Data Semantics

Sommario

La *Data Semantcs* si occupa di comprendere il significato dei dati durante la scrittura di un programma. Nell'ambito dell'analisi è fondamentale nell'integrazione di più dataset e nella loro condivisione. Altra problematica che la *Data Semantics* si pone di risolvere è l'organizzazione di dati non strutturati in modo da facilitarne l'analisi.

Scopo del corso è costruire dei modelli per attribuire valore semantico ai dati, in modo tale da facilitarne la fruizione; si tenta inoltre di capire il ruolo della semantica nell'integrazione dei dataset. Le finalità del corso rientrano nell'ambito dei *big data*, nella costruzione di *knowledge graphs* (ovvero la costruzione di relazioni interne a un database) e la costruzione di sistemi di raccomandazione. Saranno inoltre analizzate alcune tecniche di *natural language processing* per costruire rappresentazioni a partire dai dati.
Le esercitazioni si occuperanno di interrogare *knowledge graphs* e integrare fonti di dati.

L'esame orale sarà accompagnato da un progetto software (effettuato in gruppo di - circa - 3 persone), di cui sarà fatta una presentazione orale; in alternativa al progetto è possibile scrivere un articolo di approfondimento su una tematica a scelta. La preparazione dovrà essere "ragionevolmente" dettagliata su tutti gli argomenti, e particolarmente approfondita sull'argomento del progetto.

Indice

Parte I
Grafi di conoscenza ed introduzione ai concetti

Il termine "grafo di conoscenza" è stato coniato da Google per indicare uno strumento usato dal suo motore di ricerca per fornire informazioni aggiuntive durante le interrogazioni. La struttura a grafo permette di essere processata facilmente da una macchina e allo stesso tempo garantisce un livello di astrazione soddisfacente; si possono anche effettuare query come in un database a grafo (Neo4j). Il modello a grafo permette inoltre una facile integrazione di sorgenti diverse rendendo naturali collegamenti e relazioni.

Il web *semantico* ha costruito linguaggi e strumenti, approvati dal W3C, per definire, interrogare e fare inferenza su grafi di conoscenza; tuttavia nel mondo reale questi strumenti non hanno largo utilizzo. La costruzione di grafi di conoscenza è spesso effettuata a mano da una moltitudine di utenti (chiedendo recensioni tramite *app*, ad esempio). Internet produce enormi quantità di dati diversi tra di loro, usati spesso per altri fini: la semantica dei dati si occupa di integrare grandi quantità (*data volume*) da diverse fonti di dati (*data variety*). Tali dati possono essere sfruttati nella costruzione di intelligenze artificiali, ovvero di programmi che eseguono task tipicamente umani con risultati simili. In particolare è così possibile effettuare compiti particolarmente ardui per un umano, come l'integrazione di serie storiche con fonti multiple appartenenti a domini vari o poco noti.

Esistono inoltre grafi di conoscenza *open*, quali DBpedia, Yago o Wikidata, ma alcune aziende private, oppure anche organizzazioni ed istituzioni, sviluppano il proprio grafo per facilitare l'attività imprenditoriale o di gestione del proprio DB.

Introduzione al principio matematico: Lo spazio vettoriale

In un grafo di conoscenza, query e documenti sono interpretabili come vettori (in un sistema tradizionale di information retrieval si agisce attraverso un modello matematico, il *term document matrix* in cui ogni cella ci dice se le parole ricorrono o meno nel documento): è possibile calcolare misure di distanza come la *distanza euclidea* oppure calcolare misure di similarità come la *distanza coseno*.

$$sim(r,u) = cos(\theta) = \frac{uq}{|u||q|}$$

La rappresentazione vettoriale è *alla base* dell'analisi testuale. Mentre in passato si sfruttavano sistemi basati essenzialmente sulla presenza-assenza di una parola e quindi sui valori 0 e 1 oggi si usa un sistema di pesi più sofisticato per valutare diversamente l'importanza di una parola all'interno di un documento, quindi con valori *compresi* tra 0 e 1. Una parola è tanto più importante nel documento quante più ricorrenze ci sono nel documento e meno per quante ricorrenze ha negli altri documenti (in percentuale).

$$tf_{i,j} = \frac{n_{i,j}}{|d_j|}$$

$$idf_i = \log \frac{|D|}{|\{d : i \in d\}|}$$

$$tfidf_{i,j} = tf_{i,j} \times idf_i$$

Le entità, le loro proprietà e i collegamenti tra di loro sono iscritti nel grafo di conoscenza. I grafi di conoscenza sono basi di conoscenza che suppportano task di data analytics perchè possiedono un contesto ricco, servizi di collegamento e supportano accesso via web; inoltre rappresentano l'astrazione più importante per la data semantics. Esistono linguaggi formali, definiti a inizio '900, che permettono di dichiarare relazioni tra entità ma che non sono leggibili da una macchina. Inoltre si possono usare assiomi logici per definire le ricorrenze nel testo.

Linking Data

I dati sono spesso collegati in modo autonomo grazie a programmi di apprendimento automatico. Una ricerca su internet non è fatta più fatta per documenti, come accadeva nei primi anni 2000, ma per contenuti: un motore di ricerca in passato offriva una serie di documenti senza offrire informazioni aggiuntive; oggi invece tenta direttamente di rispondere con *factual information* rilevanti nella ricerca attraverso delle schede.

Per *informazione fattuale* si intende un'informazione interpretabile come vera o falsa (al contrario, alcuni dati quali immagini o suoni non sono interpretabili per veridicità). I fatti costituiscono un elemento centrale per l'analisi. Le risposte fattuali sono personalizzate in base alla natura della ricerca, secondo criteri *data driven* (ovvero statistici). Informazioni di tipo diverso hanno caratteristiche diverse: si produce un grafo di conoscenza per gestire meglio entità di tipo diverso con caratteristiche peculiari diverse. Le *preview* dei contenuti sono generate chiedendo agli sviluppatori di inserire dei contenuti nel codice HTML che sono poi interpretati dal motore di ricerca. I chatbot, ad esempio, sono costruiti con l'ausilio di reti neurali e rappresentazioni del mondo reale.

1 Lo standard per il *web semantico*

RDF (*Resource Description Framework*) è il linguaggio standrad per il web semantico, ovvero per *l'internet elaborabile dalle macchine*. L'unità base per rappresentare l'informazione è rappresentata da triple (affermazioni), ovvero grafi etichettati, identificati da URI (*Unique Resource Identifier*). Esempi di triple sono:

```
<Electric Piano, label, "Electric Piano"@en>
<Elton John, instrument, Electric Piano>
<Sails, artist, Elton John>
```

Le triple sono rappresentabili come un grafo diretto, aciclico ed etichettato:

```
     Elton John -[artist]- Sails
     /          \
  [instrument]   [artist]- Empty Sky
   /
 Electric Piano
    \
     [label]- "Electric Piano"@en
```

Alle triple si applicano degli identificativi globali (una sorta di chiave primaria SQL, gli URI, appunto): si usano generalmente identificativi web (abbreviati, detti prefissi), che specificano

come recuperare l'informazione (qualsiasi cosa a cui si può attribuire un valore costante è definibile come URI). Tutti gli URI sono risorse, e rappresentano l'ontologia del dominio.

Le triple sono strutturate con un **soggetto**, un **predicato** e un **oggetto** (Turtle syntax). Il soggetto è composto da un URI o da un *blank node* (ovvero costanti *locali*), il predicato può essere composto solo da un URI mentre gli oggetti possono essere URI, *blank nodes* oppure letterali, che possono essere *plain literal*, ovvero letterali puri, oppure *typed literal*, a cui è attribuito un tipo (stringhe, date o booleani) per permettere operazioni. I *blank nodes* sono nodi anonimi per consentire una buona costruzione del grafo.

1.1 Best practices in pubblicazione

I *linked data* riassumono delle best practices per pubblicare e unire dati provenienti dal web:

- usare URI come nomi delle cose;
- usare indirizzi HTTP come URI;
- inserire informazioni utili su un URI;
- includere link ad altri URI.

Sono solamente una serie di principi, non rendono i dati *open* (*Linked Open Data*).

Altro approccio per pubblicare i dati sul web è usare l'*annotazione semantica*: si inseriscono annotazioni (tag) in una pagina web che specifichino il significato del contenuto o altri meta-dati per permettere e facilitare l'elaborazione dei contenuti da parte di agenti intelligenti. La sintassi è simile a XML: RDF in precedenza usava il formato XML, non Turtle, ma è stato abbandonato per la sua struttura ad albero in favore di una struttura a grafo. L'approccio è tornato in auge grazie ai *crawler*: dei motori di ricerca che annotano le pagine in base al loro significato semantico[1].

Non tutti i formati sono compatibili con RDF (Microdata ed hCard non lo sono) mentre altri sì (RDFa, JSON-LD); nonostante questo non ci sono differenze significative tra i vari formati (il modello è il medesimo):

- hCard e vCard sono basati su HTML, ponendo delle convenzioni su come gestire le informazioni;
- I cosiddetti Microdata usano tag speciali introdotti in HTML5 per definire *itemscope*, *itemtype* e *itempropriety*: si definisce (*itemscope*) un oggetto di un certo tipo (*itemtype*) con determinate proprietà (*itemprop*), che in RDF si inseriscono tramite *blank nodes*. I microdati sono parti di codice integrate dentro al testo, visualizzabili solamente nel codice sorgente. Si usano dei tipi definiti da vocabolari[2], utilizzabili anche in RDF, adottati come standard;

[1] `https://webdatacommons.org` è un archivio di crawl trovati sul web.

[2] `https://schema.org/` è un dizionario che definisce in modo non eccessivamente prescrittivo oggetti e proprietà.

- RDFa specifica all'inizio il vocabolario adottato, mentre il codice rimane scritto in HTML; JSON-LD (JSON *Linked Data*) integra un file `.json` (in un tag `<script>`) in una pagina HTML per attribuire significati semantici. Non si ha bisogno di un parser apposito, e le triple sono facilmente ricostruibili a partire dalla pagina web. I motori di ricerca, processando in modo automatico, mostrano il contenuto in base al significato semantico.

1.2 RDF query language: SPARQL

Introdotto come linguaggio per interrogare *linked data*, è un linguaggio simile a SQL caratterizzato (dalla versione 1.1) da quattro elementi:

- SPARQL Query;
- SPARQL Algebra;
- SPARQL Update;
- SPARQL Protocol.

Permette di interrogare (query), modificare (update) e integrare database (effettuare cioè federated query) composti da triple. Il principio di SPARQL è quello di *pattern matching*, ovvero le query descrivono sottografi del grafo principale e quelli che matchano rappresentano i risultati. Si definisce il concetto di *triple pattern*: si sostituiscono uno o più elementi di una tripla con una variabile.

```
                                    # importante il punto alla fine
dbpedia:The_Beatles foaf:name "The Beatles" .
dbpedia:The_Beatles foaf:name ?album .
?album mo:track ?track .
?album ?p ?o .
```

Si possono così costruire query strutturate.

```
# prologo: si definiscono le fonti
PREFIX dbpedia:<http://dbpedia.org/resource/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
...
```

Esistono diversi tipi di query effettuabili in SPARQL:

```
# ASK | SELECT | DESCRIBE | CONSTRUCT
# Query di tipo SELECT
  SELECT ?album
    FROM <http://musicbrainz.org/20130302/>
   WHERE {
            # il cuore della query:
            # il titolo di tutte le tracce degli album dei Beatles
            dbpedia:The_Beatles foaf:made ?album .
            ?album  a(shortcut per rdf:type) mo:Record;
            dc:title ?title .
```

```
        }
ORDER BY ?title
```

`ASK` ritorna `true/false` se esiste una soluzione alla query:

```
  ASK
WHERE {
        dbpedia:The_Beatles mo:member .
        dbpedia:Paul_McCartney .
      }
```

```
=> true
```

`SELECT` ritorna una lista di elementi che hanno corrispondenza nel grafo:

```
SELECT ?album_name ?track_title
       ?date       ?album       .
 WHERE {
        # tutti gli album dei Beatles
        dbpedia:The_Beatles foaf:name ?album .

        # tutte le tracce degli album
        ?album              dc:title  ?album;
        mo:track ?track .
        ?track            dc:title  ?track_title;

        # filtra per durata
        mo:duration ?duration .
        FILTER (?duration > 300000 && ?duration < 400000)
       }
```

`CONSTRUCT` è simile a `SELECT` ma ritorna un grafo e non una lista:

```
CONSTRUCT {
            # riscrive con nuova terminologia
            ?album dc:creator dbpedia:The_Beatles
          }
    WHERE {
            # esegue la query
            dbpedia:The_Beatles foaf:made   ?album;
            ?album              mo:track    ?track .
          }
```

Le basi di conoscenza sono interrogate grazie ai vocabolari (anche detti *ontologie*), che permettono quindi la costruzione di query. Le ontologie ci permettono di conoscere la struttura del *Knowledge Graph* così da poter effettuare interrogazioni più o meno precise. I diversi *Knowledge Graph* sono integrati grazie a predicati come ad esempio (`same as`) che indicano quali entità si riferiscono allo stesso oggetto reale nelle varie fonti.

SPARQL 1.1, inoltre, permette di integrare un sistema di ragionamento automatico per l'uguaglianza delle entità (introduce il concetto di *entailment*), ovvero un *reasoner* in

grado eventualmente anche di inferire conoscenza non direttamente esplicitata nel grafo di conoscenza, ma solo conseguenza dei vincoli e delle proprietà presenti in esso.

2 Vocabolari (i.e. Ontologie)

In RDF si ha un modello dei dati che prevede di utilizzare URI per rappresentare sia le risorse sia le proprietà. L'uso di URI per specificare proprietà e relazioni definiscono separatamente l'insieme di proprietà e tipi usati per descrivere un dominio: si definisce così un vocabolario per descrivere i dati, permettendo una standardizzazione della tipizzazione del dato.

Un esempio di ciò è il predicato `rdf:type` che introduce un vocabolario che permette di catalogare risorse definendo i tipi di variabile. Si definiscono quindi ontologie comuni sfruttabili da più *data provider* per uniformare la rappresentazione dei dati. A tutti gli effetti un vocabolario è riconducibile allo schema dei RDBMS, dato che lo scopo di un'ontologia, semplificando, è proprio quello di definire lo schema dei dati. Per definire un vocabolario si definiscono prima le classi (ovvero i tipi di dato) con proprietà, eventualmente diverse (ovvero usando certe proprietà solo su certe classi).

Uno dei vocabolari più importanti (perché introdotto da tempo) è **FOAF** (*Friend of a Friend*); si ricordano anche **DC** (*Dubling Core*), che specifica una serie di predicati per rappresentare i documenti come *titolo, autore* ad esempio, e **SKOS** (*Simple Knowledge Organizating System*). Più vocabolari possono riferirsi allo stesso oggetto e per definizione non è lecito unificare i vocabolari in quanto dovrebbero adattarsi alla specificità del dominio. C'è quindi una certa libertà nel descrivere gli oggetti perchè ognuno può usare vocabolari differenti.

Alcuni dei più grandi provider internet hanno però unificato vocabolari col progetto `schema.org`, molto utilizzato nel web e in continua evoluzione per dati strutturati. Su tale sito sono raccolte tutte le classi disponibili in questo vocabolario e per ogni classe sono specificate alcune proprietà per descriverla, oltre a fornirci anche gli *expected type* di queste proprietà (questo concetto risulta molto importante nella definizione dei vocabolari).

A seconda del task occorre sempre trovare il vocabolario più semplice per coprire tutti i nostri bisogni e, nel caso si rivelasse insufficiente, è consigliabile espanderlo all'occorrenza, piuttosto che crearne uno nuovo dal nulla. Si possono unire più vocabolari grazie a collegamenti, a livello di istanza o di schema. Un computer ha bisogno di definire un sistema per attribuire un significato al significante.

Si possono definire tre[3] sistemi per definire un'ontologia, ovvero uno schema per definire gli oggetti esistenti. L'uso di sistemi diversi rende complicata l'integrazione di più fonti.

2.1 Tesauri

Tra i diversi tipi di ontologie il tesauro (anche detto ontologia lessicale) è il più obsoleto. Il tesauro può essere definito anche come ontologia lessicale, ovvero i significati delle parole sono esplicitati in termini di relazioni lessicali tra queste. Generalmente definisce sostantivi (semplici o composti), aggettivi e verbi.

Le relazioni lessicali sono:

- **sinonimia**: più parole con significati simili;

[3]Esiste un quarto metodo che sarà approfondito più avanti.

- **antinomia**: più parole hanno significato opposto;
- **polisemia**: più significati della stessa parola;
- **iponimia** (e iperonimia): quando un significato rappresenta un sottoinsieme dell'altro (casa - costruzione);
- **associazione**: più parole strettamente legate tra di loro.

Tuttavia esistono dei problemi: i sinonimi non sono mai perfettamente interscambiabili e l'iponimia (o iperonimia) sottintende molte sfumature.

Alcuni esempi di ontologie lessicali sono:

- **Tesauri di dominio**
- **EuroWordnet**
- **WordNet** è un vasto database lessicale sviluppato dagli anni '90 che rappresenta i significati (o concetti) in una struttura ad albero. La polisemia è gestita definendo più entità diverse con lo stesso nome. Data una parola, si ha quindi un insieme finito di significati distinti tra di loro.
- **MultiWordnet**

2.2 Tassonomia

Si definisce tassonomia un grafo di classificazione gerarchica dei concetti, generalmente con struttura ad albero. Le tassonomie sono strutture usate in informatica e nello specifico in Data Science e Software Engineering, prese in prestito dalla biologia (per classificare le forme viventi), sulle quali tuttavia sono presenti dibattiti. La valenza può anche non essere perfetta ma ha grande importanza in ambito scientifico ed economico. Si usa una forma ad albero (per facilitare la condivisione tra varie fonti) i cui nodi sono concetti (ogni nodo può avere un solo padre e uno o più figli).

Il significato è spiegato in classi, superclassi e sottoclassi (relazione di tipo `is-a`) grazie al meccanismo dell'ereditarietà: i nodi più specializzati ereditano le proprietà di quelli più generali. È necessario specificare anche quale criterio usare per effettuare la divisione dell'albero. Si può anche effettuare una ripartizione dell'oggetto dividendolo nei sotto-oggetti specifici che lo compongono (operazione detta di mereologia), nell'ambito di una relazione `part-of`, ovvero una relazione transitiva e riflessiva .

2.3 Ontologie assiomatiche

Nelle ontologie assiomatiche il significato è attribuito attraverso una serie di *assiomi logici* per specificare le definizioni dei termini utilizzati, con riferimento alle implicazioni.

Le caratteristiche fondamentali di un ontologia sono che essa sia:

- Concettuale: deve cioè essere un modello astratto di un certo fenomeno del mondo reale attraverso l'identificazione dei concetti rilevanti che lo caratterizzano;

- Esplicita: i concetti, le proprietà ed i vincoli che caratterizzano un fenomeno devono essere esplicitamente definiti;
- Formale: deve cioè essere *machine-readable*, escludendo l'uso del linguaggio naturale. Il ragionamento automatico è infatti fondamentale per assicurare la qualità di un'ontologia a design-time, ovvero verificare che i concetti non siano contraddittori ed estrarre conoscenza implicita, e per sfruttare la semantica con lo scopo di inferire conoscenza in fase di ricerca delle informazioni;
- Condivisa: è infatti fondamentale che le descrizioni siano condivise tra individui di un gruppo (`schema.org` è un esempio di questo principio).

Componenti principali dell'ontologia assiomatica sono:

- **Concetti o classi**: un concetto può rappresentare un oggetto, una nozione o un'idea. I concetti possono essere astratti o concreti, elementari o composti, reali o fittizi. Un concetto può essere qualsiasi cosa su cui si può dire qualcosa e, perciò, può anche essere la descrizione di un task, una funzione, un'azione, una strategia, un processo di ragionamento;
- **Assiomi**: sono affermazioni, *sempre vere*, sul modello definito dall'ontologia e servono per specificare la semantica dei concetti. Generalmente specificano il modo in cui il vocabolario concettuale (concetti e relazioni) può essere usato.
- Istanze: le componenti base ("ground level") che popolano l'ontologia.

Si definisce quindi un linguaggio L di assiomi logici con una semantica formale che determina il significato in modo univoco.

3 Inferenza nell'ontologia

L'esistenza (la definizione) di un oggetto nella realtà porta naturalmente con sè delle implicazioni: è necessario quindi fare inferenza, ovvero mettere in atto un processo deduttivo per rendere esplicita la conoscenza implicita. L'inferenza (un meccanismo fondamentale della semantica) nasce infatti dal ragionamento su delle premesse, con l'obiettivo di estrarre *conoscenza vera* da *premesse vere*, effettuando una generalizzazione con un certo margine di errore. Dal punto di vista dell'informatica, l'uso della logica (proposizionale e del primo ordine) permette la deduzione da parte di calcolatori. Oltre all'induzione classica, esistono altre pratiche per fare inferenza. Le implicazioni ricavate sulla base di ciò che è definito nel dizionario, dal punto di vista inferenziale, sono la semantica reale delle informazioni.

L'inferenza ha come obiettivi la riduzione della complessità del sistema trovando un modo per ottenere informazioni velocemente ed l'aumento della flessibilità dello schema (ad esempio per poter essere aggiornato in tempo reale con le modifiche dell'ambiente).

3.1 Schema dell'ontologia

Lo schema prescrive come deve essere organizzato il dato per entrare nel database; tuttavia i dati sono flessibili per loro stessa natura, dunque l'operazione non è immediata. È necessario

definire uno schema per dare una struttura ai dati: non è lecito usare nell'approccio semantico un database NoSQL troppo flessibile.

Dall'altro lato però, rispetto all'approccio relazionale, lo schema semantico è più flessibile: SQL non permette di modificare la struttura una volta dichiarata, usa cioè un approccio *prescrittivo*. Quello del *semantic web* è invece un approccio *deduttivo*: ovvero non ci dice come devono essere strutturati i dati per essere inseriti, ma quali informazioni possiamo estrarne (validazione indiretta); in sintesi tiene conto del fatto che i dati, nel tempo, possano cambiare. Questo approccio è quindi flessibile e disaccoppia lo schema dai dati (ovvero lo schema può essere modificato successivamente e ciò potrebbe provocare anche incoerenze nel breve periodo).

Potendo esserci contraddizione nei dati, si sono costruiti linguaggi come RDF Schapes e SHACL che permettono una validazione delle informazioni (estratte per mezzo di inferenza) attraverso la definizione di vincoli.

4 Reasoning

I linguaggi RDFS e OWL permettono l'operazione di *reasoning*, ovvero l'inferenza di nuova conoscenza a partire da più ontologie esistenti. Oltre ad avere sistemi deduttivi, la logica proposizionale su cui sono basati garantisce che l'algoritmo termini (sia, cioè, *decidibile*: ovvero date le premesse è sempre possibile stabilire in un tempo finito se una FBF - Formula Ben Formata - è vera o meno), mentre la logica del primo ordine è solo semi-decidibile.

Anche una procedura che termini e confermi la verità di una formula, però, non è utile in pratica: il tempo impiegato potrebbe essere particolarmente lungo e quindi rendere inutile il programma. In caso di linguaggi non decidibili, occorre semplificare il linguaggio per trovare un compromesso tra tempo impiegato ed espressività: l'esistenza di più linguaggi è dovuta al fatto che il compromesso può essere trovato in modo diverso.

Inoltre esistono altri linguaggi *a regole* (rules-based), ovvero linguaggi non puramente logici, ma che vi si ispirano, utilizzabili per fare deduzioni su RDF, che risultano più efficienti per certi tipi di operazioni.

4.1 RDFS (RDF-Schema)

L'idea alla base di RDFS è di costruire un vero e proprio *reticolo* che parte da un nodo e garantisce ereditarietà (anche multipla) delle classi: viene definito, cioè, un *ordine parziale*.

RDFS introduce inoltre implicazioni sul tipo di dato: si delinea un approccio secondo cui il mondo è composto da *classi* e *individui*; quindi si hanno insiemi ed elementi degli insiemi. Una classe è composta da individui i quali, tramite *proprietà*, entrano in relazione tra di loro; inoltre le relazioni *tra classi* comportano vincoli sulle relazioni tra individui, data l'appartenenza di questi ultimi alle classi. Questo approccio è coerente con il modello relazionale, e caratterizza in modo molto forte la semantica dei dati strutturati.

Il concetto di classe vincola l'uso del tipo: una classe non può essere del tipo di un'altra classe. I letterali vengono esplicitamente distinti dagli URI.

Con RDFS si possono rappresentare e vincolare relazioni tra classi, organizzandole in vere e proprie gerarchie: si organizzano così tassonomie che garantiscono *nativamente* inferenze semplici (dotate, cioè, di proprietà transitiva). Anche le proprietà possono essere organizzate

gerarchicamente, con proprietà che derivano da altre (come ad esempio: `figlioDi` è sotto-proprietà di `parenteDi`).

La semantica dei predicati è definita da regole che determinano l'ereditarietà (classi derivate), la transitività di classi e relazioni.

I predicati `rdfs:domain` e `rdfs:range` indicano che tutti gli oggetti di una classe appartengono anche a un'altra classe. Questi due predicati non sono usati in `schema.org` perché troppo rigidi. Lo schema non può dedurre contraddizioni, dunque.

Alcuni esempi di quanto detto sono:

```
se [x subClassOf di y]
e  [a type x]
allora [a type y] (i tipi si ereditano)

se [x subClassOf di y]
e  [y subClassOf di z]
allora [x subClassOf di z] (proprietà transitiva)

se [x a y]
e  [a subPropertyOf di b]
allora [x b y]
(i precedenti esempi valgono anche per le proprietà)

se [a subPropertyOf di b]
e  [b subPropertyOf di c]
allora [a subPropertyOf di c]

se [x a y]
e  [a domain z]
allora [x type z]
(tutti quei soggetti che hanno una proprietà parte
di un dominio, sono dello stesso tipo del dominio)

se [x a u]
e [a range z]
allora [u type z]
(stesso ragionamento del dominio ma il range vale
per l'oggetto, non per il soggetto.)
```

Nonostante tutti i pregi, RDFS presenta alcuni svantaggi:

- Non ha restrizioni locali all'appartenenza di domini;
- Non ha vincoli di cardinalità;
- Le classi non possono essere combinate;
- Le proprietà non sono transitive, inverse e simmetriche;
- Non permette i disjoint;

- Non permette combinazioni booleane delle classi.

OWL è stato introdotto per colmare alcune di queste lacune.

4.1.1 SPARQL 1.1

La versione 1.1 di SPARQL, come già accennato, introduce gli *Entailment Regimes*: ovvero procedure che calcolano tutte le implicazioni degli assiomi; uno dei sistemi più noti è Virtuoso. I risultati possono essere esportati come `.xml`, `.csv` o `.tsv`.

4.2 OWL

È un motore inferenziale composto da più sotto-linguaggi, con l'obiettivo di ridurre la confusione lasciata da RDFS; tuttavia ne aumenta la complessità computazionale. È un linguaggio *formale* (è un linguaggio logico processabile da elaboratori), *esplicito* (non ambiguo) e *condiviso*; gode perciò di tutte le caratteristiche chiave di un'ontologia assiomatica.

OWL2 comprende tre "sottoinsiemi" disponibili anche singolarmente:

- OWL 2 EL: limitato a classificazioni di base, ma la complessità è polinomiale;
- OWL 2 QL: che usa una sintassi simile a SQL;
- OWL 2 RL: è usato per essere implementato efficientemente in sistemi *rule-based*.

È frequente definire ontologie senza usare tutta la capacità espressiva di OWL, ma utilizzando in equilibrio con RDFS per arricchirlo.

Rispetto a RDFS si introduce l'equivalenza tra classi con `owl:equivalentClass` e tra individui con `owl:sameAs`, tuttavia il loro uso ha delle implicazioni semantiche molto importanti tra le quali:

- Riflessività;
- Transitività;
- Simmetria.

Inoltre l'equivalenza può essere fatta anche a livello di proprietà con `owl:equivalentProperty`. Si può anche specificare la disgiunzione tra due elementi con `owl:differentFrom`, in opposizione alla UNI (*Unique Name Assumption*), secondo cui oggetti con nomi diversi sono necessariamente distinti. Le proprietà possono essere disgiunte con `owl:propertyDisjointWith`.

Le proprietà di OWL sono distinte in due categorie:

- *Object Properties*: la proprietà è una risorsa;
- *Datatype Properties*: la proprietà è un letterale.

Questa distinzione facilita la costruzione di grafi specificando quali proprietà sono utilizzabili per attraversare il grafo stesso. La relazione tra classi e sottoclassi è, matematicamente parlando, un ordinamento parziale: si può costruire un reticolo finito (quindi con limite superiore e inferiore, rispettivamente uguali all'insieme *universo* e all'insieme *vuoto*).

Una relazione in OWL può possedere determinate proprietà:

- **Simmetria**;

- **Transitività**;
- **Transitività inversa**;
- **Funzionalità**: ovvero si ha al più un valore per un argomento permettendo l'uso di identificativi e di funzioni inverse;
- **Funzionalità inversa**: quest'ultima proprietà è fondamentale nelle fasi di *Record Linkage* o *Deduplication*.

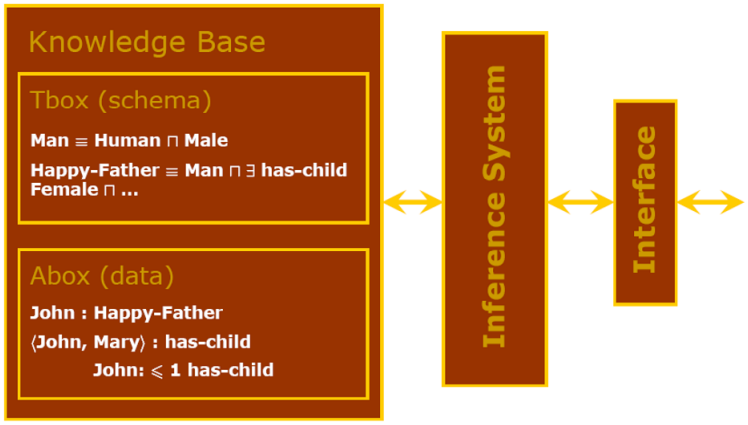
4.2.1 OWL Description Logic

Le logiche descrittive sono una famiglia di logiche usate per rappresentare il grafo di conoscenza e sono state introdotte per lo studio del linguaggio naturale prima del *deep learning*; più una logica è complessa ed espressiva maggiori sono il tempo di esecuzione e la complessità.

OWL DL (*OWL Description Logic*) si basa su SHIQ (è equivalente a SHOINDn). Una logica è composta da una TBox (*Terminological Box*), ovvero l'ontologia della logica, il suo schema, e una ABox (*Assertional Box*), che comprende i dati veri e propri, in forma di asserzioni, che possono essere di due tipi:

- Un individuo appartiene ad una classe;
- Due individui sono legati da una relazione.

Nelle logiche descrittive la difficoltà risiede nel costruire classi complesse (coi relativi costruttori).



Una classe può quindi essere definita basandosi su più classi tramite una serie di operatori logici. La semantica dunque si definisce ricorsivamente dagli elementi più semplici ai più complessi.

Costructor	Sintassi DL	Esempio	Sintassi modale
intersectionOf	$C_1 \sqcap \ldots \sqcap C_n$	Human $\sqcap$ Male	$C_1 \wedge \ldots \wedge C_n$
unionOf	$C_1 \sqcup \ldots \sqcup C_n$	Doctor $\sqcup$ Lawyer	$C_1 \vee \ldots \vee C_n$
complementOf	$\neg C$	$\neg$ Male	$\neg C$
oneOf	$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$	{ John } $\sqcup$ { Mary }	$x_1 \vee \ldots \vee x_n$
allVeluesFrom	$\forall P.C$	$\forall$ hasChild.Doctor	$[P]C$
someValuesFrom	$\exists P.C$	$\exists$ hasChild.Doctor	$(P)C$
maxCardinality	$\leq nP$	≤ 1 hasChild	$[P]_n + 1$
minCardinality	$\geq nP$	≥ 2 hasChild	$(P)_n$

Dove C è un concetto, P una proprietà e x un individuo.
L'*interpretazione* dei costrutti è di natura insiemistica:

$$\begin{aligned}
(C \sqcap D)^I &= C^I \sqcap D^I \\
(C \sqcup D)^I &= C^I \sqcup D^I \\
(\neg C)^I &= \Delta^I \backslash C^I \\
\{x\}^I &= \{x^I\} \\
(\exists R.C)^I &= \{x | \exists y. < x, y > \in R^I \wedge y \in C^I\} \\
(\forall R.C)^I &= \{x | \forall y.(x, y) \in R^I \Rightarrow y \in C^I\} \\
(\leq R)^I &= \{x | \#\{y | < x, y > \in R^I\} \leq n\} \\
(\geq R)^I &= \{x | \#\{y | < x, y > \in R^I\} \geq n\} \\
(R^-)^I &= \{(x, y) | (y, x) \in R^I\}
\end{aligned}$$

Dove $\square^I$ indica l'interpretazione di $\square$ secondo la funzione I.
Gli assiomi usati in queste logiche sono estremamente semplici ma si applicano ugualmente anche a classi più complesse. Avendo infatti complicato le classi (ovvero avendo definito alcune classi come risultati di costruzioni sfruttando classi più basilari), possiamo ottenere assiomi più *semplici*:

Sintassi OWL	Sintassi DL	Esempio
subClassOf	$C_1 \sqsubseteq C_2$	Human $\sqsubseteq$ Animal $\sqcap$ Biped
equivalentClass	$C_1 \equiv C_2$	Man $\equiv$ Human $\sqcap$ Male
subPropertryOf	$P_1 \sqsubseteq P_2$	hasDaughter $\sqsubseteq$ hasChild
equivalentProperty	$P_1 \equiv P_2$	cost $\equiv$ price
trnasitiveProperty	$P^+ \sqsubseteq P$	ancestor$^+$ $\sqsubseteq$ ancestor

- Sono sottintese le equivalenze FO/Modali come $DL : C \sqsubseteq D \quad FOL : \forall x C(x) \quad ML : C \rightarrow D$
- Si distinguono due tipi di assiomi delle TBox:
  - Definizioni: $C \sqsubseteq D$ o $C \equiv D$ (dove C è un concetto)
  - GCIs (*General Concept Inclusion axioms*: dove C può essere complesso

Possiamo dunque definire un'*interpretazione* anche per gli assiomi oltre che per le classi. L'interpretazione I di un assioma soddisfa l'assioma A ($I \models A$) solamente se sono rispettate alcune regole: con una teoria logica composta da assiomi, se ne possono avere un'infinità

di interpretazioni, ma ogni assioma riduce l'insieme di interpretazioni possibili, poichè ogni assioma aggiuntivo rappresenta un'ulteriore vincolo sull'ontolgia.

Un'ontologia ha esattamente questo compito: ridurre il numero di interpretazioni (riducendo così l'ambiguità) e permettere l'inferenza.

- An interpretation $\mathcal{I}$ satisfies (models) an axiom A ($\mathcal{I} \vDash$ A):
  - $\mathcal{I} \vDash C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ $\quad$ $\mathcal{I} \vDash C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$
  - $\mathcal{I} \vDash R \sqsubseteq S$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$ $\quad$ $\mathcal{I} \vDash R \equiv S$ iff $R^{\mathcal{I}} = S^{\mathcal{I}}$
  - $\mathcal{I} \vDash x \in D$ iff $x^{\mathcal{I}} \in D^{\mathcal{I}}$
  - $\mathcal{I} \vDash \langle x,y \rangle \in R$ iff $(x^{\mathcal{I}},y^{\mathcal{I}}) \in R^{\mathcal{I}}$
  - $\mathcal{I} \vDash R^{+} \sqsubseteq R$ iff $(R^{\mathcal{I}})^{+} \subseteq R^{\mathcal{I}}$
- $\mathcal{I}$ satisfies a Tbox $\mathcal{T}$ ($\mathcal{I} \vDash \mathcal{T}$) iff $\mathcal{I}$ satisfies every axiom A in $\mathcal{T}$
- $\mathcal{I}$ satisfies an Abox $\mathcal{A}$ ($\mathcal{I} \vDash \mathcal{A}$) iff $\mathcal{I}$ satisfies every axiom A in $\mathcal{A}$
- $\mathcal{I}$ satisfies an KB $\mathcal{K}$ ($\mathcal{I} \vDash \mathcal{K}$) iff $\mathcal{I}$ satisfies both $\mathcal{T}$ and $\mathcal{A}$

Un concetto si dice:

- *Soddisfacibile* se esiste almeno una sua interpretazione;
- *Sussunto* a un altro concetto se qualsiasi interpretazione del primo concetto è sottoinsieme di un interpretazione del secondo;
- Due concetti si dicono *equivalenti* se l'interpretazione del primo concetto è sempre uguale a quella del secondo in tutte le interpretazioni della teoria, al contrario si dicono *disgiunti*.

Il concetto di interpretazione in logica quindi è puramente teorico: è lo strumento matematico usato per attribuire un significato ai dati; l'inferenza permette di attribuire significati aggiuntivi indipendentemente dalla semantica iniziale.

È possibile utilizzare un *reasoner* per verificare la correttezza delle relazioni. La conoscenza ha senso solo se è *soddisfacibile*: una teoria è incosistente se non ha interpretazioni che la rendono vera; caso analogo si verifica quando una classe è *sussunta* all'insieme *vuoto*: in tal caso non ha interpretazioni possibili.

Da un punto di vista pragmatico, con un *reasoner* possiamo:

- Fare un check di consistenza;
- Inferire nuove relazioni;
- Inferire gerarchie;
- Ereditare proprietà;
- Inferire la classe degli individui.

In proposito ad OWL *DL* occorre specificare una distinzione molto netta tra:

- Assunzione del *mondo chiuso*, tipica dei database relazionali, per cui tutto ciò che è al di fuori del database è da ritenersi falso;

- Assuzione del *mondo aperto*, tipica di ontologie e logiche descrittive, per cui tutto ciò che è fuori *potrebbe* essere vero tanto quanto ciò che è dentro.

Inoltre sempre riguardo OWL, in quanto base di conoscenza, occorre fare riferimento all'assuzione denominata **Unique Name Assumption**, che in OWL appunto non ha valore, poichè due oggetti con lo stesso nome possono indicare due cose distinte, non necessariamente essere la stessa cosa.

4.3 Rule based reasoning

I rules-based reasoning, a differenza dell'OWL reasoning permettono di definire delle funzioni al loro interno in grado ad esempio di effettuare ragionamenti di natura quantitativa.

SWRL (Semantic Web Rule Language) è un linguaggio rules-based che combina OWL con RuleML, ma non è **decidibile**. Alternativo a SWRL in questo aspetto è DLP, un linguaggio che permette di instaurare delle relazioni d'ordine per gli oggetti, come ad esempio *antenato-genitore*. RuleML (Rule Markup Language) è una famiglia di linguaggi sviluppati per esprimere sia regole *forward* che *backward* in XML per effettuare deduzioni ed operazioni inferenziali.

5 Search Engine semantici

Un *search engine* semantico sfrutta una base di conoscenza per effettuare delle query più mirate. Per la gestione di pattern di architettura ci sono tre possibili approcci:

- **Crawling**: i dati sono memorizzati in un database locale;
- **On-the-fly deferencing**: gli URI sono deferenziati quando il programma richiede i dati;
- **Federated query**: ogni richiesta è delegata ad un servizio diverso.

Anche per il *reasoner* ci sono strategie diverse.

6 Semantic Framework

Si usano dei framework per archiviare i dati in grafi per la facilità della gestione delle relazioni (si interroga per ottenere nuova conoscenza):

- **RDF4J** è un framework per Java per gestire, archiviare (in diversi modi) e interrogare documenti RDF, organizzando tramite un paradigma ad oggetti;
- **Apache Jena** è un'api per RDF, offre un reasoner organizzando i dati ad oggetti;
- Un **triplestore** archivia le triple e le filtra per elementi fissati ed è fatto per ricevere query SPARQL. Ci permette di fare inferenza per mezzo di reasoner o entailment ma non ha struttura interna, poichè non ha struttura per le proprietà. Alcuni esempi di triplestore sono *Openlink Virtuoso*, che permette di avere più grafi e di avere quindi strutture quadruple e non triple e *AllegroGraph*;

- **Property Graph** a differenza del triplestore ha una struttura interna a costo però di perdere la possibilità di effettuare inferenza. Alcuni esempi di property graph sono *GraphDB8* e *Neo4j*.

Tra i reasoner più conosciuti vi sono:

- Pellet (OWL DL);
- Racer (OWL DL);
- Hermit;
- Reasoner nativi di Jena, GraphDB8 e AllegroGraph.

Parte II
Problemi di matching ed integrazione semantica

1 Integrazione di grafi di conoscenza

L'integrazione virtuale (l'uso di predicati `sameAs` e simili) non è una funzione particolarmente scalabile con grossi database: per fare questo è necessario effettuare un'integrazione in altro modo (*data integration*). La *data integration* nel caso dei grafi di conoscenza parte da una base di diversi oggetti, nel caso più semplice una *source* ed un *target*, che è necessario unire per aumentare il potere informativo creando un unica base di conoscenza più ampia e completa. Per effettuare con successo questa operazione è però necessario usare dei *mapping*.

L'operazione di *linking data* avviene a due livelli distinti:

- A livello di schema: effettuando un *mapping* tra concetti e proprietà (anche detto *ontology mapping*);
- A livello di istanze: ovvero effettuando un *mappping* tra entità descrittive in sorgenti eterogenee. Ciò può avvenire tramite *entity co-resolution* (`sameAs`), *link discovery*, *instance matching* o *record linkage*).

Quando si costruisce un grafo di conoscenza (soprattutto verticale, ovvero su un solo tema ma in modo approfondito), si parte innanzitutto da un'operazione di *data ingestion*.

Generalmente questa operazione avviene da fonti di dati strutturate o semi-strutturate come tabelle o database relazionali. Si può effettuare una traduzione semantica della tabella (*mapping*) per convertire una tabella in formato RDF, però facendo questo si trascurano le relazioni e le loro proprietà e non si ha controllo sugli elementi duplicati. Un tool molto usato per assolvere a questo compito di conversione da formato tabellare ad RDF è **DataGraft**.

La trasformazione da tabelle a *Knowledge Graph* può presentare una serie di criticità che occorre affrontare predendo determinati accorgimenti:

- Normalizzare la tabella;
- Annotare i diversi componenti della tabella con i componenti del *KG*, sia a livello di schema assegnando concetti e proprietà alle colonne, sia a livello di istanze assegnando entità ai valori della tabella;
- Si possono usare strumenti quali R2ML per annotare le tuple del database relazionale e mappandole con programmi scritti *ad hoc*. Questo avviene per operazioni di *mapping* particolarmente complesse. Gli strumenti di *semantic table annotation and integration* tentano di stabilire quali entità esistono in un database e aiutano nella costruzione di un grafo di conoscenza;
- Solo dopo questi accorgimenti e passaggi è possibile eseguire il *mapping* finale e rappresentare il contenuto della tabella come *KG*.

Un altro modo per accrescere un grafo di conoscenza è quello di processare testi (prendento informazioni dall'abstract di `wikidata.org`, per esempio): strutturare informazioni in formato tabellare è un processo lungo e costoso, dunque è indispensabile sfruttare tecniche di *Natural Language Processing* (NLP) per estrarre informazioni direttamente da testi, tweet, abstract; tuttavia l'uso di abbreviazioni o espressioni gergali rende difficile un processo automatico del testo. Da una parte il problema è strutturare informazioni da un testo (*information extraction*), tipico caso di studio di intelligenze artificiali e semantica computazionale, mentre dall'altra è difficile rendere utile l'informazione estratta integrandola all'interno della base di conoscenza. Gli obiettivi dell'estrazione delle informazioni sono la *named entity recognition* e la *relation extraction*: la prima serve per riconoscere un'entità di un testo, mentre la seconda tenta di stabilire le relazioni tra queste entità. Per integrare le entità in un grafo di conoscenza, si effettua un'opearazione di *entity linking*.

1.1 Matching

L'operazione di matching può risultare molto complessa a seconda del caso specifico e necessita di una serie di passaggi:

- Per prima cosa, occorre valutare la *similarità* tra due entità appartenenti a fonti diverse tramite un opportuno indice; il problema (nei sistemi classici) tuttavia è trovare un criterio di similarità opportuno poichè a seconda del problema specifico la misuradi similarità più opportuna potrebbe variare.

- Successivamente occorre combinare i risultati in modo ottimale: ogni indice si sofferma su un aspetto diverso del problema e devono essere combinati in modo opportuno. Questo processo è fondamentale per interrogare diverse basi di dati con una sola query. Di fatto quest ultimo problema è risolto tramite tecniche di apprendimento automatico (*machine learning* o *deep learning*), che ricavano una funzione di sintesi in modo automatico.

- Combinati i valori, si ha comunque un problema di decisione (cosa includere nel grafo di conoscenza), introducendo un valore di soglia o tramite tecniche di ottimizzazione, o ancora tramite tecniche di apprendimento automatico.

In rarissimi casi tuttavia un algoritmo riuscirebbe a ottenere prestazioni pari a quelle umane: integrando grandi quantità di dati l'automatizzazione è necessaria ma situazioni critiche richiedono l'intervento umano. Si deve quindi trovare un compromesso tra automatizzazione e intervento umano grazie all'*interattività*, soprattutto in caso di risultato incerto.

Oltre tutto ciò occorre anche definire che tipo di similarità stiamo ricercando, poichè ne esistono di diveri tipi:

- Similarità sintattica, per cui si ricerca un'uguaglianza tra due parole che hanno la stessa forma, ma questo presta il fianco alla possibilità di effettuare un matching errato assumendo che due parole uguali rappresentino sicuramente la stessa cosa;

- Similarità lessicale, basata sui sinonimi, che però rischia di creare molta ambiguità;

- Similarità strutturale, che tiene conto di come gli oggetti che intendiamo matchare siano inseriti nelle rispettive strutture, ma a seconda delle basi di dati potremmo trovare

strutture completamente differenti e, a quel punto, questo particolare tipo di similarità provocherebbe molti problemi.

L'approccio *pay as you go* tenta di integrare dando una priorità ai dati, che nel tempo saranno consolidati e modificati: così facendo si possono automatizzare dei processi, offrendo un risultato sporco, in attesa di una rifinitura umana.

Si tenta di sfruttare le stesse ontologie per fonti diverse, nonostante questo non sia sempre possibile (perchè di fatto possono essere preferiti linguaggi propri anche con *open data*).

1.2 Ontology matching: *Agreement Maker*

Come già detto, l'*ontology matching* è il task che ha l'obiettivo di troavare corrispondenze tra entità appartenenti ad ontologie differenti. A questo proposito *Agreement Maker* è un *tool* per aiutare esperti di dominio proprio nel processo di *ontology matching* (ovvero matching a livello di schema): è considerato uno dei migliori esistenti a livello internazionale.

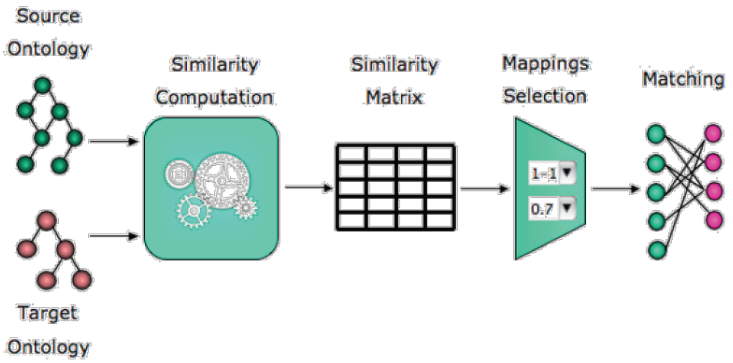
I metodi utilizzati dal software possono essere diversi:

- *Concept based*
- *Structure based*
- *Instance based*

Si tenta quindi di costruire delle corrispondenze tra termini appartenenti a ontologie diverse: la prima si chiama *source ontology* che si allinea alla *target ontology*. Mappando la *source ontology* si calcola un valore di confidenza per il processo di matching; inoltre si può stabilire la cardinalità tra sorgente e obiettivo: una cardinalità `1:1` stabilisce una relazione unica tra due entità, così come `1:n` stabilisce un rapporto per cui ad un'entità della fonte ne corrispondono più nell'obiettivo.

In sostanza il software esegue la definizione dei matching seguendo dei passaggi precisi:

1. Prende in input la *source ontology* e la *target ontology*;
2. Effettua il calcolo della misura di similarità scelta;
3. Costruisce una matrice di similarità;
4. Applica i vincoli di cardinalità;
5. Infine definisce i matching risultanti.



Il software nella versione normale calcola una matrice di similarità di dimensioni $N \times M$ (dove N ed M sono le cardinalità delle due ontologie) usata per individuare relazioni di equivalenza. Si effettua un *match* solamente sugli elementi che superano una certa soglia. Le dimensioni della matrice però degenerano velocemente e quindi la gestione della memoria è ardua: la versione *lite* tuttavia conserva in memoria solamente i valori superiori alla soglia, così da ridurre la memoria utilizzata.

Gli indici utilizzati sono:

- La *precision* $P = \frac{TP}{TP+FP}$ (dove TP sono i valori classificati correttamente come positivi e FP i valori classificati scorrettamente come positivi);
- La *recall* $R = \frac{TP}{TP+FN}$ (dove FN sono i valori classificati scorrettamente come negativi);
- La loro media armonica *F-measure* $F = 2\frac{P \cdot R}{P+R}$

Queste misure sono usate in *machine learning* per misurare la bontà di adattamento di un classificatore generico.

Il processo attraverso cui Agreement Maker (come altri tool) sottopone i dati si articola su 3 livelli, che a loro volta si articolano nell'utilizzo e nella combinazione dei risultati di più matcher basati su aspetti differenti tra loro:

- **First layer**: concettuale
  - BSM Base Similarity Matcher
  - PSM Parametric String-based Matcher
  - VMM Vector-based Multi-term Matcher
- **Second layer**: strutturale
  - DSI Descendant's Similarity Inheritance
  - SSC Sibling's Similarity Contribution
- **Third layer**
  - LWC Linear Weighted Combination

Con particolare riferimento al **VMM**, un testo è riassunto come insieme di termini, costruendo documenti virtuali per gli oggetti da comparare con pesi preimpostati: si calcola quindi la *cosine similarity* usando le frasi come vettori:

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

Al secondo livello, prendendo in input i risultati dei matcher precedenti e basandosi sul principio che la similarità è ereditabile e può anche essere comparata tra nodi fratelli, si possono calcolare indici come il **DSI** ed il **SSC**, entrambi basati sul prendere in input i risultati del **BSM**.

L'indice $DSI \in [0; +1]$ modifica la similarità base tenendo conto della similarità dei nodi ascendenti.

$$DSI = MCP \cdot bas\_sim(C, C^*) + \frac{2(1 - MCP)}{n \cdot (n + 1)} \sum_{i=1}^{n} (n + 1 - i) \cdot base\_sim(parent_{C_i}, parent_{C_i^*})$$

Dove *MCP*, il *main contribution percentage*, empiricamente è un valore intorno a 0.75 in quanto la similarità dei nodi ascendenti tende a 0.25.

L'indice SSC invece modifica la similarità base considerando la similarità tra i nodi fratelli:

$$SSC = MCP \cdot bas\_sim(C, C^*) + \frac{1 - MCP}{n} \sum_{i=1}^{n} \max\{base\_sim(S_i, S_1)...base\_sim(S_i, S_m)\}$$

Sia in riferimento ai valori di misure come *Precision* e *Recall*, che in merito a tempi di esecuzione, l'indice *DSI* si dimostra spesso il migliore.

Una volta superata la fase di calcolo della similarità e di creazione della corrispondente matrice occorre passare attraverso la fase di *selezione* dei mappings che prende in input:

- Matrice di similarità;
- Un valore soglia definito dall'utente;
- Una cardinalità per le corrispondenze tra source e target.

Per assolvere a questo scopo, con particolare riferimento a relazioni di cardinalità 1:1, è spesso utilizzato il sisterma *greedy*: tuttavia questo massimizza solamente il match del singolo elemento e non complessivamente; si tratta di un problema di *assegnamento*, risolvibile tramite metodi combinatori (*Hungarian Method*, di complessità cubica) ma molto pesanti computazionalmente. Approccio ottimale per risolvere il problema è selezionare solamente i match che superano una certa soglia e utilizzare l'algoritmo *maximum matching*, che ha complessità quadratica (e quindi computazionalmente più efficiente). Proprio per questo motivo, Agreement Maker nella sua versione Lite conserva una matrice di similarità sparsa.

Grazie a tecniche di *machine learning* si possono calcolare soglie, combinazioni lineari tra i *matcher* o ancora costruire *matcher* nuovi (nonostante sia richiesta una gran quantità di esempi).

Un approccio molto diffuso, alternativo alla semplice combinazione dei risultati dei diversi matcher, per rendere il matching più affidabile e snello è quello della combinazione dei risultati dei matcher basata però sulla qualità del matching stesso. Un esempio di misura di qualità, se è noto, è un *gold standard* (ovvero un esempio i cui risultati sono già noti). Molto raramente si presenta questa possibilità ed è quindi necessario ricorrere a misure di qualità differenti:

- A livello globale abbiamo più che delle misure, delle vere e proprie euristiche: quella di conservazione della distanza che si basa sul fatto che un match F tra due concetti in due ontologie non dovrebbe distorcere la distanza che c'è tra i concetti stessi e quella di conservazione dell'ordine che afferma la stessa cosa ma in merito all'ordine che i concetti possiedono all'interno delle rispettive ontologie;
- A livello locale con il calcolo di una vera e propria misura di confidenza.

Un buon *matcher* dovrebbe dare un'alta affidabilità a pochi collegamenti, e dovrebbe restituire un'affidabilità nulla in caso di elementi non compresi nell'ontologia. Inoltre, può capitare che certi concetti siano identificati meglio di altri: si calcola così il livello di confidenza locale

$$LC_M(c) = \frac{\sum_{c' \in m_M(c)} sim_M(c, c')}{|m_M(c)|} - \frac{\sum_{c' \in (T - m_M(c))} sim_M(c, c')}{|T - m_M(c)|}$$

Il livello di confidenza così calcolato a livello locale ci permette di definire dei pesi basati sulla qualità dei matching dei singoli matcher e, una volta definiti i pesi, di utilizzarli per combinare i risultati dei diversi matcher in una vera e propria media pesata *LWC*. A questo punto, sulla scia di quanto detto in precedenza, si può procedere a calcolare i diversi indici già noti, come il *DSI*. In sostanza l'approccio basato sulla qualità ha lo scopo primario di dare un peso diverso all'apporto di ogni matcher sul risultato finale basandosi sulla qualità del matcher misurata *localmente*.

Quanto detto finora vale soprattutto per il matching e la valutazione di ontologie reali molto grandi ma esistono tutta una serie di task analoghi che presentano ancora molte criticità e sfide:

- Per le ontologie *linked open data* le cose sono più complesse;
- Cross-lingual ontology matching;
- Metodi di matching interattivi;
- Semantic table annotation.

1.3 *Instance matching*

È un problema relativo alle basi di dati: si tratta di effettuare un *record linkage* (se fatto su due dataset diversi) o *deduplication* (se fatto sullo stesso dataset). Lo stesso oggetto del mondo reale può corrispondere a più oggetti in un database, e servono tecniche per identificare le corrispondenze; inoltre i valori possono essere comunque diversi; tra due database diversi persino l'identificativo può non corrispondere, complicando il processo. Ci possono essere persino errori voluti: in piattaforme di compravendita, il criterio più usato per ordinare è per prezzo crescente, quindi venditori di accessori possono dichiarare di vendere il prodotto per salire più in alto nella lista.

La tecnica più diffusa di matching prevede che siano dati per corretti link con valore superiore ad una certa soglia, e incerti se superiori ad un'altra soglia più bassa (tecnica della doppia soglia), che saranno poi confermati dall'utente.

Le tecniche si dividono in varie categorie.

Empirico. È un sistema che calcola la similitudine tra due tuple considerando solo la prossimità ortografica (tenendo anche conto della distanza dei tasti sulla tastiera). È usato per migliorare il matching dopo l'uso di altre tecniche.

Probabilistico. Usa delle tecniche probabilistiche per calcolare la distanza tra due tuple.

Misti. Mischia le ultime due tecniche per effettuare un match migliore.

Le dimensioni delle ontologie possono essere problematiche dato che le comparazioni da effettuare sono $n_1 \times n_2$, il prodotto cartesiano tra i due dataset. Si usano quindi tecniche di *blocking* per ridurre il numero di comparazioni e quindi lo spazio della ricerca. Anche in database a grafo e rappresentazioni RDF si hanno problemi simili con delle differenze minime. Ridotto lo spazio di ricerca, si applica un modello che distingua tra *match*, *possible match* o *not match*.

L'approccio probabilistico usa tecniche di *machine learning* per identificare i *match*, generando un algoritmo proprio. L'algoritmo sceglie una funzione di distanza (o una combinazione di più funzioni) e valori soglia in modo automatico.

Preprocessing. Per prima cosa i dati devono essere normalizzati eliminando variazioni sintattiche o armonizzando gli standard. Inoltre si tenta già di filtrare (tramite *parsing* o *machine learning*) i dati in modo da estrarre entità dal dato.

Blocking. Per ridurre il numero di comparazioni da effettuare, si cerca una chiave su cui ordinare i record in entrambe le tabelle; si usa quindi una finestra mobile per cercare collegamenti in posizioni simili. La complessità passa da $O(n^2)$ a $O(\frac{m^2 n}{m})$ (dove m è la dimensione della finestra).

Distanza. La distanza tra due tuple è l'elemento su cui è effettuata l'indagine statistica: minore è la distanza, maggiore è la percentuale di *match*; altro punto cruciale è la scelta della soglia. Il valore deve dare delle buone prestazioni ma allo stesso tempo ridurre il numero di richieste fatte all'utente. Si considerano analoghe due tuple con distanza minore della soglia.

Il processo è effettuato su un campione di dati di una certa dimensione; per migliorare ulteriormente la qualità dell'algoritmo si possono anche usare tecniche di *active learning* usando solamente osservazioni critiche per costruire il campione.

1.4 *Pay as you go.*

Si usa l'approccio *pay as you go* per aggiungere osservazioni da analizzare in modo automatico e quindi migliorare la classificazione in momenti successivi alla pubblicazione dell'algoritmo.

1.5 Gestione dei conflitti con la textitdata fusion.

La gestione dei conflitti può essere fatta rimuovendo il match, aggiungendo metadati per far sì che la scelta rimanga all'utente o impostare una gerarchia tra le osservazioni (come far prevalere la più recente). Questi tuttavia non sono metodi finali di risoluzione ma solamente sistemi di ausilio.

2 Analisi del testo.

Si usano tecniche di analisi testuale per processare i testi:

Stop words. Si eliminano le parole più diffuse che non alterano il senso principale della frase (nell'analisi), come congiunzioni o articoli.

Tokenizzazione. Si regolarizzano i nomi eliminando la punteggiatura.

Lemmatizzazione. Si riconduce una parola alla sua radice, in modo da estrapolare più facilmente il significato; tuttavia si rischia di sottovalutare l'ambiguità sintattica.

Streming. TO DO.

Si usano quindi misure di similarità sulla tupla processata per eseguire i match. Questi comparano i singoli elementi della tupla, quindi il risultato delle funzioni sopracitate. La similarità deve essere $sim(x, y) = 1$ in caso di corrispondenza perfetta.

Si parla anche di *metrica* di dimensionalità, se sono rispettate certe caratteristiche:

$$\begin{aligned} &dist(x, x) = 0 \\ &dist(x, y) \geq 0 \\ &dist(x, y) = dist(y, x) \\ &dist(x, y) \leq dist(x, z) + dist(y, x) \end{aligned}$$

Le ultime due proprietà tuttavia non sono sempre rispettate da tutti gli indici.

Esiste una funzione di *edit distance* chiamata Levenshtein, dove ogni modifica (inserimento, cancellazione o sostituzione di caratteri) che deve essere apportata per passare da una stringa v_1 a una stringa v_2: non misura una distanza ma informa quanto il primo valore è accurato rispetto al secondo. Tuttavia la sostituzione in caso di errori di battitura è più importante di sostituzioni casuali, e l'inserimento di spazi o altri segni di punteggiatura è eseguito in modo casuale, inoltre la modifica di cifre latine è particolarmente significativo; sono quindi calcolati pesi per le singole sostituzioni, eseguiti in modo automatico.

Altra tecnica di analisi è processare le stringhe in sottostringhe di lunghezza definita (n-grammi): si calcola poi una distanza considerando le tuple di n-grammi; questo indice è robusto rispetto agli errori di battitura ma poco rispetto alle sostituzioni (al contrario dell'indice di Jaccard).

3 Grafo di vicinato.

Si considera *grafo di vicinato* il sottografo comprendente gli elementi raggiungibili in h passi da un nodo i. Si comparano i grafi grazie a misure di similarità su grafi, che però sono costose: si preferisce stabilire uno spazio di *features* di dimensione pari al numero di entità nel grafo, così che ogni entità abbia un suo peso. Così si può attribuire un peso proporzionale alla distanza o al numero di relazioni che un'entità ha con le altre.

4 *Matching cross language.*

Si possono cercare match tra più lingue diverse rappresentando ogni parola come un vettore di concetti pesato. Si cerca quindi un sistema di disambiguazione leggera: si calcola il vettore

centroide (ovvero medio) tra le due stringhe per calcolare la distanza. Questo sistema funziona bene per testi brevi, ma la complessità degenera con la lunghezza del documento.

Parte III
Esercitazioni

1 SPARQL

Si interroga `https://dbpedia.org` per ottenere una lista di vulcani:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT distinct ?v
  FROM <http://dbpedia.org>
 WHERE {
         ?v rdf:type dbo:Volcano .
       }
 LIMIT 100
```

I prefissi si aggiungono inserendo l'url tra simboli `<` e `>`. `rdf:type` può essere sostituito da una `a`:

```
?v a dbo:Volcano .
```

Una query SPARQL è strutturata specificando:

- prefissi
- tipo di query
- (fonte)
- (filtri)
- eventuali altre clausole

Le specifiche SPARQL possono essere ottenute all'indirizzo `https://www.w3.org/TR/rdf-sparql-query/`

Esercizio 1

Si tenta di trovare il numero di dipendenti Google dalla pagina `http://dbpedia.org/page/Google`:

```
PREFIX res: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?num
  FROM <http://dbpedia.org>
 WHERE {
         res:Google dbo:numberOfEmployees ?num .
       }
```

```
# o piu' semplicemente:
SELECT ?num
  FROM <http://dbpedia.org>
 WHERE {
          dbr:Google dbo:numberOfEmployees ?num .
        }
```

Esercizio 2

Si cerca di ottenere una lista di trombettisti leader di band:

```
SELECT ?person
 WHERE {
       # soggetto    proprieta'       oggetto
         ?person     dbo:instrument   dbr:Trumpet .
         ?person     dbo:occupation   dbr:Bandleader .
        }
```

Esercizio 3

Si cercano i Paesi con più di due grotte:

```
  SELECT ?country
   WHERE {
           ?cave      rdf:type       dbo:Cave .
           ?cave      dbo:location   ?country .
           ?country   rdf:type       dbo:Country .
         }
GROUP BY ?country
  HAVING (COUNT(?cave) > 2)
```

Esercizio 4

Si vogliono trovare tutti i software sviluppati da società californiane.

```
SELECT DISTINCT ?software
 WHERE {
         ?company rdf:type dbo:Organisation .
         {           # dbpedia validata
           ?software dbo:developer ?company .
         } UNION {   # dbpedia da validare
           ?software dbp:developer ?company .
         }
         ?software rdf:type dbo:Software .
         ?company dbo:foundationPlace dbr:California .
        }
```

Esercizio 5

Verificare se Natalie Portman è nata negli Stati Uniti:

```
ASK
WHERE {
        dbr:Natalie_Portman dbo:birthPlace ?city .
        ?city dbo:Country dbr:United_States .
      }
```

Esercizio 6

Verificare se Roma è la capitale d'Italia:

```
ASK
WHERE {
        dbr:Italy dbo:capital dbr:Rome .
      }
```

Esercizio 7

Verificare se è disponibile la data di nascita di Elizabeth Taylor.

```
ASK
WHERE {
        dbr:Elizabeth_Taylor dbo:birthDate ?_ .
      }
```

Esercizio 8

Verificare se Milano è in Germania.

```
ASK
WHERE {
        dbr:Germany dbo:city dbr:Miland .
      }
```

Esercizio 9

Verificare se Michelle Obama è sposata con Barack Obama.

```
ASK
WHERE {
        dbr:Michelle_Obama dbo:spouse dbr:Barack_Obama .
      }
```

Esercizio 10

Si cercano i giocatori di Football americano che pesano più di 100kg.

```
CONSTRUCT {
            ?x rdf:type dbo:AmericanFootballPlayer .
          }
    WHERE {
            ?x dbo:weight ?kg .
            FILTER(?kg > 1000000000)
          }
```

2 Protégé

Progetto *Open Source* per gestire ontologie, permette di caricare direttamente l'ontologia di `dbpedia.org` o di `schema.org`. DBPedia usa più di 400 classi, usandone anche di *sporche* (ovvero ripetute o poco utili). Usando il *reasoner*, sono evidenziate le incongruenze (ovvero classi inconsistenti), che non possono avere dunque istanze.

Costruzione di un'ontologia

Si costruisce una semplice ontologia per la costruzione di un semplice domino geografico.

`City < Region < Country < Continent`

In Protégé, si definisce una nuova ontologia con `File > New` e inserendo poi sottoclassi di `owl:Thing`. Popolata l'ontologia con delle istanze arbitrarie, si definiscono le proprietà. Si cerca di ridurre al minimo il numero di relazioni (si ha un'intuizione della definizione di contenimento), ma non è possibile tentando di ridurre al minimo le classi. Nel menù `Object properties` è possibile definire nuove proprietà e definire dominio e immagine; si attribuiscono poi le proprietà alle istanze andando su `Entities > Individuals` e cliccando su `Object properties assertions` (a destra).

3 *Ontology matching*

Si tenta di fare prima dei *match* a mano tramite un'analisi esplorativa, eliminando entità troppo specifiche e selezionando le corrispondenti. Questo approccio è estremamente lento e dispendioso, per cui si tenta di automatizzarlo. Si introducono degli operatori di *matching* per indicare il rapporto tra le relazioni:

$$\begin{array}{ll} a = b & a \text{ corrisponde a } b \\ a \leq b & a \text{ è più specifico di } b \\ a \geq b & a \text{ è meno specifico di } b \end{array}$$

Si indica con $\square^*$ se la relazione non è certa ma solamente ipotizzata. In SKOS sono presenti tutte le relazioni, con proprietà di transitività e associatività.