i risultati, in modo da poter raggiungere una precision e una recall entrambe di almeno 80 punti percentuali.

Grazie all'inserimento del sistema che convalida i risultati con *conditions* multiple, sono riuscito a raggiungere valori di precision e recall di 80%/85% e, rieseguendo il programma sul dataset precedente, ho ottenuto invece una precision del 97% e una recall del 85%, che ha finalmente dimostrato che CALID è un valido tool per risolvere il problema della disambiguazione degli autori, in tempi contenuti.

Di seguito tratterò l'analisi del costo computazionale dell'algoritmo e mostrerò, nel dettaglio, i risultati descritti in precedenza, per entrambi i dataset utilizzati.

4.1 Costo computazionale dell'algoritmo

Il calcolo esatto del costo computazionale dipende dalla complessità generica delle varie fasi, ma anche da alcune principali variabili come la lunghezza del dataset e la quantità dei campi da utilizzare nei confronti.

Per i riferimenti successivi, indicherò con **N** il numero di record prodotti da una query sparql, con $\mathbf{n_o}$ il numero di oggetti autore ricavati nella fase di preprocessing, e con **c** il numero dei campi invitati al confronto.

La prima fase di CALID, processa *N* record e per ognuno di essi prepara *c* elementi, tanti quanti sono i campi inseriti nel file di configurazione che partecipano ai confronti.

Per ognuno degli elementi di un singolo campo vengono applicate le funzioni di preparazione che hanno quasi tutte un costo pari a O(n) dove *n*, in questo caso, indica il numero delle parole che compongono una stringa.

Nel caso in cui si voglia modificare l'elemento per essere utilizzato dal metodo *name_filter()*, la funzione che riceve in input un nome composto da n_w parole, ne processa $n_w+n_d+n_s$, dove n_d è il numero dei diminutivi corrispondenti, uguale al massimo ad n_w, e n_s il numero delle parole abbreviate calcolate del totale, senza ripetizioni, anch'esso identificato al massimo con n_w; semplificando abbiamo un costo di $O(3n_w)$. Vengono poi calcolate le $n_w!$ permutazioni secondo le quali le parole possono essere riordinate e poi unite per rilevare eventuali errori; considerate, per ogni permutazione, n_p parole, con $n_p>1$, ed n_p-1 spazi tra esse, si ottiene una complessità dell'intera funzione, uguale a

$$3n_w! \sum_{k=2}^{n_w-1} C_{(n_p-1,\,k)}\, n_p$$

ed in totale, la fase di preparazione avrà quindi una complessità massima di:

$$N\left(c+3n_w! \sum_{k=2}^{n_w-1} C_{(n_p-1,k)}\, n_p\right)$$

Riepilogando, la fase di preparazione su un dataset contenente circa 11'000 autori, termina la sua esecuzione in un tempo compreso tra i 5s e i 10s; ma se si vogliano processare nomi composti da 3 parole, i tempi cambiano considerevolmente, arrivando a toccare dai 60s ai 120s.

Durante la fase di *processing* invece, il costo dell'algoritmo varia a seconda del numero delle coppie di autori da processare, e della lunghezza del numero di elementi aventi ogni campo.

Più precisamente, i confronti da effettuare sono le combinazioni $C(n_o,2)$ e per ognuna di esse vengono analizzati *c* elementi, quanti sono i campi da considerare; durante il confronto, poiché ogni elemento appartenente ad un campo è una lista, vengono confrontate $n_{l1}*n_{l2}$ coppie. La complessità massima sarà:

$$c\, n_{l1} n_{l2} C\left(n_o,2\right)$$

Infine, la fase di postprocessing effettuerà due cicli annidati per recuperare le coppie di autori e incrementare la recall, creando un costo del tipo $O(n_r^2)$ con n_r uguale al numero degli insiemi di autori che vengono creati durante la convalida.

La somma dei costi delle 3 fasi producono un risultato che varia fortemente dalla quantità dei dati, e dal modo in cui essi sono confrontati, portando il programma a terminare la sua esecuzione a volte in alcuni minuti, a volte in 2/3 ore.

Riporto, qui di seguito, i test da me effettuati con diversi file di configurazione su diversi dataset, mostrando i tempi di esecuzione in dipendenza del numero dei dati e dei metodi di confronto eseguiti su di essi.

4.2 Semantic Lancet Dataset Test

Il dataset ricavato dal triple-store RDF, Semantic Lancet [BCINPV14] tramite query sparql, tratta 919 autori di pubblicazioni scientifiche, senza omonimi, in quanto i diversi nomi propri di persona di un autore, sono indicati come lista all'interno dello stessa risorsa che lo identifica.

L'algoritmo è riuscito a produrre il 97% di precision e l'85% di recall in 47 secondi, considerando la fase di preprocessing.

La seguente tabella mostra come il programma si è comportato su una macchina Intel Core i7 Quad_Core 2.80GHz e 8 GB DDR3 di RAM, eseguendo CALID due volte per un tempo totale di 47 secondi.

	Precision %	***Recall %***	***TP***	***FP***	***FN***	***Tempo***
Filter	23	100	34	111	0	45 s
Conditions1 OR Conditions2	97	85	29	1	5	2 s

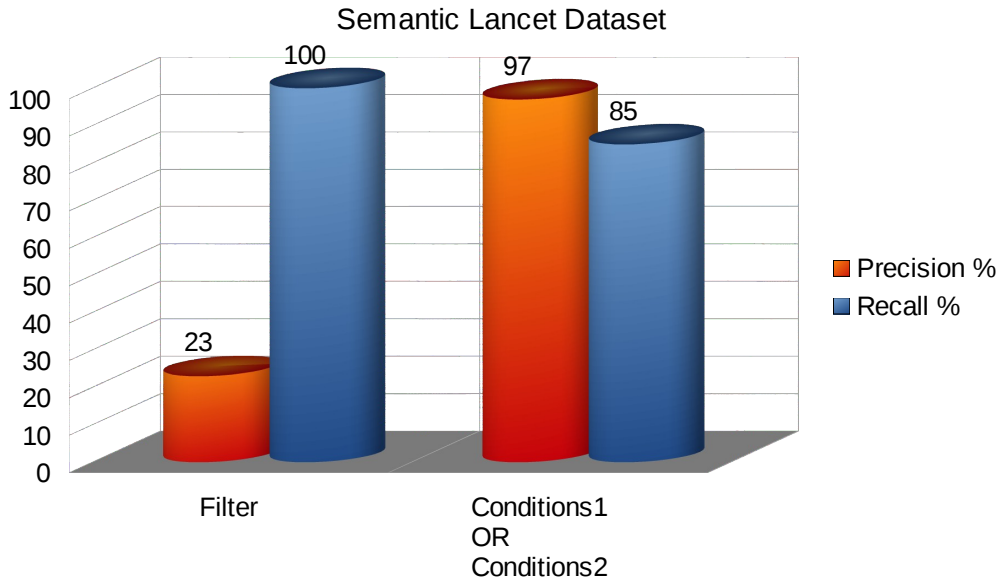
Da notare come la prima esecuzione di *filtering* produce una *Recall* del 100%, configurando il programma in modo da calcolare la similarità dei soli cognomi con il metodo ***"word",*** indicante il confronto tramite la distanza di Levenshtein, e una sola *condition* che recupera le coppie di autori il cui cognome presenta similarità massima.

Il file di configurazione utilizzato nella seconda fase permette di confrontare i campi relativi al nome proprio *(given_name)* dell'autore tramite la funzione ***"name"*** e il titolo della sua pubblicazione *(title)* tramite la funzione ***"string"***, e recupera, secondo la *condition1*, gli autori aventi almeno una parola estesa *(long-name)* del nome in comune, oppure, secondo la *condition2*, gli autori aventi almeno tutte le iniziali del nome in comune e almeno una pubblicazione che faccia match al 100%.

Durante questo test, CALID ha recuperato 29 coppie di autori su 34, e scartato le 5 restanti, andando a pesare di più sulla *recall*, dovute soprattutto ad incompatibilità del nome proprio, in quanto presentavano il caso in cui uno dei due era abbreviato utilizzando la prima lettera e una consonante interna del nome intero.

Il grafico ottenuto evidenzia come nella prima fase di filtering il programma recuperi tutte

le coppie di autori aventi uguale cognome, e scarti nella seconda fase gli omonimi che presentano bassa similarità, fino ad arrivare ad una *precision* massima del 97% e una *recall* di 85%.



4.3 DBLP Dataset Test

Il dataset in questione è ricavato da dati provenienti da *"DBLP computer science bibliography"*, all'indirizzo *http://dblp.uni-trier.de/db/*, contenente 11724 autori e 7447 pubblicazioni.

Con questo dataset, l'algoritmo è riuscito ad estrarre le coppie di autori con una precision del 71% e una recall del 85%, in poco meno di un'ora, utilizzando 2 fasi di filtering sul cognome e sul nome, e una finale sui campi riguardanti *nome*, *affiliazione* e *coautori*.

La tabella seguente illustra i progressi ottenuti nelle varie fasi e per ognuna il tempo impiegato:

	Precision %	***Recall %***	***TP***	***FP***	***FN***	***TN***	***Tempo***
Filter	~0	100	891	524203	0	64532000	56 min 34 s
Filter	0,87	100	891	102207	0	675184	2 min 20 s
Condition1 OR Condition2	71	85	757	309	134	97797	*1m 30s*

Nella prima fase di filtering, CALID ha recuperato tutte le coppie di autori aventi uguale cognome, con una *recall* del 100% ma *precision* vicina allo 0, impiegando 57 minuti per

terminare i confronti.

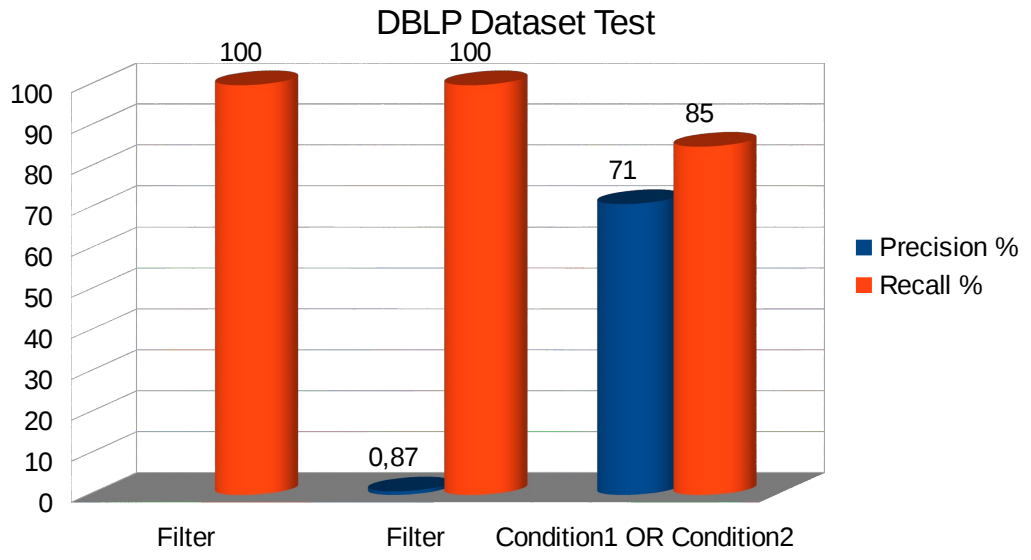
Nella seconda fase di filtering, CALID ha recuperato, le coppie rimanenti che presentavano similarità 100 nel nome proprio, secondo la funzione *name_filter,* in circa 2 minuti, ottenendo sempre una *recall* del 100%.

Nell'ultima fase, CALID effettua il confronto finale e convalida i valori tramite due *conditions:*

- a prima recupera tutti gli autori aventi iniziale del nome proprio uguale, e almeno un coautore in comune, indicando 66% come valore minimo di match sul given_name e l'1% sui coautori , utilizzando rispettivamente i metodi *name* e *intersection*.
- la seconda invece recupera gli autori aventi nome, in forma estesa, identico e almeno un'affiliazione in comune, ottenuta immettendo, come valori minimi di match, il 100% su entrambi i campi, confrontati tramite le funzioni *name* e *string*.

In quest'ultima fase, CALID ha recuperato 757 coppie di autori su 891 e scartato erroneamente 134, con conseguente recall totale del 85%; la *precision* invece è risultata il 71% in quanto il programma ha erroneamente recuperato anche 309 coppie di falsi duplicati.

Anche in questo caso il grafico mostra come le fasi di filtering mantengano al 100% la *recall*, e la fase successiva invece si incentri sulla *precision*, scartando le coppie di autori omonimi.



Conclusioni

I problemi incontrati durante lo sviluppo di CALID durante la sua creazione mi ha permesso di capire come essi dovevano essere affrontati. Di conseguenza, creare algoritmi adatti e veloci mi ha fornito una sempre più chiara idea delle difficoltà che possono sorgere durante l'analisi di un qualsiasi dataset.

Avendo a che fare con errori, mancanze, e carenze di dati, il programma è dovuto crescere ed espandersi al minimo problema incontrato, risolvendolo ma incontrandone altri, alcuni per i quali erano necessarie poche modifiche e altri per i quali bisognava “migrare” verso architetture più complesse, mantenendo costante la facilità di utilizzo.

Nonostante tutto CALID si è dimostrato valido nell'analisi di entrambi i dataset: li dove l'eterogeneità dei dati è maggiore, permette di ottenere ottimi risultati; altrimenti, come lo è stato per il dataset estratto da DBLP, si è stabilizzato restituendo valori accettabili di precision e recall.

Sicuramente, una possibile evoluzione del programma, basata su ulteriori analisi del dataset, ricercando i casi più complicati, potrebbe permettere di raggiungere valori migliori: ad esempio, si potrebbero applicare più metodi di confronto sullo stesso campo in modo da calcolare le similarità e combinarle per rendere il valore ottenuto “umanamente logico”. Questo, però, porterebbe ad un incremento notevole del costo computazionale, direttamente proporzionale all'aggiunta di ogni metodo di confronto per campo, e potrebbe far degenerare il programma, rendendolo inusabile, in tempi molto lunghi.

Quindi bisognerebbe trovare un punto di incontro tra accuratezza e velocità, magari iniziando a modificare il programma rendendolo multiprocessore, visto che Python fornisce librerie apposite per gestire i processi in concorrenza. In questo modo, già su un sistema quad-core, il tempo potrebbe essere notevolmente ridotto.

Vista la difficoltà e le numerose prove per riuscire a trovare una soluzione soddisfacente, si potrebbe “automatizzare” il processo di ricerca del giusto file di configurazione attraverso algoritmi che calcolano la vicinanza al risultato ottimale, modificando parametri, auto-aggiornandosi intelligentemente.

Al momento, sto già lavorando all'implementazione di CALID per renderlo multiprocessing, e come obbiettivo principale mi sono proposto di riuscire ad ottenere risultati soddisfacenti in termini di velocità, soprattutto durante la prima fase di filtering dell'estratto del DBLP dataset, passando da 54 minuti ad un massimo di 15 minuti.

Infine, potrebbe sicuramente portare giovamento all'utente, l'inserimento di un'interfaccia grafica che possa permettere una più facile comprensione e utilizzo del programma, annullando del tutto il tempo sprecato per correggere gli errori nel file di configurazione a

causa di modifiche incompatibili con lo stesso.

Bibliografia

- [LEV66] Levenshtein, V. I. (1966, February). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (Vol. 10, No. 8, pp. 707-710).
- [HM02] Holmes, D., & McCabe, M. C. (2002, April). Improving precision and recall for soundex retrieval. In *Information Technology: Coding and Computing, 2002. Proceedings. International Conference on* (pp. 22-26). IEEE.
- [RDM13] Roy, S. B., De Cock, M., Mandava, V., Savanna, S., Dalessandro, B., Perlich, C., ... & Hamner, B. (2013, August). The microsoft academic search dataset and kdd cup 2013. In *Proceedings of the 2013 KDD cup 2013 workshop* (p. 1). ACM.
- [SHY11] Sun, Y., Han, J., Yan, X., Yu, P. S., & Wu, T. (2011). Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *VLDB'11*.
- [LLL13] Liu, J., Lei, K. H., Liu, J. Y., Wang, C., & Han, J. (2013, August). Ranking-based name matching for author disambiguation in bibliographic data. In *Proceedings of the 2013 KDD Cup 2013 Workshop* (p. 8). ACM.
- [VP14] VITALI, F., & PERONI, S. UN APPROCCIO PER LA DISAMBIGUAZIONE DI AUTORI DI ARTICOLI SCIENTIFICI: ALGORITMO E IMPLEMENTAZIONE.
- [MIL13] Milojević, S. (2013). Accuracy of simple, initials-based methods for author name disambiguation. *Journal of Informetrics*, *7*(4), 767-773.
- [NEW01] Newman, M. E. (2001). The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, *98*(2), 404-409.
- [WC15] Precision and recall. (2015, May 4). In *Wikipedia, The Free Encyclopedia*. Retrieved 16:15, June 12, 2015, from http://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=660679169
- [HUA08] Huang, A. (2008, April). Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand* (pp. 49-56).

- [BCINPV14] Andrea Bagnacani, Paolo Ciancarini, Angelo Di Iorio, Andrea Giovanni Nuzzolese, Silvio Peroni e Fabio Vitali. The Semantic Lancet Project: a Linked Open Dataset for Scholarly Publishing. To appear in Proceedings of Satellite Events of EKAW 2014, Lecture Notes in Artificial Intelligence. 2014. url: http://speroni.web.cs.unibo.it/publications/bagnacani-in-press-semantic-lancet-project.pdf.