TLN Relazione Radicioni

Umberto Baldi, Rosita Gerbo

July 2019

1 Conceptual Similarity with WordNet

Durante questa esercitazione si è cercato di assegnare un punteggio di similarità alle coppie di parole presenti nel file *WordSim353.tab* attraverso tre divere misure di similarità (Wu & Palmer, Shortest Path e Leacock Chodorow) sfruttando la struttura di WordNet. In seguito, si è cercato di valutare tali similarità confrontandole con quelle date nel file *WordSim353.tab* utilizzando gli indici di correlazione di Pearson e Spearman.
A questo scopo sono stati creati due file: *conc_sim_WN.py* e *word_sim_structure.py*.
Si va ora a descrivere questi file nel dettaglio.

1.1 word_sim_structure.py

All'interno del file *word_sim_structure.py* è stata definita la classe *WordSim*: una struttura che permette la gestione del contenuto del file *WordSim353.tab*.
Ogni oggetto *WordSim* è composto da tre elementi: due parole e un valore numerico rappresentante la similarità a loro attribuita.
I metodi presenti all'interno del file sono dei *getter* e dei *setter*.

1.2 con_sim_WN.py

Nel file *conc_sim_WN.py* vi è la vera e propria computazione delle similarità e degli indici di correlazione.
Di seguito viene data una breve descrizione dei metodi principali.

- `wu_&_palmer(word1, word2)` riceve come parametri due parole (word1 e word2), trova tutti i loro synset presenti in WordNet e restituisce la similarità di Wu & Palmer più alta tra quelle trovate mettendo in relazione tutti i synset trovati.
  Il calcolo della similarità avviene sfruttando i metodi
  `lowest_common_subsumer(sense1, sense2)` e `max_depth()` da noi implementati.
  Se non viene trovato nessun senso (per una o per entrambe le parole), o se nessuna combinazione dei loro sensi ha una radice in comune all'interno della struttura di WordNet, la similarità restituita è pari a zero.

- `max_depth(synset)` riceve come argomento un synset di WordNet e, attraverso delle chiamate ricorsive che sfruttano le relazioni di iperonimia presenti in WordNet, restituisce la lunghezza del percorso più lungo che va dall'argomento alla radice della struttura WordNet.

- `lowest_common_subsumer(synset1, synset2)` riceve come parametri due synset di WordNet, attraverso il metodo `hypernym_paths(synset)` da noi implementato ottiene tutti i percorsi che dai sensi arrivano fino alla radice e, confrontando tutti i percorsi trovati, restituisce il senso che si trova più in basso nella struttura di WordNet e che è comune ai due sensi passati come parametro.
  Se questo senso non è trovato, viene restituito il valore None.

- `hypernym_paths(synset)` riceve come parametro un synset di WordNet e, percorrendo ricorsivamente la struttura di WordNet attraverso il metodo `hypernyms()` di WordNet, restituisce una lista di liste ciascuna contenente un percorso che parte da una radice e arriva al synset passato come parametro.

- `sht_path_sim(word1, word2)` riceve come parametri due parole (word1 e word2), trova tutti i loro synset presenti in WordNet e, combinando questi ultimi, restituisce una similarità basata sul percorso più corto da fare all'interno della struttura di WordNet per passare da un senso all'altro tra quelli trovati. Il calcolo del percorso più corto viene effettuato dal metodo `shortest_path(synset1, synset2)` da noi implementato.
  Se non è stato trovato nessun percorso, viene resituito il valore zero.

- `shortest_path(synset1, synset2)` riceve come parametri due synset, trova tutti i percorsi che dalla radice arrivano ai sensi attraverso il metodo `hypernym_paths(synset)` da noi implementato e, analizzando ogni percorso trovato partendo dal fondo della struttura (e quindi dai sensi), quando trova un iperonimo comune ai due sensi calcola la distanza dei sensi dall'iperonimo trovato e, se questa è inferiore a quella calcolata in precedenza, viene salatava.
  Se non viene trovato nessun iperonimo comune, allora viene restituito il valore None.

- `leacock_chodorw(word1, word2)` riceve come parametri due parole (word1 e word2), trova tutti i loro sensi presenti in WordNet e, combinando questi ultimi, restituisce la similarità di Leacock Chodorow più alta trovata.
  La similarità viene calcolata utilizzando il metodo `shortest_path(synset1, synset2)` da noi implementato per trovare la distanza tra i sensi analizzati ed un accorgimento: nel caso in cui i sensi siano uguali viene aggiunta una unità per evitare che venga calcolato il logaritmo di zero.
  Se non viene trovata alcuna similarità, viene ritornato il valore zero.

- `WN_DepthMax()` ritorna la profondità massima della struttura WordNet.

2 Semantic Evaluation

In questa esercitazione si è implementato l'algoritmo di Lesk in versione semplificata. Lo scopo di questo algoritmo è quello di fare Word Sense Disambiguation (WSD). Dopo aver implementato l'algoritmo è stato utilizzato il file *sentences.txt* per disambiguare i termini polisemici, identificati tra due asterischi, al suo interno. Inoltre ogni frase viene riscritta sostituendo il termine polisemico con tutti i sinonimi eventualmente presenti nel synset. Infine vengono estratte le prime 50 frasi dal corpus annotato *SemCor* e viene disambiguato il primo sostantivo trovato all'interno della frase. Al termine di tutto, il programma indica l'accuratezza con cui ha disambiguato i termini polisemici. L'esercizio è stato diviso in due files *lesk_algoritm.py* e *wsd.py*

2.1 wsd.py

All'interno di *wsd.py* si trova il main del programma. Quest'ultimo si occupa di leggere il file, di parsificare le frasi (togliere caratteri superflui e riconoscere la parola da disambiguare) e inoltre utilizza l'implementazione dell'algoritmo di Lesk, all'interno di *lesk_algorithm.py* chiamando `simplifiedLesk(word,sentence)`. Il main si occupa anche di sostituire la parola disambiguata con gli eventuali sinonimi, all'interno della frase. Infine prende le prime 50 frasi all'interno di *SemCor* ed esegue la disambiguazione sul primo sostantivo trovato all'interno di una frase. Al termine di tutto restituisce l'accuracy ottenuta sulle frasi di *SemCor*.

- `find_suitable_word(words)` è un metodo un po' grezzo di trattare i sostantivi composti da più parole. Viene utilizzato poichè in wordnet il separatore di più parole può essere sia il carattere vuoto che il carattere '_' . Un altro metodo un po' meno grezzo che si sarebbe potuto usare sarebbe stato quello di evitare i sostantivi composti da più parole.

2.2 lesk_algoritm.py

All'interno del file *lesk_algorithm.py* viene implementato l'algoritmo di Lesk semplificato, con i relativi metodi di appoggio. Si è cercato di mantenere per quanto possibile la struttura dello pseudocodice fornito nel testo dell'esercizio. Infatti molti metodi di appoggio probabilmente sono superflui per l'implementazione, tuttavia sono stati mantenuti anche per leggibilità.

- `simplifiedLesk(word,sentence)` come è ovvio che sia è l'implementazione dell'algoritmo di Lesk semplificato, *word* è la parola da disambiguare, *sentence* è il resto della frase.
- `most_frequent_sense(word)` restituisce il senso più frequente (il primo)
- `word_set(sentence)` toglie dalla *sentence* le stopwords e restituisce il *context*

- `senses(word)` restituisce una lista di sensi di una *word*
- `compute_overlap(signature, context)` esegue il calcolo dell'overlap (parole in comune) tra due set *signature* e *context*. *signature* è un set di parole che compaiono nella definizione e negli esempi di un determinato senso.

3 Riassunti Automatici utilizzando NASARI

Lo scopo di questa esercitazione è quello di definire un creatore di riassunti estrattivi che funzioni in modo automatico.
Per fare ciò è stato utilizzato il file *dd-small-nasari-15.txt* contenente un sottoinsieme di vettori NASARI composti da 15 feature ed un numero di lemmi inferiore rispetto a quello presente originariamente in NASARI. La struttura all'interno del quale il file è stato salvato è un dizionario il quale ci ha permesso, ponendo i lemmi come chiavi, di ottenerne i vettori.
Si va ora ad analizzare i metodi che portano alla creazione dei riassunti automatici.

- `nasari_to_dict(nasari)` restituisce un dizionario contenente come chiavi i lemmi definiti nel file *nasari* passato come parametro. Come valori corrispondenti alle chiavi è stata inserita una stringa contenete le righe presenti nel file a meno dell'identificatore e della parola stessa.
  I lemmi che definiscono le chiavi sono stati inseriti in formato minuscolo.
- `words_to_dict(sentence, stop_words, nasari_dict)` restituisce un dizionario contenente come valori dei vettori NASARI a partire dalle parole presenti all'interno del parametro *sentence*. Le chiavi del dizionario sono le parole presenti in *sentence* da cui sono state eliminate stop_words (salvate all'interno della variabile *stop_words* grazie a *nltk.corpus* e passate come parametro) e punteggiatura (eliminata grazie `str.maketrans()` che ci ha permesso di sostituire la punteggiatura con stringhe vuote). Inoltre, prima di inserire le parole come chiavi nel dizionario, queste sono state riscritte minuscole e nella forma del loro lemma.
  Al fine di creare il vero e proprio dizionario sono stati usati i metodi `init_dict(dictionary, nasari_dict)` e `increase_dict(dictionary, nasari_dict)` da noi creati.
- `init_dict(dict, nasari_dict)` inizializza il dizionario passato come parametro andando ad inserire i valori copiandoli dal dizionario creato dal file di NASARI.
  Non è presente alcun valore di ritorno perché, essendo il parametro *dict* un oggetto mutabile, questo viene modificato direttamente.
- `increase_dict(dict, nasari_dict)` ritorna un dizionario espanso. Questa caratteristica rende più probabile trovare una parola all'interno del dizionario, considerando che il vettore NASARI che viene fornito all'inizio è in forma semplificata e non è così ampio. Il funzionamento del metodo

è il seguente: prende in input il dizionario e il vettore nasari, scorre il dizionario e ricerca ogni valore per ogni chiave all'interno del vettore nasari aggiungendo poi il valore come chiave all'interno di un nuovo dizionario che verrà ritornato. Inoltre aggiorna anche i vari pesi normalizzandoli.

- `title_cohesion(title_dict, sentence_dict)` ritorna la coesione tra il titolo (*title_dict*) ed una frase del testo (*sentence_dict*) attraverso la *weighted overlap*. Quest'ultima viene calcolata richiamando il metodo `weigheted_overlap(vector1, vector2)` per ogni coppia di parole presenti nel titolo e nella frase.

- `weigheted_overlap(vector1, vector2)` calcola la weighted overlap tra due vettori (*vector1* e *vector2*). Per ogni coppia di parole uguali presente all'interno dei due vettori, all'interno della variabile *overlap* viene salvata una tripla contenente la parola seguita dai due pesi che ha nei due vettori. Questi due pesi vengono poi utilizzati nella formula per clcolare la weigthed overlap vera e propria.

Dopo aver utilizato i metodi appena descritti, per creare le strutture adatte, viene eseguito il vero e proprio lavoro riassuntivo.
Per fare ciò, le frasi vengono riordinate in ordine decrescente in base alla loro coesione con il titolo dell'articolo. Successivamente si prendono le prime tot frasi in base alla percentuale di testo originale che si vuole mantenere.
A questo punto le frasi scelte vengono riordinate in base al testo originario e poi stampate.

3.1 Commenti sui riassunti creati

Non tutte le frasi che noi riteniamo importanti sono selezionate dall'algoritmo. In generale, però, il risultato permette di comprendere di cosa si parla all'interno degli articoli originali.
L'algoritmo restituisce non paragrafi, ma frasi. Questo, a nostro parere, permette (vista anche l'esigua dimensione degli articoli) di avere una visuale più ampia e varia del testo originale. Infatti, in questo modo vi è la possibilità che vengano scelte frasi appartenenti a paragrafi diversi aumentando la probabilità di restituire frasi con all'interno argomenti differenti e diminuendo così la monotematicità.

4 Annotazione di Corpora e Sense Identification

In questa utlima esercitazione si è andati ad annotare manualmente e individualmente la similarità tra cento parole date; le annotazioni sono state salvate nei file `words_umbo` e `words_ro` i quali, in ogni riga, contengono le due parole date, la similarità decisa dall'annotatore ed un eventuale commento che spieghi la decisione presa. Si sono poi calcolati gli indici di Pearson e di Spearman tra i due file
In seguito, utilizzando la risorsa BabelNet, grazie a dei vettori NASARI sono

stati trovati gli ID e le relative glosse dei BabelNet synset che rendevano massima la cosine similarity per ogni coppia di parole. Si sono infine confrontate le glosse trovate con i sensi che si erano pensati durante lo svolgimento dell'anotazione.
Si va ora a descrivere nel dettaglio i file ed i metodi utilizzati.

4.1 words.txt, words_ro.txt & words_umbo.txt

All'interno del file `words.txt` sono presenti 100 coppie di parole estrapolate dalle 500 presenti nel file `it.test.data.txt` utilizzando `sem_eval_mapper.py` a cui è stato passato come parametro di input il valore *gerbo*.
I file `words_ro` e `words_umbo` rappresentano lo stesso file a cui è stato aggiunto un valore di similarità per ogni coppia di parole con un eventuale commento a chiarificare la scelta fatta. In `words_ro` i valori sono stati assegnati da Rosita Gerbo, in `words_umbo` da Umberto Baldi.
In questi file vi è quindi la parte di annotazione dell'esercitazione.

4.2 corpora_sense_annotation.py

All'interno del `main` del file `corpora_sense_annotation.py` viene effettuata una valutazione sui valori dati in `words_ro` e `words_umbo` e, successivamente, vi è la vera e propria parte di identificazione del senso.
Per la parte di valutazione, per ogni coppia di parole viene effettuata la media dei valori assegnati. Vengono inoltre calcolati gli indici di correlazione di Pearson e Spearman utilizzando i metodi presenti nella libreria `scipy`.
Per la parte di identificazione del senso sono stati utilizzati alcuni metodi che verranno ora descritti.

- `nasari_to_vectors()` ritorna una lista di vettori NASARI inserendovi quanto trovato nel file *mini_NASARI.tsv*
- `sem_eval_to_vectors()` ritorna una lista all'interno del quale ogni elemento rappresenta una parola seguita dai suoi sysnset di BabelNet. Le parole inserite all'interno della lista, sono quelle presenti all'interno del file *SemEval17_IT_senses2synsets.txt*
- `senseIdentification(words, nasari_vectors, sem_eval_vectors)` ritorna, per ogni coppia di parole presente nell'argomento *words*, i sensi che hanno una cosine similarity maggiore (le parole all'interno di una coppia fanno quindi da disambiguatore l'una per l'altra).
  Per arrivare a questo risultato, sono state effettuate alcune operazioni.
  Per ogni coppia di parole presenti in *words*, vengono confrontati tutti i sensi della prima parola all'interno della coppia con tutti i sensi della seconda. Durante il confronto viene cercato un vettore NASARI per ogni senso e, se questo si trova per entrambi i sensi presi in considerazione, viene calcolata la cosine similarity. Se la cosine similarity è migliore di quella calcolata in precedenza, i due sensi vengono salvati e, una volta confrontati tutti

i sensi per la coppia di parole, i due sensi migliori vengono inseriti nella variabile di ritorno *best_senses*.

- `find_gloss(bn_id)` ritorna la glossa corrispondente al synset del BabelNet ID passato come parametro attraverso delle richieste al server di BabelNet.

4.3 Accuratezza

L'accuratezza tra i sensi in noi evocati durante l'annotazione e quelli evocati dalle glosse restituite dall'algoritmo è stata calcolata con una frazione: il numeratore indica il numero di sensi interpretati allo stesso modo; il denominatore indica il numero di elementi presi in considerazione. Per fare ciò è stato utilizzato il metodo `compute_accuracy(accuracies)`.
Sono state calcolate due tipologie di accuratezza: una mette a confronto i sensi delle parole prese singolarmente; l'altra i sensi delle parole prese secondo le coppie presenti nel file *words.txt*.