**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*

*-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 *Baselines.py* che invece contiene le due baseline richieste (una più semplice e l’altra basata sui MEMM). Entrambi questi file verranno importati come moduli nel file principale, *NER-tagging.ipynb*, che abbiamo deciso di sviluppare come notebook jupyter.

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.

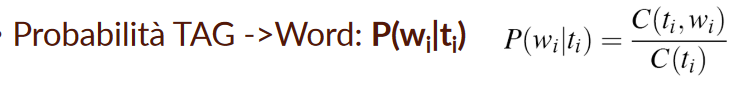
Questa è la parte di codice da cui siamo partiti, e abbiamo dunque deciso di creare due array (‘tags’ e ‘words’) e due tabelle (‘transition\_P’ ed ‘emission\_P’). I due array servono principalmente per contenere tutti i tag e tutte le parole presenti nel corpus; infatti vengono aggiornati ogni volta 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 tagi-1 sulle righe e al tagi 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 tagi sulle righe e le wordi sulle colonne.

‘Tags’ è inizializzato a [‘Start’, ‘End’], i due tag di inizio e fine frase che ci serviranno nell’algoritmo di Viterbi. ‘Words’ all’inizio è un array vuoto in quanto non contiene ancora 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.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

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 varie celle delle matrici. Si utilizzano anche alcune variabili come ‘tag’ ‘word’ ‘tag\_prec’ ‘tpi’ che tengono traccia del valore del tagi-1 e degli indici del tagi della wordi.

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 di un certo tag specifico.  
Questo valore è utilizzabile per il calcolo delle due probabilità:   
Immagine che contiene Carattere, testo, linea, calligrafia

Descrizione generata automaticamente

Come si può notare, le righe di ‘emission\_P’ fanno riferimento al valore ti-1, mentre le righe di ‘transition\_P’ si riferiscono al valore ti-1.

Una volta effettuato il calcolo delle occorrenze totali di ogni tag…

**viterbi.py**

**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.

Abbiamo deciso di implementare 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, così da mantenere il notebook principale il più “pulito” possibile.

**NER-tagging.ipynb**

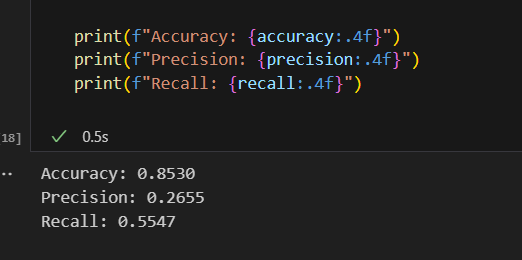
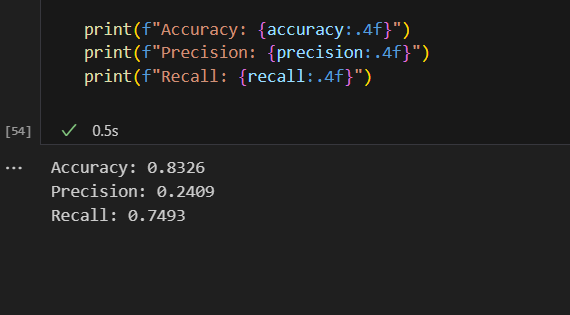
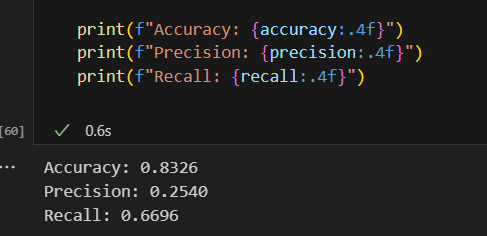
Si tratta del file principale, e per questo motivo abbiamo deciso di svilupparlo all’interno di un notebook per alcune motivazioni che ora illustreremo.

**Risultati e considerazioni**

**easy\_baseline**

Con la baseline più semplice i risultati sono i seguenti:

baseline en baseline es baselline it

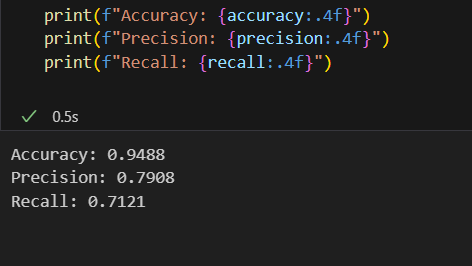
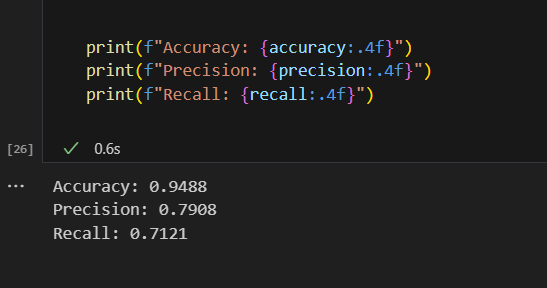
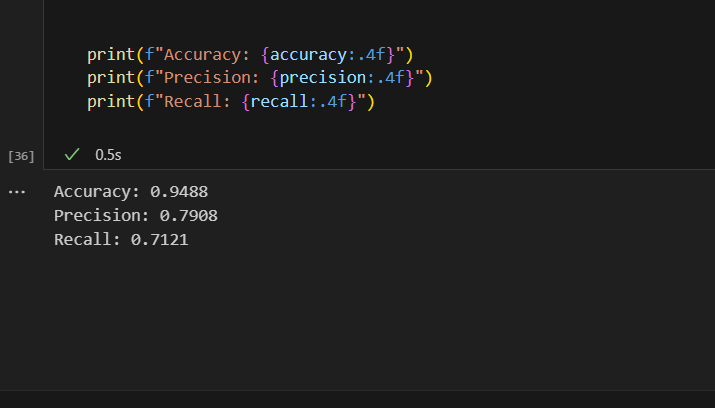
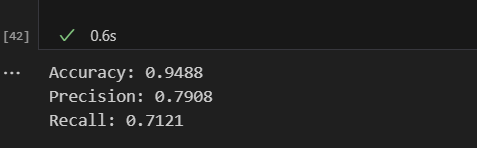
  

**Viterbi**

Per viterbi faremo invece confronti basati sulla stessa lingua, con i diversi tipo di smoothing.

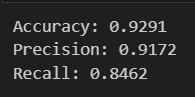
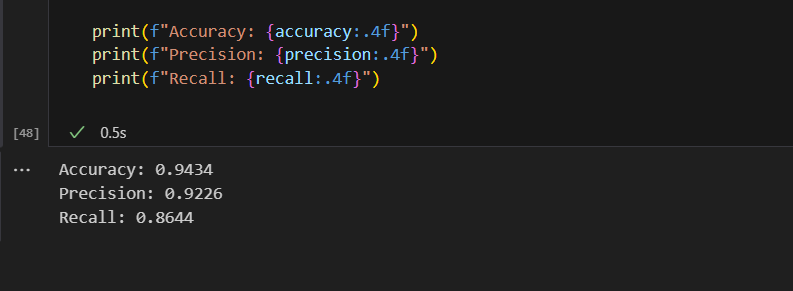
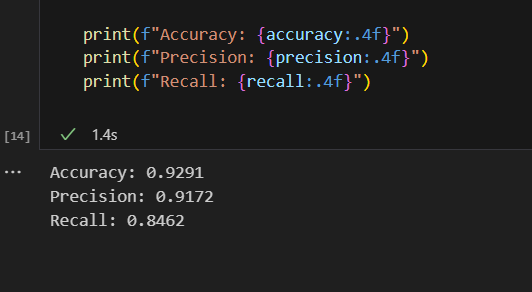
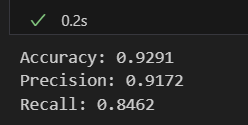
* **wikineural\_en**

no smoothing P(unk|O) = 1 P(unk|O) = P(unk|B-MISC) = 0.5 P(unk|tagi) = 1/len(tags)

**   **

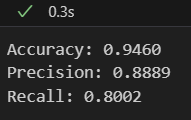
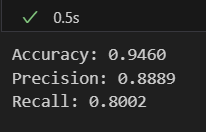
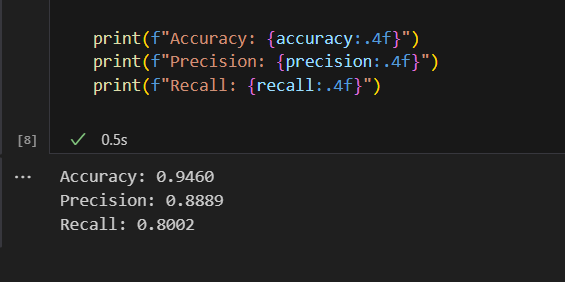
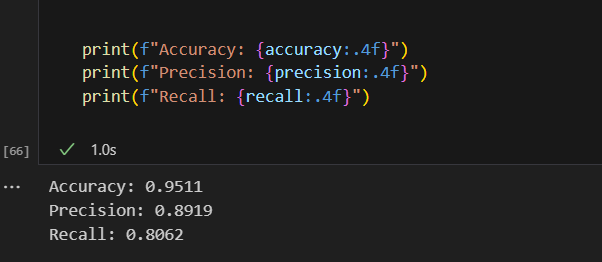
* **wikineural\_es**

no smoothing P(unk|O) = 1 P(unk|O) = P(unk|B-MISC) = 0.5 P(unk|tagi) = 1/len(tags)

**** 

* **wikineural\_it**

no smoothing P(unk|O) = 1 P(unk|O) = P(unk|B-MISC) = 0.5 P(unk|tagi) = 1/len(tags)

**   **

**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 10 punti percentuali 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 a prima vista deludente dal punto di vista dei confronti è il fatto che solo l’ultimo tipo di smoothing sembra migliorare effettivamente le prestazioni del nostro notebook: ciò comunque potrebbe indicare che le parole sconosciute sono equamente 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.

AGGIUNGERE PARTE CON LOG

**Conclusioni**