Roberto Gjokaj

Alberto Alpe

Università degli studi di Torino

Corso di Laurea Magistrale di Informatica

Tecnologie del Linguaggio Naturale

Parte 1 – Prof. Alessandro Mazzei

NER Tagging with HMM

Panoramica del Progetto

La richiesta del progetto è quella di sviluppare un modello basato sugli Hidden Markov Models e sull'algoritmo di Viterbi in grado di assegnare correttamente i cosiddetti NER tags alle parole di una frase.
Il NER (Named Entity Recognition) tagging si occupa di riconoscere particolari entità come persone, organizzazioni o luoghi geografici rispetto alle parole di uso comune, che invece vengono solitamente taggate tramite il POS tagging. L'importanza e la difficoltà di questa fase dell'analisi di una frase è data dal fatto che queste entità possono essere composte da più parole.
L'approccio seguito è dunque quello del BIO tagging, secondo il quale si vogliono distinguere i tag che non riguardano entità NER (quindi tag di tipo O = "Outside") da quelli che invece le riguardano, discriminando ulteriormente tra il primo termine di un'entità (B = "Begin") e quelli al suo interno (I = "Inside").

Viene richiesto di lavorare su tre dataset "wikineural", di cui uno in inglese, uno in spagnolo e uno in italiano. Ciascuno di questi dataset è suddiviso in tre file: "train.conllu", "test.conllu" e "val.conllu", i quali sono già 'puliti', ovvero contengono diverse frasi già suddivise parola per parola (compresa la punteggiatura), ognuna con il suo NER tag corretto. L'obiettivo del progetto è sviluppare le parti di learning e di decoding necessarie a ottenere un modello funzionante, oltre alla valutazione dello stesso e al confronto rispetto a due baselines.

Scelte implementative

Abbiamo deciso di organizzare il progetto nel seguente modo:

I tre dataset wikineural sono stati inseriti in tre cartelle con i rispettivi nomi.

I file contenenti il codice sono stati inseriti tutti nella cartella principale, e sono stati nominati:

- *PoS-Probabilities.py*

-viterbi.py (+ Viterbi_no_log.py)

-NER-tagging.ipynb

-baseline.py

La prima parte di codice sviluppata è stata inserita nel file *PoS_probabilities.py*, che si occupa della parte di learning. Dopodiché ci siamo occupati del file *viterbi.py*, che sviluppa l'omonimo algoritmo affrontato a lezione, e poi il file *baseline.py* che invece contiene la baseline più semplice. Entrambi questi file verranno importati come moduli nel file principale, *NER-tagging.ipynb*, che abbiamo deciso di sviluppare come notebook jupyter per diversi motivi che illustreremo nella sezione dedicata.

Nei paragrafi seguenti andremo a discutere le scelte implementative per ogni file.

PoS-probabilities.py

Come affrontato a lezione, l'aspetto legato al learning negli Hidden Markov Models è piuttosto semplice e riguarda il calcolo di due probabilità: le probabilità di emissione e di transizione. Nello specifico, la probabilità di transizione riguarda la probabilità che in una frase, dato un certo tag (es. O), questo sia seguito da un altro tag (es. B-PER), mentre la probabilità di emissione si riferisce alla probabilità che un certo tag(es. O) riguardi una specifica parola(es. "the"). Queste probabilità si calcolano in due fasi, ossia contando prima le occorrenze ci transizione ed emissione, e poi calcolandone le rispettive frequenze.

Questa è la parte di codice da cui siamo partiti, e abbiamo dunque deciso di creare due array ('tags' e 'words') e due tabelle (`transition_P` ed `emission_P`). I due array servono principalmente per contenere tutti i tag e tutte le parole presenti nel corpus; infatti vengono aggiornati ogni volta che si trova una parola o un tag nuovo. L'indice di una parola e un tag ci serviranno quindi per accedere alla cella corretta delle matrici.
Ipotizzando che 'i' sia la riga che stiamo leggendo nel file 'train.conllu', la matrice di transizione deve fare riferimento al tag_{i-1} sulle righe e al tag_i sulle colonne, dunque sarà una matrice quadrata con tante righe e colonne quanto è lungo l'array 'tags'.
Un approccio simile è usato per la matrice di emissione, la quale contiene $i\ tag_i$ sulle righe e le $word_i$ sulle colonne.

`Tags` è inizializzato come array contenente i valori ['Start', 'End'], ossia i due tag di inizio e fine frase che ci serviranno nell'algoritmo di Viterbi. `Words` inizialmente è un array vuoto in quanto non contiene ancora le parole del corpus. Di conseguenza `transition_P` è inizialmente composto da due righe e due colonne contenenti i valori 0, mentre `emission_P` è composto solo da due righe vuote.

```
emission_P = [[], []]     #matrice iniziale composta da due righe vuote
transition_P = [[0, 0], [0, 0]]  #matrice iniziale 2x2
tags = ['START', 'END']
words = []
```

Leggendo il file riga per riga con un ciclo for è così possibile aggiornare dinamicamente gli array e le matrici con gli elementi nuovi del corpus e al contempo aggiornare i valori delle occorrenze nelle specifiche celle delle matrici. Si utilizzano anche alcune variabili come 'tag' 'word' 'tag_prec' 'tpi' che tengono traccia del valore del tag_{i-1} e degli indici del tag_i della $word_i$.

Una volta letto tutto il file 'train.conllu' avremo quindi ottenuto le matrici delle occorrenze, che devono ancora essere aggiornate per diventare effettivamente matrici di probabilità.

Un aspetto interessante di queste strutture è dato dal fatto che, sommando i valori dentro ogni riga della matrice di emissione, avremo le occorrenze totali per un certo tag specifico.
Questo valore è utilizzabile per il calcolo delle due probabilità:

$\mathbf{P(w_i|t_i)}$ $P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}$ $\mathbf{P(t_i|t_{i-1})}$ $P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$

Come si può notare, le righe di 'emission_P' fanno riferimento al valore t_{i-1}, mentre le righe di 'transition_P' si riferiscono al valore t_{i-1}.

Per questo calcolo abbiamo deciso di utilizzare la libreria *numpy*. In questo modo, con poche righe di codice si possono sommare tutti i valori in una certa riga della matrice e poi si possono dividere i valori (diversi da zero) presenti in ogni cella della riga della suddetta riga per il valore ottenuto(se diverso da zero), così da ottenere delle matrici con le probabilità di transizione ed emissione da usare per i calcoli dell'algoritmo di Viterbi. Di seguito l'esempio per la matrice di transizione:

```
# Calcolo delle probabilità di transizione
row_sums = np.sum(transition_P, axis=1, keepdims=True)
non_zero_rows = row_sums.squeeze() != 0
transition_P[non_zero_rows] = transition_P[non_zero_rows] / row_sums[non_zero_rows]
```

In questa fase abbiamo anche aggiunto il calcolo delle statistiche per le parole che compaiono una sola volta nel file:

```
# Calcolo statistiche per development set:
# se la somma dei valori in una colonna di emission_P è 1, allora quella parola compare una volta
# gli indici di 'tags' e 'array_dev_set' coincidono
# per ogni tag calcoliamo quante (singole) parole corrispondono e poi dividiamo per il totale
t = len(tags)
array_dev_set = np.zeros(t)
for col in range(emission_P.shape[1]):  # il numero di colonne in matrice
        column = emission_P[:, col]
        if np.sum(column) == 1:
            row_index = np.where(column == 1)[0][0] # tag corrispondente
            array_dev_set[row_index] += 1
tot = np.sum(array_dev_set)
for i in range(0, len(array_dev_set)):
   array_dev_set[i] /= tot
array_dev_set = np.array(array_dev_set, dtype=float)
```

Dopo aver calcolato le probabilità per le matrici di transizione ed emissione, aggiungiamo un'ultima colonna alla matrice. Questa rappresenterà proprio le parole che compaiono una sola volta nel training set:

```
array_dev_set = array_dev_set.reshape(-1, 1)
# np.column_stack aggiunge l'array come ultima colonna
emission_P = np.column_stack((emission_P, array_dev_set))
```

Le ultime righe di questo file si occupano semplicemente di salvare i due array e le due matrici risultanti in un csv chiamato 'probabilities.csv', dove ogni sezione ha il nome della struttura corrispondente, così da poterle rileggere e recuperare facilmente.

Grazie al salvataggio dei valori in file csv appositi, questo file è stato eseguito in totale tre volte (una per ogni file di training) e non dev'essere più eseguito.

viterbi.py

Il file viterbi.py contiene l'implementazione dell'algoritmo di Viterbi, che è utilizzato per trovare la sequenza di NER tag più probabile per una data sequenza di parole. L'algoritmo di Viterbi è un algoritmo di programmazione dinamica che utilizza le probabilità di transizione e di emissione calcolate durante la fase di training per effettuare l'assegnazione dei tag.

L'algoritmo di Viterbi è implementato come segue:

1. **Initialization Step**: Creiamo una matrice viterbi per memorizzare i punteggi delle probabilità più alte per ogni stato (tag) in ogni passo temporale (parola). Creiamo anche una matrice backpointer per tenere traccia dei percorsi che portano ai punteggi più alti.

```
# Algoritmo di Viterbi
def viterbi(sentence, emission_P, transition_P, tags, words):
    log_emission_P = np.log(emission_P)
    log_transition_P = np.log(transition_P)
    n = len(sentence)
    m = len(tags)
    viterbi_matrix = np.full((m, n), -np.inf)   # viterbi_matrix = np.zeros(m, n)
    backpointer = np.zeros((m, n), dtype=int)
    Oindex = tags.index('O')  #queste variabili serviranno per lo smoothing
    MISCindex = tags.index('B-MISC')

    # Initialization step
    for tag in range(2, m):  # Ignoro START e END nei calcoli
        if sentence[0] in words:
            viterbi_matrix[tag, 0] = log_transition_P[0, tag] + log_emission_P[tag, words.index(sentence[0])]
        else:       #se la parola è sconosciuta
        #vers. 3: uniforme-> P(unk|Ti)= 1/len(tags)
            viterbi_matrix[tag, 0] = log_transition_P[0, tag] * 1/m
```

2. **Recursion Step**: Nella fase di ricorsione, per ogni parola nella frase (a partire dalla seconda), calcoliamo il punteggio più alto per ogni stato considerando tutte le possibili transizioni dagli stati precedenti. Usiamo i logaritmi delle probabilità per evitare problemi di underflow numerico.

```
# Recursion step
for word in range(1, n):
    if sentence[word] in words:
        for tag in range(2, m):
            max_tr_prob = viterbi_matrix[:, word-1] + log_transition_P[:, tag]
            backpointer[tag, word] = np.argmax(max_tr_prob)
            viterbi_matrix[tag, word] = np.max(max_tr_prob) + log_emission_P[tag, words.index(sentence[word])]
    else:       #se la parola è sconosciuta
    #vers. 3: uniforme-> P(unk|Ti)= 1/len(tags)
        for tag in range(2, m):
            max_tr_prob = viterbi_matrix[:, word-1] + log_transition_P[:, tag]
            backpointer[tag, word] = np.argmax(max_tr_prob)
            viterbi_matrix[tag, word] = np.max(max_tr_prob) * 1/m
```

3. **Termination Step**: Una volta popolata la matrice viterbi, utilizziamo la matrice backpointer per ricostruire la sequenza di tag più probabile.

```
    # Termination step
    max_prob = viterbi_matrix[:, n-1] + log_transition_P[:, 1]
    best_path_pointer = np.argmax(max_prob)
    best_path = [best_path_pointer]

    for word in range(n-1, 0, -1):
        best_path_pointer = backpointer[best_path_pointer, word]
        best_path.insert(0, best_path_pointer)

    best_tag_sequence = [tags[i] for i in best_path]

    return best_tag_sequence
```

baseline.py

Questo file si occupa di implementare una baseline che segue due regole molto semplici:

- se la parola è sconosciuta(ossia non fa parte dell'array words) il tag assegnato è 'MISC';

- altrimenti viene assegnato il tag più frequente per quella parola, dunque quello con la probabilità di emissione più alta.

```
tag_sequence = []

# baseline più semplice, a ogni parola assegna il tag più frequente (o MISC se sconosciuta)
def easy_baseline(sentence, emission_P, tags, words):
    tag_sequence = []
    for word in sentence:
        if word in words:
            word_index = words.index(word)
            best_tag = np.argmax(emission_P[:, word_index])
            tag_sequence.append(tags[best_tag])
        else:
            tag_sequence.append('MISC')

    return tag_sequence
```

Abbiamo implementato questa funzione in un modulo a parte perché la nostra intenzione iniziale era mantenere entrambe le baselines nello stesso modulo; ci siamo accorti successivamente che la baseline con i MEMM non può essere richiamata come una libreria ma va, invece, mantenuta ed eseguita in un file separato. La nostra decisione finale è stata di mantenere la *easy_baseline* comunque in un modulo a parte (nonostante le poche righe di codice), così da mantenere il notebook principale il più "pulito" possibile.

NER-tagging.ipynb

Si tratta del file principale e a differenza degli altri abbiamo deciso di svilupparlo all'interno di un notebook. Il motivo principale dietro a questa scelta è dato dall'utilizzo che si farà del notebook: dovendo eseguire più volte il codice per fare dei test in cui cambia la versione di Viterbi utilizzata si potrà leggere una sola volta il file .csv contenente le probabilità ed eseguire più volte un singolo

blocco di codice(ad es. quello del learning) cambiando alcuni parametri. Inoltre, i notebook consentono di eseguire e testare singoli blocchi di codice, oppure di eseguire tutto il codice escludendo alcuni blocchi da noi specificati(marcando il linguaggio del blocco come "raw"). L'ultima ragione è prettamente estetica, in quanto i notebook consentono di inserire anche blocchi in markdown, così da scrivere titoli e sottotitoli e separare anche a livello visivo le varie sezioni del file.

- Come prima cosa vengono importati i vari moduli che useremo, compresi i moduli *baseline*, *viterbi* e *viterbi_no_log*. Poi vengono inizializzate le variabili che dovranno "ospitare" le strutture dati create precedentemente nel file *PoS-Probabilitie.py*.
- La funzione `readProb` viene definita all'inizio e servirà a leggere il file 'probabilities' dentro a una cartella wikineural specifica, estraendone i valori e inserendoli nelle variabili sopra definite.
- Seguono tre blocchi di codice (da eseguire uno per volta) che richiamano proprio la funzione appena citata, indirizzandola alla cartella *wikineural* giusta, dopodiché aprono anche il corrispettivo file 'test.conllu'.
  Nota: l'ultima riga di questi blocchi contiene (commentata) anche l'opzione [:1000], che esegue il codice solo sulle prime mille righe del file di test, nel caso si volessero fare test veloci per verificare la correttezza del modello.
- Una volta estratti gli array e le matrici si passa alla fase di decoding, nella quale si legge riga per riga il file di test e si raccolgono le parole e i tag corrispondenti. Per ogni frase (riga vuota raggiunta) si esegue il decoding sulla serie di parole ottenuta (`sentence`):

```
current_sentence = []
current_tags = []
true_tags = []
predicted_tags = []

# un ciclo for generale legge tutte le frasi del file test
for riga in righe:
    riga = riga.strip()
    if riga:
        riga = riga.split()
        current_sentence.append(riga[1])
        current_tags.append(riga[2])
    else:     # riga vuota => end of sentence
        #final_sequence = viterbi_no_log.viterbi_no_log(current_sentence, emission_P, transition_P, tags, words)
        final_sequence = viterbi.viterbi(current_sentence, emission_P, transition_P, tags, words)
        #final_sequence = baseline.easy_baseline(current_sentence, emission_P, tags, words)
```

- Infine, l'ultimo blocco si occupa della valutazione. Si utilizzano le variabili `true_tags` e `prediceted_tags`, che sono liste di liste (nelle quali ogni lista è una serie di tag corrispondenti a una frase). Per ogni sotto-lista si aggiorna il valore `total` che conta il numero totali di tags, poi si confrontano quindi i tag predetti con quelli reali così da poter calcolare la Accuracy alla fine. Precision e Recall sono invece calcolate sulla base delle entità correttamente predette (o meno). Abbiamo dunque deciso di controllare se un tag predetto inizia con 'B': ciò significa che è l'inizio di un'entità predetta e quindi salveremo i tag successivi fino al primo O in un array. Una volta arrivati a 'O' avremo dunque terminato l'entità predetta e potremo confrontarla con la serie di tag reali corrispondente. Per trovare i falsi negativi invece controlliamo se un tag reale inizia con 'B' e quello predetto non corrisponde: in tal caso siamo di fronte a un'entità non predetta.

```
    if predicted_tag.startswith("B-") or predicted_tag.startswith("I-"):
        predicted_entity.append(predicted_tag)    # crea due sequenze di entità, predetta e vera
        true_entity.append(true_tag)
    if predicted_tag == "O":
        if predicted_entity:    # se abbiamo appena superato un'entità controlliamo se è corretta
            if predicted_entity == true_entity:
                true_positives += 1
            else:
                false_positives += 1
            predicted_entity = []
            true_entity = []
    if true_tag.startswith("B-") and true_tag != predicted_tag:  # se c'è entità non riconosciuta
```

extra: cartella 'memm'

Una volta implementato Viterbi e la baseline, abbiamo provato ad implementare il NER-tagging con Maximum Entropy Markov Model dal repository GitHub indicato.

Come indicato nel file, abbiamo incluso il codice in una sottocartella apposita con i due file di train e di test rinominati come 'wsj.pos.train' e 'wsj.pos.dev'. Lanciando da terminale il comando per eseguire il codice abbiamo ricevuto il seguente errore:

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x8d in position 3260:
character maps to <undefined>
```

Dopo alcuni controlli abbiamo capito che si tratta di un errore di codifica per certi caratteri speciali presenti nel file (come il carattere 'Č'), e aggiunto l'encoding "utf-8" al codice in fase di lettura dei file(riga 175) risolvendo questo probelma.
Inoltre abbiamo sostituito `dtype=np.32` con `dtype=object` su suggerimento di alcuni compagni di corso.
Purtroppo, nonostante queste modifiche, non siamo comunque riusciti a eseguire il codice, quindi non abbiamo ottenuto risultati da confrontare con i nostri.

Risultati e considerazioni

Nelle tabelle sottostanti(una per ogni lingua IT, ES, EN) andiamo a mostrare i risultati ottenuti tra i vari modelli:

- wikineural_es

Modello	Accuracy	Precision	Recall
baseline	0.8326	0.2409	0.7493
HMM Viterbi no Log	0.9291	0.9172	0.8642
HMM Viterbi Log	**0.9461**	**0.9226**	**0.8648**

- wikineural_en

Modello	Accuracy	Precision	Recall
baseline	0.8530	0.2655	0.5547
HMM Viterbi no Log	0.9488	0.7908	0.7121
HMM Viterbi Log	**0.9497**	**0.7908**	**0.7126**

- wikineural_it

Modello	Accuracy	Precision	Recall
baseline	0.8326	0.2540	0.6696
HMM Viterbi no Log	0.9460	0.8889	0.8002
HMM Viterbi Log	**0.9600**	**0.8912**	**0.8117**

Alcune considerazioni

Come si evince dai valori ottenuti, le varie versioni di Viterbi migliorano nettamente le prestazioni rispetto alla baseline più semplice (ne sono prova i più di 10 punti percentuali di miglioramento nell'Accuracy) e portano il nostro modello a livelli di accuratezza accettabili (come visto a lezione, il 94% sarebbe il livello minimo richiesto).
L'aspetto (non evidenziato dalle immagini riportate sopra) a prima vista deludente dal punto di vista dei confronti è il fatto che solo l'ultimo tipo di smoothing (uniforme) sembra migliorare effettivamente le prestazioni del nostro modello: ciò potrebbe indicare che le parole sconosciute sono uniformemente distribuite tra tutti i tipi di tag. Se pensiamo a un task di NER-tagging (e quindi fortemente orientato all'etichettamento di entità con nomi specifici), è prevedibile che nuove parole non facenti parte del corpus siano effettivamente nuove entità, e in quest'ottica sarebbe sensato "provare tutte le vie" ossia ammettere che queste appartengano a uno qualsiasi dei vari tag piuttosto che assegnare direttamente il tag 'O' o 'B-MISC'. Quanto alle statistiche sul development set, abbiamo notato che queste migliorano le prestazioni del Viterbi normale, ma non di quello con logaritmi
Tutti i risultati dei test si trovano nella cartella 'screenshots'

Conclusioni

Questo progetto si è rivelato utile per capire meglio come funziona il POS-tagging in generale e come si lavora con le probabilità di emissione e transizione (basate sui bigrammi).
Effettuare la parte di learning ci ha messo di fronte ad alcuni dubbi come la scelta delle strutture da utilizzare e dei modi più efficienti per effettuare i calcoli, mentre l'implementazione dell'algoritmo di Viterbi e le sue varianti per il decoding ci hanno permesso anche di capire meglio la teoria e di riflettere sui risultati. Infine nell'ottica dell'interezza del corso di Tecnologie del Linguaggio Naturale possiamo affermare che, essendo questo esercizio piuttosto completo nel suo sviluppo (accesso e lettura dei file, utilizzo di strutture adeguate, elaborazione dei dati, analisi dei risultati e raffinamento dei modelli) la risoluzione di alcuni errori ci è tornata utile nell'affrontare più velocemente i problemi posti dagli altri laboratori del corso.