Pos Tagger per lingue morte: Latino e Greco

0. Introduzione

L'obiettivo dell'esercitazione è quello di implementare un Pos Tagger statistico per lingue morte (Latino e Greco) basato su **HMM** (Hidden Markov Model), il quale molto spesso viene utilizzato nella analisi delle sequenze.
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 3 fasi, ma la definizione del modello (Modelling) è già nota agli addetti ai lavori:

- **Modelling**: definizione del **modello matematico** del problema
  L'obiettivo è trovare la sequenza di tag $\hat{t}_1^n$ data la sequenza di parole osservabili w_1^n, che massimizza la distribuzione di probabilità P:

$$\hat{t}_1^n = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n, w_1^n)$$

Per rendere operazionale questo calcolo è possibile sfruttare la regola di Bayes in modo tale da ottenere una equazione che avrà più probabilità da calcolare, ma **approssimabili**. In questo modo si ottiene una nuova equazione che approssima la precedente:

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

Distinguiamo le due probabilità:

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

le quali sono rispettivamente le **probabilità di transizione** ed **emissione** utilizzate nella fase di Learning, che verranno trattate nel capitolo successivo insieme agli algortimi e strutture dati impiegati rispettivamente per ottenerli e memorizzarli

- **Learning**: apprendere i parametri da un dato corpus, che verranno utilizzati nella fase di Decoding
- **Decoding**: implementazione dell'**algoritmo** di Pos Tagging

Infine, è stata fondamentale la valutazione dei risultati in modo da comprendere quale strategia di **smoothing** risulta essere più efficace e se le performance ottenute dell'algoritmo implementato superano, e in che modo, quelle del più comune algoritmo di **baseline**. E' stato necessario, inoltre, confrontare i risultati ottenuti in base al linguaggio testato e analizzare gli **errori** più frequenti in modo tale da capire i limiti del sistema nei confronti delle lingue morte.

1. Implementazione del Learning

Le probabilità di transizione ed emissione vengono calcolate indipendentemente mediante i **train set** del greco e latino.

1.1 Probabilità di transizione

La probabilità di transizione rappresenta la probabilità di un tag dato il tag precedente. I **conteggi** vengono definiti rispettivamente dalla formula della probabilità:

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

$$C(t_{i-1}) = \text{numero di volte che il tag } t_{i-1} \text{ compare nel train set}$$

$$C(t_{i-1}, t_i) = \text{numero di volte che il tag } t_{i-1} \text{ compare prima del tag } t_i \text{ all'interno del train set}$$

```
# sent_id = train-s1749
# text = quia inter nos taliter convinet a Rachisindo notario iscrivere rogavit.
# reference = document_id='73:20'-span='110'
1   quia        quia        SCONJ
2   inter       inter       ADP i-1
3   nos         nos         PRON i
4   taliter     taliter     ADV
5   convinet    conuenio    VERB
6   a           ab          ADP
7   Rachisindo  Rachisindus PROPN
8   notario     notarius    NOUN
9   iscrivere   scribo      VERB
10  rogavit     rogo        VERB
11      .           .       PUNCT
```

Pertanto, nel file del train set, letto attraverso la libreria ***pyconll***, verrà considerata solo la lista dei tag relativi ad una sentence.

`train = pyconll.load_from_file('la_llct-ud-train.conllu')`

Ogni parola della frase ha un attributo ***upos*** mediante il quale è possibile ricavare il Pos Tag. Di seguito la funzione per calcolare la probabilità di transizione tra due tag:

```
def compute_transition_probability(train,tag1,tag2):
   count_t1_before_t2 = 0
   count_t1 = 0
   for sentence in train:
       for i in range (len(sentence)):
           if sentence[i-1].upos == tag1 and sentence[i].upos == tag2 and i != 0:
               count_t1_before_t2 = count_t1_before_t2 + 1
           if sentence[i].upos == tag1:
               count_t1 = count_t1 + 1
   return count_t1_before_t2/count_t1
```

La funzione **compute_transition_probability** calcola la probabilità di transizione del tag2 dato il tag1, cioè conta quante volte il tag1 compare prima del tag2 e quante volte è presente il tag1 in tutte le **sentence** del train set. Restituisce la divisione tra i due conteggi.

1.2 Matrice di transizione

Per la memorizzazione delle probabilità di transizione calcolate come descritto nel capitolo precedente 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 antico.

Infatti avremo una lista di ***possible_tags***:

- Pos Tags Latino: [*'ADJ','ADP', 'ADV', 'AUX', 'CCONJ', 'DET', 'NOUN', 'NUM', 'PART', 'PRON','PROPN','PUNCT', 'SCONJ', 'VERB','X'*]
- Pos Tags Greco: [*'ADJ','ADP', 'ADV', 'CCONJ', 'DET', 'INTJ', 'NOUN', 'NUM', 'PART', 'PRON','SCONJ', 'VERB', 'X', 'PUNCT'*]

Di seguito l'algoritmo utilizzato per popolare la matrice di transizione che sfrutta la funzione vista in precedenza per calcolare le probabilità di transizione relative a due tag. Il calcolo viene effettuato per ogni coppia di tag presente nella lista ***possible_tags***:

```
def compute_trasition_matrix(possible_tags,train):
    transition_matrix = np.zeros((len(possible_tags), len(possible_tags)),
dtype='float32')
    for i,t1 in enumerate(possible_tags):
        for j,t2 in enumerate(possible_tags):
            transition_matrix[i][j] =
compute_transition_probability(train,t1,t2)
    return transition_matrix
```

Successivamente la matrice viene trasformata in un **DataFrame** per poter etichettare le righe e le colonne con i Pos Tag.

Un esempio di matrice di transizione relativa al train set per il Latino:

	ADJ	ADP	ADV	AUX	CCONJ	DET	NOUN	NUM	PART	PRON	PROPN	PUNCT	SCONJ	VERB	X
ADJ	0.089186	0.022273	0.005023	0.034594	0.047578	0.006729	0.283196	0.000758	0.000284	0.008435	0.213819	0.252867	0.000569	0.034499	0.000190
ADP	0.090429	0.002025	0.008323	0.000000	0.000000	0.244686	0.353729	0.006355	0.000000	0.105331	0.147677	0.000562	0.000169	0.040659	0.000056
ADV	0.011566	0.214399	0.048576	0.017782	0.013879	0.043516	0.080382	0.005060	0.029059	0.061009	0.129536	0.032529	0.010843	0.301720	0.000145
AUX	0.089726	0.109915	0.013459	0.000000	0.026021	0.018394	0.061014	0.001795	0.000449	0.154329	0.014805	0.275011	0.005384	0.229699	0.000000
CCONJ	0.054470	0.157828	0.050239	0.004051	0.007563	0.089943	0.263798	0.029621	0.010354	0.037364	0.054560	0.004952	0.059242	0.175925	0.000090
DET	0.061589	0.056543	0.076478	0.001415	0.024611	0.055128	0.447487	0.007629	0.000554	0.033225	0.076724	0.060604	0.014705	0.083308	0.000000
NOUN	0.101177	0.092977	0.023564	0.007742	0.076263	0.151512	0.082509	0.018716	0.000892	0.027326	0.077879	0.180141	0.007091	0.151778	0.000434
NUM	0.015735	0.189908	0.034183	0.003256	0.051546	0.010309	0.421595	0.022789	0.000000	0.003798	0.000543	0.185567	0.001085	0.059685	0.000000
PART	0.000000	0.015009	0.011257	0.223265	0.043152	0.001876	0.071295	0.000000	0.000000	0.009381	0.001876	0.000000	0.000000	0.622889	0.000000
PRON	0.005215	0.082030	0.049471	0.020414	0.039115	0.100954	0.139920	0.009835	0.001714	0.079049	0.253241	0.020861	0.003129	0.191030	0.004023
PROPN	0.032700	0.044267	0.019945	0.002438	0.049331	0.012192	0.373515	0.000813	0.000313	0.026260	0.008128	0.371764	0.002563	0.055771	0.000000
PUNCT	0.021844	0.072242	0.029219	0.000454	0.041871	0.029498	0.206801	0.001957	0.000454	0.173389	0.019223	0.002551	0.055956	0.089753	0.000035
SCONJ	0.012817	0.260805	0.219374	0.002086	0.004471	0.070045	0.035469	0.002385	0.002385	0.200894	0.014307	0.047988	0.016393	0.110581	0.000000
VERB	0.012123	0.165639	0.018164	0.037119	0.115189	0.016997	0.088735	0.008040	0.004749	0.053699	0.037494	0.291160	0.012415	0.138477	0.000000
X	0.012658	0.050633	0.000000	0.000000	0.000000	0.000000	0.240506	0.000000	0.000000	0.012658	0.000000	0.493671	0.012658	0.177215	0.000000

1.3 Probabilità di transizione iniziali

In una struttura dati separata sono state memorizzate le probabilità di transizione iniziali, cioè le probabilità che da uno stato iniziale ***START*** si passi ad uno stato relativo ad uno dei Pos Tag presenti nella lista ***possible_tags***.

Ad esempio, per il Latino:

	ADJ	ADP	ADV	AUX	CCONJ	DET	NOUN	NUM	PART	PRON	PROPN	PUNCT	SCONJ	VERB	X
START	0.046508	0.022911	0.070243	0.000412	0.191384	0.034984	0.139388	0.001372	0.000549	0.025792	0.009878	0.378241	0.003841	0.074359	0.000137

1.4 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:

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

$C(t_i, w_i)$ = numero di volte che la parola w_i è taggata con il tag t_i nel train set

$C(t_i)$ = numero di volte che il tag t_i compare nel train set

1.5 Dizionario di emissione

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.

Inoltre, il calcolo avviene a priori per tutte le parole del train set in modo tale da NON costruire un dizionario in maniera dinamica ogni volta che viene analizzata una sentence e rendere l'algoritmo di Decoding, presentato in un capitolo successivo, più veloce.

La seguente funzione restituisce due dizionari: il **dizionario di emissione** e quello delle parole con il loro relativo conteggio all'interno del train set. Questo secondo dizionario servirà nella fase di Decoding per verificare se una parola della sentence analizzata è sconosciuta o no.

```
def compute_emission_probabilities(train):
    word_tag_set = []
    tags_set = []
    words_set = []
    for sentence in 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
```

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

Un estratto del dizionario di emissione associato al train set del Latino:

```
{...,('Dei', 'PROPN'): 0.057208953357509064, ('nomine', 'NOUN'):
0.01572524239062274, ('regnante', 'VERB'): 0.015205799033494418, ('domno',
'NOUN'): 0.014205778785393855, ('nostro', 'DET'): 0.035255029840644804,
('Carulo', 'PROPN'): 0.004689258471926972, ('rege', 'NOUN'):
0.0035695335487916646, ('Francorum', 'NOUN'): 0.0027977425112150887,...}
```

Attraverso gli attributi ***form*** e ***upos*** è possibile accedere rispettivamente alla stringa del token della sentence in oggetto e al Pos Tag assegnato nel train set.

2. Implementazione del Decoding

La fase di Decoding permette di capire qual è l'algoritmo che permette di applicare le probabilità apprese nella fase di Learning per poter restituire la sequenza di tag ottimale per una determinata frase in input.

Dal momento che, la soluzione ottimale è la **sequenza che massimizza la probabilità** vista nel capitolo 0, per poterla ottenere si dovrebbero provare tutte le possibili combinazioni di tag associate ad una frase, calcolare la probabilità e scegliere quella massima. E' subito chiaro che c'è un problema! C'è un esplosione di casi e si avrebbe una **complessità esponenziale**.

La soluzione è quella di applicare la **dynamic programming** e passare da una complessità esponenziale ad una **polinomiale** attraverso l'**approssimazione Markoviana**.

Le due assunzioni fondamentali dell'HMM sono:

- La probabilità che una parola appaia dipende solo dal suo stesso tag ed è indipendente dalle parole vicine e dagli altri tag
- La probabilità di un tag dipende solo dal tag precedente anzichè che dall'intera sequenza di tag precedenti

L'idea è che la programmazione dinamica permette di non ripetere gli stessi calcoli più volte e consente di memorizzare i dati parziali, passando da una complessità temporale ad una spaziale.

L'**algoritmo di Viterbi** permette di fare tutto ciò, e nel capitolo seguente verrà spiegata la sua implementazione all'interno del progetto.

2.1 Algoritmo di Viterbi

L'algoritmo di Viterbi implementato è una **variante** di quello tradizionale, ma con la stessa finalità. Si cercheranno di fare analogie con il secondo in modo tale da rendere più efficace la comprensione.

Input dell'algoritmo:

- `sentence_tokens`: una lista di parole della frase analizzata
- `possible_tags`: lista dei possibili tag relativi dal train set utilizzato (Greco o Latino)
- `transition_matrix`
- `emission_probabilities`: dizionario delle probabilità di emissione
- `initial_transition_probabilities`
- `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

Per la realizzazione dell'algortimo non viene costruita una **matrice di Viterbi**, ma le colonne vengono simulate attraverso un **vettore** `p` dichiarato all'inizio del ciclo for il quale analizza una singola parola.

L'algoritmo si compone quindi di due cicli, uno più esterno che scorre le parole della frase in input e uno interno che scorre su tutti i tag, ovvero gli **stati**.

Gli stati con le probabilità della matrice (i valori di Viterbi) vengono simulati e memorizzati nel vettore `p`, il quale si svuota ad ogni iterazione su una singola parola.

Il vettore `states` è utilizzato per tenere traccia della sequenza di tag da restituire: ad ogni iterazione su una parola, viene scelto il tag dalla lista `possible_tags` in base all'indice del valore di Viterbi massimo nel vettore `p`, e questo viene inserito nella lista `states`.

```
def viterbi_algorithm(sentence_tokens, possible_tags, transition_matrix,
                      emission_probabilities, initial_transition_probabilities,
                      count_word,smoothing_strategy,
                      oneshot_words_tag_distribution):
    states = []
    for key,word in enumerate(sentence_tokens):
        p = []
        for t,tag in enumerate(possible_tags):
            emission_p = 0
            if key == 0:
                trasition_p = initial_transition_probabilities.loc['START',tag]
            else:
                trasition_p = transition_matrix.loc[states[-1]][tag]
            try:
                count_word[word]
            except KeyError: #unknown_word
                emission_p = unknown_word_emission_p(smoothing_strategy, tag,
possible_tags, oneshot_words_tag_distribution)
            emission_p = emission_probabilities.get((word,tag),emission_p)
            if emission_p != 0 and trasition_p != 0:
                state_probability = math.log(emission_p) + math.log(trasition_p)
            else:
                state_probability = -sys.maxsize
            p.append(state_probability)
        pmax = max(p)
        state_max = possible_tags[p.index(pmax)]
        states.append(state_max)
    return states
```

All'interno del ciclo più annidato vengono ricavate le probabilità di **transizione** ed **emissione**.

- Se l'indice della parola da analizzare è pari a 0 vuol dire che la probabilità di transizione è quella iniziale, memorizzata nella matrice `initial_transition_probabilities`, altrimenti viene ricavata prendendo in considerazione il tag per il quale il valore di viterbi della colonna precedente è massimo (ultimo elemento della lista `states`) e il tag corrente; questi due indici vengono usati per ricavare la probabiltà all'interno della matrice di transizione `transition_matrix` .
- La probabilità di emissione viene ricavata dal dizionario di emissione `emission_probabilities` se la parola considerata è conosciuta, altrimenti verrà generato un `KeyError` ; nel blocco di eccezione la funzione `unknown_word_emission_p` deciderà quale probabilità di emissione restituire in base alla **strategia di smoothing** utilizzata.
- La probabilità di uno stato sarà la somma dei logaritmi della due probabilità se entrambe sono diverse da 0, altrimenti avrà valore uguale a -infinito.

3. Strategie di Smoothing

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

- $P(unk|NOUN) = 1$: 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 quella parola sconosciuta è un **sostantivo**
- $P(unk|NOUN) = P(unk|VERB) = 0.5$: 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
- $P(unk|t_i) = \frac{1}{|POSTAGs|}$: la proabilità di emissione per qualsiasi tag sarà la 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
- **Statistica PoS sul development set**: 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 associerà 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 0.

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

  ```
  [('NOUN', 0.19340159271899887), ('PROPN', 0.38680318543799774), ('VERB',
  0.229806598407281), ('ADJ', 0.080773606370876), ('CCONJ',
  0.0011376564277588168), ('DET', 0.051194539249146756), ('NUM',
  0.012514220705346985), ('ADP', 0.009101251422070534), ('ADV',
  0.022753128555176336), ('PRON', 0.004550625711035267), ('AUX',
  0.005688282138794084), ('SCONJ', 0.0011376564277588168), ('PART',
  0.0011376564277588168), ('PUNCT', 0), ('X', 0)]
  ```

  E' possibile notare un'alta percentuale di nomi propri (PROPN).

4. Valutazione del Sistema e Analisi degli errori

4.1 Risultati per il Latino

Baseline

L'algoritmo di Baseline ottiene **ottimi** risultati con un accuratezza di quasi 96%. I vantaggi di questo algoritmo sono l'estrema semplicità e la velocità di esecuzione, infatti risulta essere quasi istantaneo. Gli errori sono principalmente relativi alla valutazione dei **nomi propri** e **verbi**.

```
Pos Tag corretti:  23220
Pos Tag sbagliati:  969
Totale parole valutate:  24189
Accuratezza:  95.99 %
Conteggi errori:  {'PROPN': 351, 'VERB': 234, 'DET': 184, 'ADV': 85, 'NOUN': 7,
'SCONJ': 23, 'AUX': 6, 'CCONJ': 12, 'ADP': 14, 'ADJ': 20, 'NUM': 28, 'PRON': 5}
Tempo di esecuzione:  0.08  sec
```

Viterbi

L'algoritmo di Viterbi ottiene risultati leggermente migliori del Baseline per quanto riguarda tre 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 sono i **nomi propri** e i **verbi**; 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 "l'azzardo" effettuato dalla strategia di smoothing utilizzata.

```
Tipologia di smoothing:  UNKNOWN_NAME
Pos Tag corretti:  23179
Pos Tag sbagliati:  1010
Totale parole valutate:  24189
Accuratezza:  95.82 %
Conteggi errori:  {'PROPN': 354, 'VERB': 191, 'DET': 161, 'ADV': 82, 'AUX': 66,
'PRON': 60, 'NOUN': 9, 'SCONJ': 23, 'CCONJ': 6, 'ADP': 17, 'ADJ': 15, 'NUM': 23,
'PUNCT': 2, 'X': 1}
Tempo di esecuzione:  27.99  sec
```

```
Tipologia di smoothing:  UNKNOWN_NAME_VERB
Pos Tag corretti:  23221
Pos Tag sbagliati:  968
Totale parole valutate:  24189
Accuratezza:  96.00 %
Conteggi errori:  {'PROPN': 354, 'VERB': 142, 'DET': 161, 'ADV': 82, 'AUX': 66,
'NOUN': 24, 'PRON': 52, 'SCONJ': 23, 'CCONJ': 6, 'ADP': 17, 'ADJ': 15, 'NUM': 23,
'PUNCT': 2, 'X': 1}
Tempo di esecuzione:  28.28  sec
```

```
Tipologia di smoothing:  UNKNOWN_ALL
Pos Tag corretti:  23244
Pos Tag sbagliati:  945
Totale parole valutate:  24189
Accuratezza:  96.09 %
Conteggi errori:  {'PROPN': 289, 'VERB': 182, 'DET': 161, 'ADV': 82, 'AUX': 67,
'NOUN': 26, 'PRON': 52, 'SCONJ': 23, 'CCONJ': 6, 'ADP': 16, 'ADJ': 15, 'NUM': 23,
'PUNCT': 2, 'X': 1}
Tempo di esecuzione:  28.50  sec
```

```
Tipologia di smoothing:  UNKNOWN_DISTRIBUTION_ONESHOT_WORDS
Pos Tag corretti:  23307
Pos Tag sbagliati:  882
Totale parole valutate:  24189
Accuratezza:  96.35 %
Conteggi errori:  {'VERB': 152, 'DET': 161, 'ADV': 82, 'AUX': 67, 'PROPN': 255,
'NOUN': 26, 'PRON': 52, 'SCONJ': 23, 'CCONJ': 6, 'ADP': 17, 'ADJ': 15, 'NUM': 23,
'PUNCT': 2, 'X': 1}
Tempo di esecuzione:  27.03  sec
```

4.2 Risultati per il Greco

Baseline

Le performance per il Greco calano di molto. L'algoritmo di Baseline ha un'accuratezza del 73.5 % 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
Accuratezza:  73.53 %
Conteggi errori:  {'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}
Tempo di esecuzione:  0.07  sec
```

Viterbi

L'algoritmo di Viterbi, attraverso le strategie di smoothing, migliora le performance di molto che rimangono comunque molto basse. Infatti l'accuratezza oscilla tra il 72 e il 76%. Anche in questo caso la strategia di smoothing relativa alla distribuzione di probabilità delle parole che compaiono una sola volta nel dev-set risulta la migliore;

```
Tipologia di smoothing:  UNKNOWN_NAME
Pos Tag corretti:  15504
Pos Tag sbagliati:  5455
Totale parole valutate:  20959
Accuratezza:  73.97 %
Conteggi errori:  {'VERB': 1974, 'ADV': 1664, 'PRON': 476, 'ADJ': 969, 'CCONJ':
130, 'DET': 113, 'SCONJ': 51, 'NOUN': 48, 'ADP': 19, 'PUNCT': 6, 'NUM': 1,
'INTJ': 3, 'X': 1}
Tempo di esecuzione:  23.13  sec
```

```
Tipologia di smoothing:  UNKNOWN_NAME_VERB
Pos Tag corretti:  15859
Pos Tag sbagliati:  5100
Totale parole valutate:  20959
Accuratezza:  75.67 %
Conteggi errori:  {'ADV': 1663, 'PRON': 459, 'VERB': 748, 'NOUN': 943, 'ADJ':
970, 'CCONJ': 130, 'DET': 113, 'SCONJ': 44, 'ADP': 19, 'PUNCT': 6, 'NUM': 1,
'INTJ': 3, 'X': 1}
Tempo di esecuzione:  23.09  sec
```

```
Tipologia di smoothing:  UNKNOWN_ALL
Pos Tag corretti:  15158
Pos Tag sbagliati:  5801
Totale parole valutate:  20959
Accuratezza:  72.32 %
Conteggi errori:  {'VERB': 985, 'ADV': 1630, 'PRON': 442, 'NOUN': 1234, 'ADJ':
970, 'CCONJ': 131, 'DET': 132, 'PUNCT': 200, 'SCONJ': 53, 'ADP': 19, 'NUM': 1,
'INTJ': 3, 'X': 1}
Tempo di esecuzione:  23.20  sec
```

```
Tipologia di smoothing:  UNKNOWN_DISTRIBUTION_ONESHOT_WORDS
Pos Tag corretti:  15928
Pos Tag sbagliati:  5031
Totale parole valutate:  20959
Accuratezza:  76.00 %
Conteggi errori:  {'ADV': 1663, 'PRON': 461, 'VERB': 586, 'NOUN': 1035, 'ADJ':
970, 'CCONJ': 130, 'DET': 113, 'SCONJ': 43, 'ADP': 19, 'PUNCT': 6, 'NUM': 1,
'INTJ': 3, 'X': 1}
Tempo di esecuzione:  22.22  sec
```