Relazione TLN 2021/2022

Matricola **836201**

Nome Cognome: **Luca Frigato**

NER Tagger

Introduzione

Lo scopo del progetto è quello di implementare un sistema di Named Entity Recognition (NER) utilizzando il modello Hidden Markov Model (HMM). In particolare, il progetto prevede di implementare le fasi di Learning e Decoding del modello HMM, addestrare il sistema su due dataset di testo in lingua inglese e italiana provenienti da Wikipedia, valutare il sistema utilizzando diverse strategie di smoothing per entrambe le lingue e confrontarlo con una baseline facile e una difficile. L'obiettivo finale è di valutare l'efficacia del sistema NER implementato con HMM rispetto ad altre tecniche di NER e ad alcune baseline.

Dataset Wikineural

Il dataset utilizzato per addestrare il sistema di Named Entity Recognition (NER) è stato preso dalle risorse di Wikineural di Babelscape. Si tratta di due dataset, uno per la lingua inglese e uno per la lingua italiana, entrambi basati su articoli di Wikipedia.

Il dataset in lingua inglese contiene un totale di circa 116k frasi, mentre il dataset in lingua italiana contiene circa 111k frasi. Ogni frase è stata annotata manualmente con le entità nominate (NER), che sono:

- B-PER: inizio di un'entità nominata di tipo persona
- I-PER: entità nominata di tipo persona
- B-ORG: inizio di un'entità nominata di tipo organizzazione
- I-ORG: entità nominata di tipo organizzazione
- B-LOC: inizio di un'entità nominata di tipo luogo
- I-LOC: entità nominata di tipo luogo
- B-MISC: inizio di un'entità nominata di tipo vario (miscellanea)
- I-MISC: entità nominata di tipo vario (miscellanea)
- O: parole che non fanno parte di nessuna entità nominata

Le frasi nei dataset sono state pre-elaborate in modo da rimuovere la punteggiatura e le righe vuote. Le frasi sono state quindi convertite in sequenze di parole, ognuna delle quali è stata etichettata con il rispettivo tag NER. Questo formato di input è stato utilizzato per l'addestramento del sistema NER.

Il dataset di Wikineural è stato scelto per la sua qualità e la varietà delle entità nominate presenti. Inoltre, essendo basato su Wikipedia, il dataset copre una vasta gamma di argomenti e stili di scrittura, il che lo rende un'ottima fonte di dati di addestramento per il NER.

Analisi esplorativa del dataset

Molte caratteristiche del dataset possono influenzare le prestazioni del modello di Named Entity Recognition (NER), come **la lunghezza delle frasi,** la **distribuzione dei tag NER** e la **frequenza delle parole**.

La distribuzione della lunghezza delle frasi può influenzare la scelta della lunghezza massima della sequenza da utilizzare durante il training del modello. Questo perché il modello HMM utilizza **la probabilità condizionata** delle parole date le etichette nascoste (i.e**. i tag NER**) per effettuare le predizioni. Se la sequenza di parole è troppo lunga, il modello può avere difficoltà a catturare tutte le informazioni pertinenti per effettuare una predizione accurata. Al contrario, se la sequenza di parole è troppo corta, il modello può non avere abbastanza informazioni per distinguere correttamente le diverse entità nominate. Pertanto, la distribuzione della lunghezza delle frasi potrebbe essere utilizzata per determinare la lunghezza massima della sequenza da utilizzare durante il training del modello HMM per il task di NER.

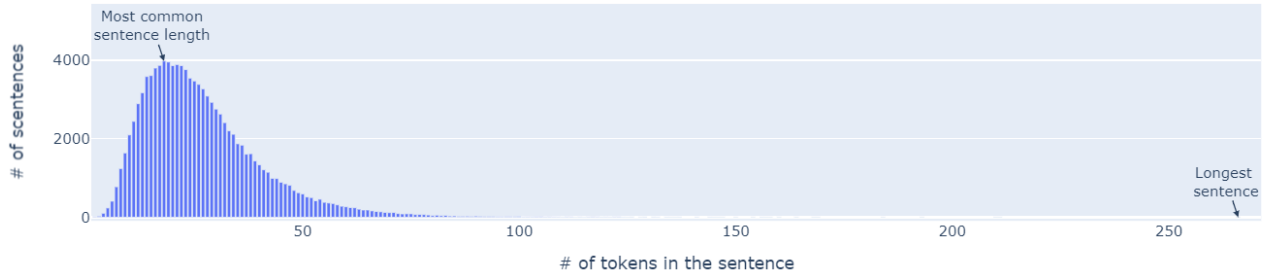

Figura 1 Gaussiana che rappresenta la distribuzione della lunghezza delle frasi
Valore medio: 18 parole per frase – Lingua Italiana

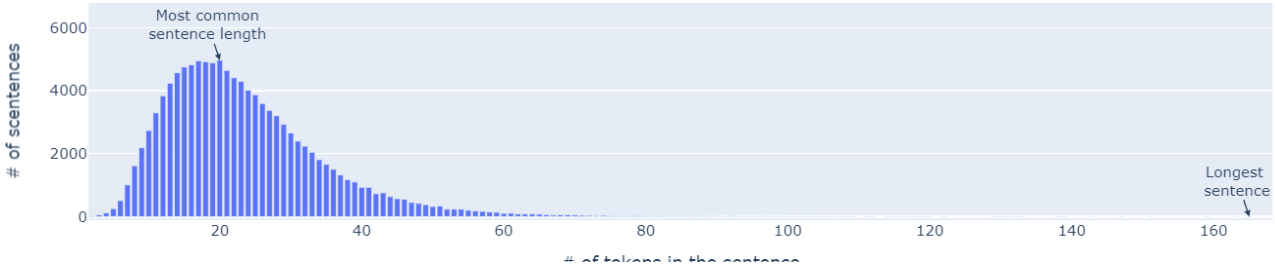

Figura 2 Gaussiana che rappresenta la distribuzione della lunghezza delle frasi

Valore medio: 20 parole per frase – Lingua Italiana

La distribuzione dei tag NER si riferisce alla proporzione di istanze di ogni classe di entità nominata all'interno del dataset. Se questa distribuzione non fosse bilanciata, il modello potrebbe apprendere a predire in modo sbilanciato e meno accurato per le classi meno rappresentate. Per questo motivo, è possibile utilizzare tecniche di upsampling (aumentare il numero di istanze delle classi meno rappresentate) o downsampling (ridurre il numero di istanze delle classi più rappresentate) per bilanciare le classi durante il training del modello. In alternativa, sarebbe possibile utilizzare tecniche di pesatura dei dati, dove viene assegnato un peso maggiore alle istanze delle classi meno rappresentate per dare loro maggiore importanza durante il training.

Figura 3 Distribuzione TAG sul dataset combinato

"O" è prevalente sul resto, "B-LOC" è rappresentato il 50% in più di "B-PER", ancora meno gli altri

- Lingua Italiana

Figura 4Figura 1 Distribuzione TAG sul dataset combinato
"O" è prevalente sul resto, "B-LOC" e "B-PER" sono equamente rappresentati, meno gli altri

- Lingua Inglese

La frequenza delle parole uniche si riferisce alla quantità di parole presenti nel dataset che compaiono solo una volta. Queste parole uniche possono essere una fonte di rumore per il modello di NER, poiché il modello potrebbe avere difficoltà a generalizzare a queste parole che non hanno molte occorrenze nel training set. Inoltre, se il numero di parole uniche è troppo elevato, la complessità del modello aumenterà, portando ad una possibile overfitting.

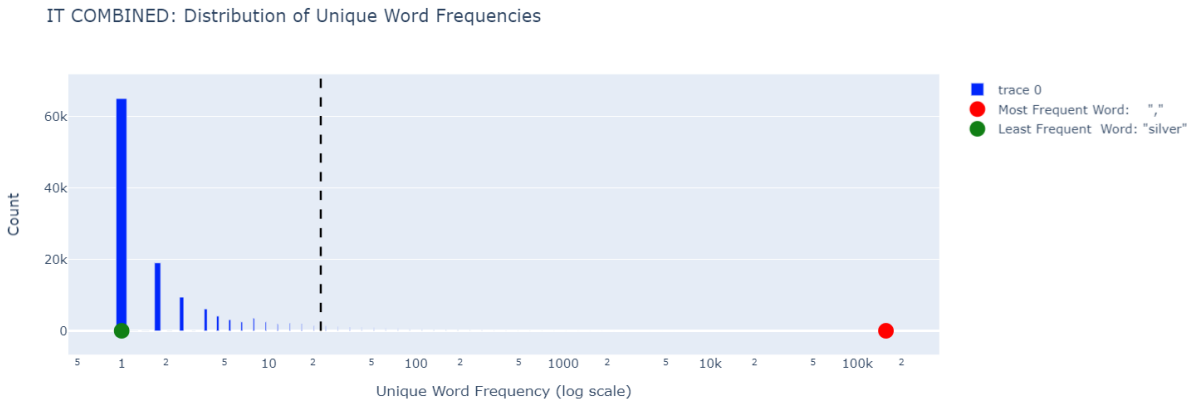

*Figura 5 Lingua Italiana - Circa 64k parole uniche compaiono una sola volta, mentre in media una parola è ripetuta almeno 22 volte tra le varie frasi. La meno frequente è **Silver**, la più frequente è la **virgola***

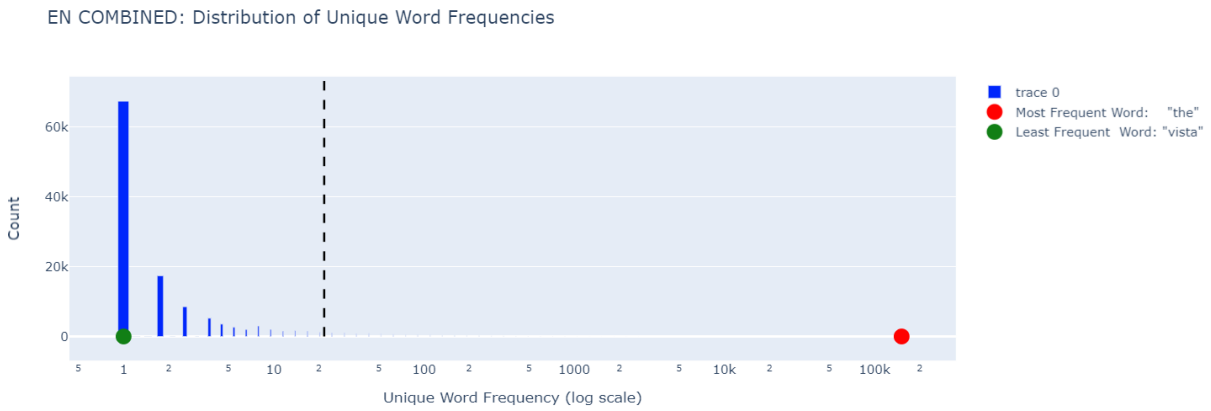

*Figura 6 Lingua Inglese - Circa 67k parole uniche compaiono una sola volta, mentre in media una parola è ripetuta almeno 21 volte tra le varie frasi. La meno frequente è **visita**, la più frequente è **the***

La correlazione tra la frequenza delle parole uniche e la distribuzione dei tag NER può essere utile in diversi modi. Ad esempio, può aiutare a identificare parole che sono particolarmente rilevanti per alcune classi di tag, ad esempio nomi propri o termini tecnici. Queste parole potrebbero essere incluse in una lista di stopword, oppure potrebbe essere applicata una tecnica di pesatura per assegnare loro maggior importanza durante la fase di tagging. Inoltre, l'analisi della correlazione tra la frequenza delle parole uniche e la distribuzione dei tag NER può aiutare a identificare eventuali errori o ambiguità nel dataset, come ad esempio termini che vengono etichettati con classi di tag sbagliate.

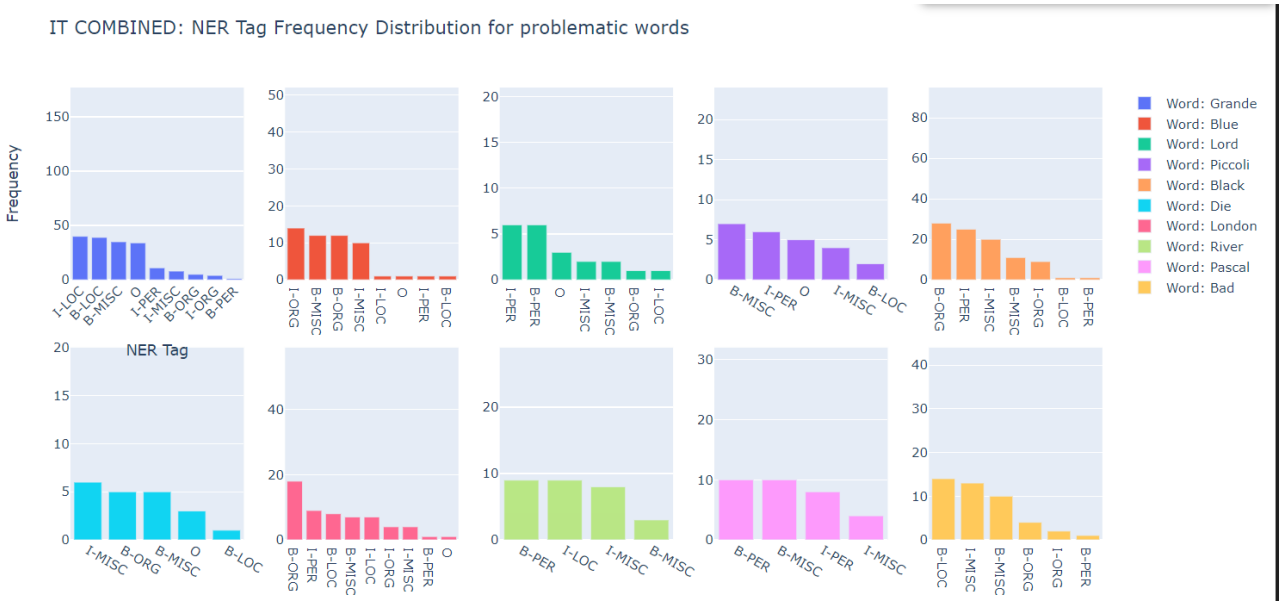

Figura 7 Un gruppo di parole 'problematiche' per il NER, in quanto nelle varie frasi sono etichettate con tag NER diversi e non esiste una vera dominanza di uno tra gli altri - Lingua Italiana

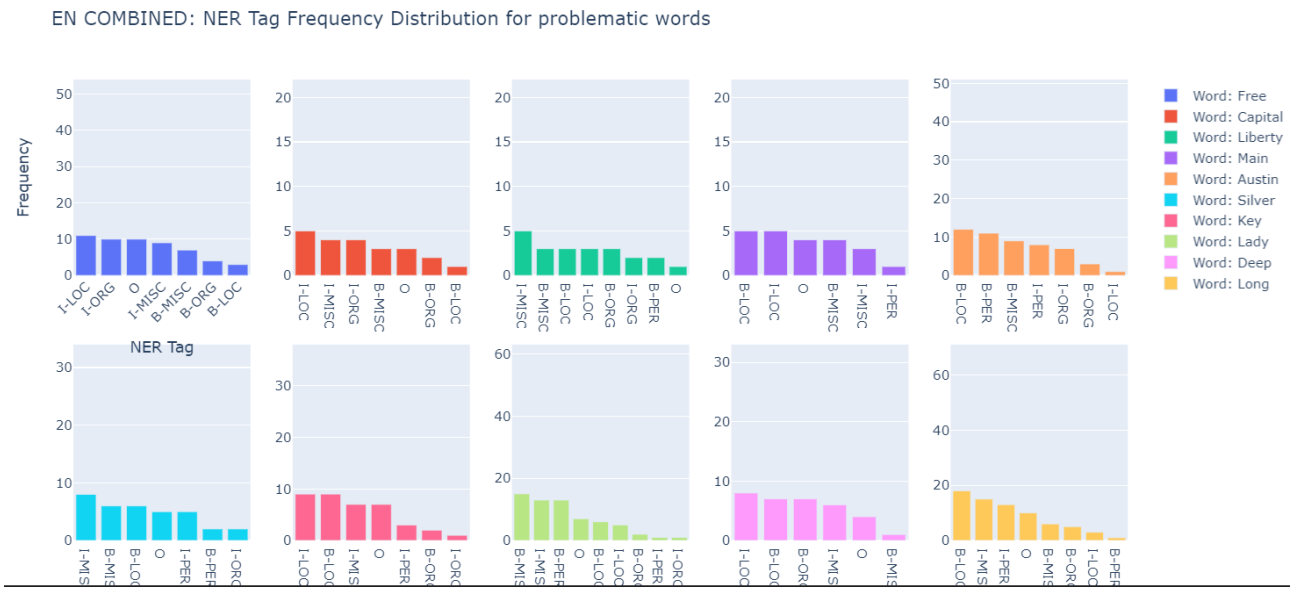

Figura 8 Un gruppo di parole 'problematiche' per il NER, in quanto nelle varie frasi sono etichettate con tag NER diversi e non esiste una vera dominanza di uno tra gli altri - Lingua Inglese

Struttura del progetto

Il progetto si compone di due jupyter book:

- **TLN_HMM_NER_Exploratory,** utilizzato per la fase di analisi esplorativa sul dataset
- **TLN_HMM_NER_Project,** training e testing dei modelli (e valutazione delle performance)

Al path src/hmm

- **Model**, contiene il codice relativo all'implementazione custom dell'Hidden Markov Model usato per risolvere il task di NER tagging
- **Memm_tagger, random_tagger, majority**, rappresentano gli altri modelli utilizzati per il confronto (baselines)
- **Evalutation** contiene le funzioni per la valutazione delle performance del modello

Al path src/tests

- Sono presenti gli script per eseguire i test delle performance sul modello MEMM_tagger, sull' NLTK.HiddenMarkowModel e sulla implementazione custom dell'HMM

Al path src/utils

- Funzionalità di utility come, ad esempio, per la conversione dei file CONNLU in un formato digeribile da parte de MEMM Tagger.

Implementazione

Il tagger di "NER tags" proposto utilizza una Hidden Markov Chain e l'algoritmo di Viterbi Decoding per risolvere il problema del sequence labeling, ovvero **l'assegnazione di un tag ad ogni parola di una frase.** L'obiettivo del modello HMM è di trovare la sequenza di NER tags che massimizza la probabilità condizionata data una sequenza di token osservati in input. Cioè assegnare ad ogni parola del testo una classe di tag NER, sfruttando l'informazione contenuta nelle parole **precedenti** e **successive**.

$P(t_1...t_n|w_1...w_n)$ is highest.

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

Il modello HMM è composto da un insieme di stati nascosti (hidden states) e di simboli osservati (observed symbols). Nel contesto del NER tagging, gli stati nascosti rappresentano le diverse classi di tag NER, mentre i simboli osservati corrispondono alle parole del testo.

Algoritmo

L'implementazione dell'algoritmo è suddiviso in due macro-fasi: una fase di **Learning**, in cui la probabilità di transizione e la probabilità di emissione vengono costruite; e una fase di **Decoding**, in cui viene restituita la sequenza di tag più probabile data una nuova frase in ingresso. Per ottenere tale sequenza, utilizzeremo l'Algoritmo di Viterbi, una tecnica di programmazione dinamica.

Learning

```
def fit(self, traing_path, tagName='lemma'):
    self.__init__(self.tagset) # Initialize the model

    # Count number of sentences for calculating probability of transition with the starting tag Q0
    nSentences = 0;

    # Open the training file
    file = open(traing_path, "r", encoding="utf-8")

    # Iterate over each sentence in the file
    for sentence in parse_incr(file):
        prev_tag = 'Q0'; # Set the previous tag to Q0 at the start of each sentence
        nSentences = nSentences + 1; # Increment sentence counter for each new sentence

        # Iterate over each token in the sentence
        for token in sentence:
            word = token["form"]; # Get the word for the token
            tag = token[tagName]; # Get the tag for the token

            # Count the word/tag pair and calculate emission probability
            self.count(word, tag);
            self.calculateEmissionProbability(word,tag);

            # Calculate transition probability using the previous tag and the current tag
            self.calculateTransitionProbability(prev_tag, tag, nSentences)

            # Set the previous tag to the current tag for the next iteration
            prev_tag = tag

    # Return the calculated probabilities and statistics
    return self.emission_probs, self.transition_probs, self.statistics;
```

L'algoritmo di Viterbi presenta una fase di **apprendimento** in cui vengono stimati i **conteggi delle transizioni** tra i tag (probabilità di un tag dato il tag precedente) e **delle emissioni** (probabilità di una parola dato il tag). Questi conteggi possono essere usati per calcolare le probabilità di transizione e di emissione, che vengono poi utilizzate successivamente nell'algoritmo di decodifica di Viterbi per trovare la sequenza di tag più probabile per una determinata frase.

Nella implementazione del **Learning**, gli handler **count, calculateEmissionProbabiliy e CalculateTransitionProbability** gestiscono l'aggiornamento dei contatori che avviene sequenzialmente così da ridurre al minimo gli impatti computazionali di questa fase.

Il tag Q0 viene utilizzato per rappresentare l'inizio di una frase e non fa parte del tagset originale. È stato aggiunto specificamente per l'uso all'interno del modello HMM, cioè per rappresentare lo stato iniziale del modello, prima che siano state fatte delle osservazioni.

Questo tag viene solo utilizzato come punto di partenza per calcolare le probabilità di transizione dallo stato iniziale al primo **tag della frase**. In questo modo, si può sfruttare il fatto che il tag Q0 è sempre presente all'inizio di ogni frase e utilizzarlo come punto di partenza fisso per i calcoli.

Vengono dunque computati:

- **count_tags:** è un array numpy che viene utilizzato per memorizzare il conteggio delle occorrenze di ciascuna etichetta (o tag) all'interno del corpus di addestramento. L'array è inizializzato con tutti gli elementi a zero e viene aggiornato con la funzione count() che viene chiamata per ogni coppia (parola, tag) presente nel corpus. E' importante per calcolare le probabilità di transizione tra i vari tag, in quanto permette di contare quante volte ogni tag compare in sequenza con ogni altro tag.

$$C(t_{i-1}, t_i)$$

```
def count(self, word, tag):
    index_of_tag = self.tagset.index(tag)

    # increment count of this tag
    self.count_tags[index_of_tag] = self.count_tags[index_of_tag] + 1
```

- **count_words**: È un dizionario che contiene **il conteggio delle parole emesse** per ogni tag, **utilizzato per calcolare le probabilità di emissione**. Le chiavi sono le parole e i valori sono dizionari con chiave i vari tag NER che hanno emesso quella parola, contenenti il conteggio di quella parola emessa. Ad esempio, se il tag 'PER' viene emesso con le parole 'John' e 'Mary', e 'John' viene emesso 50 volte e 'Mary' 30 volte, avremo:
count_words['John'] sarà {'PER': 50} count_words['Mary'] sarà {'PER': 30}

  Nota: il fatto che **non siano state salvate** le conte in count_words come count_words [tag][word_i] ma come count_words [word][tag_i] ma come, è unicamente per ragioni implementative.

$$C(t_i, w_i)$$

```
# increment count of this word for this tag
if word in self.count_words.keys():
    self.count_words[word][index_of_tag] = self.count_words[word][index_of_tag] + 1
else:
    # create a new row for the word
    word_row = np.zeros(self.totTags + 1, dtype=int)
    word_row[index_of_tag] = 1
    self.count_words[word] = word_row
```

- **transition**: È un dizionario che contiene **il conteggio delle transizioni tra tag**, cioè il numero di volte che un tag segue un altro nei dati di addestramento. Le chiavi del dizionario sono tuple di due tag (t1, t2) e i valori sono i conteggi di tali transizioni. Ad esempio, la voce ('ORG', 'PER') : 100 significa che il tag 'PER' è apparso immediatamente dopo il tag 'ORG' 100 volte nei dati di addestramento.

```
def calculateTransitionProbability(self, prev_tag, tag,  nSentences):
    #natural sequence when scanning senteces
    prev_tag_to_tag = "%s_%s" % (prev_tag,tag);
    #Sequence used to saved
    tag_to_prev_tag = "%s_%s" % (tag,prev_tag);

    if(prev_tag_to_tag in self.transition.keys()):
        self.transition[prev_tag_to_tag] = self.transition[prev_tag_to_tag] + 1;
    else:
        self.transition[prev_tag_to_tag] = 1;
```

- **transition_probs**: È un dizionario che contiene le **probabilità delle transizioni tra i tag**, calcolate normalizzando i conteggi delle transizioni nel dizionario delle transizioni. Le chiavi e i valori sono gli stessi del dizionario delle transizioni, ma i valori sono ora **probabilità** anziché conteggi. Ad esempio, se il conteggio totale delle transizioni dal tag 'ORG' è 1000 e il conteggio delle transizioni da **'ORG'** a **'PER'** è 100, allora la **probabilità di transizione da 'ORG' a 'PER' è 0,1.**

$$P(t_i|ti-1)$$

```
if(prev_tag_to_tag in self.transition_probs.keys() and prev_tag != 'Q0'):
    index_of_tag = self.tagset.index(prev_tag);
    self.transition_probs[tag_to_prev_tag] = self.transition[prev_tag_to_tag] /self.count_tags[index_of_tag];
else:
    self.transition_probs[tag_to_prev_tag] = self.transition[prev_tag_to_tag] / nSentences;
```

- **emission_probs**: È un dizionario che contiene **le probabilità di emissione d**i una parola data da un tag, calcolate dividendo il conteggio della parola emessa per un determinato tag per il conteggio totale delle parole emesse per quel tag. Le chiavi del dizionario sono tuple di una parola e di un tag (w, t), e i valori sono le probabilità di emettere la parola w dato il tag t. Per esempio, se il conteggio della parola "Giovanni" emessa con il tag "PER" è 50 e il conteggio totale delle parole emesse con il tag "PER" è 500, allora la probabilità di emissione di "Giovanni" con il tag "PER" è 0,1.

$$P(w_i|t_i)$$

```
def calculateEmissionProbability(self, word, tag):
    # Get index of tag in tagset
    index_of_tag = self.tagset.index(tag)

    # Calculate emission probability of word given tag
    emissionProb = self.count_words[word][index_of_tag] / self.count_tags[index_of_tag]

    # If word already exists in emission_probs, update its probability for the current tag
    if(word in self.emission_probs.keys()):
        self.emission_probs[word][index_of_tag] = emissionProb

    # If word doesn't exist in emission_probs, create a new row for the word and initialize with the current
    tag's  probability
    else:
        prob_row = np.zeros(self.totTags + 1)
        prob_row[index_of_tag] = emissionProb
        self.emission_probs[word] = prob_row;
```

Decoding

L'implementazione dell'algoritmo di Decoding di Viterbi è contenuta all'interno del **HiddenMarkovModel.predict(words,smoothingStrategy)** ed utilizza 3 strutture dati principali:

```
def predict(self, words, smoothingStrategy=0):
    # Initialize variables
    start_tag = "Q0"
    viterbi_matrix = np.zeros((self.totTags, len(words)))
    backtrace = np.zeros(len(words), dtype=int)
    probabilities = np.zeros(len(words))
```

- Una **matrice numpy bidimensionale** chiamata **viterbi_matrix,** utilizzata per memorizzare la **massima probabilità** che la sequenza di tag più probabile finisca con un particolare tag a ogni passo temporale. Ogni riga di questa matrice corrisponde a un tag e ogni colonna a una posizione nella sequenza di input.
- Un array numpy unidimensionale chiamato **backtrace**, utilizzato per tenere traccia dell'indice del tag più probabile per ogni posizione nella sequenza di input. L'elemento i-esimo di questo array memorizza l'indice del tag più probabile per la posizione i-esima nella sequenza di input.
- Un array numpy unidimensionale chiamato **probabilites**, viene utilizzata per mantenere le probabilità più alte calcolate fino a quel punto durante l'esecuzione del viterbi algorithm per ciascuna parola
- **Smoothing Strategy** si tratta di un parametro del metodo **Predict** che permette di applicare uno tra gli smoothing disponibili. Il parametro accetta valori interi che rappresentano diversi tipi di strategie di smoothing. La selezione dello smoothing è affidata all'handler selectSmoothing.

Successiva all'istanziazione delle stutture dati ViterbiMatrix e backpointers seguono tre steps principali:

Inizializzazione:

L'algoritmo inizializza alcune variabili, tra cui il tag iniziale, la matrice viterbi, il backtrace e le **probabilites** iniziali

Inizializza la prima colonna della matrice di viterbi calcolando la probabilities iniziale per ogni tag, in base alle probabilità di emissione e transizione **della prima parola della frase**.

Inoltre, imposta i valori iniziali di backtrace e probabilities per la prima parola.

for each state s **from** 1 **to** N **do**
$viterbi[s,1] \leftarrow a_{0,s} * b_s(o_1)$
$backpointer[s,1] \leftarrow 0$

```
# First iteration of Viterbi algorithm to initialize first column
for tag_idx, tag in enumerate(self.tagset):
    # Get emission probability for the first word (here we can apply smoothing)
    probE = self.selectSmoothing(words[0],tag_idx,smoothingStrategy)
    if probE == 0:
        # if probE is zero, set it to a small value to avoid log(0) error
        probE = np.log(0.00001)
    else:
        probE = np.log(probE)
    # Set initial transition probability for  Q0
    probT = np.log(0.00001);
    tran_tag = "%s_%s" % (tag,start_tag);
    # Calculate initial viterbi probability for each tag
    if tran_tag in self.transition_probs.keys():
        probT = np.log(self.transition_probs[tran_tag]);
    viterbi_matrix[tag_idx][0] = probE + probT;

index_max_values = np.argmax(viterbi_matrix[:,0]);
backtrace[0] = index_max_values;
probabilites[0] = viterbi_matrix[index_max_values,0];
```

Passo ricorsivo:

L'algoritmo aggiorna la matrice di Viterbi per ogni parola successiva della frase.

- Per ogni etichetta, calcola la probabilità di emissione della parola corrente, data l'etichetta, e la probabilità di transizione da ogni etichetta precedente all'etichetta corrente.
- Quindi valuta il prodotto delle probabilità viterbi precedenti, la probabilità di transizione e la probabilità di emissione per tutti i possibili tag precedenti, per ottenere la massima probabilità viterbi per il tag corrente.
- Infine, aggiorna la matrice viterbi con la probabilità massima per ogni tag e registra l'indice del tag con la probabilità massima all'interno di backtrace e di probabilities.

for each time step t **from** 2 **to** T **do** ; recursion ste
  for each state s **from** 1 **to** N **do**

$$viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$$

$$backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s}$$

-

```
# Update Viterbi matrix for each word in sentence
t= 1;
for word in words[1:]:
    # Calculate viterbi probability for each tag
    for tag_idx, tag in enumerate(self.tagset):
        # Get emission probability for current word (here we can apply smoothing)
        probE = self.selectSmoothing(word, tag_idx, smoothingStrategy);
        if probE == 0:
            probE = np.log(HiddenMarkovModel.ZERO_PROB)
        else:
            probE = np.log(probE)

        # Evaluate "max vt-1*aij*bj" between the N tags to get the maximum
        # viterbi probability for current tag
        max_tmp = np.zeros(self.totTags);
        for i in range(0,self.totTags):
            tran_tag = "%s_%s" % (tag,self.tagset[i]);
            probT = np.log(HiddenMarkovModel.ZERO_PROB);
            if tran_tag in self.transition_probs.keys():
                probT = np.log(self.transition_probs[tran_tag]);
            max_tmp[i] = viterbi_matrix[i,t-1] + probT;
        # Update Viterbi matrix with the maximum probability for current tag
        viterbi_matrix[tag_idx,t] = np.max(max_tmp) + probE;
```

Finalizzazione: l'algoritmo identifica il tag con la massima probabilità per l'ultima parola e lo memorizza come tag finale nell'elenco di backtrace.

- Quindi, identifica ricorsivamente i tag più probabili per le parole precedenti della frase, seguendo **il percorso di massima probabilità** memorizzato in **backtrace**.
- Infine, restituisce backtrace e probabilities come output dell'algoritmo.

$$viterbi[q_F,T] \leftarrow \max_{s=1}^{N} viterbi[s,T] * a_{s,q_F} \quad ; \text{termination step}$$

$$backpointer[q_F,T] \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T] * a_{s,q_F} \quad ; \text{termination step}$$

```
    # Update backtrace and probability lists with the max computed
    index_max_values = np.argmax(viterbi_matrix[:,t]);
    backtrace[t] = index_max_values;
    probabilites[t] = viterbi_matrix[index_max_values,t];
    t= t +1;
return backtrace,probabilites;
```

Smoothing

Spesso ci sono parole sconosciute che non sono presenti nel set di addestramento del modello, il che rende difficile assegnare loro un tag. Inoltre, le distribuzioni di probabilità di alcune parole potrebbero essere molto sbilanciate, il che può portare ad un bias del modello verso alcune categorie di entità a discapito di altre. Per ovviare a questi problemi, è possibile applicare una tecnica di smoothing, che consente di "distribuire" la probabilità mancante sulle parole sconosciute e di ridurre l'effetto dello sbilanciamento delle distribuzioni di probabilità. In questo modo, si può migliorare la capacità del modello di classificare correttamente le parole sconosciute e di garantire una distribuzione di probabilità più equilibrata tra le diverse categorie di entità.

Le strategie di smoothing sono applicate durante il calcolo di **probE** dunque della probabilità di emissione e sono definite all'interno di **hmm.selectSmoothing:**

- "**Sempre O**": questa opzione assegna sempre il tag "O" alle parole sconosciute. Ciò significa che non viene fatta alcuna previsione riguardo al tipo di entità nominata a cui potrebbero appartenere queste parole sconosciute. Questa opzione è molto semplice da implementare e può funzionare bene se le parole sconosciute non sono troppo frequenti. Tuttavia, potrebbe portare a risultati imprecisi se ci sono molte parole sconosciute in una frase.

```
elif smoothingStrategy == 1 and word not in self.emission_probs.keys():
    prob_row = np.zeros(self.totTags + 1);
    index_of_o = self.tagset.index("O");
    prob_row[index_of_o] = 1;
    return prob_row[index_of_tag];
```

- "**Sempre O MISC**": questa opzione assegna il tag "O" o i tag "MISC" (sia I e B di miscellaneous) alle parole sconosciute con probabilità 0,33. Anche questa opzione è semplice da implementare ed è un po' più flessibile rispetto all'opzione precedente. Tuttavia, ancora una volta, potrebbe non funzionare bene se le parole sconosciute sono troppo frequenti.

```
elif smoothingStrategy == 2 and word not in self.emission_probs.keys():
    prob_row = np.zeros(self.totTags + 1);
    index_of_i_misc = self.tagset.index("I-MISC");
    index_of_b_misc = self.tagset.index("B-MISC");
    index_of_o = self.tagset.index("O");
    prob_row[index_of_i_misc] = 0.33;
    prob_row[index_of_b_misc] = 0.33;
    prob_row[index_of_o] = 0.33;
    return prob_row[index_of_tag];
```

- "**Uniforme**": questa opzione assegna alle parole sconosciute una probabilità uniforme di appartenere a qualsiasi tag di entità nominata. In questo modo, si suppone che ogni tag abbia la stessa probabilità di essere assegnato alle parole sconosciute. Questa opzione è molto utile quando non si dispone di molte informazioni sulle parole sconosciute. Tuttavia, potrebbe non funzionare bene se ci sono tag molto rari o inusuali.

```
elif smoothingStrategy == 3 and word not in self.emission_probs.keys():
    unk_prob = 1 / self.totTags;
    prob_row = np.full(self.totTags + 1, unk_prob);
    return prob_row[index_of_tag];
```

- "**Statistica sul development set**": questa opzione assegna alle parole sconosciute il tag che appare meno frequentemente nel set di **validazione**. In questo modo, si suppone che le parole sconosciute siano associate al tag meno comune tra quelli del set di sviluppo. Questa opzione funziona bene se il set di sviluppo è rappresentativo del corpus completo. Tuttavia, potrebbe portare a risultati imprecisi se il set di sviluppo non è sufficientemente rappresentativo.

```
elif smoothingStrategy == 4 and word not in self.emission_probs.key():
    return self.statistics[index_of_tag];
```

Risultati e considerazioni

Sono stati condotti due esperimenti per valutare le prestazioni di diversi algoritmi di classificazione: il primo ha esaminato le strategie di **smoothing** utilizzando il dataset di test, mentre il secondo ha confrontato quattro modelli di classificazione NER basati su algoritmi diversi. Nello specifico:

Random Baseline: un modello "dummy" in cui ogni token nella sequenza in input viene etichettato casualmente dal tagset.

Majority Baseline: un modello "dummy" in cui ogni token nella sequenza in input viene etichettato con il tag più frequente nel set di addestramento.

HMM + Uniform: un modello basato su Hidden Markov Model e Viterbi decoding con una strategia di smoothing uniforme.

NLTK.TAG. HiddenMarkovModelTrainer: un modello basato su Hidden Markov Model e Viterbi decoding fornito dalla libreria NLTK. Lo script utilizzato si trova sotto **NER/src/tests/nltk_test**

MEMM: un modello discriminativo basato su HHM e modelli di max entropy. E' stato utilizzato il modello di **Gan-Tu/ML-DL-NLP/blob/master/MEMM-POS-Tagger/memm_tagger.py**

Nota a margine: Ho riscontrato lunghissimi tempi di esecuzione e l'unica statistica stampata dal **NER/src/tests/memm_tagget.test** è stata l'accuracy. L'esecuzione è stata interrotta dopo 10min di ottimizzazioni della soluzione sia per il dataset italiano sia per quello inglese .

LANG	**Smoothing**	**Accuracy**	**Precision**	**Recall**	**F1-score**
IT	HMM O -> 1	0.944	0.941	0.944	0.937
IT	HMM O && I-MISC && B-MISC-> .33	0.944	0.941	0.944	0.937
IT	HMM Uniform	0.946	0.943	0.946	0.940
IT	HMM Statistic	0.945	0.942	0.945	0.938
EN	HMM O -> 1	0.931	0.923	0.931	0.921
EN	HMM O && I-MISC && B-MISC-> .33	0.931	0.923	0.931	0.922
EN	HMM Uniform	0.933	0.926	0.933	0.924
EN	HMM Statistic	0.931	0.923	0.931	0.922

Figura 9 Statistiche dei diversi smoothing

Dalle statistiche ottenute, sembra che l'opzione di smoothing "HMM Uniform" abbia ottenuto i risultati migliori per entrambe le lingue, con l'accuracy più alta e il F1-score più elevato. Tuttavia, è importante notare che il tipo di smoothing migliore dipende dal corpus specifico su cui si sta lavorando e da quanto le parole sconosciute sono presenti. In generale, l'opzione di smoothing "Uniforme" può funzionare bene quando si dispone di poche informazioni sulle parole sconosciute, mentre l'opzione "Statistiche sul set di sviluppo" può essere più efficace quando il set di sviluppo è sufficientemente rappresentativo del corpus completo. Infatti, come è stato osservato nella parte introduttiva di esplorazione dei dati, la distribuzione dei NER tag non è uniforme nel corpus. Ciò significa che alcune entità nominate sono più frequenti di altre.

Ad esempio, nel corpus italiano, l'entità "B-LOC" (località) è molto più frequente **(il 50% in più)** dell'entità "B-PER" (persona). Questa disuguaglianza nella distribuzione delle etichette di entità nominata può influire sulle prestazioni di alcune strategie di smoothing. Ad esempio, l'opzione "uniforme" assegna la stessa probabilità a tutte le etichette di entità nominata, ma questo potrebbe non essere adatto quando alcune etichette sono molto più frequenti di altre. In generale, una strategia di smoothing che tiene conto della frequenza delle etichette di entità nominata nel corpus di addestramento potrebbe produrre risultati migliori rispetto a quelle che non lo fanno.

LANG	**Modelli**	**Accuracy**	**Precision**	**Recall**	**F1-score**
IT	HMM Uniform	0.946	0.943	0.946	0.940
IT	RandomTagger	0.111	0.779	0.111	0.177
IT	MajorityTagger	0.957	0.952	0.957	0.953
IT	NLTK. HiddenMarkovModelTrainer	0.949	0.947	0.949	0.943
IT	MEMM	0.949	N\A	N\A	N\A
EN	HMM Uniform	0.933	0.926	0.933	0.924
EN	RandomTagger	0.111	0.775	0.111	0.177
EN	MajorityTagger	0.944	0.937	0.944	0.938
EN	NLTK. HiddenMarkovModelTrainer	0.941	0.936	0.941	0.934
EN	MEMM	0.956	N\A	N\A	N\A

Figura 10 Confronto tra modelli

Il modello custom HMM uniform ha ottenuto un buon punteggio di precisione (0,946), pari a quello del modello NLTK HiddenMarkovModelTrainer (0,949). Tuttavia, va notato che il modello custom HMM uniform ha ottenuto un punteggio di recall leggermente inferiore rispetto a NLTK HiddenMarkovModelTrainer.

I modelli NLTK HiddenMarkovModelTrainer e MEMM sono modelli ben più elaborati e quindi potrebbero avere prestazioni migliori rispetto al modello custom HMM uniform in determinati contesti. Tuttavia, ciò non significa che il modello custom HMM uniform realizzato non sia un buon modello, in quanto ha dimostrato di avere una buona precisione.

Si riportano le prestazioni della Random Baseline solamente per completezza, in quanto tale baseline è significativa solamente nel caso di dataset equilibrati, cosa che non si verifica in questo caso come evidenziato. Dalla Tabella 13, si evince che l'accuracy del test set dei due tagger HMM e MEMM è pressoché identica a quella della Majority Baseline. Tale risultato non sorprende particolarmente, considerando che il POS Tagging è un task meno complesso rispetto ad altre attività più difficili come la Disambiguazione del Sense delle Parole, la Risoluzione delle Domande, la Risoluzione delle Coreferenze, ecc.

Confusion Matrix

Analizziamo ora le matrici di confusione e, in primo luogo, possiamo notare che il modello si è comportato bene con la classe "O", che rappresenta i token che non appartengono ad alcuna entità denominata. Ciò è prevedibile, poiché la maggior parte dei token in un testo non fa parte di un'entità denominata e infatti la distribuzione del tag “O” nel dataset è dell’88% . Per questa ragione la classe “O” è stata omesso dalla visualizzazione della matrice di confusione, in modo da poter analizzare meglio quanto accade con il resto dei TAG.

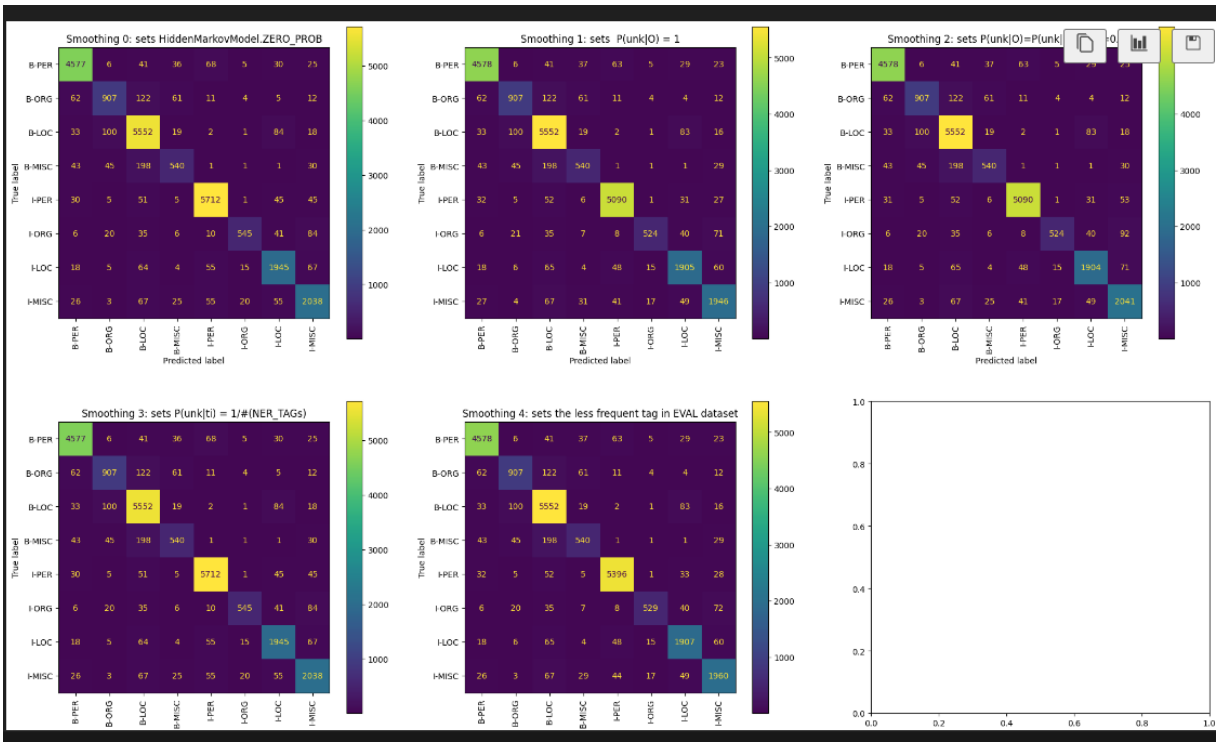

Figura 11 Confustion Matrix relative ai diversi tipi di smoothing applicati sul dataset di test Itailano

Nel dataset italiano, il modello ha avuto maggiori difficoltà a identificare correttamente le entità denominate nelle classi "B-ORG" e "B-LOC", confondendole spesso con la classe "O". Ciò è un problema significativo in applicazioni NLP come il recupero di informazioni o la risposta a domande, in cui la corretta identificazione di organizzazioni e luoghi è importante. Il modello ha ottenuto prestazioni relativamente migliori nelle classi "B-PER" e "B-MISC", ma ha commesso ancora un numero significativo di errori, confondendo talvolta i token di queste classi con quelli delle classi "I-PER" e "I-MISC". Inoltre, il modello ha commesso errori nella gestione dei tag "B" e "I", assegnando erroneamente un'etichetta "I" a un token che avrebbe dovuto essere etichettato come "B" e viceversa.

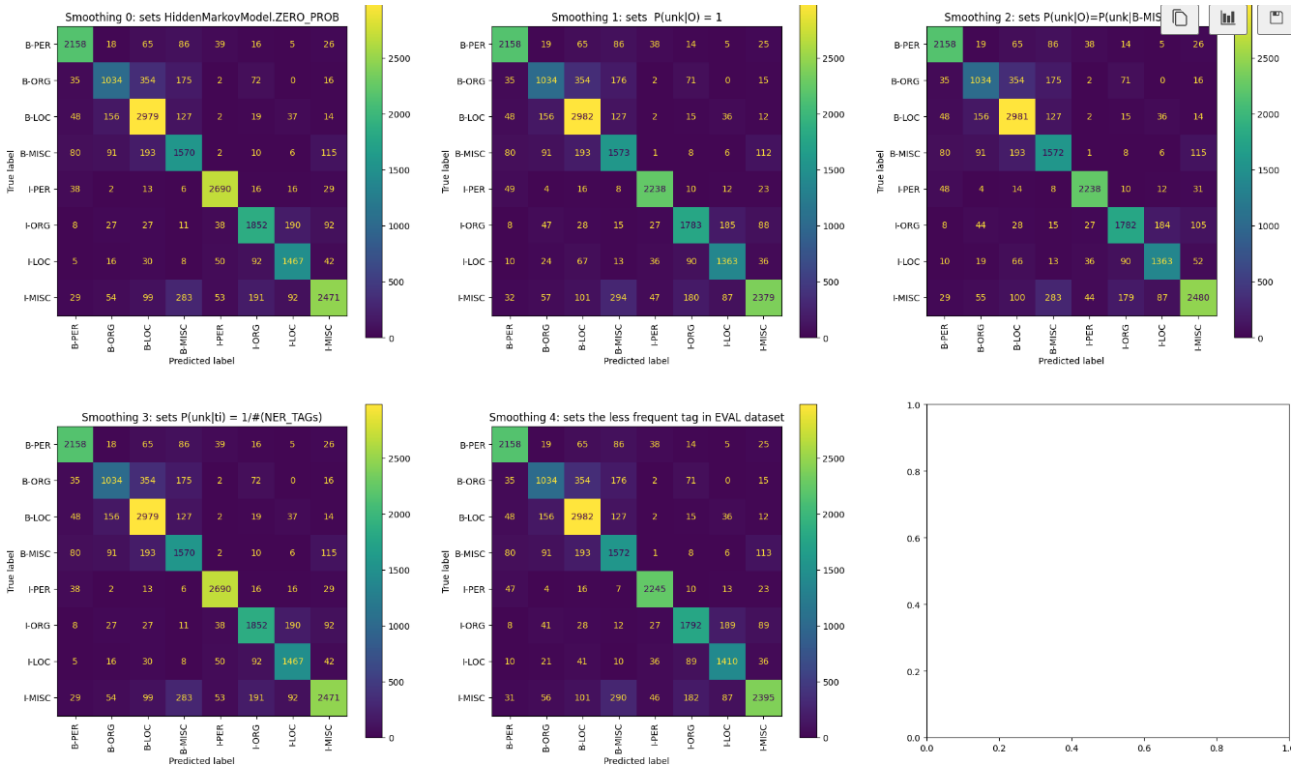

Figura 12 Confustion Matrix relative ai diversi tipi di smoothing applicati sul dataset di test Inglese.

Nel dataset inglese, la confusione tra le classi "B-LOC" e "B-ORG" è peggiorata rispetto all'italiano, così come la confusione tra le classi "B-MISC" e "I-MISC", "I-ORG" e "I-MISC", e "I-LOC" e "I-ORG". Inoltre, c'è stata una maggiore confusione tra le classi "B-LOC" e "I-MISC". Questi risultati possono essere influenzati dalla struttura grammaticale diversa e dalla distribuzione dei tag delle entità denominate nelle due lingue:

- **Differenze nella struttura della lingua**: L'inglese e l'italiano hanno strutture grammaticali diverse, che possono influire sulle prestazioni di un tagger HMM. Ad esempio, l'inglese ha un sistema di tempi verbali più complesso e un numero maggiore di verbi irregolari rispetto all'italiano, il che può rendere più difficile per un tagger HMM identificare correttamente le entità nominate.
- **Differenze nella distribuzione dei tag**: La distribuzione dei tag delle entità denominate in inglese può essere diversa da quella in italiano, il che potrebbe influire sulle prestazioni del tagger HMM. Ad esempio, se il tag "B-LOC" è più ambiguo in inglese che in italiano, potrebbe essere più difficile per il tagger HMM identificarlo correttamente.

Test su frasi di Harry Potter

E' stato inoltre eseguito un test per verificare le performance del modello su un set di frasi contenenti parole mai viste dal modello, per cui è stata eseguita manualmente l'etichettatura. Si sono ottenute le seguenti statistiche:

Accuracy: **0.811** Precision: **0.677** Recall: **0.811** F1-score**: 0.738**

Nota: precision_recall_fscore_support(y_true, y_pred, average='weighted',**zero_division=0**)

Il fatto che l'accuracy sia piuttosto alta (0.811) indica che il modello ha generalizzato bene su parole mai viste prima durante il test.

La precision, che è pari al rapporto tra il numero di entità correttamente etichettate dal modello e il numero totale di entità etichettate dal modello, è relativamente bassa (0.677), il che significa che il modello ha classificato erroneamente alcune entità. La recall, che rappresenta il rapporto tra il numero di entità correttamente etichettate dal modello e il numero totale di entità presenti nella valutazione manuale, è alta (0.811), il che significa che il modello ha identificato la maggior parte delle entità presenti. Tuttavia, l'F1-score, che è la media armonica tra precision e recall, è inferiore alla recall (0.738), il che indica che il modello ha difficoltà a classificare correttamente alcune entità.

Test on Harry Potter Phrases

```
sentences = ["La vera casa di Harry Potter è il castello di Hogwarts.",
             "Harry le raccontò del loro incontro a Diagon Alley.",
             "Mr Dursley era direttore di una ditta di nome Grunnings, che fabbricava trapani."]

sentences_words = [  ["La", "vera", "casa", "di", "Harry", "Potter", "è", "il", "castello", "di", "Hogwarts", "."],
  ["Harry", "le", "raccontò", "del", "loro", "incontro", "a", "Diagon", "Alley", "."],
  ["Mr", "Dursley", "era", "direttore", "di", "una", "ditta", "di", "nome", "Grunnings", ",", "che", "fabbricava", "trapani", "."]
]

sentences_tags = [  ["O", "O", "O", "O", "B-PER", "I-PER", "O", "O", "O", "O", "B-LOC", "O"],
  ["B-PER", "O", "O", "O", "O", "O", "O", "B-MISC", "I-MISC", "O"],
  ["B-ORG", "I-ORG", "O", "O", "O", "O", "O", "O", "O", "I-ORG", "O", "O", "O", "O", "O"]
]
```