UNIVERSITÀ
DEGLI STUDI
DI TORINO

PoS Tagger per lingue morte

a cura di

Pier Felice Balestrucci
Miriam Fasciana

Relazione del progetto di TECNOLOGIE DEL LINGUAGGIO NATURALE

Università degli studi di Torino 2020/2021

Indice

1. Introduzione

L'esercizio proposto per la prima parte d'esame è l'implementazione di un PoS tagger statistico basato su HMM per il greco antico e per il latino. Nel dettaglio, si è implementata la fase di learning e decoding con l'algoritmo di Viterbi e si è valutato il sistema applicando diverse strategie di smoothing. Infine, i risultati sono stati confrontati rispetto ad una baseline.
Il progetto è consultabile all'indirizzo Github: TLN2021/PosTagger.

2. Fase di learning

Come prima cosa è stata implementata la fase di learning in cui si sono calcolate le probabilità di transizione e quelle di emissione.
Per svolgere questa e le successive fasi è stato necessario ricercare i possibili PoS tag presenti nei Treebank del greco e del latino del progetto UD.
Di seguito è mostrato il risultato della ricerca.

POS - latino	POS - greco
PUNCT	VERB
ADP	ADV
PROPN	ADJ
NOUN	NOUN
VERB	PUNCT
DET	CCONJ
CCONJ	ADP
PRON	DET
ADJ	PRON
NUM	SCONJ
AUX	INTJ
SCONJ	NUM
ADV	X
PART	PART
X	Start
Start	End
End	

Fig. 1. Tag per il latino e il greco

Successivamente sono state calcolate le probabilità di transizione e di emissione sulla base della modellazione formale del problema del PoS tagging.
Più in particolare, le probabilità di transizione sono state calcolate dividendo la frequenza all'interno del corpus del tag t_i preceduto dal tag t_{i-1}, per il numero di volte in cui compare il tag t_{i-1}.

$$P(t_1^n) \approx \prod_{n=1}^{n} P(t_i|t_{1-1})$$

Queste sono state memorizzate in una *matrice* di dimensione $pos * pos$ in cui ogni cella indica il numero di volte in cui il tag in una determinata colonna precede quello nella riga.

Le probabilità di emissione sono state calcolate dividendo il numero di volte con cui è assegnato un certo tag al termine per il numero di volte in cui quello stesso tag compare in

tutto il corpus.

$$P(w_1^n|t_1^n) \approx \prod_{i=1} P(w_i|t_i)$$

In questo caso si è optato per un *dizionario* per memorizzare le probabilità dove le chiavi sono le parole del corpus e i valori di queste sono le probabilità per cui una parola essere etichettata con un tag.

3. Fase di decoding

Nella fase di decoding si è implementato l'algoritmo di Viterbi. Si precisa che, affinché i dati non fossero affetti da errori di approssimazione essendo le probabilità molto piccole, i calcoli sono stati effettuati convertendoli in forma logaritmica.
L'algoritmo è suddiviso in quattro parti:

1. **fase di inizializzazione**: nella matrice di Viterbi viene inserita la probabilità che un certo stato sia il primo PoS della sequenza
2. **fase ricorsiva**: per tutte le parole dalla seconda all'ultima viene calcolato il valore di viterbi utilizzando la formula $v_t(j) = max_{i=1,..,n} v_t(j) * a_{ij} * b_j(o_t)$ dove:
   - $v_t(j)$ = valore di viterbi per la parola t ed il pos j
   - a_{ij} = probabilità di transizione dallo stato i allo stato j
   - $b_j(o_t)$ = probabilità di emissione per la parola t ed il pos j
3. **fase di terminazione**: si calcola la probabilità per cui un certo tag sia quello finale
4. **ultima fase**: utilizza il vettore dei backpointer per risalire al giusto percorso di stati che porta dallo stato finale a quello iniziale della matrice di Viterbi.

```python
# codifica dello pseudocodice dell'algoritmo di Viterbi
def viterbiAlgorithm(sentence, pos, transitionProbabilityMatrix, emissionProbabilityDictionary, smoothingVector):
    sentenceList = utils.tokenizeSentence(sentence)
    # convertiamo i valori delle strutture create in precedenza coi log
    logTPM = utils.matrixToLogMatrix(transitionProbabilityMatrix)
    logEPD = utils.dictToLogDict(emissionProbabilityDictionary)
    logSV = utils.matrixToLogMatrix(smoothingVector)

    # fase di inizializzazione
    viterbi = np.ones((len(pos), len(sentenceList)))
    backpointer = np.zeros((len(pos), len(sentenceList)))
    # considera il pos 'start'
    for index,p in enumerate(pos):
        if sentenceList[0].lower() in emissionProbabilityDictionary.keys():
            viterbi[index, 0] = logTPM[len(pos)-2, index] + logEPD[sentenceList[0].lower()][index]
        else:
            viterbi[index, 0] = logTPM[len(pos)-2, index] + logSV[index]

    # fase ricorsiva
    for t,word in enumerate(sentenceList):
        for s,p in enumerate(pos):
            if word.lower() in emissionProbabilityDictionary.keys():
                viterbi[s, t] = np.max(viterbi[:, t-1] + logTPM[:, s] + logEPD[sentenceList[t].lower()][s])
            else:
                viterbi[s, t] = np.max(viterbi[:, t - 1] + logTPM[:, s] + logSV[s])

            backpointer[s, t] = np.argmax(viterbi[:, t - 1] + logTPM[:, s])

    # fase di terminazione
    # considera il pos 'end'
    viterbi[len(pos)-1, len(sentenceList)-1] = np.max(viterbi[:, len(sentenceList)-1] + logTPM[:, len(pos)-1] )
    backpointer[len(pos)-1, len(sentenceList)-1] = np.argmax(viterbi[:,  len(sentenceList)-1] + logTPM[:, len(pos)-1])

    # calcolo del percorso migliore usando il backpointer
    bestPath = np.zeros(len(sentenceList))
    bestPath[len(sentenceList) - 1] =  viterbi[:, -1].argmax() # last state
    for t in range(len(sentenceList)-1, 0, -1): # states of (last-1)th to 0th time step
        bestPath[t-1] = backpointer[int(bestPath[t]),t]

    # ritrovamento dei pos in base all'indice
    bestPos = []
    for bp in bestPath:
        bestPos.append(pos[int(bp)])
    return bestPos
```

Fig. 2. Codice algoritmo di Viterbi

4. Strategie di smoothing

Come si vede dalla figura 2, lo smoothing permette di gestire casi in cui il sistema deve affrontare termini sconosciuti.
Gli approcci adottati sono diversi:

- la prima strategia è assumere che tutte le parole sconosciute siano dei nomi;
- un secondo modo, invece, è quello di assumere che queste siano o nomi o verbi;
- è possibile distribuire una probabilità omogenea a tutti i PoS;
- un'altra tecnica molto ingegnosa è quella di allenare il sistema su un secondo corpus - in questo caso il development set del progetto UD - calcolando le probabilità che parole sconosciute possano essere etichettate con un certo PoS;
- l'ultimo approccio implementato è basato sulla sintassi: si controlla che un termine abbia un certo suffisso o prefisso per dedurre che questo possa essere classificato in un certo modo.

Di seguito sono riportate le feature sintattiche - suffissi - usate per la classificazione:

ADJ - latino		NOUN/ADJ - latino		VERB - latino	
0	aceus	0	gen	0	esco
1	alis	1	arium	1	ico
2	andus	2	arius	2	ito
3	endus	3	atus	3	sco
4	iendus	4	cola	4	so
5	ans	5	colum	5	sso
6	antis	6	dicus	6	to
7	ens	7	ellus	7	urio
8	entis	8	genus		
9	iens	9	gena		
10	ientis	10	mentum		
11	anus	11	aria		
12	aticus	12	or		
13	atus	13	tas		
14	bilis	14	tus		
15	bundus	15	ter		
16	ellus	16	tio		
17	ensis	17	tor		
18	esimus	18	trix		
19	eus	19	trina		
20	ilis	20	tudo		
21	inus	21	unculus		
22	ior	22	ura		
23	ius				
24	issimis				
25	imus				
26	osus				
27	torius				
28	timus				
29	ulus				

Fig. 3. Feature sintattiche per il latino

NOUN - greco		ADJ - greco		ADV - greco		VERB - greco	
0	της	0	ῖος	0	ως	0	ίζω
1	τής	1	εῖος				
2	ίτης						
3	ώτης						

Fig. 4. Feature sintattiche per il greco

Nota: gli accorgimenti sintattici ritrovati per il greco sono inferiori a quelli latini per motivi di ritrovamento. Seppur in numero basso essi hanno portato come si vedrà comunque a miglioramenti.

5. Valutazione

Nella seguente sezione si mostrano i risultati ottenuti dal sistema.

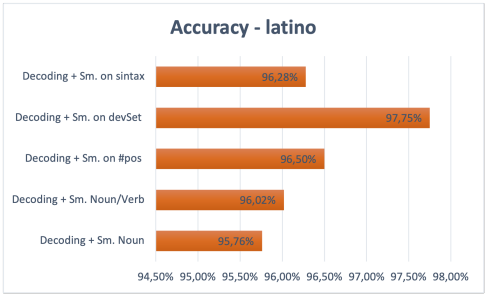

Fig. 5. Score per il latino

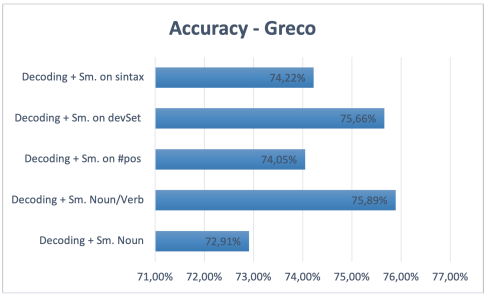

Fig. 6. Score per il greco

Generalmente entrambe le lingue presentano analogie per quanto riguarda il testing applicando le diverse strategie di smoothing. Infatti, la strategia di smoothing basata sui nomi è per entrambe la peggiore, contrariamente a quella basata sul development set che si comporta decisamente meglio. Lo smoothing basato sulle regole sintattiche migliora leggermente lo score rispetto allo smoothing basato sui nomi, non riuscendo tuttavia a posizionarsi nella parte alta della classifica delle strategie.

Accanto a questa analisi, poniamo di seguito quella dei tempi:

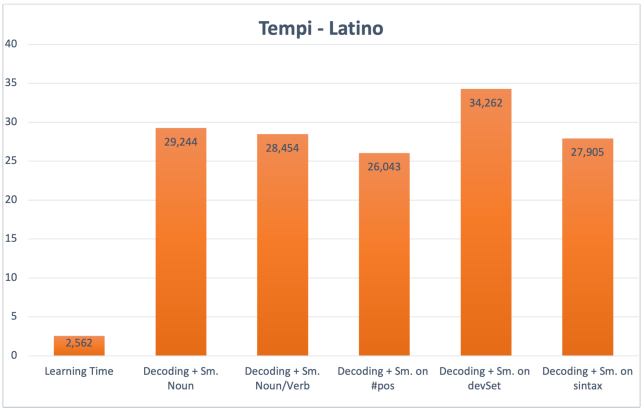

Fig. 7. Tempi in secondi per il latino

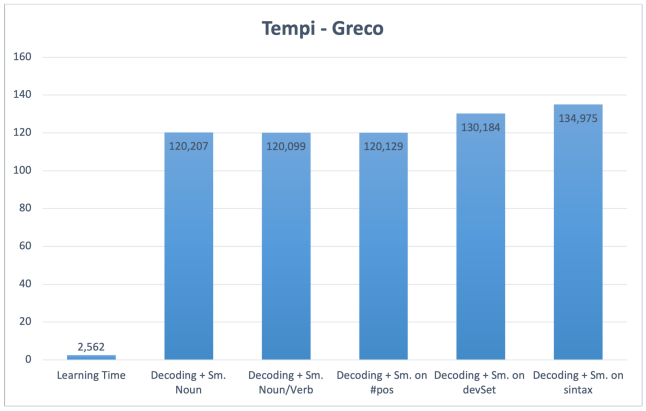

Fig. 8. Tempi in secondi per il greco

Quello che risalta è la differenza in termini di ordine di grandezza fra le due lingue. La causa è dovuta probabilmente all'encoding che l'interprete Python ha dovuto effettuare per l'alfabeto greco. Ciononostante, per tutti i test con i vari smoothing il tempo di esecuzione è molto simile.
Come era prevedibile sia per il greco che per il latino i tempi di esecuzione minori sono per le strategie di smoothing che impiegano meno tempo ad essere calcolate: per entrambe infatti, la strategia basata sul development set è la più lenta.

Per quanto riguarda gli errori più comuni commessi dal sistema: generalmente i tag che inducono maggiormente in errore sono per il latino i PROPN, i VERB e gli ADJ; invece, per il greco sono gli ADV, gli ADJ e i VERB. La motivazione è probabilmente dovuta alla natura delle termini: ad esempio i nomi propri sono molti e vari quindi, con poche occorrenze nel training corpus; mentre nel greco gli aggettivi neutri possono fungere da avverbi e condurre il sistema in errore.

6. Simple Baseline e confronto

L'ultima parte dell'analisi dei dati ha previsto il confronto dei risultati con una semplice baseline. Ne è stata, quindi, implementata una con le seguenti caratteristiche:

- nella fase di learning si esaminano le parole del training set e si memorizzano i tag più frequenti per queste.
- nella fase di decoding si assegnano alle parole del test set i tag ritrovati nel passo precedente. Qualora ci siano termini sconosciuti sarà assegnato automaticamente il tag 'NOUN'.

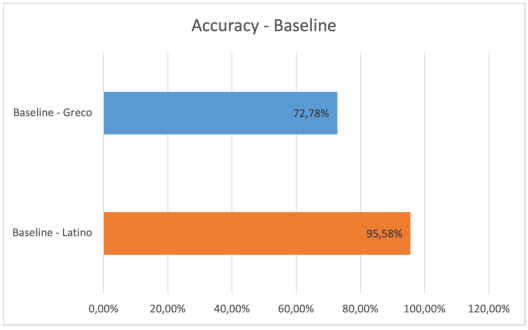

Fig. 9. Score per la baseline - semplice

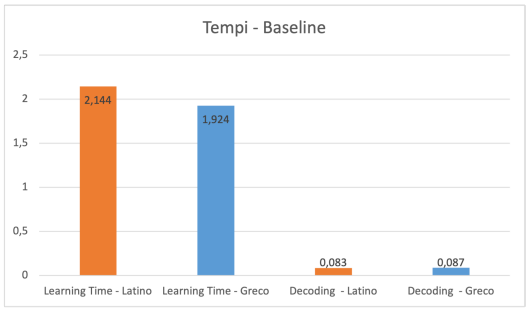

Fig. 10. Tempi per la baseline - semplice

Da questi dati scaturisce una domanda nota in letteratura: 'Quanto è vantaggioso il compromesso tra tempo e accuratezza?'
Di fatti esaminando solo le percentuali di accuratezza per la lingua latina si evince come la differenza tra questa e ad esempio quella implementata con la strategia di smoothing basata sui nomi è di circa due decimi. Tuttavia, il tempo necessario per computare i due risultati è notevolmente diverso: il sistema della simple baseline è ovviamente molto più veloce. Tali risultati sono ancora più enfatizzati per la lingua greca.

Prima di giungere alle conclusioni, si può trarre un'analogia tra gli errori più comuni ritrovati in output e il sistema implementato. Anche in questo caso, quelli più comuni riguardano i tag elencati nella sezione precedente confermando l'ambiguità di questi.

Il sistema implementato nel progetto è infine un buono strumento per effettuare PoS tagging. Le percentuali di accuratezza per ogni strategia di smoothing sono maggiori di quella di una semplice baseline e questi si avvicinano ai punteggi umani. Tuttavia, in base al task richiesto e alla capacità di calcolo del sistema potrebbe essere meglio rivolgere lo sguardo verso sistemi più semplici che garantiscano risultati leggermente peggiori, ma con vantaggi significativi dal punto di vista computazionale. Un esempio è per lingue complesse come il greco dove la complessità temporale gioca un ruolo importante; al contrario è un'ottima soluzione usare il PoS tagger con smoothing basato su un development set

qualora si disponga di corpus ricchi.

Nota: i risultati sono stati calcolati con una macchina con le seguenti specifiche hardware e software: