UNIVERSITÀ DEGLI STUDI ROMA TRE

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Disambiguazione lessicale con il grafo di BabelNet

Laureando

Silvio Severino

Matricola 499262

Relatore

Prof. Francesco Benedetto

Correlatore

Prof. Roberto Navigli

Anno Accademico 2017/2018

In memoria di mio padre

Ringraziamenti

Sebbene io sia una persona introversa, ci tenevo a dedicare qualche riga per ringraziare tutte le persone che mi sono state vicine durante il mio percorso di studi. Inizio ringraziando il professor Francesco Benedetto, mio relatore, per la sua professionalità, la sua disponibilità e per l'aiuto che mi ha dato in questi mesi. Ringrazio il professor Roberto Navigli, nonchè mio tutor aziendale, per i suoi consigli, riguardanti l'informatica e non solo. Vorrei ringraziare i miei colleghi universitari per avermi supportato, con i quali ho condiviso momenti che non dimenticherò mai. Un ringraziamento speciale va alla mia famiglia, in particolare a mia madre e mia nonna, che hanno sempre creduto in me e mi hanno sempre spronato in sfide che magari non avrei mai avuto il coraggio di affrontare. Di persone da ringraziare durante il corso di questi tre anni ce ne sono tante, tuttavia ne vorrei ringraziare una in particolare, con la quale ho condiviso i momenti più forti e meravigliosi della mia vita e, grazie alla quale, sono riuscito a ritrovare molto spesso il sorriso, un pezzo del mio cuore sarà sempre il suo. Infine voglio ringraziare me stesso e la mia forza di volontà, che mi ha permesso di superare momenti bui e, grazie alla quale, sono riuscito sempre a trasformare quelle difficoltà in forza.

Silvio Severino

Introduzione

Premessa

In questo elaborato verrà descritta la relazione finale all'esperienza del tirocinio formativo della durata di cinque mesi, svolto dal candidato presso l'azienda Babelscape.
Questo saggio illustrerà lo scopo del persorso formativo, finalizzato alla progettazione e all'implementazione di un disambiguatore lessicale per la lingua inglese, creato mediante l'utilizzo del dizionario enclicopedico BabelNet. Il saggio in questione sarà sviluppato attraverso l'analisi di cinque capitoli, qui di seguito descritti brevemente e successivamente spiegati nel dettaglio:

1. **Stato dell'arte**: verrà descritto il contesto nel quale il candidato ha lavorato, ponento particolare attenzione alle problematiche della **Word sense disambiguation**.
2. **Risorse metodologiche e tecnologiche**: verranno descritte le risorse grazie alle quali è stato possibile realizzare il progetto.
3. **Analisi dei requisiti**: verranno analizzate le principali problematiche riscontrate e le relative soluzioni apportate mediante scelte progettuali.
4. **Progettazione ed implementazione del disambiguatore**: verrà descritto in maniera dettagliata il processo grazie al quale è stato possibile implementare il disambiguatore.
5. **Sviluppi futuri**: verranno teorizzate le possibili migliorie apportabili all'elaborato e probabili applicazioni dello stesso in contesti tecnici.
6. **Conclusioni**: considerazioni.

Indice

Introduzione **iv**

Premessa iv

v

Indice **v**

Elenco delle figure **viii**

1 Stato dell'arte **1**

1.1 Intelligenza artificiale 1

1.2 Elaborazione del linguaggio naturale 2

1.3 Semantica lessicale 2

1.3.1 Classificazione 3

1.3.2 Relazioni fra lessemi e sensi 3

1.4 Word Sense Disambiguation 6

1.4.1 Problematiche 7

1.4.2 Metodologie di disambiguazione 7

2 Risorse metodologiche e tecnologiche **10**

2.1 BabelNet 10
2.2 WordNet 16
2.3 Word embedding 18
2.3.1 Word2Vec 19
2.4 Ambiente di sviluppo 22
2.4.1 Sistema operativo 22
2.4.2 Eclipse 22
2.4.3 Java8 23
2.5 Librerie utilizzate 23
2.5.1 JavaBabelNetAPI 23
2.5.2 JGraphT 23
2.5.3 Medallia - Word2VecJava 24
2.5.4 StanfordCoreNLP 24

3 Analisi dei requisiti **25**
3.1 Scopo del tirocinio 25
3.2 Interfacce Handler e Builder 26
3.2.1 WordNetNodeBuilder 27
3.2.2 SenseAndGlossBuilder 29

4 Progettazione ed implementazione del disambiguatore **32**
4.1 GraphBuilder 33
4.1.1 Algoritmo di costruzione 34
4.2 Componente Ranker 39
4.2.1 Algoritmo di ranking 39

5 Sviluppi futuri **41**
5.1 Migliorie all'algoritmo di ranking 41
5.2 Ottimizzazioni ulteriori 41
5.2.1 Occupazione di memoria 42
5.2.2 Riduzione delle tempistiche d'esecuzione 42
5.3 Futuro per BabelNet 42

5.3.1 Estensione ad altre lingue 43

Conclusioni **44**

Bibliografia **45**

Elenco delle figure

2.1 La struttura di BabelNet per il BabelSynset *play* 12
2.2 La pagina di dettaglio per il BabelSynset *Steve Jobs* 14
2.3 La rete relativa al BabelSynset *Steve Jobs* 15
2.4 La struttura semantica relativa al lemma *car* 17
2.5 Rete neurale del tipo *Skip-Gram* 21
2.6 L'interfaccia Graph di Jgrapht 24

3.2 Il risultato del metodo *build()* 27
3.3 Il risultato del metodo *buildSynID2Edge()* 28
3.4 Il risultato del metodo *build()* 30
3.5 Il risultato del metodo *buildExistSense()* 31

4.1 Il risultato del refactoring con BabelSenseAndEdges 33
4.2 La struttura del disambiguatore 34
4.3 Costruttore e variabili di istanza 35
4.4 Metodo *build()* 36
4.5 Algoritmo ricorsivo 38
4.6 Algoritmo di ranking 40

Capitolo 1

Stato dell'arte

1.1 Intelligenza artificiale

L'intelligenza artificiale, che successivamente verrà chiamata IA, è un ramo dell'informatica che permette di progettare e programmare sistemi sia di hardware che di software al fine di dotare le macchine di caratteristiche che vengono considerate tipicamente umane. Ad esempio, la capacità di comporre frasi di senso compiuto, agire automaticamente, riconoscere immagini e molte altre. Il vero scopo di questa disciplina è quello di rendere le macchine in grado di **comprendere** il mondo che le circonda e agire di conseguenza. Considerando che tale quesito è veramente complesso, l'IA è suddivisa in più macro aree, al fine di comporre piccoli pezzi di un puzzle, tutti funzionali e indirizzati a risolvere un unico problema [Van16]. Un famoso test per capire se una macchina è intelligente o meno, è stato proposto da Alan Turing nel 1950. Un sistema intelligente ottiene un risultato positivo, nella prospettiva di Turing, se un qualunque individuo, interagendo con una macchina e dopo averle proposto una serie di quesiti, non riesce più a rendersi conto della reale entità del suo interlocutore e quindi se i responsi provengono da un altro individuo o da un calcolatore[SR16].
Caratteristica fondamentale per una macchina intelligente è quella di essere in grado di capire il linguaggio umano. Questa macro area dell'IA è comunemente conosciuta come "*Elaborazione del linguaggio naturale*" (NLP). Essa si occupa di risolvere tutti quei problemi complessi legati all'ambiguità del linguaggio naturale, dovuta all'esistenza di

una diversità di linguaggi, e non solo. L'elaborato di questa tesi si occupa proprio della questione ivi descritta, ponendo l'attenzione, in particolar modo, sul problema del linguaggio naturale e della sua ambivalenza.

1.2 Elaborazione del linguaggio naturale

L'elaborazione del linguaggio naturale, come menzionato nella **Sezione 1.1**, consiste nello sviluppo automatico di informazioni scritte o parlate usate nel linguaggio umano, conosciuto anche come linguaggio naturale.
Questo processo è particolarmente complesso a causa di tutte le caratteristiche proprie del linguaggio. Per comprendere il significato di un fonema o di una conversazione si deve associare ad ogni singola parola il suo corretto significato. Tale associazione, che si applica in maniera inconscia e naturale, è una forma propria dell'essere umano. Esso infatti, sin dalla sua infanzia, impara ad associare una parola al suo significato, in relazione al contesto in cui il vocabolo è usato. Replicare il meccanismo sopra descritto in una macchina è estremamente difficile per una serie di ragioni. La prima tra tutte è che non è sufficiente conoscere i significati ma è altresì necessario possedere tutte le relazioni semantiche tra i vari sensi, le quali a loro volta sono legate ad uno specifico contesto. Idealmente, un computer necessita di una base di dati in cui codifica, in un formato leggibile da un'altra macchina, le relazione semantiche nonchè tutti i concetti, sia a livello lessicografico che a livello enciclopedico-linguistico. In termini tecnici questo processo è conosciuto come disambiguazione (vedi Sezione 1.4).

1.3 Semantica lessicale

La semantica lessicale è un ramo della semantica generale, cioè quella parte della linguistica che studia il significato delle parole, degli insiemi di parole, delle frasi, dei testi e così via. Essa consiste nell'analisi di come e che cosa esprimono i vocaboli di un qualunque linguaggio [sem15]. Per *significato* si intende che la materia fonica del linguaggio acquista la capacità di indicare quello che il soggetto esprime intorno alla realtà circostante. Si noti, che il pieno significato si ha soltanto entro la sintesi di un concetto, racchiuso all'interno di una frase. La seguente terminologia sarà utile al lettore

per capire alcuni termini tecnici utilizzati all'interno del seguente elaborato. Verrà presa d'esempio la seguente frase:

Cane che abbaia non morde.

1.3.1 Classificazione

Il *lessico* è l'insieme di parole per mezzo delle quali i membri di una comunità linguistica comunicano tra loro. Generalmente un lessico ha una forma molto strutturata, al fine di memorizzare significati e possibili usi di una parola, e conseguentemente codificare le diverse relazioni fra parole e significati [Svo10] [Gar10]. All'interno della frase d'esempio, ciascun elemento, così enunciato, assume uno specifico significato. Le *parole* ({cane, che, abbaia, non, morde}) infatti, rappresentano le "forme coniugate", in modo e tempo, di un lessema. Un *lessema* è l'unità minima che viene rappresentata nel lessico {cane, che, abbaiare, non, mordere}. Per comprendere, immaginando una struttura ad albero, i singoli vocaboli si collocherebbero sulle foglie, mentre il lessema ne costituirebbe la radice. Nei vocabolari non vengono elencate le parole, bensì i lessemi, ovvero la **forma di citazione** di una parola (in italiano è l'infinito per il verbo, il singolare per il nome, il singolare maschile per l'aggettivo). Un *dizionario* è una forma di lessico in cui i significati vengono spiegati mediante descrizioni ed esempi. Paradossalmente le singole voci di un dizionario non sono vere definizioni ma bensì enunciati di lessemi elaborate con altri lessemi (con la speranza che l'utente sia sufficientemente informato sugli altri termini). Tuttavia, tali enunciazioni forniscono una notevole quantità di informazioni sulle relazioni fra le parole, grazie alle quali è possibile eseguire operazioni semantiche. Un *lemma* è l'identificatore di un lessema, cioè quello di cui tratta ogni singola definizione di un dizionario.

1.3.2 Relazioni fra lessemi e sensi

Strutturalmente, determinate parti di una frase o di un discorso possono essere tra loro codificate mediante un processo che prende il nome di *tagging di parte del discorso*, comunemente noto con la sua traduzione inglese *part of speech (POS)*. Tale processo prevede la marcatura di parole in un test come corrispondente a una particolare parte

del discorso, basandosi sulla sua relazione con parole adiacenti e correlate in una frase o in un paragrafo. [Gro] In informatica, la POS, può essere eseguita mediante algoritmi stocastici; tra i più noti si ricorda il tagger di E. Brill, basato su tecniche di apprendimento supervisionato. Tuttavia una forma semplificata della POS è insegnata ai bambini in età scolastica, i quali, imparano ad identificare parole: nomi come nomi, verbi come verbi, aggettivi come aggettivi e così via.
Nel NLP, il POS è una tecnica fondamentale perchè consente di poter andare a lavorare tra le molteplici relazioni presenti fra i lessemi e i loro rispettivi sensi. Di seguito verrano approfondite alcuni tipi di legami.

1.3.2.1 Polisemia

La *polisemia* corrisponde al caso di un lessema che ha più significati correlati fra loro. Per questi tipi di lessemi occorre definire: 1) un modo per capire quanti e quali sono i sensi e se essi siano effettivamente distinti (lavoro dei lessicografi), 2) in che modalità sono correlati fra loro i vari significati e 3) come possono essere diversificati tra di loro al fine di interpretare correttamente una parola in un certo contesto di una proposizione (vedi **Sezione 1.4**). Grazie alla polisemia si possono esprimere più significati con una sola parola, ma ciò può diventare un' importante fonte di ambiguità, soprattutto in caso di contesti il cui contenuto è povero o le parole generiche. Oltre all'ambito in cui una parola è inserita, i mezzi grammaticali per questi tipi di relazioni possono aiutare notevolmente nella disambiguazione: nelle frasi {*la radio è accesa*} e {*il radio è un osso*} l'articolo davanti alla parola "radio", rispettivamente femminile e maschile, differenzia il fatto che la prima si riferisce al mezzo di comunicazione, mentre la seconda ad un osso del corpo umano. Un altro fattore che favorisce la risoluzione dell'ambiguità può essere l'utilizzo dell'aggettivo post- o pre-nominale (*un uomo povero* e *un povero uomo*). Tuttavia la fonte di ambiguità più forte nelle relazioni polisemiche è data da parole omografiche, ovvero quelle con più funzioni grammaticali: *il piano di lavoro*(nome), *il terreno piano*(aggettivo), *cammina piano!*(avverbio).

1.3.2.2 Omonimia

L'*omonimia* è una relazione fra parole che hanno la stessa forma ma significati distinti. Ad un primo sguardo, l'omonimia, può apparire uguale alla polisemia, ma il suo meccanismo è ben diverso. Un possibile esempio può essere quello della parola "vite", i cui significati possono essere: il plurale di vita, una pianta, ed altri. Questo tipo di relazione è una causa di ambiguità per l'interpretazione di una frase, in quanto genera più lessemi con la stessa grafia. Inoltre, esistono relazioni omonime che sono anche *omofone*, ovvero che hanno la stessa pronuncia. Come per le parole polisemiche il contesto e la grammatica può aiutare nel processo di disambiguazione, ma gli strumenti per le relazioni polisemiche sono poco efficaci per questi tipi di legami. Tuttavia, nel caso particolare di parole omofoniche, può essere utile approfondire lo studio della pronuncia, la quale funge da mezzo per la risoluzione dell'ambiguità. Un esempio possibile può essere dato dalle frasi *mi piace la pèsca* e *mi piace la pésca*, nelle quali la pronuncia, ovvero l'accento in questo caso, fa differenziare il frutto dallo sport.

1.3.2.3 Sinonimia

La *sinonimia* è una relazione fra due diversi lessemi con lo stesso significato (ovvero sono sostituibili in un certo contesto senza modificarne il contenuto o la correttezza). Tuttavia a volte, due lessemi sinonimi, non sono del tutto intercambiabili, per via della forma grammaticale della frase, per la fonetica o altro. Si possono avere due tipi di sinonimi, quelli assoluti e quelli approssimativi. Nel primo caso si hanno tutti quei tipi di sinonimi interamente intercambiabili tra di loro. Un possibile esempio può essere dato dalle proposizioni *tra* e *fra*, con l'unica eccezione del fattore eufonico (come suona la frase). Tuttavia la maggior parte delle relazioni sinonime è del tipo approssimativo. Questi tipi di relazioni non formano una vera e propria forma di ambiguità, tranne in alcuni casi particolari, ma possono essere processate andando semplicemente a guardare la forma grammaticale di una frase. La sinonimia è un forte strumento per le tecniche di "article spinning", ovvero quei processi che prendono una frase, un testo o un articolo e cambiano la forma degli stessi andando a modificare di volta in volta i sinonimi, senza stravolgere il senso della frase. Questo processo può essere effettuato all'infinito, fino a

quando non si esauriscono i sinonimi di una parola [Ver18]. Tipi particolari di sinonimi sono gli iperonimi e gli iponimi.

1.3.2.4 Iponimia, Iperonimia e Olonimia

Le relazioni di *iponimia* e *iperonimia* sono quei tipi di legami fra due lessemi nei quali uno rappresenta la specializzazione (iponimia) o generalizzazione (iperonimia) dell'altro. Si potrebbe dire che il restringimento di significato comporta il passaggio da un iperonimo a un iponimo, mentre l'allargamento è il suo contrario. Una relazione semantica di questo tipo è valida solo per i nomi e i verbi. Un semplice esempio per comprendere meglio queste importanti relazioni potrebbe essere: *abete* è un iponimo di *albero*, *albero* è a sua volta un iponimo di *vegetale*. Sono, altresì, verificate le relazioni opposte: *vegetale* è un iperonimo di *albero*, *albero* è un iperonimo di *abete*. Si parla, invece, di due lessemi legati da un rapporto di *olonimia* se, uno dei due, rappresenta una parte dell'altro (e.g., *finestrino* e *automobile*). Successivamente si capirà nel dettaglio l'importanza di queste relazioni; in quanto, per lo sviluppo di questa tesi, si è fatto affidamento a WordNet, il quale verrà illustrato nella **Sezione 2.2**.

1.4 Word Sense Disambiguation

La disambiguazione (in inglese Word Sense Disambiguation o, abbreviato, WSD) è il processo attraverso il quale si precisa il significato di una parola o di un insieme di parole (frase), che denotano concetti diversi a seconda dei contesti, e che quindi creano ambiguità. Questo processo si applica ai vocaboli con più significati (polisemia). Il problema della disambiguazione, mediante appositi algoritmi, riveste particolare importanza nelle ricerche sull'intelligenza artificiale e, in particolare, sull'elaborazione del linguaggio naturale. Specificamente, si prevedono benefici della disambiguazione in programmi di traduzione automatica, recupero dell'informazione, estrazione automatica di informazioni e molti altri. Per poter effettuare il processo di disambiguazione è necessario un dizionario (o un'enciclopedia) che elenca tutti i possibili sensi di una parola, in particolare per lo sviluppo di questa tesi si è scelto di utilizzare BabelNet (vedi **Sezione 2.1**).
I due esempi seguenti riguardano significati distinti della parola rombo:

Ho acquistato un rombo fresco al mercato.
Il rombo è una figura geometrica

Sebbene per un essere umano sia ovvio che il primo esempio si riferisce a una specie animale e il secondo a quello di una figura geometrica, sviluppare algoritmi per replicare questa capacità è tipicamente complesso [Nav09].

1.4.1 Problematiche

Un problema fondamentale della disambiguazione riguarda l'identificazione dei significati delle parole. Quando una parola assume più significati, essa è detta polisemica. Esistono, inoltre, altri casi di significati differenti strettamente correlati. Un esempio è quello di un significato correlato a un altro significato mediante metafora (ad esempio, divorare un patrimonio) o metonimia (bere un bicchiere). In tali casi, la suddivisione dei concetti è molto più difficile: diversi dizionari forniscono suddivisioni differenti di significati per le parole. Una soluzione adottata dai ricercatori è stata quella di scegliere un particolare dizionario della lingua inglese, WordNet (vedi **Sezione 2.2**), e utilizzare il suo insieme di significati. Ricerche sono state effettuate anche utilizzando gli equivalenti di WordNet in altre lingue (per l'italiano, esistono BabelNet, ItalWordNet e MultiWordNet).

1.4.2 Metodologie di disambiguazione

Allo stato dell'arte esistono varie metodologie di disambiguazione, tra le più note si ricordano le "Dictionary and Knowledge-based", tecniche di "Machine Learning" e infine "Supervised methods, Semi-supervised and Unsupervised methods". Nel corso di questa tesi ne sono state analizzate alcune, qui di seguito spiegate nel dettaglio.

1.4.2.1 Apprendimento supervisionato

Per *apprendimento supervisionato* si intende quella branca del machine learning (ovvero l'apprendimento automatico di una macchina) nella quale si cerca di istruire un sistema informatico in modo tale da consentirgli di risolvere determinati compiti secondo previsioni. Per poter definire un algoritmo di apprendimento supervisionato si parte dal presupposto che se si forniscono all'algoritmo un numero adeguato di esempi, esso sarà in grado di

riportare un risultato molto simile all'esempio dato in ingresso. Per poter effettuare questo, l'algoritmo necessita di una grande mole di dati in ingresso (tipicamente vettori), una grande quantità di dati in uscita e, infine, altrettante relazioni tra ingresso e uscita. Il suo funzionamento prevede di predire il senso corretto di una frase o di una conversazione mediante tecniche di classificazione quali, ad esempio quella di Naive Bayes, reti neurali, alberi decisionali e così via [wik07]. Per questa tesi tale tipo di sistema non poteva essere attuato in quanto non si potevano conoscere a priori i dati in uscita e pertanto il sistema risultava inconsistente.

1.4.2.2 Metodo basato su dizionario

Nel metodo basato sui dizionari, essi si analizzano in quanto forniscono delle informazioni di un contesto legato ai sensi delle parole. Questi tipi di informazioni prendono nome di *glosse*. Il più famoso e semplice algoritmo basato sul metodo dei dizionari è stato formulato nel 1986 da Mike Lesk, un informatico statunitense. Questo algoritmo prevede il calcolo della sovrapposizione fra le glosse, associate ai vari significati delle parole, nella frase. Successivamente si sceglie la combinazione di significati che fornisce il **massimo livello di sovrapposizione** complessiva. Tuttavia questo algoritmo è portatore di alcuni limiti: il primo tra tutti è quello che permette di ottenere prestazioni del 50-70%. Generalmente in un dizionario non sono presenti glosse o esempi legati ad un determinato senso di una parola sufficientemente consistenti da poter addestrare un classificatore, per queste ragioni le previsioni del sistema non risultano soddisfacenti [FV04]; proprio questo limite è stato motivo di eclusione per lo sviluppo di questa tesi.

1.4.2.3 Similarità delle parole

Questo metodo prevede di lavorare maggiormente sulle relazioni sinonime viste come forme strette di similarità, definendole come una distanza semantica fra più parole. Questo metodo per poter essere attuato necessita di un thesaurus (dizionario), in quanto, in esso, i significati delle parole sono legati tra loro da relazioni. Generalmente la similarità delle parole prevede il calcolo della lunghezza del cammino minimo per arrivare da una parola ad un altra usando le relazioni (come mostrato nella Figura 1.1a); tuttavia per problemi legati alla complessità algoritmica e all'occupazione di memoria,

si è preferito adottare un metodo statistico. Quest'ultimo (spiegato successivamente nella **Sezione 2.3.1**) prevede di rappresentare le parole come vettori N-dimensionali, la cui distanza è calcolata come la distanza cosinusoidale tra i due vettori (vedi Figura 1.1b). Il risultato è la probabilità con la quale due parole sono simili a seconda del contesto in cui esse si trovano.

$$\text{sim}(w_1, w_2) = -\log \text{pathlenght}(s(w_1), s(w_2))$$

(a) Word similarity

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

(b) Word cosine similarity

Capitolo 2

Risorse metodologiche e tecnologiche

2.1 BabelNet

Dalla necessità di reperire un dizionario con il quale poter andare ad estrapolare contenuti relazionali che legano più sensi di una parola, la scelta è ricaduta su BabelNet.
BabelNet (https://babelnet.org/) è un dizionario enciclopedico su larga scala nonchè una rete semantica multilingue (più di 280 lingue). Esso è stato creato integrando automaticamente la più grande enciclopedia multilingue, ovvero Wikipedia, con il più noto lessico della lingua inglese, ovvero WordNet. L'integrazione è stata effettuata per mezzo di una mappatura automatica, nella quale voci appartenenti a lingue diverse dall'inglese, sono state ottenute con l'ausilio di algoritmi di traduzione automatica. Il frutto di queste integrazioni è un ***"dizionario enciclopedico"*** che fornisce concetti e voci enciclopediche, lessicalizzate in molte lingue e collegate tra loro da grandi quantità di relazioni semantiche. BabelNet raggruppa le parole in insiemi di sinonimi, chiamati **BabelSynset**, ovvero gli insiemi dei sinonimi di BabelNet (BabelNet synonym set) e, per ciascuno di essi, fornisce definizioni testuali in diverse lingue chiamate glosse (vedi **Sezione 1.4.2.2**), ottenute dalla fusione delle definizioni date da WordNet e Wikipedia. BabelNet codifica la conoscenza come un grafo diretto etichettato $G = (V, E)$ dove V è l'insieme dei nodi - i.e., *concetti* come **"play"** e *entità* come **"Albert Einstein"** - e

$E \subseteq V \times R \times V$ è l'insieme degli *archi* che connettono coppie di concetti - e.g., play *is-a* gioco ricreativo. Ogni arco è etichettato con una *relazione semantica* (sottoinsieme di R), e.g., {*is-a*, *part-of*, ..., $\in$}, dove $\in$ rappresenta una relazione semantica non specificata. È importante sottolineare che ogni nodo $v \in V$ contiene un insieme di lessicalizzazioni di un concetto per ogni lingua, e.g., {play(EN), dramma(IT), obra(ES), ..., pièce de théâtre(FR)}. Per costruire il grafo di BabelNet, si estraggono diversi tipi di informazioni dalle fonti sopra citate: da WordNet, ad esempio, vengono dedotti tutti i sensi disponibili di una parola (come i *concetti*) nonchè tutti gli indicatori lessicali e semantici tra due *synsets* (cos'è un syntets sarà spiegato successivamente nella **Sezione 2.2**), ovvero le *relazioni*; da Wikipedia, invece, tutte le diverse Wikipages (le pagine web di Wikipedia di un certo argomento) e tutte le relazioni semantiche non specificate dai suoi link ipertestuali. Le informazioni estrapolate da WordNet e Wikipedia si sovrappongono in termini di concetti e di relazioni: questa sovrapposizione rende possibile la fusione tra due eventuali risorse, abilitando la creazione di **risorse di conoscenza unificata** (in inglese unified knowledge resources). Per consentire il multilinguismo, sono collegate le relazioni lessicali a tutti i concetti disponibili in tutte le possibili lingue. Infine, vengono connessi nel grafo i BabelSynsets stabilendo le relazioni semantiche tra di essi.
Quindi, questa metodologia consiste in tre passaggi fondamentali:

1. **Vengono integrati WordNet e Wikipedia** tramite la creazione di mappature automatiche tra i sensi, contenuti in WordNet, e le Wikipages. Questa configurazione consente di evitare che i concetti siano duplicati nonchè di ampliare l'inventario ad essi correlato, reciprocamente.

2. **Vengono raccolti i lessemi multilinguistici** dei nuovi concetti "appena creati", ovvero i BabelSynsets, usando:

   - traduzioni estratte da Wikipedia effettuate da persone.
   - sistemi di traduzione automatica funzionale a decifrare modelli di concetti contenuti all'interno di un POS (vedi **Sezione 1.3.2**)).

3. **Vengono create relazioni tra i vari BabelSynsets** mediante la raccolta di tutte le relazioni contenute in WordNet e in Wikipedia, nelle lingue di interesse.

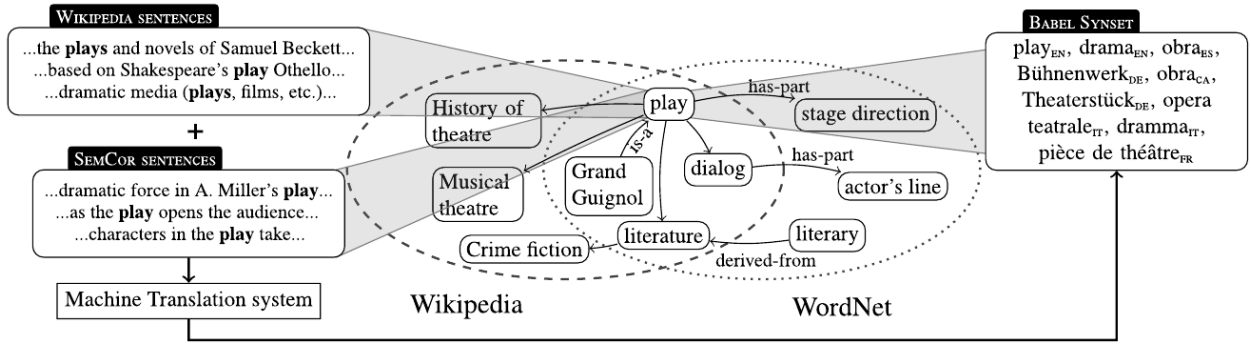


Figura 2.1: La struttura di BabelNet per il BabelSynset *play*

Per questo elaborato è stata utilizzata l'ultima versione disponibile online di BabelNet: BabelNet 4.0. In questa versione, oltre alle informazioni estratte da Wikipedia e WordNet, vengono integrate anche risorse da altre fonti note nel web:

- **OmegaWiki**, un ampio dizionario multilingue collaborativo (aggiornato a Gennaio 2017)
- **Wiktionary**, un progetto collaborativo al fine di produrre un dizionario multilingue gratuito (aggiornato a Febbraio 2018)
- **Wikidata**, un database basato sulla conoscenza - i.e., in termini tecnici base di conoscenza - creata da persone e macchine intelligenti (aggiornato a Febbraio 2018)
- **Wikiquote**, un compendio multilinguistico di citazioni online gratuito provenienti da persone importanti e opere creative (aggiornato a Marzo 2015)
- **VerbNet**, un lessico dei verbi basato su classi (aggiornato alla versione 3.6)
- **Microsoft Terminology**, una collezione delle terminologia che possono essere utilizzate per sviluppare versione adattabile di applicazioni (aggiornato a Luglio 2015)
- **GeoNames**, un database geografico gratuito contenente i nomi di tutti i paesi e tutti i rispettivi toponimi (aggiornato a Aprile 2015)

- **ImageNet**, un database di immagini organizzato secondo la gerarchia di WordNet (versione del 2011)
- **FrameNet**, un database lessicale di parole inglesi leggibile sia da umani che da sistemi intelligenti (versione 1.6)
- **WN-Map**, mappature generate automaticamente tra le versioni di WordNet (versione del 2007)
- **Open Multilingual WordNet**, una raccolta delle informazioni di WordNet tradotti in diverse lingue (inserita in BabelNet da Gennaio 2017): **WoNef**, **WoNeF**, **Albanet**, **Arabic WordNet (AWN v2)**, **BulTreeBank WordNet (BTB-WN)**, **Chinese Open WordNet**, **Chinese WordNet (Taiwan)**, **DanNet**, **Greek WordNet**, **Princeton WordNet**, **Persian WordNet**, **FinnWordNet**, **WOLF (WordNet Libre du Français)**, **Hebrew WordNet**, **Croatian WordNet**, **IceWordNet**, **MultiWordNet**, **ItalWordNet**, **Japanese WordNet**, **Multilingual Central Repository**, **WordNet Bahasa**, **Open Dutch WordNet**, **Norwegian WordNet**, **plWordNet**, **OpenWN-PT**, **Romanian WordNet**, **Lithuanian WordNet**, **Slovak WordNet**, **sloWNet**, **Swedish (SALDO)**, **Thai WordNet**.

BabelNet è disponibile online sia come sito web sia come API (Application programming interface) in diversi linguaggi di programmazione. La sua interfaccia grafica disponibile nel sito web online permette di sfruttare tutta la sua potenzialità in modo semplice e intuitivo. L'utente, collegandosi al sito di BabelNet avrà un barra di ricerca con la quale potrà ricercare tutti i sensi di un lemma, e avrà, inoltre, la possibilità di scegliere tra le 280 lingue disponibili per poter ottenere la traduzione desiderata. Nella pagina dei risultati verranno riportati tutti i sensi di quel lemma (i BabelSynsets), una rispettiva foto e una breve descrizione. L'utente, selezionandone uno, verrà riportato in una pagina di "dettaglio" nel quale potrà trovare maggiori informazioni riguardo a quel BabelSynset, come è possibile vedere dalla Figura 2.2. La pagina dei "dettagli" è divisa in due parti. La prima parte mostra un sommario per il synset contenete le seguenti informazioni:

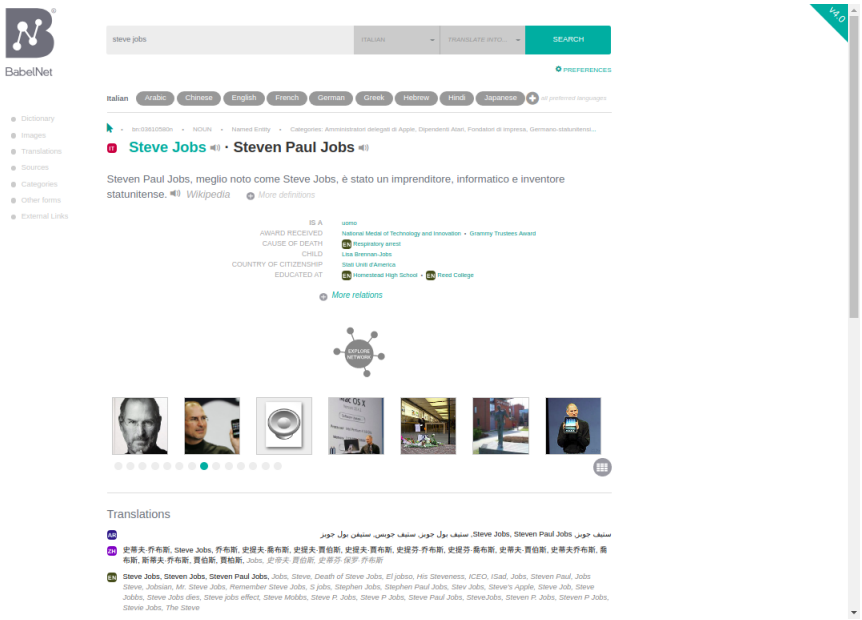


Figura 2.2: La pagina di dettaglio per il BabelSynset *Steve Jobs*

- **Dominio**, la nuova versione di BabelNet classifica i synsets in 34 domini differenti, come ad esempio *Computing Sport and recreation*, *Politics and government*, e molti altri.
- **Lemmi**, i sinonimi principali del lemma.
- **Esempi di utilizzo**, il synset utilizzato all'interno di una proposizione per facilitarne la comprensione.
- **Relazioni**, questa è la parte più importante per un synset e la vera innovazione che ha portato BabelNet nel mondo dei dizionari enciclopedici. Nell'interfaccia grafica sono presenti le principali relazioni del synset con altri synsets (iponimi, iperonimi, e così via). Ad esempio, come è possibile vedere nella Figura 2.2 relativa al synset *Steve Jobs*, sono mostrate le relazioni IS-A, AWARD RECEIVED, CAUSE OF DEATH, CHILD, COUNTRY OF CITIZENSHIP, EDUCATED AT. Inoltre è possibile "*esplorare la rete*", cliccando sul relativo pulsante "Explore network", ed ottenere una nuova interfaccia grafica che mostra una struttura a grafo nel quale sono riportati tutti i synsets correlati al synset in questione, navigare tra di essi ed ottenere nuove informazioni (vedi Figura 2.3).

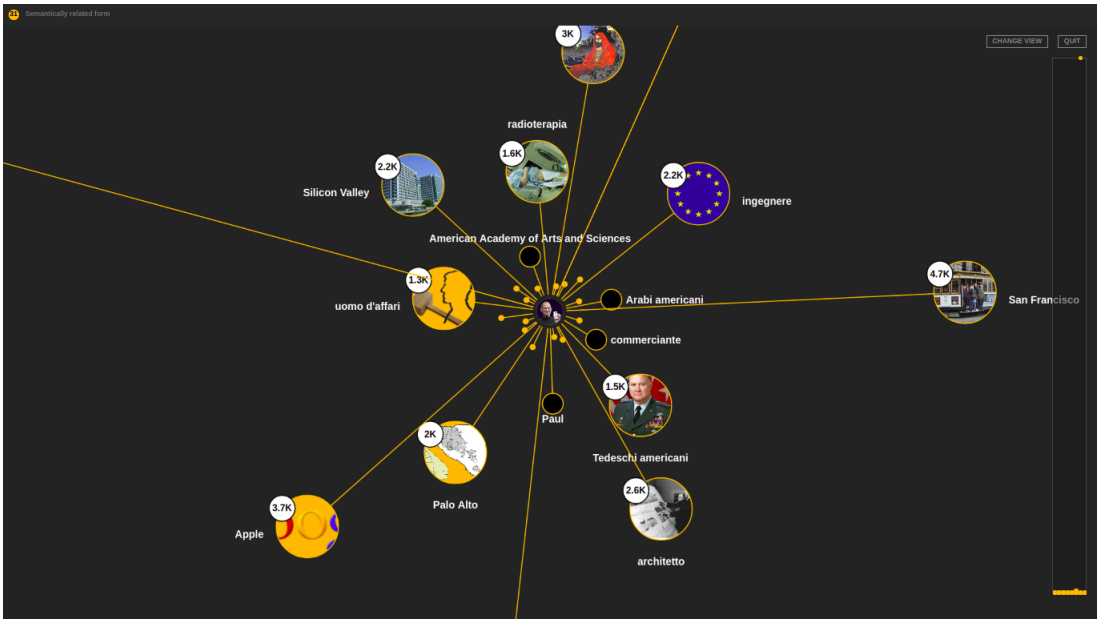


Figura 2.3: La rete relativa al BabelSynset *Steve Jobs*

- **Immagini**, le principali immagini del synset estratte da Wikipedia e ImageNet.
- **Traduzioni**, è possibile ottenere traduzioni dello stesso concetto in tutte le lingue coperte da BabelNet.

Mentre, nella seconda parte, l'interfaccia mostra informazioni riguardanti la provenienza del synset:

- **Fonte**, una lista delle fonti utilizzate per la creazione del synset (WordNet, Wikipidea, ed altre).
- **Categorie**, in generale un synset può essere classificato in diverse categorie (principalmente quelle con le quali veniva classificato in Wikipedia).
- **Altre forme**, forme diverse con le quali è possibile esprimere lo stesso concetto del lemma.
- **Link esterni**, link ad altre risorse online riguardanti il lemma.

BabelNet è stato citato sul Time magazine come il rappresentante di una nuova era di risorse lessicografiche computazionali del ventunesimo secolo [Ste16][RN12][RN10][AM14].

2.2 WordNet

In particolare, per questo elaborato, sono stati utilizzati i synsets di WordNet contenuti in BabelNet. Sono stati scelti questi in quanto presentano molte più informazioni semantiche rispetto agli altri synsets di BabelNet. Tuttavia, questa scelta ha condizionato l'elaborano nell'utilizzo di parole della lingua inglese.
WordNet è una base di dati sematico-lessicale per la lingua inglese. Esso è stato ideato da George Armitage Miller presso l'Università di Princeton. È la risorsa lessicale più famosa ed utilizzata nel mondo NLP (vedi Sezione 1.4). L'organizzazione lessicale di WordNet si avvale di raggruppamenti di termini mediante il loro senso - i.e., i synset, ovvero gli insiemi dei sinonimi (synonym set) - e del collegamento dei loro significati mediante diversi tipi di relazioni. Per esempio il concetto di **car** è espresso come segue:

$$\{car^1_n, auto^1_n, automobile^1_n, machine^6_n, motorcar^1_n\}$$

Gli elementi a destra dei lemmi (^1_n) rappresentano rispettivamente il primo senso del lemma *car* $(^1)$ e il fatto che *car* sia un nome $(_n)$. Questa figurazione è utilizzata per distinguere i principali sensi di un concetto - i.e., car^1_n è il primo senso di *car*, car^2_n è il secondo senso di *car*, e così via - e per classificare i nomi $(_n)$, i verbi $(_v)$, gli aggettivi $(_a)$ e via dicendo. I vari sensi, invece, sono espressi come segue:

- $\{car^1_n, auto^1_n, automobile^1_n, machine^6_n, motorcar^1_n\}$
- $\{car^2_n, railcar^1_n, railwaycar^1_n, railroadcar^1_n\}$
- $\{car^3_n, cablecar^1_n\}$
- $\{car^4_n, gondola^3_n\}$
- $\{car^5_n, elevatocar^1_n\}$

Oltre al raggruppamento degli insiemi dei sinonimi, ogni synsets, contiene anche una definizione testuale nella quale viene descritto cosa il concetto rappresenta {Is-a, Has-part, ..., $\in$}, dove $\in$ rappresenta l'insieme di tutte le descrizioni. Nel caso del lemma *car*, è possibile vedere la struttura nella Figura 2.4. Infine, un synset può contenere anche un esempio di utilizzo come in un vero e proprio dizionario.

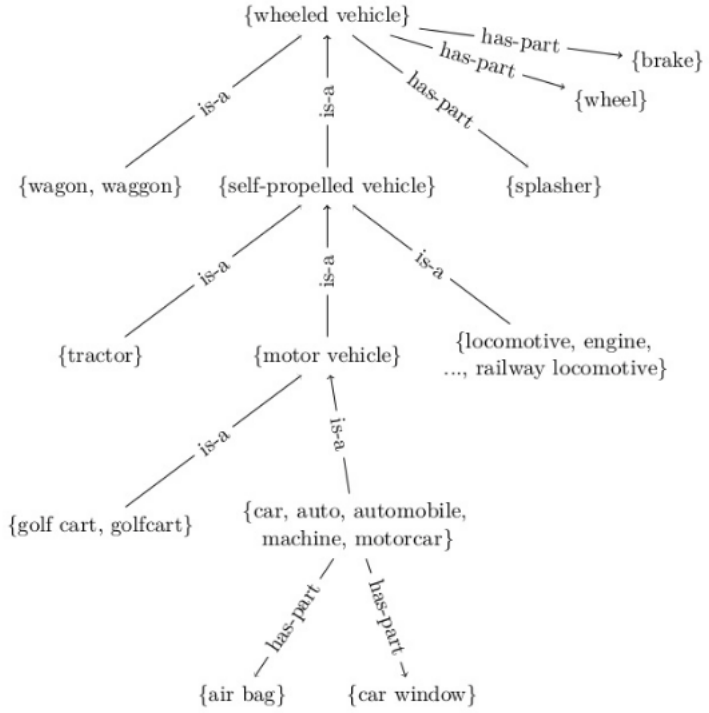


Figura 2.4: La struttura semantica relativa al lemma *car*

La parte principale di WordNet è la sua rete semantica, all'interno della quale vi sono connessioni tra più synsets attraverso *relazioni semantiche*. L'insieme delle relazioni che costituiscono la rete semantica possono essere di 12 tipi distinti: per esempio, relazioni di iperonimia (*la macchina* ***is-a*** *veicolo a motore*), relazioni di olonimia (*il finestrino* ***is a part of*** *automobile*), o ancora relazioni di iponimia (*l'ambulanza* ***is a kind of*** *macchina*) e così via. La Figura 2.4 mostra alcune di queste relazioni. Qui di seguito sono descritte le relazioni più importanti di WordNet:

- **Iperonimia** (vedi Sezione 1.3.2.4): S^1 è iperonimo di S^2 se ogni S^2 è un tipo (**kind of**) di S^1. Vale sono per i synsets nominali e verbali.
- **Iponimia** e **Troponimia** (vedi Sezione 1.3.2.4): relazione inversa a quella di iperonimia per i synsets verbali e nominali.
- **Meronimia**: S^1 è un meronimo di S^2 se S^1 è una parte di S^2. Vale solamente per i synsets nominali.

- **Olonimia** (vedi Sezione 1.3.2.4): è la relazione inversa a quella di meronimia.
- **Implicazioni**: un verbo V^1 è un'implicazione di un verbo V^2 se per fare V^2 si deve sottointendere V^1. Ad esempio: $russare_v^1$ implica $dormire_v^1$.
- **Antonimia**: S^1 è un antonimo di S^2 se esprime il concetto opposto di S^2. Ad esempio: $buono_a^1$ è un antonimo di $cattivo_a^1$.
- **Similarità**: si applica a tutte le relazioni di similarità. Un aggettivo A^1 è simile a un aggettivo A^2. Ad esempio: $bello_a^1$ è simile a $grazioso_a^1$.
- **Attributo**: un sostantivo S^1 è un attributo per il quale un aggettivo S^2 ne esprime il valore. Ad esempio: $freddo_a^1$ è un valore di $temperatura_n^1$.

Sfortunatamente, WordNet codifica distinzioni di significato troppo raffinate, portando a prestazioni di disambiguazione deludenti. Ultimamente, sono state create distinzioni di significato meno raffinate che hanno portato a prestazioni di disambiguazione per la lingua inglese tra l'80% e il 90% [RN07]. Il maggior limite di WordNet stà nel fatto che non presenta concetti quali aziende, città, gruppi musicali e così via, bensì tutti quei concetti reperibili su un dizionario. Inoltre è limitante anche il fatto che tutte le informazioni riportate sono in lingua inglese. [Van16] [wik18f] [sit18] [TW14].

2.3 Word embedding

Il *word embedding* (in italiano l'incorporamento di parole) è un insieme di tecniche utilizzate nell'elaborazione del linguaggio naturale dove, il linguaggio umano, le parole o un discorso vengono modellati. Questi modelli vengono creati secondo tecniche di mappatura e il risultato sono dei vettori di numeri reali. Prendendo una frase di N parole, il risultato sarà uno spazio vettoriale di N vettori i quali dovranno rispettare alcune condizioni empiriche:

- Parole semanticamente più vicine - e.g., con una relazione di sinonimia forte - avranno dei vettori composti da numeri reali più vicini tra loro.
- I vettori devono essere costruiti in modo tale che il modello possa capire analogie del tipo: $(\text{"}King\text{"} - \text{"}Man\text{"}) + \text{"}Queen\text{"} = \text{"}Woman\text{"}$.

Sebbene il word embedding sia una teoria remota, allo stato dell'arte non esisto molti metodi per generare queste mappature. Tra i più noti si ricordano le *reti neurali*, la *riduzione della dimensionalità* sulla matrice di co-occorrenza di parole, *modelli probabilistici*, metodo della *base di conoscenza spiegabile*. Infatti, le tecniche per trasformare parole in vettori sono state formulate già negli anni Sessanta con lo sviluppo del modello spaziale vettoriale per il recupero delle informazioni [wik18e]. Questo modello è stato un punto di partenza per una serie di metodologie ben diverse tra loro, generando un continuo e graduale sviluppo. Tuttavia, solo nel recente 2010, questa tecnologia è decollata: lo sviluppo dell'hardware e, in particolare, il miglioramento delle reti neurali, ha reso possibile che si affinassero le qualità dei vettori e che si velocizzasse l'allenamento del modello. Queste tecniche hanno portato anche notevoli miglioramenti nel campo della bioinformatica (i.e., analisi del DNA, analisi del RNA, e molti altri). Il 2013 è stato l'anno di svolta in quanto un team di Google ha creato *word2vec*.

2.3.1 Word2Vec

La *word2vec* è una tecnica di incorporamento di parole in grado di formare modelli spaziali vettoriali più veloci rispetto agli approcci precedenti. Questa tecnologia è stata ideata e realizzata da un team di Google nel 2013, guidato da Tomas Mikolov, uno statunitense che ha conseguito il dottorato di ricerca in Informatica nel 2012 e che attualmente lavora a Facebook Inc. Il paradigma word2vec consiste nel creare modelli mediante l'utilizzo di reti neurali efficienti del tipo *Skip-gram*. L'obbiettivo di ogni modello è di produrre vettori a dimensione fissa per le parole, che rispettino, inoltre, le condizioni empiriche previste dal word embedding.
La differenza sostanziale tra il word2vec e le altre tecniche del word embedding sta nel fatto che la rappresentazione di parole - i.e., la mappatura di parole in vettori - viene effettuata usando i vocaboli circostanti alla parola in questione. Il team di Mikolov, nel teorizzare un possibile vettore per un determinato fonema, ha ragionato su come il cervello umano si comporti di fronte ad una frase nella quale non viene precisato il concetto chiave che da senso alla stessa. Nella frase:

Mi piace giocare a X

non si conosce quale possa essere il significato di X, tuttavia inconsciamente l'essere umano capisce che si riferisce ad un'attività ludica e, pertanto, intuisce che questa frase riguarda un qualcosa di piacevole, in quanto deve essere una cosa con cui ci si può "giocare". Per poter arrivare a questa conclusione, il cervello, ha analizzato le parole circostanti a X: ha filtrato tra le parole "comuni" (in questo caso "Mi" e "a"), si è concentrato sul verbo "piacere", immaginandosi una sensazione bella, gratificante, piacevole appunto e, infine, ha collegato "giocare" ad X. Le procedure ivi descritte sono proprio i procedimenti che effettua word2vec nel mappare vettori a parole.
Entrando in un contesto più tecnico ed ingegneristico, i vettori di parole, sono costituiti da valori in virgola mobile ottenuti durante la fase di addestramento dei modelli: lo Skip-gram. Qui di seguito verranno descritti alcuni passaggi con il quale è possibile passare da un vocabolo ad un vettore e verrà presa d'esempio la seguente frase (i lettori che giocano a Clash Royale capiranno):

Tanto una partita si vince e una si perde

La frase d'esempio è composta da N parole, pertanto, verranno creati N vettori di dimensione N (ovvero una matrice quadrata $N \times N$). Questi vettori dovranno semplicemente differenziarsi tra di loro al fine di poter costituire N elementi di una rete neurale. Una tecnica semplice e veloce per distinguere un vettore da un altro è quella del One-hot-encoding, cioè porre tutti i bit del vettore a 0 tranne uno e scalare via via lungo la diagonale:

$$
\begin{aligned}
[1,0,0,0,0,0,0,0,0] &= \text{Tanto} \\
[0,1,0,0,0,0,0,0,0] &= \text{una} \\
[0,0,1,0,0,0,0,0,0] &= \text{partita} \\
[0,0,0,1,0,0,0,0,0] &= \text{si} \\
[0,0,0,0,1,0,0,0,0] &= \text{vince} \\
[0,0,0,0,0,1,0,0,0] &= \text{e} \\
[0,0,0,0,0,0,1,0,0] &= \text{una} \\
[0,0,0,0,0,0,0,1,0] &= \text{si} \\
[0,0,0,0,0,0,0,0,1] &= \text{perde}
\end{aligned}
$$

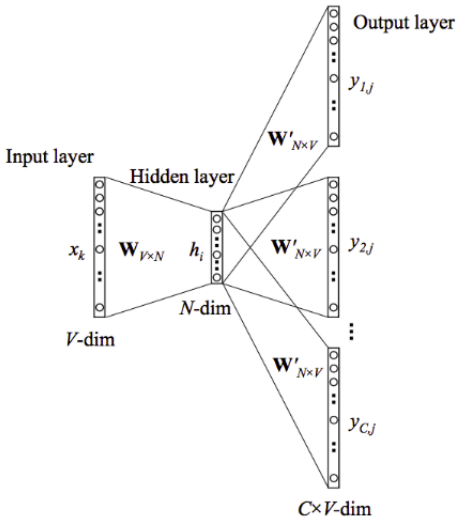


Figura 2.5: Rete neurale del tipo *Skip-Gram*

La Figura 2.5 mostra una rete neurale del tipo *Skip-Gram*. L'obbiettivo di questa rete è quello di predire, dato un target, le parole più vicine. Pertanto, l'input sarà l'insieme dei vettori codificati con la tecnica One-hot-encoding, mentre l'output saranno N parole dove N è la dimensione della finestra (windows size) del contesto; grandezza determinata a priori nel settaggio della rete. Nella frase d'esempio, supponendo di voler conoscere il significato della parola target "partita" e che le parole vicine siano "Tanto", "una" e "si", la dimensione del vettore di output sarà 3 - i.e., si avranno 3 strati per la rete neurale - mentre, i pesi saranno inizializzati con valori casuali. Dopo aver fatto attraversare il target nella rete, si otterrà la prima rappresentazione vettoriale del vocabolo "partita", ottenuto dal prodotto dei valori di layer con il target in input. Il vettore sarà simile a $[0.8, 0.4, 0.5]$. Quindi, ad esempio:

$$(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) \times \begin{pmatrix} 0.1 & 0.9 & 0.3 \\ 0.2 & 0.4 & 0.3 \\ & \vdots & \\ 0.6 & 0.5 & 0.4 \end{pmatrix} = (0.8, 0.4, 0.5) \tag{2.1}$$

(a) (b) (c)

Nell'equazione (2.1), (b) rappresenta la matrice dei pesi, ovvero una rappresentazione

matematica dei vocaboli nell'intorno della parola target. Replicando il processo ivi descritto e dopo aver effettuato backpropagation (un algoritmo per addestrare le reti neurali che prevede l'attraversamento inverso degli elementi al fine di correggere il maggior numero possibile di errori), il risultato sarà la trasformazione dei vocaboli in input sotto forma di vettori [Mik] [Ahm17] [Ron16] [ten].
Il word2vec si applica maggiormente con le frasi nelle quali vi sono molteplici relazioni di sinonimia e, pertanto, centrava a pieno gli scopi di questo elaborato.

2.4 Ambiente di sviluppo

2.4.1 Sistema operativo

Il sistema operativo su cui il tirocinante ha svolto il proprio lavoro è stato Ubuntu 18.04. *Ubuntu* è un sistema operativo nato nel 2004, focalizzato sulla facilità di utilizzo. È prevalentemente composto da software libero proveniente dal ramo unstable di Debian GNU/Linux, ma contiene anche software proprietario, ed è distribuito liberamente con licenza GNU GPL. È orientato all'utilizzo sui computer desktop, ma presenta delle varianti per server, tablet, smartphone e dispositivi IoT, ponendo grande attenzione al supporto hardware. In particolare, Ubuntu 18.04 LTS, rilasciato il 26 aprile 2018. Fra le principali novità si ricordano la presenza del kernel Linux 4.15, di GNOME 3.28, del server grafico X.Org, il passaggio definitivo dalla versione 2 di Python alla 3. Sono infine presenti le patch per Meltdown e Spectre [wik18c] [wik18d].

2.4.2 Eclipse

Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multipiattaforma, utilizzato soprattutto per la creazione di applicazioni Java. La sua scelta, infatti, è legata al fatto che per il lavoro di questa tesi si è dovuto usare il linguaggio Java8. Inoltre, Eclipse, è un ottimo strumento che permette di avere un'ampia gamma di plugin utili per facilitare la gestione del codice [wik18a].

2.4.3 Java8

In informatica *Java* è un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, specificamente progettato per essere il più possibile indipendente dalla piattaforma di esecuzione [wik18b]. In particolare è stata usata la versione Java8, rilasciata nel 2014. Questo perchè le novità che furono introdotte in quella versione, come le espressioni lamba [Ces14], permettono di usare al meglio le API di BabelNet; tecnologia fondamentale ai fini di questo elaborato.

2.5 Librerie utilizzate

2.5.1 JavaBabelNetAPI

Le API di BabelNet permettono un collegamento, tramite il linguaggio Java, agli indici del dizionario di BabelNet (vedi Sezione 2.1). Esse offrono la possibilità di poter ottenere, secondo l'evenienza, le informazioni contenute in BabelNet. Le classi principali sono: BabelNet, BabelSynset. La classe **BabelNet** viene utilizzata come punto di accesso agli indici di BabelNet. Essa è stata implementata secondo il pattern singleton, ovvero quel paradigma che ha lo scopo di garantire che di una determinata classe ne venga creata una e una sola istanza, fornendone un unico punto di accesso globale [CL16]. La classe **BabelSynset** è la rappresentazione sotto forma di oggetti Java dei BabelSynsets di BabelNet. Questa classe contiene tutte le informazioni che può contenere un elemento di BabelNet ad esempio: sensi, glosse, lemmi e così via. È stata la classe più utilizzata all'interno del progetto in quanto ha permesso la creazione del disambiguatore, sfruttando il metodo che consente di ottenere i BabelSynsets adiacenti ad un determinato BabelSynsets: *getOutgoingEdges()* [RN15].

2.5.2 JGraphT

JGraphT è una libreria Java gratuita che fornisce oggetti e algoritmi per l'implementazioni di grafi. La potenzialità di questa libreria sta nel fatto che permette di gestire grafi come se fossero le *collection* di Java. Supporta vari tipi di grafi, come è possibile vedere nella Figura 2.6. Il tipo di grafo utilizzato in questo elaborato è stato *simpleDirectedGraph<V,E>*, in quanto permette di ottenere grafi diretti non pesati, garantendo l'esclusione di cicli.

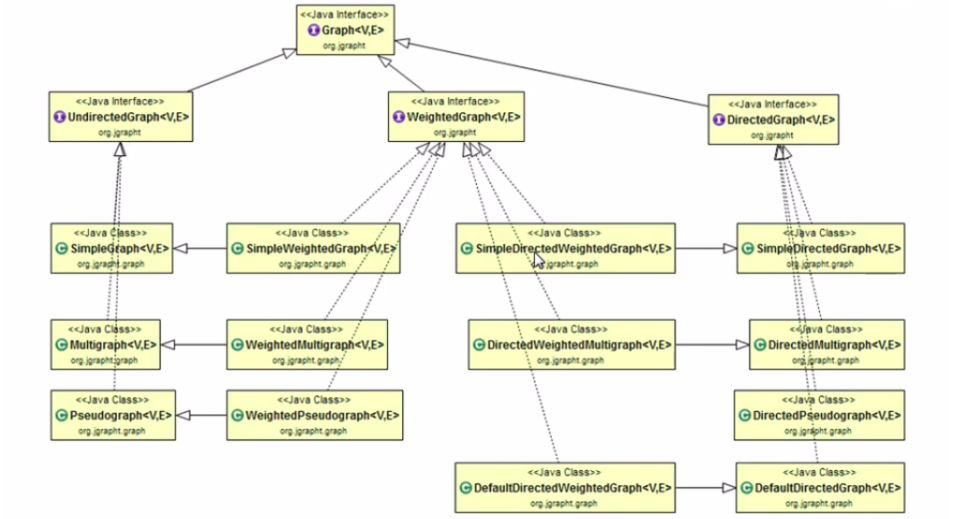


Figura 2.6: L'interfaccia Graph di Jgrapht

2.5.3 Medallia - Word2VecJava

Questa libreria, sviluppata dall'azienda statunitense Medallia Inc., è la traduzione in Java dell'originale libreria Word2Vec di Google scritta in C. Essa offre le stesse prestazioni (con i limiti tecnici che comportano un codice scritto in Java ed uno scritto in C) le stesse potenzialità e gli stessi algoritmi dell'originale. Anche questa replica utilizza una rete neurale del tipo Skip-Gram per trasformare i modelli da parole a vettori. Inoltre offre il metodo che permette di calcolare la distanza cosinusoidale tra due parole, metodo chiave per questo elaborato al fine di filtrare i vari grafi. La funzione in questione è *cosineDistance(s1, s2)*, dove s1 e s2 sono due stringhe. Il risultato è un numero reale rappresentante la probabilità di distanza tra le stringhe s1 e s2 [Med18].

2.5.4 StanfordCoreNLP

Questa è una famosa libreria sviluppata dal gruppo di elaborazione del linguaggio naturale dell'Università di Stanford. La libreria offre quasi tutti i servizi utili nel campo del NLP. In questo progetto è stato sfruttato una classe che permette di *lemmatizare* una stringa, ovvero trasformare una parola alla sua forma di citazione (vedi Sezione 1.3.1) [Uni18b].

Capitolo 3

Analisi dei requisiti

3.1 Scopo del tirocinio

L'obbiettivo del tirocinio è stato quello di creare una struttura (un dizionario) che permettesse di redigere una classifica di sensi. Il progetto doveva prendere in ingresso una frase di senso compiuto (o un'insieme di parole casuali) e confrontarla con un dizionario artificiale. La struttura doveva avere un'organizzazione tale da consentirle di ospitare al suo interno il maggior numero di sensi possibili correlati tra loro, in modo tale da permetterle di effettuare delle operazioni di matching tra la stringa e se stessa. Pertanto, immaginando una struttura a grafo con G nodi e una stringa di N parole (con $N < G$), si sarebbe dovuto contare quanti elementi della stringa erano contenuti nel grafo: se l'elemento fosse stato presente si doveva assegnare un punto, in caso contrario 0. Prevista per la fine del conteggio è stata la somma dei punti ottenuti con il conseguente assegno del punteggio all'elemento della struttura in esame. Dopo aver effettuato questa operazione di scoring, l'ultimo passaggio è stato quello di ordinare la struttura in base ai punteggi.
Si è pensato su quali nodi di BabelNet potessero andare bene per una struttura dalle caratteristiche ivi citate e la scelta è caduta sui nodi di WordNet. Il fatto che ogni elemento in BabelNet abbia un identificatore ben preciso, ha consentito di creare una mappa. Pertanto la struttura immaginata è stata una mappa di grafi (*Map<String, Graph<String, DefaultEdge»*), dove le *keys* sono tutti i sensi di WordNet contenuti in

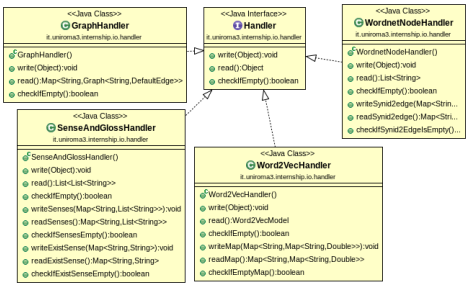
BabelNet, mentre i *values* sono i grafi creati partendo dalla rispettiva key ed effettuando una visita in profondità fino ad una distanza pari a due. In questo modo è stato possibile abbattere notevolmente il costo di ricerca nelle operazioni di matching. Tuttavia il problema principale presentato durante lo sviluppo del tirocinio è stato quello dell'algoritmo per la costruzione di tale mappa, sia in termini di tempi di esecuzione e sia in termini di occupazione di memoria. È stato quindi necessario filtrare tutte le operazioni che potenzialmente avrebbero rallentato l'algoritmo e ci si è resi conto che fossero tutte azioni statiche il cui risultato non cambiasse da esecuzione ad esecuzione, ad esempio: *estrarre i nodi di WordNet contenuti in BabelNet*, *per ogni nodo chiedere a BabelNet i nodi vicini* e così via. Pertanto si è capito che non potessero essere operazioni effettuabili durante la costruzione della struttura, bensì dovevano obbligatoriamente essere realizzate ex ante.

3.2 Interfacce Handler e Builder

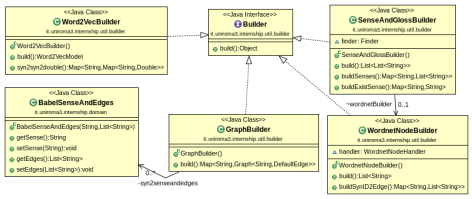
È stato quindi deciso di adottare un approccio bottom-up: suddividere il problema generale in sottoproblemi, salvare i risultati in memoria secondaria e iterare il procedimento fino ad un risultato soddisfacente.
Le interfacce Handler e Builder hanno consentito questa soluzione:

- **Handler**: questa interfaccia viene implementata da tutte quelle classi (vedi Figura 3.1a) la cui responsabilità è quella di effetturare operazioni riguardanti la memoria secondaria: in lettura (con il metodo *read()*) e in scrittura (con il metodo *write()*). Quelle classi, inoltre, hanno un metodo grazie al quale è possibile verificare se l'elemento da leggere è vuoto o no (con la funzione *checkIfEmpty()*), in questo modo è stato possibile automatizzare il processo di costruzione effettuato dai vari builders.

- **Builder**: questa interfaccia viene implementata da tutte quelle classi (vedi Figura 3.1b) la cui responsabilità è quella di costruire strutture (con il metodo *build()*), chiedendo preventivamente alla corrispondente classe Handler se l'oggetto in questione sia già presente in memoria.



(a) Interfaccia Handler



(b) Interfaccia Builder

3.2.1 WordNetNodeBuilder

Questa classe effettua tutte le operazioni legate all'estrazione dei nodi di WordNet contenuti in BabelNet. Al suo interno vi sono due metodi: *build()* e *buildSynID2Edge()*. Il primo metodo consente di creare una *List<String>* (vedi Figura 3.2) contenente tutti i nodi di WordNet. È stato necessario creare questo metodo in quanto le API di BabelNet, pur presentanto un Iterator<>, restituivano i nodi in maniera casuale e non serializzabili; quindi non scrivibili in memoria secondaria (i BabelSynsets). Considerando che si conoscevano a priori gli identificatori di quei nodi è stato possibile costruire la lista, convertendo i singoli BabelSynset in stringhe e mantenendo invariato l'hashCode() originale. In questo modo è stato possibile mantenere l'univocità dei nodi presenti in BabelNet e salvare la struttura in un file.

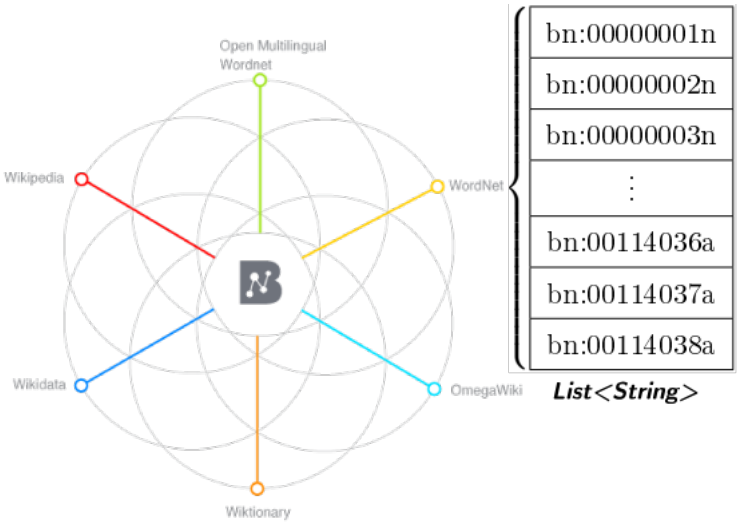


Figura 3.2: Il risultato del metodo *build()*

Il secondo metodo, invece, consente di creare una mappa di liste *Map<String, List<String»* (vedi Figura 3.3). Nelle *keys* vi sono tutti i nodi estratti mediante *build()*, mentre nei *values* vi sono le liste contenenti tutti i nodi collegati all'i-esima keys in BabelNet. Questo metodo è stato fondamentale per la costruzione della mappa di grafi descritta ad inizio capitolo; in quanto l'algoritmo presentava un collo di bottiglia proprio nella visita del grafo associato ad ogni singolo nodo. Per ottimizzare l'occupazione in RAM è stato necessario andare a filtrare alcuni nodi nel grafo di BabelNet, legati da delle relazioni semantiche ritenute fuorvianti: *REGION*, *REGION_ MEMBER*, *TOPIC*, *TOPIC_ MEMBER*, *USAGE* e *USAGE_ MEMBER*; tutte significanti collegamenti relativi a toponimi o usanze di un posto [Uni18a]. Sebbene questa selezione abbia notevolmente ridotto la dimensione della struttura finale, si aveva la necessità di avere una maggiore ottimizzazione. Per queste ragioni si è pensato di andare a sfruttare il calcolo della distanza cosinusoeidale, in modo tale da poter filtrare, nei vari grafi finali, le relazioni semanticamente più distanti tra loro. È stato necessario quindi addestrare un modello word2vec.

bn:00000001n	bn:00001383n	bn:00001354n	···	bn:00082262v	bn:00113891a
bn:00000002n	bn:00082213n	bn:00072114n	···	bn:00082115n	bn:00082116v
bn:00000003n	bn:00001563n	bn:00006352n	···	bn:00122262a	bn:00113894a
⋮					
bn:00114036a	bn:00007273n	bn:00001451n	···	bn:00000062n	bn:00113291a
bn:00114037a	bn:00004323n	bn:00021354n	···	bn:00085562v	bn:00111995a
bn:00114038a	bn:00011383n	bn:00000355n	···	bn:00122262a	bn:00113892a

Map<String, List<String>>

Figura 3.3: Il risultato del metodo *buildSynID2Edge()*

3.2.2 SenseAndGlossBuilder

Per poter addestrare una rete neurale del tipo Skip-Gram, come descritto nella Sezione 2.3.1, è necessario avere a disposizione una grande quantità di frasi di senso compiuto. Inizialmente si è pensato di utilizzare un famoso file proposto da Google: *text8*. Il file presenta circa 253000 frasi per un totale di 100MB. Tuttavia i sensi associati ai nodi di BabelNet non venivano riscontrati all'interno della rete, come se non vi fossero presenti. Per queste ragioni è stato necessario costruire un modello adatto agli scopi del tirocinio. Si è pensato di utilizzare le informazioni intrinseche ai nodi di BabelNet, ovvero i vari sensi legati ad un nodo e le opzionali glosse. La classe SenseAndGlossBuilder effettua proprio le operazioni di parsing sopra citate, costruendo una struttura adatta alla rete neurale. Al suo interno sono presenti due metodi: la funzione *build()* (intrinseca all'interfaccia handler) e la funzione proprietaria *buildExistSense()*. Il metodo principale all'addestramento della rete per il word2vec è build(). In questo metodo vengono analizzati i nodi estratti dal metodo *build()* presente in WordNetNodeBuilder e salvati in una List<String>. Per ogni elemento della lista, si estraggono prima i sensi e poi le glosse. Successivamente ogni informazione estratta viene incrociata e filtrata con le *EnglishStopWords*, ovvero le parole comuni della lingua inglese; questo per due ragioni: la prima per ridurre il peso in memoria della neo struttura, la seconda è che i vocaboli comuni, all'interno dei sensi o delle glosse, sono fuorvianti ai fini dell'addestramento della rete. Una volta finiti gli elementi da analizzare, il tutto è inserito all'interno di una *List<List<String»* (vedi Figura 3.4), struttura richiesta in input dalla libreria utilizzata per il word2vec.

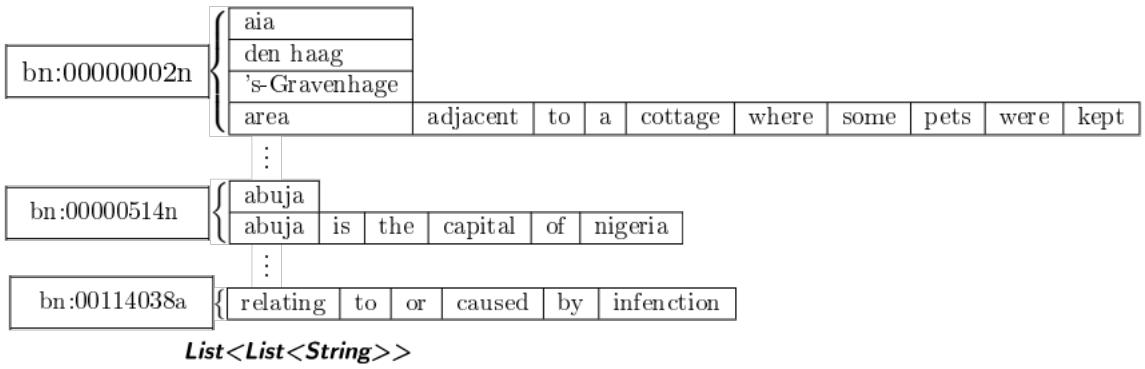


Figura 3.4: Il risultato del metodo *build()*

Tuttavia, dopo le operazioni sopra citate, è sorto un nuovo problema: sebbene i nodi di BabelNet presentano più sensi, nel modello word2vec finale vi era presente solamente uno di quei significati. Ad esempio per i BabelSynset *bn:00000002n*, corrispondente al concetto del numero tredici, i sensi ad esso associato sono in ordine {13(numero), tredici} e, nel modello word2vec, vi è solamente il senso *tredici*. Per queste ragioni, onde evitare di aumentare le tempistiche d'esecuzione del metodo per la costruzione della mappa di grafi, per via di una possibile ricerca del senso presente nella rete neurale, è stato necessario costruire una nuova struttura che associasse ad ogni nodo, il corrispondente significato presente nel modello word2vec. Questa struttura viene costruita dalla funzione *buildExistSense()* e la struttura risultante è una mappa di sensi (*Map<String, String>*) dove le *keys* sono i nodi di WordNet presenti in BabelNet, metre i *values* i sensi presenti nel modello word2vec. In questo modo è stato possibile velocizzare notevolmente il tempo d'esecuzione dell'algoritmo principale e, soprattuto, limitarne l'occupazione in memoria.

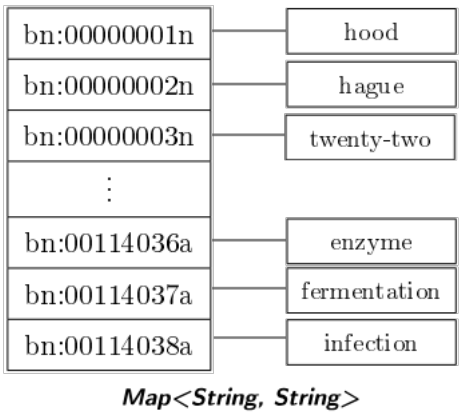


Figura 3.5: Il risultato del metodo *buildExistSense()*

Capitolo 4

Progettazione ed implementazione del disambiguatore

Nel capitolo precedente è stata mostrata la soluzione grazie alla quale è stato possibile ottimizzare sia lo spazio occupato in memoria e sia le tempistiche d'esecuzione dell'algoritmo per la costruzione della mappa di grafi, ovvero il **disambiguatore**. Sebbene l'approccio bottom-up abbia risolto notevolmente i problemi sopra citati, l'occupazione di memoria continuava ad essere eccessiva. Andando a analizzare le strutture create, ci si è resi conto che, soprattutto per le mappe, le keys erano sempre i nodi di WordNet contenuti in BabelNet e che, quindi, cambiassero solamente i values da struttura a struttura. Scendendo nel dettaglio, mantenendo questa replica di informazione, per l'algoritmo per la costruzione della mappa di grafi servivano tre strutture:

1. La lista di stringhe (List<String> chiamata nel progetto wordnetList) contenente i nodi di WordNet presenti in BabelNet. Questa lista veniva usata solamente per lo scorrimento sui nodi.

2. La mappa contenente i nodi adiacenti al nodo i-esimo di wordnetList. Questa mappa veniva utilizzata per conoscere le relazioni del nodo all'interno di BabelNet.

3. La mappa avente il senso del i-esimo nodo di wordnetList presente nel modello word2vec. Questa mappa venita usata per poter ottimizzare la ricerca del senso dell'i-esimo nodo in modo tale da poterlo confrontare tramite la distanza cosinusoeidale.

Questa ripetizione di informazione è risultata essere inutile, poichè aumentava l'occupazione di memoria e rallentava l'esecuzione dell'algoritmo in quanto, sebbene si trattasse di mappe, la cui lettura ha una costo costante, era necessario accedere a due oggetti diversi mediante lo stesso codice hash per estrarre due tipi di informazioni diverse. Per ovviare a questa inutile duplicazione di informazione, si è deciso di creare una nuova classe *BabelSenseAndEdges*. Lo scopo di questa classe era solamente quello di accorpare le strutture in modo da poter creare una singola mappa avente tutte le informazioni necessarie (vedi Figura 4.1). Per questi motivi, la BabelSenseAndEdges, presenta due campi: il primo racchiude il senso della key presente nel modello word2vec e, il secondo, i nodi di WordNet adiacenti all'interno di BabelNet. Tramite questa ritrutturazione è stato possibile ottimizzare ulteriormente l'algoritmo principale e abbattere sia l'occupazione di memoria e sia il tempo d'esecuzione.

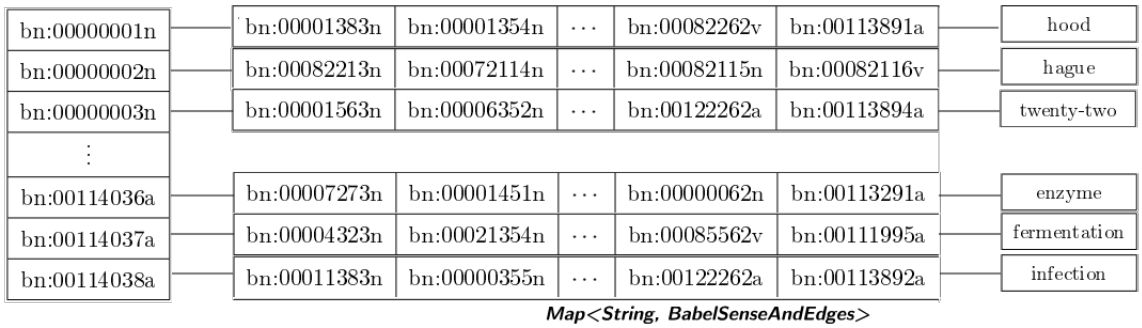


Figura 4.1: Il risultato del refactoring con BabelSenseAndEdges

4.1 GraphBuilder

L'ultima classe che verrà analizzata e che implementa l'interfaccia Builder (citata nella Sezione 3.2) è GraphBuilder. Uno degli scopi di questa classe è quello di riunire le strutture create dagli altri Builders, raggruppandole in un unica struttura comune (vedi Figura 4.1), ovvero l'operazione descritta nel punto precedente. La responsabilità di questo atto, è stata assegnata a questa classe seguendo le direttive del pattern GRASP *Creator*, in quanto soddisfava due specifiche: 1) GraphBuilder utilizza strettamente la mappa della Figura 4.1, 2) la classe ha le informazioni necessarie alla creazione della mappa [CL16]. Tuttavia, lo scopo principale della classe è quello di costruire il

disambiguatore. Nella Figura 4.2 è possibile vedere un modello della struttura risultante. Qui di seguito, inoltre, verrà analizzato in maniera dettagliata l'algoritmo di costruzione.

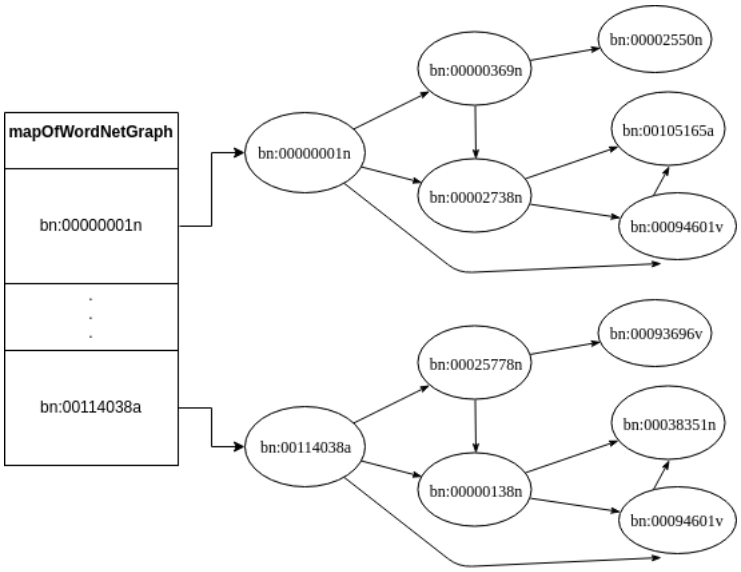


Figura 4.2: La struttura del disambiguatore

4.1.1 Algoritmo di costruzione

Nella Figura 4.3 vi è riportata la parte iniziale della classe GraphBuilder, ovvero le variabili di istanza. Si noti che sono stati appositamente mantenuti i commenti alle strutture da unire, questo per mostrare come fosse la classe precedentemente. All'interno della classe vi sono quattro variabili di istanza:

- **model**, questa variabile rappresenta il modello Word2Vec della libreria descritta nella Sezione 2.5.3. Come è possibile vedere nella parte relativa al costruttore, sempre nella Figura 4.3, essa viene riempita dalla classe Word2VecBuilder, appunto il builder per il modello.

- **searcher**, questa variabile rappresenta un oggetto contenuto nella classe Word2VecModel. Lo scopo di questo oggetto, come si può intuire dal nome, è quello di cercare all'interno del modello stesso, parole, vettori e così via. È possibile ottenere un oggetto di tipo *Searcher* mediante il metodo *forSearch()*, presente nella libreria word2vec utilizzata. È stato necessario creare una variabile di istanza di questo

tipo in quanto, ogni qualvonta che veniva invocato il metodo *forSearch()*, veniva generato un set di oggetti, riempito successivamente mediante lo scorrimento sugli elementi contenuti nel word2vec. Questa operazione rallentava notevolmente l'algoritmo in quanto, precedentemente, l'oggetto Searcher veniva invocato ad ogni ricorsione.

- **sy2senseandedges**, questa variabile rappresenta la mappa di BabelSenseAndEdges ivi descritta, creata accorpando tutte le strutture che presentano i commenti (nella Figura 4.3 quelle alle righe 32, 33 e 34).

- **mapOfWordnetSynsets**, questa variabile rappresenta il cuore di questa classe, avendo come scopo proprio la costruzione di questo oggetto.

```
1  package it.uniroma3.internship.util.builder;
2
3  import java.util.HashMap;
19
20 /**
21  * This class it's used to build the map of graphs relative at the wordnet nodes in BabelNet
22  *
23  * @author Silvio Severino
24  *
25  */
26 public class GraphBuilder implements Builder
27 {
28     private Word2VecModel model;
29     private Searcher searcher;
30     private Map<String, Graph<String, DefaultEdge>> mapOfWordnetSynsets;
31     private Map<String, BabelSenseAndEdges> syn2senseandedges;
32 //  private WordnetNodeBuilder wordnetBuilder;
33 //  private Map<String, String> synID2ExistSense;
34 //  private Map<String, List<String>> synID2Edges;
35
36     /**
37      * Constructor
38      */
39     public GraphBuilder()
40     {
41         Word2VecBuilder builder = new Word2VecBuilder();
42         this.model = builder.build();
43         this.searcher = this.model.forSearch();
44         BabelSenseAndEdgesBuilder senseAndEdgesBuilder = new BabelSenseAndEdgesBuilder();
45         this.syn2senseandedges = senseAndEdgesBuilder.read();
46         this.mapOfWordnetSynsets = new HashMap<>();
47     }
48
```

Figura 4.3: Costruttore e variabili di istanza

Nella Figura 4.4 viene rappresentato il metodo *build()*, implementazione della funzione presente nell'interfaccia Builder. Questo metodo ha lo scopo di scorrere i nodi di WordNet presenti in BabelNet e invocare il metodo ricorsivo per la costruzione dei grafi. Il metodo inizia invocando la classe GraphHandler. Questa classe, che implementa l'interfaccia Handler, ha la responsabilità di occuparsi di tutte le operazioni relative alla mappa dei grafi: sia la lettura da file e sia la scrittura su file. Inizialmente si controlla

se il grafo è già presente in memoria; se il risultato è positivo, la mappa viene caricata e ritornata al chiamante, altrimenti si inizia il processamento. Nelle righe 67 e 68, vengono create due variabili il cui scopo è quello di tener traccia della percentuale di completamento dell'algoritmo, informazione necessaria allo sviluppatore. È stato doveroso utilizzare una variabile del tipo AtomicInteger in quanto, per l'iterazione, viene utilizzata un' espressione lamba, la quale, al suo interno, non consente la modifica di variabili. Successivamente inizia la parte fondamentale dell'algoritmo. Si scorrono le keys di syn2senseandedges mediante uno *stream()*. Questo metodo, introdotto in Java8, prevede la suddivisione della struttura da scorrere in flussi, in modo da poter operare su di essa in maniera parallela. Questo ha permesso di sfruttare il maggior numero di core possibili della CPU e velocizzare le operazioni. All'interno del *forEach()* viene creata la mappa, inserendo come key l'i-esimo nodo dell'iterazione (syn) e, come values, il risultato del metodo *recursiveWordnetGraph(syn)*. Alla fine del ciclo, si salva in memoria la struttura appena creata e la si ritorna al chiamante.

```
60    @Override
61    public Map<String, Graph<String, DefaultEdge>> build()
62    {
63        GraphHandler handler = new GraphHandler();
64        if(handler.checkIfEmpty())
65        {
66            AtomicInteger counter = new AtomicInteger(0);
67            int size = this.syn2senseandedges.keySet().size();
68            this.syn2senseandedges.keySet().stream().forEach(syn ->
69            {
70                try
71                {
72                    this.mapOfWordnetSynsets.put(syn, recursiveWordnetGraph(syn));
73                } catch (UnknownWordException e){}
74
75                System.out.println(this.mapOfWordnetSynsets.get(syn).vertexSet().size());
76                System.out.print(("Processing---------"+((float)(counter.addAndGet(1)*100))/size) + "%\r");
77                System.out.flush();
78            });
79            handler.write(this.mapOfWordnetSynsets);
80            return this.mapOfWordnetSynsets;
81        }
82        else
83            return handler.read();
84    }
85
```

Figura 4.4: Metodo *build()*

Nella Figura 4.5 vengono mostrati due metodi, la parte più corposa dell'algoritmo. La prima funzione (*recursiveWornetGraph(String root)*) è un metodo di appoggio creato per astrarre il chiamante (*build()*) dal metodo vero e proprio (*auxRecursiveWordnetGraph()*). Il suo scopo è quello di inizializzare il metodo sottostante, al fine di consentirgli la ricorsione. Il metodo principale della classe che si sta analizzando è *auxRecursiveWordnetGraph*.

Questa funzione non è nient'altro che una visita in profondità di un grafo (*DFS - Depth first search*). Sebbene la libreria JGrapht (vedi Sezione 2.5.2) implementasse una DFS efficente, è stato necessario riscrivere l'algoritmo in quanto era incompatibile con gli scopi di questo elaborato. La funzione auxRecursiveWordnetGraph prende in input quattro parametri:

1. **String root**, è la variabile rappresentante il nodo di partenza dal quale verrà costruito il grafo.

2. **int deep**, è la profondità alla quale dovrà essere visitato il grafo.

3. **String rootSense**, è il senso del nodo di partenta. Questo significato è recuperato mediante il metodo *getSense()*, metodo presente all'interno della classe BabelSenseAndEdges. Si è preferito mantenere l'informazione in un'apposita variabile in quanto, durante la ricorsione, venivano effettuati molti confronti tra il senso del nodo di partenza e i sensi dei nuovi nodi da aggiungere al grafo, raffronti necessari a calcolare la distanza cosinusoeidale. Pertanto, invece che invocare *getSense()* per tutti i confronti, si è preferito garantire un accesso direttamente alla variabile contenente l'informazione necessaria.

4. **Graph<String, DefaultEdge> graph**, come è intuibile, questa variabile rappresenta il grafo che si stà costruento. Viene passato come parametro in quanto i nodi vengono aggiunti durante la ricorsione.

Si noti che il metodo che si stà analizzando prevede il controllo di una *UnknownWordException*, eccezione prevista nella libreria Medillia - Word2Vec. Questa eccezione viene lanciata nel momento in cui si cerca di calcolare la distanza cosinusoeidale tra due stringhe non presenti nel modello. L'algoritmo inizia verificando se il nodo corrente è presente o meno nel grafo e, nell'eventualità in cui non lo è, viene aggiunto. Successivamente vengono salvati all'interno di una lista i nodi adiacenti al nodo corrente. Questi nodi, contenuti all'interno della mappa dei BabelSenseAndEdges, vengono restituiti mediante il metodo *getEdges()*. A questo punto, mediante un ciclo for, viene scorso l'oggetto contenente i nodi. In questa iterazione si effettuano tutte le operazioni tra il nodo "padre" (root) e i nodi "figli" (childID). All'interno del ciclo sono presenti due strutture condizionali.

Nella prima condizione si verifica se childID è presente nel grafo corrente, nel caso in cui lo è, viene controllato ulteriormente se è uguale al "padre" root, questo per evitare creazioni di cicli. Solamente se anche quest'ultima condizione non viene verificata, si procede con la costruzione del grafo aggiungendo un arco tra il nodo padre e childID. Se l'*if* non viene verificato, si entra nel caso in cui bisogna controllare se il nodo che si stà analizzando è da aggiungere al grafo o meno. In questo caso viene utilizzato il word2vec, come è possibile vedere alla riga 96 della Figura 4.5. Si calcola la distanza cosinusoeidale tra il senso del padre e il senso di childID. Solamente se la distanza tra i due concetti supera una percentuale dello 0.7% (cifra consigliata da Google per la disambiguazione), viene aggiungo childID al grafo e viene creato un nuovo collegamento tra root e il figlio. Nella seconda struttura condizionale si verifica se si è scesi alla profondità voluta, effettuando o meno la chiamata ricorsiva. Alla fine del processo ricorsivo viene ritornato il grafo completo al chiamante.
Sarà possibile effettuare il ranking della frase in input solamente dopo che il disambiguatore sarà ultimato, ovvero dopo che sono stati scorsi tutti i nodi di WordNet contenuti in BabelNet.

```
76    private Graph<String, DefaultEdge> recursiveWordnetGraph(String root) throws UnknownWordException
77    {
78        return auxRecursiveWordnetGraph(root, 0, this.syn2senseandedges.get(root).getSense(), new SimpleDirectedGraph<>(DefaultEdge.class));
79    }
80
81    private Graph<String, DefaultEdge> auxRecursiveWordnetGraph(String root, int deep, String rootSense, Graph<String, DefaultEdge> graph)
82            throws UnknownWordException
83    {
84        if(!graph.containsVertex(root))
85            graph.addVertex(root);
86        List<String> childs = this.syn2senseandedges.get(root).getEdges();
87        for(String childID : childs)
88        {
89            if(graph.containsVertex(childID))
90            {
91                if(!root.toString().equals(childID))
92                    graph.addEdge(root, childID);
93            }
94            else
95            {
96                if(this.searcher.cosineDistance(rootSense, this.syn2senseandedges.get(childID).getSense()) >= 0.7)
97                {
98                    graph.addVertex(childID);
99                    graph.addEdge(root, childID);
100               }
101           }
102           if(deep < 1)
103               auxRecursiveWordnetGraph(childID, deep+1, rootSense, graph);
104       }
105       return graph;
106   }
```

Figura 4.5: Algoritmo ricorsivo

4.2 Componente Ranker

La componente *Ranker*, visibile nella Figura 4.6, è la classe la cui responsabilità è quella di effettuare operazioni di matching tra il disambiguatore e la frase in input. L'unica variabile di istanza che presenta è proprio quella della mappa dei grafi. La componente Ranker è l'ultimo passo di questo progetto riguardante la disambiguazione. Dopo l'esecuzione dell'unico suo algoritmo, viene restituito al chiamante una lista di *BabelScore* ordinata per punteggio, contenente tutti i grafi presenti nel disambiguatore e la loro relativa votazione. Qui di seguito verrà analizzato in maniera dettagliata l'algoritmo di ranking.

4.2.1 Algoritmo di ranking

Il metodo *ranking* è il cuore di questa classe. Prende in input un solo parametro: *input2synId*. Questa mappa, precedentemente costruita, ha come *keys* l'insieme di parole della frase da disambiguare, mentre come *values* una lista contenente, per ogni vocabolo, l'insieme dei sensi presenti in BabelNet. L'algoritmo presenta due cicli for. Nel primo ciclo viene scorsa la mappa dei grafi precedentemente creata da GraphBuilder. Successivamente ogni elemento della mappa, quindi un grafo, viene incrociato con tutti gli oggetti presenti nell'input2synId mediante il secondo ciclo for. All'interno di quest'ultimo vi è un istruzione condizionale, la quale verifica se nel grafo vi sono tutti i sensi legati all'i-esimo elemento della mappa dei concetti e, nell'eventualità in cui la verifica si concretizzasse, viene assegnato un punto al grafo. Quando è stata scorsa tutta l'input2synId, si crea una nuova instanza di un oggetto BabelScore, vi viene assegnato il grafo e il relativo punteggio. Infine l'oggetto è aggiunto alla lista risultante. Il procedimento poi si sviluppa per tutti gli elementi della mappa dei sensi e per tutti i grafi. Alla fine del processo si ordina la lista dei grafi, rappresentante una classifica, secondo la votazione ottenuta da ogni grafo e viene riportato il risultato al chiamante.

```java
/**
 * @author Silvio Severino
 */
public class Ranker
{
    private Map<String, Graph<String, DefaultEdge>> mapOfWordnetGraph;

    public Ranker()
    {
        GraphBuilder builder = new GraphBuilder();
        this.mapOfWordnetGraph = builder.build();
    }

    @SuppressWarnings("unlikely-arg-type")
    public List<BabelScore> ranking(Map<String, List<BabelSynsetID>> input2synId)
    {
        List<BabelScore> listOfGraphs = new LinkedList<>();
        this.mapOfWordnetGraph.keySet().forEach(elem ->
        {
            AtomicInteger counter = new AtomicInteger(0);
            input2synId.keySet().forEach(input -> {

                if(this.mapOfWordnetGraph.get(elem).vertexSet().stream().anyMatch(input2synId.get(input)::contains))
                    counter.getAndAdd(1);
            });
            BabelScore score = new BabelScore(this.mapOfWordnetGraph.get(elem), counter.get());
            listOfGraphs.add(score);
        });

        Collections.sort(listOfGraphs);
        return listOfGraphs;
    }
}
```

Figura 4.6: Algoritmo di ranking

Capitolo 5

Sviluppi futuri

5.1 Migliorie all'algoritmo di ranking

Sebbene la versione implementata dell'algoritmo di ranking abbia già discreti risultati, esso, presenta numerose anomalie. Il fatto che venga attribuito un punteggio unitario sulla base della presenza o meno dei sensi nei grafi, senza tener conto della profondità alla quale il concetto è situato, limita notevolmente il processo di disambiguazione. Considerando che è stato utilizzato BabelNet come dizionario dalla quale estrarre informazioni, e che esso sia basato sulle relazioni semantiche, implica che un nodo derivato, quindi a distanza due, abbia meno rilevanza rispetto ad un nodo con un legame diretto. Pertanto per le ragioni ivi citate, sarebbe più opporto rivisitare l'algoritmo di ranking andando a considerare non solo la distanza alla quale il noto è situato e quindi, conseguentemente, assegnare un punteggio opportuno, sarebbe altresì opportuno assegnare una votazione anche sulla base del tipo di relazione che lega due nodi.

5.2 Ottimizzazioni ulteriori

Nel corso dei capitoli precedenti sono state descritte alcune ottimizzazioni apportate al progetto del tirocinio. Tuttavia sarebbe opportuno rafforzare i miglioramenti al fine di rendere il progetto più efficente.

5.2.1 Occupazione di memoria

Il lato dell'occupazione di memoria è la parte che più necessiterebbe di perfezionamenti. La sezione del progetto che fa da collo di bottiglia per la RAM è la parte dell'algoritmo per la costruzione del disambiguatore. Sebbene sia stato l'argomento del tirocinio nel quale vi sono stati il maggior numero di scelte progettuali e ragionamenti tecnici, l'occupazione ad oggi risulta ancora del 40%. Pertanto, considerando che la macchina sulla quale è stato realizzato l'elaborato abbia 8GB di RAM, l'occupazione totale risulta di circa 3GB. Le migliorie apportabili protebbero riguardare un ulteriore filtraggio dei nodi del disambiguatore, magari tenendo conto delle relazioni semantiche fuorvianti al fine di un processo di disambiguazione.

5.2.2 Riduzione delle tempistiche d'esecuzione

Le tempistiche d'esecuzione sono già ad un discreto risultato, basti pensare che l'algoritmo di ranking riporti un risultato in mediamente 2.1s. Tuttavia questo dato verrebbe influenzato notevolmente se le strutture necessarie non siano presenti in memoria, in particolare il disambiguatore. Infatti il processo completo, dall'estrazione dei nodi di WordNet contenuti in BabelNet, fino alla classificazione dei grafi, impiega circa 3h. Sebbene il tempo non sia eccessivo, migliorie simili a quelle descritte nel paragrafo dell'occupazione di memoria e modifiche all'algoritmo mediante tecniche di programmazione dinamica, potrebbero notevolmente ottimizzarlo. Si consideri inoltre che l'elaborato è stato sviluppato su un calcolatore portatile il quale potrebbe influenzare le tempistiche in termini di prestazioni.

5.3 Futuro per BabelNet

Oggigiorno BabelNet basa il suo contenuto sull'estrazione di informazioni presenti in altri dizionari. Nel settore dell'intelligenza artificiale questo tipo di approccio prende il nome di *Knowledge Based System*. Un possibile scopo per l'argomento esposto nella tesi, potrebbe essere quello di un probabile strumento utile al processo di conversione di BabelNet da una *Knowledge Based* ad un *Apprendimento non supervisionato*. Immaginando che un altro sistema riesca a generare frasi di senso compiuto finalizzate all'apprendimento

non supervisionato di BabelNet, il disambiguatore creato nel corso di questo elaborato, potrebbe verificare se il concetto della frase sia proprio quello voluto, analizzando semplicemente i primi grafi della classifica.

5.3.1 Estensione ad altre lingue

Un fattore limitante per questo progetto è proprio quello delle lingue. Il disambiguatore, ad oggi, riesce ad elaborare solamente concetti in lingua inglese, idioma sicuramente fondamentale a livello mondiale e parlato da innumerevoli persone. Tuttavia esistono altre lingue che presentano un numero maggiore di popolazioni che la parlano, ad esempio la lingua cinese, o la lingua russa e così via. Sarebbe possibile sfruttare BabelNet in quanto contenente traduzioni in 283 idiomi diversi per poter ampliare in range di disambiguazione dell'elaborato, in modo da poter soddisfare problemi di ambiguità in quelle lingue più necessitarie rispetto all'inglese.

Conclusioni

Il candidato, grazie a questa esperienza, è riuscito ad ampliare il proprio background di conoscenze tecniche, lavorando con professionisti esperti nel settore e approcciandosi, muovendo i primi passi, nel mondo del lavoro. Inoltre, grazie ai consigli del tutor, ha appreso nozioni utili non solo allo sviluppo di questo elaborato, bensì anche al mondo della programmazione, in particolare quella riguardante il natural language processing. Per questi motivi l'esperienza è risulta molto soddisfacente ed ha aiutato notevolmente il candidato alla scelta della propria laurea magistrale, quindi del proprio futuro.

Bibliografia

[Ahm17] Ahmed. Understanding word2vec for word embedding, 2017.

[AM14] R. Navigli A. Moro, A. Raganato. Entity linking meets word sense disambiguation: a unified approach. 2014.

[Ces14] C. De Sio Cesari. *Manuale di Java 8. Programmazione orientata agli oggetti con Java standard edition 8.* 2014.

[CL16] a cura di L. Cabibbo C. Larman. *Applicare UML e i pattern. Analisi e progettazione orientata agli oggetti.* 2016.

[FV04] G. Lapalme F. Vasilescu, P. Langlais. Evaluating variants of the lesk approach for disambiguating words. 2004.

[Gar10] K. Garajová. *IL LESSICO ITALIANO E LA SEMANTICA*. 2010.

[Gro] The Stanford Natural Language Processing Group. Stanford log-linear part-of-speech tagger.

[Med18] Medallia. Word2vecjava, 2018.

[Mik] T. Mikolov. Word2vec.

[Nav09] R. Navigli. Word sense disambiguation: A survey. 2009.

[RN07] O. Hargraves R. Navigli, K. Litkowski. Semeval-2007 task 07: Coarse-grained english all-words task. 2007.

[RN10] S. P. Ponzetto R. Navigli. Babelnet: Building a very large multilingual semantic network. 2010.

[RN12] S. P. Ponzetto R. Navigli. Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 2012.

[RN15] F. Cecconi R. Navigli, S. P. Ponzetto. Babelnetapi, 2015.

[Ron16] X. Rong. word2vec parameter learning explained. 2016.

[sem15] Semantica lessicale, 2015.

[sit18] Wordnet, a lexical database for english, 2018.

[SR16] P. Norvig S. Russell. *Artificial Intelligence. A Modern Approach.* 2016.

[Ste16] Katy Steinmetz. Redefining the modern dictionary. *Time magazine*, 23 maggio 2016.

[Svo10] M. Svolacchia. *Da dove vengono le nuove parole ?* 2010.

[ten] Vector representations of words.

[TW14] H. Chang T. Wei, Y. Lu. A semantic approach for text clustering using wordnet and lexical chains. 2014.

[Uni18a] Princeton University. Glossary of wordnet, 2018.

[Uni18b] Stanford University. Stanfordcorenlp, 2018.

[Van16] D. Vannella. *Enriching, Validating and Publishing a Large Multilingual Semantic Network.* PhD thesis, University of Rome "La Sapienza", 2016.

[Ver18] A. Verdolini. Progettazione e realizzazione di un sistema per article spinning, 2018.

[wik07] Intelligenza artificiale/apprendimento supervisionato, 2007.

[wik18a] Eclipse (informatica), 2018.

[wik18b] Java (linguaggio di programmazione), 2018.

[wik18c] Ubuntu, 2018.

[wik18d] Versioni di ubuntu, 2018.

[wik18e] Word embeddings, 2018.

[wik18f] Wordnet, 2018.