Roberto Gjokaj

Alberto Alpe

Università degli studi di Torino

Corso di Laurea Magistrale di Informatica

Tecnologie del Linguaggio Naturale
Parte 2 – Prof. Daniele Radicioni

Progetto 1: Conceptual Similarity con WordNet

Panoramica del Progetto

L'obiettivo del progetto è stabilire la similarità semantica tra coppie di termini.
L'input è costituito da 353 coppie di termini contenute nel file "WordSim353.csv". Per ogni riga del file, oltre ai due termini, c'è anche un valore di similarità da confrontare con i valori ottenuti.
La richiesta è di implementare tre metriche di similarità le cui formule vengono fornite nella consegna:
-metrica "Wu & Palmer"
-metrica "shortest path"
-metrica "Leakcock & Chodorow"
Infine bisogna calcolare gli indici di correlazione di Spearman e di Pearson tra i risultati ottenuti e quelli target(annotati nel file).

Per sviluppare il progetto e calcolare le metriche si esplorano i vari synsets di WordNet, di cui si è trattato durante il corso.

Scelte implementative
Abbiamo implementato il codice in un notebook chiamato "1-ConceptSimilarity.ipynb".
Tra le varie librerie importate c'è `nltk` (natural language toolkit), che useremo per accedere a WordNet.
Abbiamo implementato la Maximum Similarity con una funzione `max_similarity(term1, term2, similarity_function)`.
Abbiamo poi definito funzioni per calcolare le varie metriche di similarità, così da poterle richiamare insieme in un blocco unico.
Di seguito le funzioni che implementano le tre metriche:

Similarità di Wu-Palmer

```
def wu_palmer_similarity(syn1, syn2):
    lcs_candidates = syn1.lowest_common_hypernyms(syn2)
    if not lcs_candidates:
        return 0
    lcs = lcs_candidates[0]
    depth_lcs = lcs.max_depth()
    depth_syn1 = syn1.max_depth()
    depth_syn2 = syn2.max_depth()
    return (2 * depth_lcs) / (depth_syn1 + depth_syn2)
```

Calcolo della Shortest path similarity

```
def shortest_path_similarity(syn1, syn2):
    path_length = syn1.shortest_path_distance(syn2)
    if path_length is None:
        return 0
    return 2 * depth_max - path_length
```

Calcolo della similarità di Leacock&Chodorow

```
def leacock_chodorow_similarity(syn1, syn2):
    if depth_max == 0:
        return 0
    path_length = syn1.shortest_path_distance(syn2)
    if path_length is None or path_length == 0:
        return 0
    return -np.log((path_length) / (2 * depth_max))
```

Il secondo e il terzo calcolo utilizzano la variabile `depth_max`, che abbiamo calcolato all'inizio per non dover ripetere ogni volta il calcolo.

```
depth_max = max(max(len(hyp_path) for hyp_path in ss.hypernym_paths()) for ss in wn.all_synsets())
```

Dopo aver aperto "WordSim353.csv", salviamo in una lista di liste `combo_parole` ogni riga del dataset, ottenendo così una serie di triple (le coppie di termini da confrontare e il valore di similarità da confrontare con quelli ottenuti):

```
    combo_parole = []

    for riga in righe:
        riga = riga.strip().split(',')
        combo_parole.append([riga[0], riga[1], riga[2]])

    print(combo_parole)
```

✓ 0.0s

```
[['love', 'sex', '6.77'], ['tiger', 'cat', '7.35'], ['tiger', 'tiger', '10.00'], ['book', 'paper', '7.46'],
```

Infine proprio su `combo_parole` vengono richiamate le tre funzioni per il calcolo della similarità, insieme a quelle per i due indici di correlazione.
Viene eseguita la funzione `calculate_correlations`, la quale a sua volta richiama la funzione `max_similarity` che calcola la massima similarità tra due termini secondo una delle metriche possibili(che viene passata come parametro).

Osservazioni e conclusioni

Di seguito sono riportati i risultati ottenuti.

Wu & Palmer
Spearman: `correlation=0.3504207555845377, pvalue=1.2310344613866503e-11,`
Pearson: `statistic=0.2961610096453874, pvalue=1.4082869911851914e-08`

Shortest Path
Spearman: `correlation=0.2895253335747299, pvalue=3.0336884637747205e-08,`
Pearson: `statistic=0.16653216769600954, pvalue=0.0016911511526584368`

Leacock & Chodorow
Spearman: `correlation=0.2236163845432123, pvalue=2.232370373511577e-05,`
Pearson: `statistic=0.2397364846884062, pvalue=5.242105939464922e-06`

Da quanto abbiamo potuto constatare, sia per l'indice di Spearman che per quello di Pearson, i valori ottenuti indicano una correlazione debole/moderata, mentre il pvalue basso indica che dal punto statistico la correlazione è altamente significativa e poco casuale.
Inoltre Wu & Palmer risulta essere la metrica con le correlazioni più alte, mentre Leacock & Chodorow ha le più basse.
Immaginando che i valori target originali siano stati assegnati "a mano", e sebbene la nostra sia una prima implementazione che può certamente essere migliorata, i risultati potrebbero andare nella direzione di quanto spiegato a lezione, ossia che WordNet è una risorsa importante che però cerca di adempiere a task molto complicati come l'identificazione automatica dei significati e della correlazione tra gli stessi, e che quindi raggiungere risultati simili a quelli umani sia piuttosto impegnativo.

Progetto 2: Word Sense Disambiguation

Panoramica del Progetto

Il progetto si concentra sull'implementazione e valutazione dell'algoritmo Lesk per il disambiguamento semantico delle parole (Word Sense Disambiguation, WSD). Il disambiguamento semantico è il processo di identificazione del significato corretto di una parola ambigua all'interno di un contesto specifico. L'algoritmo Lesk è una tecnica consolidata che si basa sull'analisi dell'overlap tra il contesto della parola ambigua e le definizioni fornite da WordNet. Per la valutazione delle prestazioni dell'algoritmo, è stato utilizzato il corpus SemCor, un corpus annotato semanticamente.

Scelte implementative

1 **Preprocessamento del testo:**

- **Rimozione della punteggiatura:** Utilizzo di espressioni regolari per eliminare la punteggiatura dal testo.
- **Conversione in minuscolo:** Tutti i caratteri sono stati convertiti in minuscolo per garantire l'uniformità.
- **Tokenizzazione e lemmatizzazione:** Impiego del `MWETokenizer` per riconoscere espressioni multi-parola e del `WordNetLemmatizer` per ridurre le parole alle loro forme base.
- **Rimozione delle stop words:** Le stop words sono state eliminate utilizzando il set di stop words di NLTK.

2 **Algoritmo Lesk:**

- L'algoritmo Lesk determina il significato corretto di una parola confrontando il contesto della frase con le definizioni e gli esempi forniti da WordNet per ciascun possibile significato della parola.
- Viene calcolato l'overlap (intersezione) tra il contesto della frase e le definizioni/esempi di ciascun significato per determinare il significato con il massimo overlap.

3 **Valutazione dell'algoritmo:**

- È stato selezionato un campione di 50 frasi annotate dal corpus SemCor.
- Per ciascuna frase, è stata scelta una parola casuale e il significato previsto dall'algoritmo è stato confrontato con il significato annotato nel corpus.
- Il processo di valutazione è stato ripetuto 10 volte per ottenere una misura più affidabile dell'accuratezza dell'algoritmo.

Osservazioni e conclusioni

Durante la fase di valutazione, in una delle varie esecuzioni, l'algoritmo ha mostrato le seguenti prestazioni in ciascuna delle 10 esecuzioni:

1. accuracy: 0.4090909090909091
2. accuracy: 0.43243243243243246
3. accuracy: 0.3695652173913043
4. accuracy: 0.43478260869565216
5. accuracy: 0.5238095238095238
6. accuracy: 0.45652173913043476
7. accuracy: 0.36585365853658536
8. accuracy: 0.5
9. accuracy: 0.38095238095238093
10. accuracy: 0.4666666666666667

La precisione media dell'algoritmo è stata calcolata come `0.433967513670589`, indicando che l'algoritmo è riuscito a prevedere correttamente il significato delle parole nel 43.4% dei casi. Questa precisione, non molto alta, fa capire come il WSD rimanga un problema tutt'ora aperto.

Progetto 3: Twitting like (a) Trump

Panoramica del Progetto

L'obiettivo di quest'ultimo progetto è sperimentare direttamente come si costruisce un language model basato sugli n-grams.

Nello specifico, data una raccolta di tweet di Donald Trump (il "Trump twitter archive") contenente circa 300 tweet, la richiesta è di costruire un modello basato su bigrammi e i trigrammi in grado di generare a sua volta delle frasi basate proprio sulle probabilità calcolate inizialmente, così da "twittare come Trump".

Scelte implementative
Abbiamo scelto di sviluppare il codice in un unico jupyter notebook che abbiamo chiamato "twitting_trump.ipynb", così da poter effettuare (e testare) ogni porzione di codice singolarmente, suddividendo le varie fasi in pulizia del dataset, creazione delle matrici di probabilità basate su bigrammi e trigrammi, e infine generazione dei tweet utilizzando gli stessi modelli ottenuti.

Pulizia dei tweet
Il primo aspetto di cui si occupa il codice, dopo aver aperto e letto il file "tweets.csv" contenuto nella cartella "trump_twitter_archive", è la lettura e pulizia dei singoli tweet per trattarli come una lista di parole.
Dal momento che la mole di dati a disposizione non è particolarmente grande, abbiamo deciso di eseguire una fase di pulizia sul singolo tweet. Tramite la libreria *re* si effettuano operazioni con le regular expressions che permettono di eliminare i link, avere tutte le parole in minuscolo ed eliminare la punteggiatura: in questo modo le parole "Most" e "most" saranno trattate allo stesso modo e non come due termini differenti, così come "most:" e "most".
Se il dataset fosse più ampio si potrebbe evitare almeno parte di questa fase di pulizia: ad esempio le parole maiuscole potrebbero rimanere tali, e si potrebbero anche utilizzare come "inizio frase" così da aumentare la probabilità di ottenere tweet verosimili.
Nel processo di pulizia dei tweet abbiamo anche inserito una variabile che somma la lunghezza in parole di tutti tweet, così da poter poi calcolare la `medium_length` per usarlo, eventualmente, come parametro per la generazione dei tweet.

Creazione dei modelli a bigrammi e trigrammi
In questa fase del progetto abbiamo usato la libreria *nltk*.
Quest'ultima permette di ottenere delle strutture a dizionario in cui, per ogni parola (o coppia di parole nei trigrammi), corrisponde una serie di parole che nel corpus la seguono, con un valore corrispondente alla frequenza e quindi utilizzabile come probabilità per quello specifico bigramma(o trigramma).
Otteniamo dunque le due strutture `model_bigrams` e `model_trigrams`.

Generazione di tweet basati sui modelli ottenuti

Abbiamo scritto due funzioni per generare i tweet:
- `bi_tweet_generator(length, temperature)`
- `tri_tweet_generator(length, temperature)`

Il generatore di tweet basato su bigrammi sceglie casualmente una parola presa dal corpus ottenuto con i tweet di Trump. Dopodiché inizia un ciclo for: a ogni iterazione la parola appena scelta diventa la `current_word` e viene usata per scegliere la `next_word`.

```
for i in range(length):
    next_words = list(model_bigrams[current_word].keys())
    if next_words:
        probabilities = list(model_bigrams[current_word].values())
        probabilities = apply_temperature(probabilities, temperature)
        new_word = random.choices(next_words, weights=probabilities, k=1)[0]
    else:
        break   # Non ci sono parole disponibili, esci dal ciclo
```

Questo ciclo for viene eseguito fino a che non si raggiunge la lunghezza massima impostata per tweet.
Il generatore basato su trigrammi invece esegue una volta il bi_tweet_generator con length impostata a 1 così che crei un tweet di due parole, che vengono usate appunto per generare trigrammi fino al limite massimo. L'unica differenza rilevante è il fatto che si accede appunto alla struttura `model_trigrams` piuttosto che a `model_bigrams`.

Risultati e raffinamento

Dopo aver ottenuto i modelli e generato i primi tweet, abbiamo notato alcune possibili migliorie. Inizialmente il nostro algoritmo sceglieva semplicemente la parola più probabile tra quelle date dai bigrammi, ma questo rendeva il generatore troppo deterministico e spesso causava dei loop: infatti 'looser' e 'loosers' sono termini molto utilizzati dall'ex Presidente degli Stati Uniti nei suoi tweets, il che portava a loop del tipo *'a loser who is a loser who is…'* al primo articolo della frase.
Abbiamo dunque implementato la possibilità di stabilire la temperatura dei generatori:
la "temperatura" è un parametro che può essere utilizzato per controllare la casualità della generazione di testo. Temperature più basse rendono il modello più deterministico (selezionando più spesso le parole con probabilità più alte), mentre temperature più alte rendono il modello più casuale (selezionando parole con probabilità più basse più frequentemente).

```
def apply_temperature(probabilities, temperature):
    probabilities = np.array(probabilities)
    probabilities = probabilities ** (1.0 / temperature)
    probabilities = probabilities / np.sum(probabilities)
    return probabilities
```

Nello specifico, effettuando alcune ricerche abbiamo trovato questa funzione che eleva ciascuna probabilità alla potenza di 1/T, dove T è la temperatura. Quando la temperatura è bassa le probabilità più alte aumentano ulteriormente rispetto a quelle più basse. Quando la temperatura è alta, le differenze tra le probabilità si appiattiscono.

Avevamo aggiunto anche questa linea di codice per eliminare i tag di utenti:

```
    riga[1] = re.sub(r"@\S+", "", riga[1])
```
ma, facendo un controllo, questa eliminava per qualche motivo più parole di quante fosse necessario, quindi abbiamo deciso di lasciare semplicemente i tag con la @ inclusa.
Il ragionamento dietro a questa scelta è che comunque i tag di utenti non avranno probabilità particolarmente alta di essere 'estratti' durante la generazione di tweet, e più verosimilmente potrebbero essere scelti come parola iniziale del tweet(dove la scelta è randomica), il che non è un problema dal momento che solitamente i tweet iniziano in questo modo quando sono risposte ad altri utenti.
Un altro termine che abbiamo eliminato con le espressioni regolari è `&` che sostituisce nel file csv i caratteri non codificati. Eliminando solo la punteggiatura ottenevamo frasi in cui compariva il termine "amp" sporcando inutilmente la frase generata, così abbiamo deciso di eliminare direttamente il termine speciale nel dataset originale.

Abbiamo effettuato alcune prove modificando la temperatura, e la nostra impressione è che i risultati migliori si ottengano con temperature medie. Ad esempio, con temperature estremamente basse (es. 0.1) il sistema diventa quasi deterministico, andando il loop sulle parole "looser" e "loosers". Con una temperatura di 0.3 o 0.4 la situazione non migliora molto per i bigrammi ma lo fa per i trigrammi, riuscendo a non rimanere 'intrappolato' nel loop appena incontra la parola "looser". Per ottenere risultati migliori con i bigrammi bisogna spostarsi sullo 0.6.
Per temperature maggiori l'impressione è che il risultato diventi più variegato e quindi non migliore o peggiore in senso assoluto: le frasi possono certo essere più realistiche ed evitare loop, ma anche perdere totalmente di significato.

Un esempio di possibile miglioramento potrebbe essere fatto in fase di pulizia, ad esempio salvando i segni di punteggiatura come singoli termini, così da provare a ottenere tweet che li contengono. Un'altra miglioria potrebbe essere anche quella di tenere traccia delle parole di "inizio tweet" più frequenti e applicare una temperatura anche a quelle, mentre al momento la scelta è randomica e distribuita tra tutti i termini.

Conclusioni

Il notebook fornisce un esempio interessante e divertente di come si possa analizzare e generare testo basandosi su modelli statistici semplici come bigrammi e trigrammi. Nonostante la semplicità dei modelli utilizzati, i risultati ottenuti sono un buon punto di partenza per ulteriori approfondimenti e miglioramenti.