Relazione PoS Tagging per Latino e Greco

PARISI MATTEO

INTRODUZIONE

L'obiettivo di questo progetto è quello di realizzare l'algoritmo di Viterbi per risolvere il task di PoS Tagging su lingue morte come Latino e Greco.
Quindi, data una frase, il PoS Tagging consiste nell'assegnare una parte del discorso (Part of Speech) identificata da un tag ad ogni parola della frase stessa. L'algoritmo di Viterbi è basato sulle HMM (Hidden Markov Model).

L'operazione di Pos Tagging consiste in:

- **Input**: una frase, vista come una sequenza di parole;
- **Output**: una lista di PoS Tag. Ogni tag è associato ad una parola della frase presa in input dall'algoritmo.

Questo problema, secondo lo standard del machine learning, è stato affrontato in tre fasi:

1. **Modelling**: viene fornito un modello formale del problema;
2. **Learning:** si cerca di capire come, dato un corpus, sia possibile settare i parametri in grado di generare il modello migliore;
3. **Decoding:** si cerca di capire qual è l'algoritmo che permette di applicare al meglio i parametri appresi nella fase precedente, per poter recuperare la soluzione ottimale dato un certo input.

MODELLING

La prima fase, la fase di Modelling, si occupa di definire il modello matematico del problema.
Dal punto di vista statistico l'obiettivo è quello di ottenere la sequenza di tag più probabile data la sequenza di parole nella frase:

$$\hat{t}_1^n = argmax_{t_1^n} P(t_1^n | w_1^n)$$

Per rendere operazionale questo calcolo è possibile sfruttare la regola di Bayes:

$$\hat{t}_1^n = argmax_{t_1^n} P(w_1^n | t_1^n) P(t_1^n)$$

In questo modo si ottiene una nuova equazione che approssima la precedente:

$$\hat{t}_1^n = argmax_{t_1^n} P(t_1^n | w_1^n) \approx argmax_{t_1^n} \prod_{i=1}^{n} P(w_i | t_i)\, P(t_i | t_{i-1})$$

Quindi la sequenza di tag più probabile è determinata dal prodotto di due tipi di probabilità:

$$P(w_i | t_i) \text{ e } P(t_i | t_{i-1})$$

Dove la prima rappresenta la probabilità di transizione, mentre la seconda rappresenta la probabilità di emissione, che saranno poi calcolate nella fase di learning.

LEARNING

La seconda fase, la fase di learning, si occupa di apprendere le **probabilità di osservazione** e le **probabilità di transizione** basandosi sui training set delle due lingue prese in esame. Analizziamo le probabilità:

PROBABILITÀ DI EMISSIONE

La probabilità di emissione rappresenta quanto un certo tag sia una certa parola. Anche in questo caso i conteggi vengono definiti dalla formula per il calcolo della probabilità:

$$P(w_i|t_i) = \frac{C(t_i,w_i)}{C(t_i)}$$

Dove:

- $C(ti, wi)$ è il numero di volte che la parola wi è taggata con ti;
- $C(ti)$ è il numero di volte che il tag ti compare nel training set.

Per la memorizzazione delle probabilità di emissione è stato utilizzato un dizionario in quanto una matrice di emissione avrebbe avuto tante colonne quante sono le parole nel train set e non sarebbe una struttura dati ottimale.

```python
def compute_emission_probabilities(train):
    word_tag_set = []
    tags_set = []
    words_set = []
    for sentence in parse_incr(train):
        for token in sentence:
            word_tag_set.append((token["form"],token["upos"]))
            tags_set.append(token["upos"])
            words_set.append(token["form"])

    count_word_tag = dict(Counter(word_tag_set))
    count_tags = dict(Counter(tags_set))
    count_word = dict(Counter(words_set))

    emission_dict = dict()
    for key in count_word_tag:
        emission_dict[(key[0],key[1])] = count_word_tag[key]/count_tags[key[1]]
    return emission_dict,count_word,count_word_tag
```

La funzione *compute_emission_probabilities* restituisce tre dizionari:

- ***emission_dict***: dizionario di emissione, descritto nella fase di Decoding.
- ***count_word***: dizionario delle parole con il relativo conteggio all'interno del train set. Servirà nella fase di Decoding per verificare se una parola della sentence analizzata è conosciuta apriori o meno.
- ***count_word_tag***: dizionario delle parole conteggiate con i pos tag. È utile nell'algoritmo di Baseline.

La funzione prende in input il train set (del Latino o del Greco) e restituisce come primo elemento *emission_dict*, un dizionario che avrà come chiavi delle coppie (word, tag) e come valori le probabilità di emissione relative alle coppie.

PROBABILITÀ DI TRANSIZIONE

La probabilità di transizione indica quant'è probabile che un tag t_i segua un tag t_i−1. È calcolata utilizzando la seguente formula:

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

Dove:

- $C(t_i$−1, $t_i)$ è il numero di volte in cui il tag t_i−1 precede il tag t_i, nel train set;
- $C(t_i$−1) è il numero di volte in cui il tag t_i−1 compare nel train set.

Per la memorizzazione delle probabilità di transizione è stata utilizzata una matrice etichettata sia sulle righe che sulle colonne con i Pos Tag per poter accedere alle probabilità in maniera semplice, specificando i due tag relativi ad essa.

Le matrici di transizione cambiano in base al linguaggio utilizzato per il training perchè l'insieme dei Pos Tag relativi al train set del latino è diverso da quello relativo al greco.

Infatti, avremo una lista di tag possibili (*possible_tags*) a cui è stato aggiunto lo stato iniziale 'START' e lo stato finale 'END'. Quest'ultimi NON sono associati in generale a nessuna osservazione (parola) ma saranno utili nella fase di Learning per il calcolo delle probabilità di transizione iniziali e finali utilizzate successivamente nella fase di Decoding.

Pos Tag per il Graco e Latino:

```
Lingua:  GREEK
['ADJ', 'ADP', 'ADV', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NUM', 'PART', 'PRON', 'SCONJ', 'VERB', 'X', 'PUNCT']
```

```
Lingua:  LATIN
['ADJ', 'ADP', 'ADV', 'AUX', 'CCONJ', 'DET', 'NOUN', 'NUM', 'PART', 'PRON', 'PROPN', 'PUNCT', 'SCONJ', 'VERB', 'X']
```

Di seguito l'algoritmo *compute_transition_matrix* utilizzato per popolare la matrice di transizione. Il calcolo viene effettuato per ogni coppia di tag presente nella lista *possible_tags*:

```
def compute_transition_matrix(possible_tags, train):
    transition_matrix = np.zeros((len(possible_tags), len(possible_tags)), dtype='float32')

    counter_dict = dict()
    count_initial_dict = dict()
    transition_counter_dict = dict()

    #FASE 1
    for tag1 in possible_tags:
        counter_dict[tag1] = 0
        count_initial_dict[tag1] = 0
        for tag2 in possible_tags:
            transition_counter_dict[(tag1, tag2)] = 0
```

FASE 1: vengono inizializzati i tre dizionari utilizzati per memorizzare i conteggi relativi ai tag. In particolare, abbiamo:

- *counter_dict*: dizionario relativo ai conteggi dei tag singoli all'interno del train set. Sarà formato da coppie (TAG, conteggio) tale che il valore 'conteggio' indica quante volte il TAG compare all'interno del train set.
- *count_initial_dict*: dizionario relativo ai conteggi dei tag singoli che compaiono per primi nelle sentence del train set. Sarà formato da coppie (TAG, conteggio) tale che il valore 'conteggio' indica quante volte il TAG è associato alla prima parola delle sentence all'interno del train set.
- *transition_counter_dict*: dizionario relativo al conteggio delle coppie di tag che compaiono uno dopo l'altro nelle sentence del train set. Sarà formato da coppie ((TAG1, TAG2), conteggio), tale che il valore 'conteggio' indica quante volte il TAG1 compare immediatamente prima del TAG2 nelle sentence del train set.

  In precedenza, viene inizializzata la matrice di transizione come una matrice di zeri, quindi vuota.

```
#FASE 2
sentence_n = 0
for sentence in parse_incr(train):
    sentence_n += 1
    for i in range(len(sentence)):
        word_before = sentence[i-1]
        word = sentence[i]
        if i == 0:
            if word["upos"] in count_initial_dict.keys():
                count_initial_dict[word["upos"]] = count_initial_dict[word["upos"]] + 1
        if (word_before["upos"], word["upos"]) in transition_counter_dict.keys() and i != 0:
            transition_counter_dict[(word_before["upos"], word["upos"])] = transition_counter_dict[(word_before["upos"], word["upos"])] + 1
        if word["upos"] in counter_dict.keys():
            counter_dict[word["upos"]] = counter_dict[word["upos"]] + 1
        if i == len(sentence) - 1:
            if (word["upos"], 'END') in transition_counter_dict.keys():
                transition_counter_dict[(word["upos"], 'END')] = transition_counter_dict[(word_before["upos"], word["upos"])] + 1
```

FASE 2: vengono popolati i tre dizionari con i relativi conteggi. Viene analizzata ogni sentence del train set in modo tale da effettuare tutti i conteggi e memorizzarli all'interno dei dizionari.

```
#FASE 3
#probabilità di transizione iniziali
for i,t in enumerate(possible_tags):
    transition_matrix[0][i] = count_initial_dict[t]/sentence_n
#probabilità di transizione intermedie
for i,t1 in enumerate(possible_tags):
    for j,t2 in enumerate(possible_tags):
        if i >= 1 and j >= 1 and i < (len(possible_tags) - 1):
            transition_matrix[i][j] =  transition_counter_dict[(t1,t2)]/counter_dict[t1]
train.seek(0)
return transition_matrix
```

FASE 3: viene popolata la matrice di transizione con le probabilità di transizione. La probabilità del tag2 dato il tag1 è memorizzata nella cella della matrice avente come riga il tag1 e come colonna il tag2. In particolare, vengono effettuate due operazioni:

- Viene popolata la prima riga della matrice di transizione con le probabilità di transizione iniziali. Ogni cella è calcolata dividendo il numero di volte che il tag relativo alla colonna è associato alla prima parola di ogni sentence per il numero di sentence totali.
- Vengono popolate le righe successive alla prima con le probabilità di transizione calcolate dividendo il numero di volte che il tag1(riga) compare immediatamente prima del tag2 (colonna) per il numero di volte che compare il tag1.

Dunque, ogni riga della matrice di transizione (tranne l'ultima) rappresenta una distribuzione di probabilità e la somma dei suoi valori è sempre pari a 1.

Nell'ultima colonna della matrice di transizione sono memorizzate le probabilità di transizione finali.

DECODING

In questa fase si utilizzano le probabilità calcolate nella fase di learning per ottenere i PoS delle parole all'interno delle frasi del test set. Per fare ciò è stato implementato l'algoritmo di Viterbi che sfrutta la **programmazione dinamica** e **l'approssimazione Markoviana**.
La prima consente di dividere il problema in due sotto-problemi più semplici, la seconda riduce la complessità computazionale del problema da esponenziale a polinomiale. In particolare, data una sequenza di tag che finisce allo stato j con un tag T, la probabilità di questa sequenza può essere ridotta in due fattori: la probabilità della miglior sequenza di tag fino allo stato j-1 e la probabilità di transizione dal tag allo stato j-1 al tag T.
L'approssimazione Markoviana, inoltre, ci consente di determinare la probabilità di un tag considerano solo il tag precedente.

ALGORITMO DI VITERBI

L'algoritmo di Viterbi riceve in input i seguenti parametri:

- *sentence_tokens*: lista di parole della frase analizzata;
- *possible_tags*: lista dei possibili tag relativi dal train set utilizzato (Greco o Latino);
- *transition_matrix*: matrice di transizione che memorizza tutte le probabilità di transizione;
- *emission_probabilities*: dizionario delle probabilità di emissione;
- *count_word*: dizionario delle parole con i relativi conteggi nel train set;
- *smoothing_strategy*: strategia di smoothing utilizzata, relativa al train set;
- *oneshot_words_tag_distribution*: distribuzione probabilistica dei tag relativi alle parole che compaiono una sola volta nel dev set.

```
def viterbi_algorithm(sentence_tokens, possible_tags, transition_matrix, emission_probabilities, count_word, smoothing_strategy, oneshot_words_tag_distribution):

    viterbi_matrix = np.zeros((len(possible_tags), len(sentence_tokens))) #matrice di viterbi
    backpointer = dict() #dizionario di dizionari
```

- *viterbi_matrix*: matrice di dimensione *possible_tags* * *sentence_tokens*, che sono rispettivamente il numero di tag possibili dai quali sono stati rimossi 'START' ed 'END' ed il numero di termini della frase in input.
  Una differenza importante con l'algoritmo standard di Viterbi visto a lezione è la struttura della matrice di Viterbi: non è stato considerato necessario ampliare la dimensione della matrice con le righe corrispondenti ai tag di 'START' ed 'END' per due motivi:

1. gli stati corrispondenti a questi tag non vengono calcolati. Ciò che è importante, invece, sono le probabilità presenti nella matrice di transizione.
2. viene restituita dall'algoritmo solo la sequenza più probabile di tag, non la sua probabilità, la quale non viene memorizzata in nessuno stato finale.

Ogni cella della matrice che chiameremo 'stato' ha l'utilità di memorizzare la probabilità più alta di una sottosequenza della sequenza di tag, questo ci permette di ridurre la complessità perchè NON vengono calcolate iterativamente tutte le possibili sequenze, il tutto per ottenere la soluzione migliore sfociando in un'esplosione combinatoria.

- *backpointer*: dizionario di dizionari. Ad ogni chiave (colonna) corrisponde un dizionario che rappresenta una colonna. Ogni colonna è rappresentata da un insieme di chiavi (righe) a cui corrisponde il puntatore alla riga della colonna precedente.

  Questa struttura dati è utile per memorizzare il riferimento di ogni stato allo stato precedente che ha dato maggior contributo nel calcolo del suo valore di Viterbi.

```
#FASE 1: inizializzazione della prima colonna
for s,tag in enumerate(possible_tags):
    transition_p = transition_matrix.loc['START',tag]
    emission_p = get_emission_p(emission_probabilities, sentence_tokens[0], tag, count_word, smoothing_strategy, oneshot_words_tag_distribution, possible_tags)

    if transition_p == 0 : transition_p = np.finfo(float).tiny
    if emission_p == 0 : emission_p = np.finfo(float).tiny

    viterbi_matrix[s,0] = math.log(transition_p) +  math.log(emission_p)
```

FASE 1: Viene inizializzata la prima colonna della matrice di Viterbi: per ogni stato rappresentato da un tag viene calcolata la somma dei logaritmi tra la probabilità di transizione iniziale relativa al tag e la probabilità di emissione relativa alla coppia (parola, tag). Se la probabilità di transizione o quella di emissione è pari a 0, la si transforma nel numero reale più piccolo positivo, in modo tale da poterne calcolare il logaritmo.

```
def get_emission_p(emission_probabilities, word, tag, count_word, smoothing_strategy, oneshot_words_tag_distribution, possible_tags):
    emission_p = 0
    try:
        count_word[word]
    except KeyError: #unknown_word
        emission_p = unknown_word_emission_p(smoothing_strategy, tag, possible_tags, oneshot_words_tag_distribution)
        return emission_p
    try:
        emission_p = emission_probabilities[(word,tag)]
    except KeyError: #tag never emitted word
        emission_p = 0
    return emission_p
```

La funzione *get_emission_p* restituisce la probabilità di emissione presente nel dizionario *emission_probabilities* oppure la calcola in base alla strategia di smoothing utilizzata.

```
#FASE 2: Inizializzazione delle colonne intermedie
#Si cicla prima sulle colonne e poi sulle righe
for t in range(1,len(sentence_tokens)):
    backpointer_column = dict()
    for s, tag in enumerate(possible_tags):
        max_ , backpointer_column[s] = get_max_argmax_value(possible_tags, viterbi_matrix, transition_matrix, t, s)
        emission_p = get_emission_p(emission_probabilities, sentence_tokens[t], tag, count_word, smoothing_strategy, oneshot_words_tag_distribution, possible_tags)
        if emission_p == 0: emission_p = np.finfo(float).tiny
        viterbi_matrix[s,t] = max_ + math.log(emission_p)
    backpointer[t] = backpointer_column
```

FASE 2 - Step iniziale: Vengono calcolati i valori di Viterbi degli stati delle colonne successive alla prima e man mano viene popolato il dizionario backpointer.

```
def get_max_argmax_value(possible_tags, viterbi_matrix, transition_matrix, t, s):
    max_ = -sys.maxsize
    argmax = None
    for s1, tag in enumerate(possible_tags):
        transition_p = transition_matrix.loc[tag,possible_tags[s]]
        if transition_p == 0 : transition_p = np.finfo(float).tiny
        val = viterbi_matrix[s1, t-1] + math.log(transition_p)
        if val >= max_: max_ = val; argmax = s1
    return max_, argmax
```

La funzione *get_max_argmax_value* restituisce:

1. *max_*: il prodotto massimo tra ogni valore di Viterbi della colonna precedente e la probabilità di transizione tra lo stato a cui fa riferimento e lo stato corrente.

$$\max_{s'=1}^{N} \; viterbi[s', t-1] \; * \; a_{s',s}$$

2. la riga s' che massimizza questo prodotto. Inoltre, l'indice della riga viene memorizzato nel dizionario *backpointer*.

Il valore di Viterbi dello stato corrente è calcolato sommando *max_* con il logaritmo della probabilità di emissione del token rappresentato dalla colonna corrente.

```
#FASE 2: Step finale (argmax)
max_ = -sys.maxsize
best_path_pointer = None
for s,tag in enumerate(possible_tags):
    end_transition = transition_matrix.loc[tag,'END']
    if end_transition == 0: end_transition = np.finfo(float).tiny
    val = viterbi_matrix[s,len(sentence_tokens) - 1] + math.log(end_transition)
    if val >= max_: max_ = val ; best_path_pointer = s
```

FASE 2 - Step finale: Calcolo del *best_path_pointer*, ovvero il riferimento alla riga della cella dell'ultima colonna che massimizza la probabilità della sequenza di tag migliore.

$$\max_{s=1}^{N} \; viterbi[s,T] * a_{s,q_F}$$

Quest'ultima è calcolata moltiplicando il valore di Viterbi della cella in questione con la probabilità di transizione finale.

```
#FASE 3: Backtracking
#Recupero tramite backtracking della sequenza di PoS
states = []
states.append(best_path_pointer)
t = len(sentence_tokens) - 1
s = best_path_pointer
while t >= 1:
    states.append(backpointer[t].get(s))
    s = backpointer[t].get(s)
    t = t -1
```

FASE 3 - Backtraking: Partendo dal *best_path_pointer* viene srotolata la sequenza dei tag più probabile sfruttando il dizionario *backpointer*.

SMOOTHING

Sono state utilizzate diverse strategie di smoothing per le parole sconosciute:

- **UNKNOWN_NAME**: se il tag preso in considerazione per la parola sconosciuta è NOUN allora la probabilità di emissione sarà uguale a 1, ovvero si decide con estrema certezza che la parola sconosciuta è un sostantivo.

$$P(unk|NOUN)=1$$

- **UNKNOWN_NAME_VERB**: se il tag preso in considerazione per la parola sconosciuta e NOUN o VERB allora la probabilità di emissione sarà uguale a 0.5, ovvero la parola può essere un sostantivo o un verbo con la stessa probabilità

$$P(unk|VERB)=P(unk|NOUN)=0.5$$

- **UNKNOWN_TAG**: la proabilità di emissione per qualsiasi tag sarà 1 diviso la cardinalità dell'insieme dei tag possibili, ovvero una parola sconosciuta può essere un qualsiasi tag della lista dei tag possibili con la stessa probabilità.
- **UNKNOWN_DEV:** la probabilità di emissione viene calcolata sulla base di una distribuzione estratta da un determinato corpus. In pratica, la distribuzione è relativa ai tag associati alle parole che compaiono una sola volta nel dev-set.

  Il procedimento consiste nel collezionare tutte le parole che compaiono una sola volta (one shot word) e verificare quante di queste sono NOUN, VERB, ADJ, ecc...

  Come tutte le distribuzioni, quella calcolata assocerà ad ogni tag una probabilità e la somma di tutte le probabilità sarà 1. Nell'algoritmo *compute_oneshot_words_distributions*, che calcola e restituisce la distribuzione, verranno considerati anche i tag non associati a nessuna one shot word presente nel dev-set; a questi verrà assegnata probabilità nulla.

  Ad esempio, la distribuzione di probabilità per i tag relativi alle one shot word del dev-set sarà definita come una lista di coppie (tag, probabilità):

  Latino:

```
[('NOUN', 0.196405648267009), ('PROPN', 0.417201540436457), ('VERB', 0.21951219512195122), ('DET', 0.04749679075738126), ('ADJ',
0.06161745827984596), ('PRON', 0.005134788189987163), ('ADV', 0.02053915275994865), ('ADP', 0.011553273427471117), ('NUM',
0.014120667522464698), ('SCONJ', 0.0012836970474967907), ('AUX', 0.005134788189987163), ('START', 0), ('CCONJ', 0), ('PART', 0), ('PUNCT',
0), ('X', 0), ('END', 0)]
```

  Greco:

```
[('ADJ', 0.1627680311890838), ('VERB', 0.47797270955165694), ('NOUN', 0.31539961013645224), ('CCONJ', 0.0005847953216374269), ('PRON',
0.010916179337231967), ('ADV', 0.027485380116959064), ('DET', 0.0009746588693957114), ('ADP', 0.00253411306042885), ('NUM',
0.0003898635477582846), ('SCONJ', 0.0009746588693957114), ('START', 0), ('INTJ', 0), ('PART', 0), ('X', 0), ('PUNCT', 0), ('END', 0)]
```

È possibile notare che, per quanto riguarda il Latino, c'è un'alta percentuale di nomi propri (PROPN) perchè rappresentano quella parte del discorso che viene menzionata più raramente, specialmente se trattiamo un corpus con sentence che non sono in relazione contestuale tra di loro. Mentre per il Greco c'è un'alta percentuale di verbi (VERB).

RISULTATI

Per valutare l'implementazione dell'algoritmo di Viterbi, è stata misurata l'accuratezza per ogni strategia di smoothing, il tempo totale di esecuzione e gli errori commessi dall'algoritmo. Per avere un metro di misura è stata implementata una baseline, per entrambi i dataset, che attribuisce ad ogni parola nel test set il tag più frequente per quella parola nel training set, se la parola non è presente nel training set, viene attribuito tag "NOUN".

BASELINE LATINO

L'algoritmo di Baseline sul latino ottiene ottimi risultati. I vantaggi di questo algoritmo sono l'estrema semplicità e la velocità di esecuzione, infatti risulta essere quasi istantaneo. Gli errori più comuni sono principalmente relativi alla valutazione dei nomi propri e verbi.

```
PoS Tag corretti:  22969
PoS Tag sbagliati:  1110
Totale parole valutate:  24079
Errori per PoS Tag:  {'VERB': 248, 'PROPN': 471, 'ADV': 56, 'DET': 142, 'NUM': 34, 'ADJ': 83, 'NOUN': 15, 'CCONJ': 35, 'ADP': 6, 'SCONJ':
15, 'AUX': 3, 'PRON': 2}

Accuratezza:  95.39 %
Tempo di esecuzione:  0.37  sec
```

BASELINE GRECO

Le performance per il Greco calano di molto. L'algoritmo di Baseline ha un'accuratezza del 73.53 % e gli errori più comuni riguardano i verbi, gli avverbi e i nomi.

```
PoS Tag corretti:  15411
PoS Tag sbagliati:  5548
Totale parole valutate:  20959
Errori per PoS Tag:  {'VERB': 1978, 'ADV': 1823, 'PRON': 462, 'ADJ': 965, 'CCONJ': 133, 'DET': 88, 'SCONJ': 25, 'NOUN': 50, 'ADP': 16,
'NUM': 1, 'PUNCT': 3, 'INTJ': 3, 'X': 1}

Accuratezza:  73.53 %
Tempo di esecuzione:  0.32  sec
```

VITERBI

L'algoritmo di Viterbi ottiene risultati leggermente migliori del Baseline per quanto riguarda le quattro strategie di smoothing, ma in tutti i casi il tempo di esecuzione risulta essere estremamente più lungo. Anche in questo caso gli errori più comuni riguardano la valutazione dei nomi propri; questo accade perchè i nomi propri sono quelli che appaiono più raramente nel train set e molti di questi sono trattati come parole sconosciute; di conseguenza risulta essere determinante la scelta effettuata dalla strategia di smoothing utilizzata.

Notiamo che le strategie di smoothing sono fondamentali perchè per loro natura tendono a far diminuire gli errori riguardo i tag che scelgono; nel primo smoothing, il conteggio degli errori relativi ai nomi è molto basso, nel secondo invece aumenta ma diminuisce quello relativo ai verbi.

LATINO

La strategia di smoothing migliore risulta essere l'ultima, perchè viene ottenuta un'approssimazione molto vicina a quella che sarebbe la distribuzione di probabilità dei tag relativi alle parole sconosciute.

```
Tipologia di smoothing:  UNKNOWN_NAME
PoS Tag corretti:  23110
PoS Tag sbagliati:  969
Totale parole valutate:  24079
Errori per PoS Tag:  {'VERB': 200, 'PROPN': 471, 'ADV': 54, 'NUM': 15, 'PRON': 32, 'DET': 25, 'ADJ': 82, 'NOUN': 19, 'AUX': 31, 'CCONJ': 21,
'PUNCT': 5, 'SCONJ': 12, 'ADP': 2}

Accuratezza:  95.98 %
Tempo di esecuzione:  40.80  sec
```

```
Tipologia di smoothing:  UNKNOWN_NAME_VERB
PoS Tag corretti:  23170
PoS Tag sbagliati:  909
Totale parole valutate:  24079
Errori per PoS Tag:  {'VERB': 121, 'PROPN': 471, 'ADV': 49, 'NUM': 15, 'PRON': 22, 'DET': 25, 'ADJ': 82, 'NOUN': 56, 'AUX': 29, 'CCONJ': 21,
'PUNCT': 4, 'SCONJ': 12, 'ADP': 2}

Accuratezza:  96.22 %
Tempo di esecuzione:  41.26  sec
```

```
Tipologia di smoothing:  UNKNOWN_TAG
PoS Tag corretti:  23216
PoS Tag sbagliati:  863
Totale parole valutate:  24079
Errori per PoS Tag:  {'VERB': 148, 'PROPN': 387, 'ADV': 48, 'NUM': 15, 'PRON': 22, 'DET': 23, 'ADJ': 82, 'NOUN': 69, 'AUX': 29, 'CCONJ': 21,
'PUNCT': 5, 'SCONJ': 12, 'ADP': 2}

Accuratezza:  96.42 %
Tempo di esecuzione:  41.37  sec
```

```
Tipologia di smoothing:  UNKNOWN_DEV
PoS Tag corretti:  23409
PoS Tag sbagliati:  670
Totale parole valutate:  24079
Errori per PoS Tag:  {'VERB': 159, 'PROPN': 184, 'ADV': 49, 'NUM': 15, 'PRON': 22, 'DET': 25, 'ADJ': 82, 'NOUN': 66, 'AUX': 29, 'CCONJ': 21,
'PUNCT': 4, 'SCONJ': 12, 'ADP': 2}

Accuratezza:  97.22 %
Tempo di esecuzione:  40.87  sec
```

GRECO

Anche per il Greco, la strategia di smoothing migliore risulta essere quella relativa alla distribuzione di probabilità dei tag relativi alle parole che compaiono una sola volta nel dev set.

```
Tipologia di smoothing:  UNKNOWN_NAME
PoS Tag corretti:  15426
PoS Tag sbagliati:  5533
Totale parole valutate:  20959
Errori per PoS Tag:  {'VERB': 1974, 'ADV': 1710, 'PRON': 527, 'ADJ': 959, 'CCONJ': 125, 'DET': 103, 'SCONJ': 61, 'NOUN': 47, 'ADP': 18,
'PUNCT': 4, 'NUM': 1, 'INTJ': 3, 'X': 1}

Accuratezza:  73.60 %
Tempo di esecuzione:  29.95  sec
```

```
Tipologia di smoothing:  UNKNOWN_NAME_VERB
PoS Tag corretti:  16020
PoS Tag sbagliati:  4939
Totale parole valutate:  20959
Errori per PoS Tag:  {'VERB': 725, 'ADV': 1646, 'PRON': 501, 'NOUN': 780, 'ADJ': 966, 'CCONJ': 130, 'DET': 112, 'SCONJ': 52, 'ADP': 18,
'PUNCT': 4, 'NUM': 1, 'INTJ': 3, 'X': 1}

Accuratezza:  76.43 %
Tempo di esecuzione:  31.43  sec
```

```
Tipologia di smoothing:  UNKNOWN_TAG
PoS Tag corretti:  15497
PoS Tag sbagliati:  5462
Totale parole valutate:  20959
Errori per PoS Tag:  {'VERB': 974, 'ADV': 1638, 'PRON': 467, 'NOUN': 1079, 'ADJ': 970, 'CCONJ': 131, 'DET': 129, 'SCONJ': 50, 'ADP': 16,
'PUNCT': 3, 'NUM': 1, 'INTJ': 3, 'X': 1}

Accuratezza:  73.94 %
Tempo di esecuzione:  30.68  sec
```

```
Tipologia di smoothing:  UNKNOWN_DEV
PoS Tag corretti:  15982
PoS Tag sbagliati:  4977
Totale parole valutate:  20959
Errori per PoS Tag:  {'ADV': 1633, 'PRON': 494, 'VERB': 380, 'NOUN': 1172, 'ADJ': 968, 'CCONJ': 131, 'DET': 123, 'SCONJ': 49, 'ADP': 18,
'PUNCT': 4, 'NUM': 1, 'INTJ': 3, 'X': 1}

Accuratezza:  76.25 %
Tempo di esecuzione:  30.45  sec
```