ALTAMURA SANTE - 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:

- **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** calcolate 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

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
```

Ogni parola della frase ha un attributo ***upos*** e un attributo **form** mediante i quali è possibile ricavare rispettivamente il Pos Tag e la forma (token) della parola stessa.

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 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 Tags Latino:['START','ADJ','ADP', 'ADV', 'AUX', 'CCONJ', 'DET', 'NOUN',
                   'NUM', 'PART', 'PRON','PROPN','PUNCT',
  'SCONJ','VERB','X','END']
  ```

- ```
  Pos Tags Greco: ['START','ADJ','ADP', 'ADV', 'CCONJ', 'DET', 'INTJ', 'NOUN',
                     'NUM', 'PART', 'PRON','SCONJ', 'VERB', 'X', 'PUNCT','END']
  ```

Di seguito l'algoritmo `compute_transition_matrix` nel file **learning2.py** 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')

    transition_counter_dict = dict()
    counter_dict = dict()
    count_initial_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 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 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
```

- Nella **prima fase** 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.
    - count_initial_dict: dizionario relativo ai 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.
- Nella **seconda fase** i tre dizionari vengono popolati 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.

- Nella **terza fase** la matrice di transizione viene riempita con le probabilità di transizione. La probabilità del tag2 dato il tag1 è memorizzata nella cella delle matrice avente come riga il tag1 e come colonna il tag2.

  In particolare vengono effettuate due operazioni:

  1. Viene popolata la prima riga della matrice **(riquadro verde)**. 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.
  2. Vengono popolate le righe successive alla prima **(riquadro blu)** con le probabilità di transizione calcolate come è stato descritto nel capitolo precedente, 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 **(riquadro rosso)**. E' importante notare che sono tutte pari a 0 tranne quella relativa a PUNCT; questo accade perchè tutte le sentence del train set terminano con un simbolo di punteggiatura.

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:

	START	ADJ	ADP	ADV	AUX	CCONJ	DET	NOUN	NUM	PART	PRON	PROPN	PUNCT	SCONJ	VERB	X	END
START	0.0	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	0.000000
ADJ	0.0	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	0.000000
ADP	0.0	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	0.000000
ADV	0.0	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	0.000000
AUX	0.0	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	0.000000
CCONJ	0.0	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	0.000000
DET	0.0	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	0.000000
NOUN	0.0	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	0.000000
NUM	0.0	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	0.000000
PART	0.0	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	0.000000
PRON	0.0	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	0.000000
PROPN	0.0	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	0.000000
PUNCT	0.0	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	0.244303
SCONJ	0.0	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	0.000000
VERB	0.0	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	0.000000
X	0.0	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	0.000000
END	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

1.3 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

1.4 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.

La funzione `compute_emission_probabilities` nel file **learning2.py** restituisce tre dizionari:

- il **dizionario di emissione**, descritto a breve.
- il **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 o no
- il **dizionario delle parole** conteggiate con i **pos tag**. E' 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.

Di seguito, un piccolo 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,...}
```

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.

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.

2.1 Algoritmo di Viterbi

L'algoritmo di Viterbi implementato è una **leggera "variante"** di quello tradizionale in quanto utilizza diverse strutture dati, ma con la stessa finalità. L'algoritmo, con le sue funzioni di supporto, è implementato nel file **viterbi.py**.

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`: 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

Strutture dati principali:

```
viterbi_matrix = np.zeros((len(possible_tags), len(sentence_tokens)))
backpointer = dict()
```

- `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 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.

L'algoritmo è diviso in 4 fasi:

1.Inizializzazione della prima colonna

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.

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.

```
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)
```

2.Calcolo delle colonne successive

Vengono calcolati i valori di Viterbi degli stati delle colonne successive alla prima e man mano viene popolato il dizionario `backpointer`.

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.

```
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
```

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

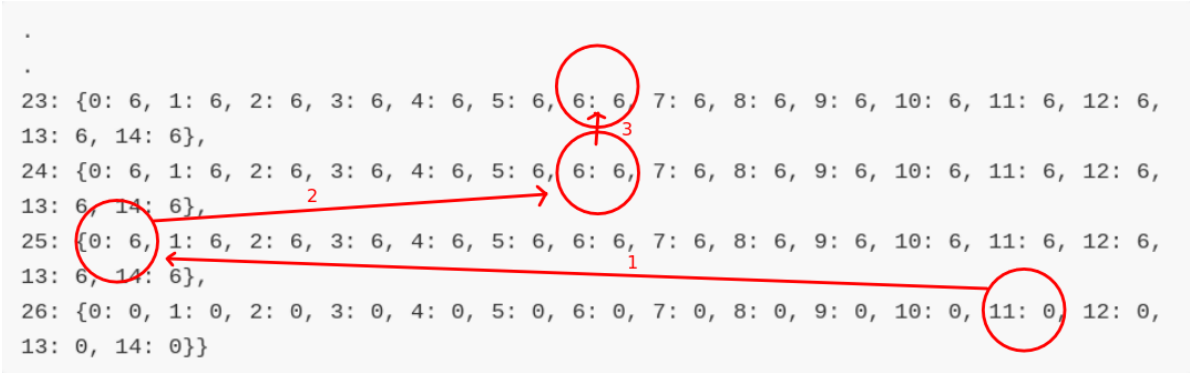
```
max_ = -sys.maxsize
best_path_pointer = None
for s,tag in enumerate(possible_tags):
end_transition = transition_matrix.loc[tag,'END']
if end_transition == 0: end_transition = np.finfo(float).tiny
val = viterbi_matrix[s,len(sentence_tokens) - 1] + math.log(end_transition)
if val >= max_: max_ = val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

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

4.Backtracking (riga 50 del codice)

Partendo dal **best_path_pointer** viene srotolata la sequenza dei tag più probabile sfruttando il dizionario `backpointer`. Attraverso la seguente immagine possiamo intuire facilmente il processo:

`best_path_pointer = 11`



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 la 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 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 assocerà ad ogni tag una probabilità e la somma di tutte le probabilità sarà 1. Nell'algoritmo `compute_oneshot_words_distributions` nel file **learning2.py**, 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 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) 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.

4. Valutazione del Sistema e Analisi degli errori

4.1 Risultati per il Latino

Baseline

L'algoritmo di Baseline 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**.

```
Accuratezza:  95.39 %
Conteggi errori:  {'VERB': 248, 'PROPN': 471, 'ADV': 56, 'DET': 142, 'NUM': 34,
'ADJ': 83, 'NOUN': 15, 'CCONJ': 35, 'ADP': 6, 'SCONJ': 15, 'AUX': 3, 'PRON': 2}
Tempo di esecuzione:  0.52  sec
```

Viterbi

L'algoritmo di Viterbi ottiene risultati leggermente migliori del Baseline per quanto riguarda le 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 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 "l'azzardo" effettuato 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 su cui azzardano; nel primo smoothing, il conteggio degli errori relativi ai nomi è molto basso, nel secondo invece aumenta ma diminuisce quello relativo ai verbi.

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
Accuratezza:  95.98 %
Conteggi errori:  {'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}
Tempo di esecuzione:  61.71  sec
```

```
Tipologia di smoothing:  UNKNOWN_NAME_VERB
Accuratezza:  96.22 %
Conteggi errori:  {'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}
Tempo di esecuzione:  59.65  sec
```

```
Tipologia di smoothing:  UNKNOWN_ALL
Accuratezza:  96.42 %
Conteggi errori:  {'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}
Tempo di esecuzione:  58.13  sec
```

```
Tipologia di smoothing:  UNKNOWN_DISTRIBUTION_ONESHOT_WORDS
Accuratezza:  97.22 %
Conteggi errori:  {'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}
Tempo di esecuzione:  58.18  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**.

```
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:  0.49  sec
```

Viterbi

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
Accuratezza:  73.60 %
Conteggi errori:  {'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}
Tempo di esecuzione:  43.34  sec
```

```
Tipologia di smoothing:  UNKNOWN_NAME_VERB
Accuratezza:  76.43 %
Conteggi errori:  {'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}
Tempo di esecuzione:  43.31  sec
```

```
Tipologia di smoothing:  UNKNOWN_ALL
Accuratezza:  73.94 %
Conteggi errori:  {'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}
Tempo di esecuzione:  43.07  sec
```

```
Tipologia di smoothing:  UNKNOWN_DISTRIBUTION_ONESHOT_WORDS
Accuratezza:  76.25 %
Conteggi errori:  {'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}
Tempo di esecuzione:  43.81  sec
```