Università degli Studi di Torino
Dipartimento di Informatica

Relazione seconda parte Laboratorio Tecnologie del Linguaggio Naturale

Professore:
Prof. Daniele Radicioni

Candidati:
Lorenzo Giacometti
Matteo Giovanale
Daniel Haures

ANNO ACCADEMICO 2023/2024

Indice

Esercizio 1 - conceptual similarity with WordNet

L'esercitazione 1 consiste nell'implementare tre misure di similarità basate su WordNet: Wu & Palmer, Shortest Path e Leakcock&Chodorow.
Per ciascuna di queste misure di similarità, siamo andati a calcolare i coefficienti di correlazione di Spearman e i coefficienti di correlazione di Pearson fra i risultati ottenuti e quelli "target" presenti nel file annotato "WordSim353.csv".
Tutte le formule utilizzano il metodo *LCS*, il quale permette sia del Lowest Common Subsumer che il calcolo del cammino più corto tra due synset.
Il calcolo dei coefficienti di correlazione è agevolato dalle librerie scipy.

LCS

I parametri del metodo sono i due synset (*s1* e *s2*) di cui si desidera calcolare il lowest common subsumer.
Le strutture dati implementate all'interno del metodo sono due HashMap, contenenti ciascuna come chiave l'unità lessicale e come valore un intero. In aggiunta a queste sono state implementate due liste.
L'algoritmo esegue in maniera speculare e iterativa una risalita gerarchica sui due synsets forniti in input, collocati rispettivamente nel ramo sinistro e destro dell'albero.
Prima di ogni ciclo di interazione, nelle liste verranno inseriti i synset di cui si desidera l'espansione degli iperonimi. Queste sono perciò inizializzate reciprocamente con *s1* e *s2*.
Successivamente, si procede con l'espansione degli elementi inseriti in lista, ossia la ricerca dei loro iperonimi.
Se l'espansione di synset sinistro genera un iperonimo già presente all'interno della hashmap reciproca destra, allora l'iperonimo in questione è il Lowest Common Subsumer.
In alternativa, l'algoritmo procede con l'inserimento degli iperonimi all'interno dell'hashmap del ramo espanso e all'interno della lista, in attesa di essere esplorato.
Come è possibile dedurre, le due HashMap conterranno tutti i synset che formano un potenziale percorso per arrivare al minimo antenato in comune, e a ognuno sarà associato un numero rappresentante il ciclo di iterazione in cui è stato aggiunto, ossia una sorta di "livello" di esplorazione gerarchica.

Il nostro metodo ritornerà il Lowest Common Subsumer e lo shortest path tra i synset *s1* e *s2*, ricavato dalla somma del livello in cui è stato trovato lcs all'interno delle due hashmap.

Criticità

Il lower common subsumer trovato dall'algoritmo è corretto perché la risalita gerarchica alternata tra i due rami esclude la possibilità di ignorare percorsi di risalita più brevi.
A presentare più criticità è il calcolo dello shortest path, è infatti possibile che un iponimo dei termini esplorati porti a un percorso più corto. Tuttavia la complessità computazionale di un algoritmo capace di considerare queste occorrenze rende preferibile l'utilizzo del lowest common subsumer.

Esercizio 2 - WSD

Nella prima versione del nostro programma andavamo ad estrarre 50 frasi dal Corpus Semcor, per ogni frase andavamo a salvare in due liste distinte tutte le parole che formavano quella frase e tutti i synset corretti associati a quelle parole. Successivamente richiamavano l'algoritmo di Lesk su una parola estratta randomicamente dalla lista, procedevamo a predire il senso associato a quest'ultima e successivamente lo andavamo a confrontare con il synset corretto; andando a segnare se la previsione era corretta o meno. Questo processo veniva ripetuto per ognuna delle 50 frasi.
Abbiamo poi deciso di implementare l'algoritmo di Lesk migliorato con Gemini per la Word Sense Disambiguation (WSD).
In questa variante continuiamo a utilizzare il corpus di SemCor per estrarre 50 frasi su cui basare l'esercizio, ma nell'algoritmo di Lesk per calcolare l'overlap, non andiamo più ad utilizzare la gloss estratta da WordNet ma, tramite un loop su tutti i possibili sensi della parola andiamo a generare, con Gemini, per ogni senso una gloss usando questo prompt: "["write some example which contains the word", word,"defined as", sense.definition()]".
Utilizzando il metodo "get_overlap()" otteniamo l'overlap della gloss appena generata e se quest'ultimo è maggiore del max_overlap allora facciamo l'update con l'overlap corrente e settiamo il best_sense con il senso corrente.

Criticità

Durante i test della nostra implementazione del Lesk, utilizzando Gemini per creare la gloss dei termini, abbiamo notato che, con l'impostazione di default di "temperature": 0.7, l'output di Gemini era molto verboso e a nostro parere poco coerente. Per ottenere delle frasi di esempio che si avvicinano il più possibile a quelle di WordNet abbiamo impostato il parametro della temperatura a 0.3, ottenendo risultati soddisfacenti.
Per quanto riguarda il parametro "top_k", lo abbiamo aumentato leggermente poiché, dovendo generare più frasi, desideravamo ottenere esempi più diversificati tra loro.

default:

```
generation_config = {
  "temperature": 0.3,
  "top_p": 0.9,
  "top_k": 50
}
```

nostro:

```
generation_config = {
  "temperature": 0.3,
  "top_p": 0.9,
  "top_k": 70
}
```

Esercizio 3 - twitting like (a) Trump

In questo esercizio abbiamo preso il Trump Twitter Archive (~290 tweet attribuiti all'ex Presidente US, disponibile fra i materiali della lezione) e abbiamo acquisito due language models,uno a bi-grammi e uno a tri grammi, su questo set di testi. Abbiamo utilizzato questi due modelli per produrre tweets ex-novo.

Criticità

Inizialmente i valori generati dall'algoritmo non erano estratti tramite un criterio randomico pesato, ma utilizzavano un criterio basato sulla prima parola più probabile, di conseguenza la frase generata era sempre la stessa, siccome l'algoritmo selezionava sempre il trigramma/bigramma avente la probabilità condizionata maggiore.
Un'altro problema riscontrato è legato all'incapacità dell'algoritmo di porre fine alla generazione di una frase. Questa criticità è stata risolta negando al modello la possibilità di inserire trigrammi già presenti nella frase.

```
no_repeated_key = list(filter(lambda key: key not in "
".join(sentence), chosen_key))
```

Con questo filtro andiamo a cercare se la parola che abbiamo estratto non è contenuta nel tweet, se è così andremo a fare l'append di no_repeated_key al tweet che stiamo generando.