Relazione Tecnologie del Linguaggio Naturale

Pio Raffaele Fina

aa 2020/2021

1 Introduzione

Il seguente documento costituisce la relazione per il progetto d'esame del corso *Tecnologie del Linguaggio Naturale* a.a. 2020/2021 tenuto dal Prof. Alessandro Mazzei.

Come progetto è stato scelto di implementare un **POS tagger** basato sull' *Algoritmo di Viterbi*. L'obiettivo principale è l'implementazione e la valutazione del POS tagger con l'utilizzo di differenti strategie di smoothing.

Organizzazione della repository La repository del progetto è organizzata come segue:

- `data`: contiene le risorse lessicali utilizzate nell'esercitazione.
- `src`: directory del codice sorgente.
- `esercitazione.ipynb`: jupyter notebook contenente un'analisi esplorativa dei dati e la valutazione.
- `requirements.txt`: python dependencies utilizzate dal progetto.

Riproducibilità Per riprodurre i risultati riportati nel seguente documento, effettuare i successivi passi:

1. Effettuare il download della repository:

```
git clone https://github.com/PRFina/TLN2021.git
```

2. Creare un virtual environment (eg. conda):

```
conda create -n TLN python=3.8; conda activate TLN
```

3. Installare le dipendenze del progetto:

```
pip install -r requirements.txt
```

4. Eseguire il notebook:

```
cd Mazzei/esercitazione1
```

```
jupyter notebook esercitazione.ipynb
```

2 Materiale e Metodi

2.1 Dati

Cosi come da specifiche, sono stati utilizzati due corpus del progetto *Universal Dependencies*. Dei 200 treebanks messi a disposizione, sono stati utilizzati: il corpus *Late Latin Charter Treebank (LLCT)*, ottenuto con un processo di conversione semi-automatica dal corpus *LLCT2* [1]; e il corpus *Ancient Greek Perseus (AGP)*, ottenuto, come il primo, dal corpus *Ancient Greek and Latin Dependency Treebank 2.1* [2].

I corpus sono in formato *CoNLL-U* (.conll), a questo proposito, il parsing del contenuto è stato effttuato con l'ausilio della libreria `pyconll`. Ciascun dataset viene distribuito con 3 split, *train*, *dev* e *test*, generati con random sampling. I 3 split rappresentano rispettivamente l'80%, il 10% e il 10% del intero curpus.

In fase preliminare, si è proceduto ad un'analisi esplorativa dei dati (`esercitazione.ipynb`, sezione "EDA"). In Figura1 e Figura 2 è riportato il *treemap* della distribuzione dei POS tags nelle rispettive classi, associati a ciascun token del dataset. In questo modo è possibile conoscere quali tags sono presenti nel *tagset* e in che quantità.

In Figura 3 e Figura 4 è riporta la distribuzione delle lunghezze delle frasi in termini di numero di tokens. Quest'ultimo plot è utile in quanto l'algoritmo di viterbi prevede una catena di moltiplicazioni, di conseguenza, la lunghezza delle frasi può ricoprire un fattore determinante in termini di stabilità numerica dell'algoritmo.

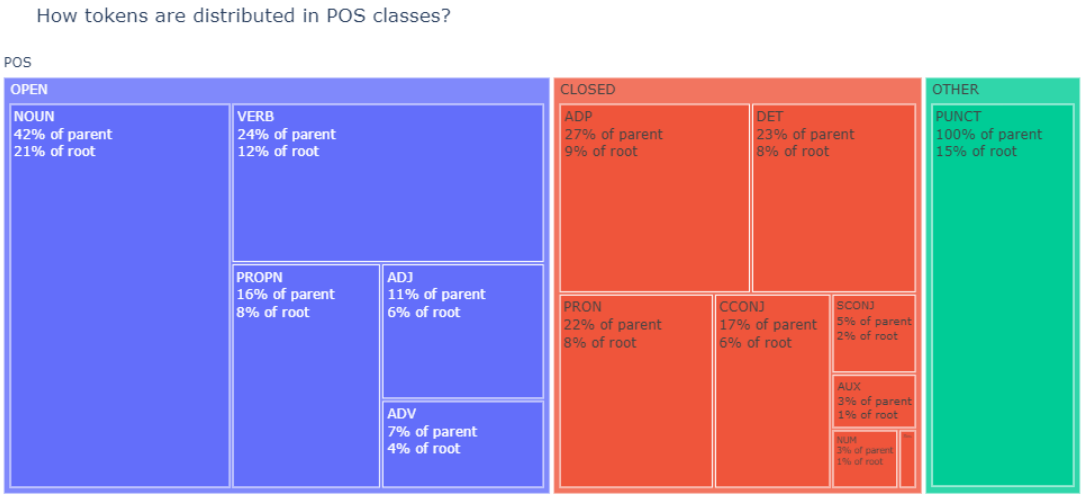

Figura 1: Distribuzione dei POS tags nelle rispettivi classi per il corpus LLCT (train + dev + test)

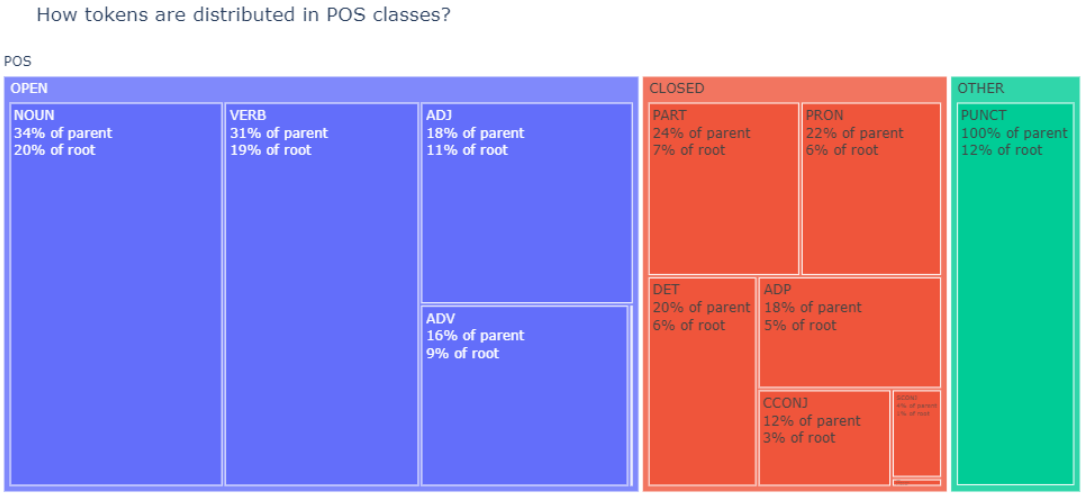

Figura 2: Distribuzione dei POS tags nelle rispettivi classi per il corpus AGP (train + dev + test)

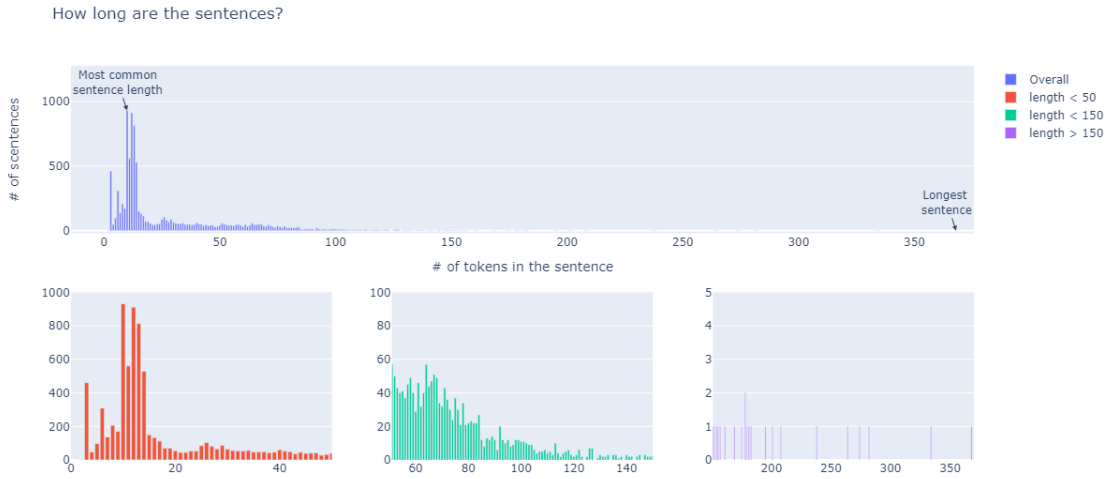

Figura 3: Distribuzione della lunghezza delle frasi, in termini di numero di tokens, per il corpus LLCT (train + dev + test)

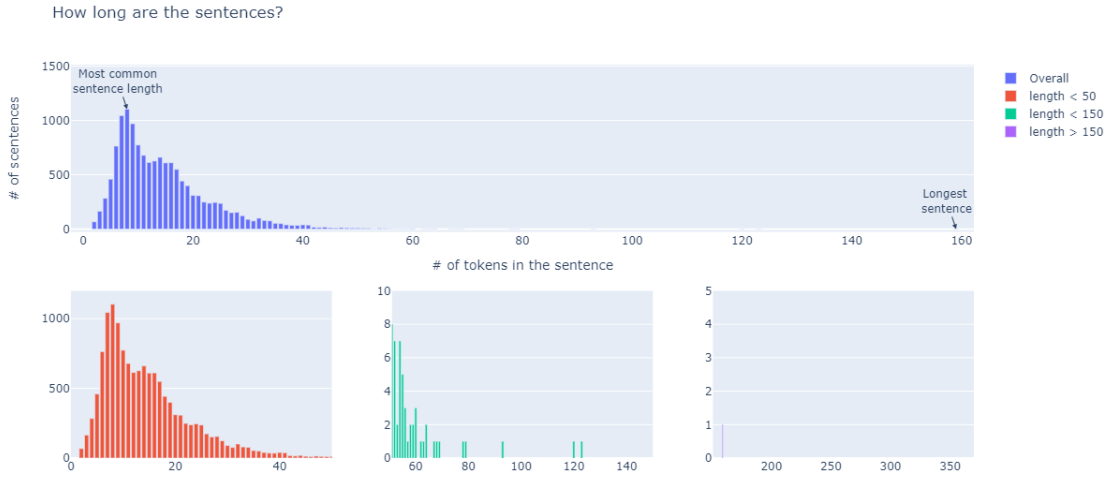

Figura 4: Distribuzione della lunghezza delle frasi, in termini di numero di tokens, per il corpus AGP (train + dev + test)

3 Implementazione

Il pos tagger qui proposto è modellato da una *Hidden Markov Chain* e sottostante al task di *sequence labeling* vi è l'algoritmo di *Viterbi Decoding*. In breve, il problema che si vuole risolvere può essere formulato come segue:

Data una frase in input (sequenza di token) $w_{1:n}$, trovare la sequenza di POS tags $\hat{t}_{1:n}$ tale che:

$$\hat{t}_{1:n} = \underset{t_{1:n}}{\operatorname{argmax}} \; P(t_{1:n}|w_{1:n})$$

Di seguito verranno riportate alcune parti salienti dell'implementazione. Per ulteriori dettagli si rimanda alle *docstrings* presenti nel codice.

3.1 Strutture dati

Sono state utilizzate tre principali strutture dati:

- `DefaultDict[str, collections.Counter]`[1]: stuttura dati composita, utilizzata per memorizzare e aggiornare le *transition probabilities* e *emission probabilities*. `DefaultDict` specializza il dizionario base di Python, modificando la semantica di accesso. La sostanziale differenza consiste nel creare degli oggetti di default piuttosto che lanciare un'eccezione `KeyError` se si prova ad accedere ad una chiave non presente nel dizionario. Questa dinamica, ritorna particolarmente utile quando, durante la fase di learning, si aggiunge un **nuovo** token/POS non presente nel dizionario:

  ```
  emission_counts[token][pos_tag] += 1
  ```

  in questo caso, se `token` non è presente nel dizionario, verrà instanziato in automatico un nuovo `Counter`. L'oggetto `Counter`, anch'esso specializza un dizionario python, permettendo di aggiornare agevolmente il conteggio delle chiavi al suo interno. La scelta per l'utilizzo di queste due strutture dati è motivata puramente dal rendere il codice meno verboso e maggiormente leggibile.

- `ViterbiMatrix`: rappresenta un "thin wrapper" che incapsula un `pandas.DataFrame` per un accesso diretto ai dati. La matrice, creata in fase di decoding, è di dimensione $N \times T$ con $N = |TAGSET|$ e T la dimensione della sequenza di token in input. Un esempio è rappresentato in Figura 5.

  La motivazione principale dietro l'utilizzo di questa struttura dati è da ricercare nel fatto che l'accesso ad un dataframe è principalmente *label-based*. Questo pone un problema per accedere alle colonne della matrice in quanto un token può essere ripetuto molteplici volte all'interno di una frase. In questi casi, l'indicizzazione `dataframe[token]` restituisce tutte le colonne associate al token, creando ambiguità. L'ambiguità nasce in quanto la semantica di accesso per le colonne dovrebbe essere *index-based*, mentre quella per righe (POS tags) dovrebbe essere *label-based*. Dato questo *mismatch*, si è provveduto a realizzare la classe `ViterbiMatrix` con l'intento di offrire un accesso uniforme e una notazione più compatta:

  ```
  dataframe.loc[pos].iloc(axis=1)[token_index] = value
                              ↓
  viterbi_matrix.assign(pos, token_index, value)
  ```

[1] la notazione utilizzata fa riferimento al meccanismo di type hinting di Python

	+	in	Deus	omnipotens	nomen	,	regno	domnus	noster	Carolus	diuinus	faueo	clementia	imperator	augustus	,	annus
ADJ	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ADP	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ADV	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
AUX	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
CCONJ	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
DET	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NOUN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NUM	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
PART	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
PRON	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
PROPN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
PUNCT	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
SCONJ	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
VERB	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
X	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figura 5: Esempio di un oggetto `ViterbiMatrix` istanziato sulla frase "*+ in Deus omnipotens nomen, regno domnus noster Carolus diuinus faueo clementia imperator augustus, annus ...*". I valori `nan` vengono sostituti in fase di decoding con gli effettivi valori di viterbi $v_t(i)$.

- `Smoother`: classe utilizzata per l'implementazione di differenti strategie di smoothing. Anche in questo caso uno smoother è riconducibile ad un dizionario. Il metodo `Smooth.get(key_from, key_to)` permette di recuperare l'emission probabilities $P(w_i|t_i)$. Nel caso $w_i = UNK$ (token sconosciuto) il valore di probabilità restituito dal metodo, dipende dall'effettiva strategia di smoothing implementata. In caso contrario, se $w_i \neq UNK$ il metodo restituisce l'emission probability calcolata durante la fase di leaning. Ulteriori dettagli sulle strategie di smoothing sono riportate in sezione 3.3.

  L'utilizzo dello smoother si basa sul *decorator pattern*. In fase di inizializzazione del tagger, viene passato come argomento un oggetto di classe `smoothing.BaseSmoother` (o una sua specializzazione):

  ```
  HMMPosTagger(smoother=Smoother())
  ```

  dopo aver calcolato le emission probabilities in fase di learning, quest'ultime vengono "wrappate" dallo smoother:

  ```
  Smoother.add_probabilities(emission_probabilities)
  ```

  permettendo, in fase di decoding, di accedere alle probabilità $P(UNK|t_i)$. Ad esempio, supponendo UNK = abacus:

  ```
  Smoother.get("NOUN", "abacus")
  ```

  ritornerà il valore "smoothed" di emission probability.

3.2 Algoritmo

3.2.1 Learning

La fase di learning è implementata dal metodo `HMMPosTagger.fit()`. Nel caso dei modelli HMM basati sul language model a bigrammi, il training si riduce a stimare le probabilità:

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

$$P(w_i|t_i) = \frac{C(w_i, t_i)}{t_i} \tag{2}$$

il metodo `HMMPosTagger.fit()` effettua un unico pass sull'intero training set contando le occorrenze come specificato in (1) e (2) e memorizzando tali occorrenze nelle strutture `transition_probs` ed `emission_probs` .

```
def fit(self, X, y):

    pos_counts = Counter()
    transition_counts = defaultdict(Counter)
    emission_counts = defaultdict(Counter)

    for sentence_tokens, sentence_tags in zip(X,y):

        # add once for each sentence
        pos_counts[HMMPosTagger.START_TOKEN] += 1
        pos_counts[HMMPosTagger.END_TOKEN] += 1

        # emission counts
        for token, pos_tag in zip(sentence_tokens, sentence_tags):
            emission_counts[token][pos_tag] += 1
            pos_counts[pos_tag] += 1

        # transtions counts
        curr_tags = sentence_tags + [HMMPosTagger.END_TOKEN]
        prev_tags = [HMMPosTagger.START_TOKEN] + sentence_tags
        for prev_tag, curr_tag in zip(prev_tags, curr_tags):
            transition_counts[prev_tag][curr_tag] += 1

        # normalization
        ...

        self.emission_probs_smoother.add_probabilities(self.emission_probs)
```

Da notare l'aggiunta di due ulteriori tag "fittizi" `START` e `END` che indicano rispettivamente l'inizio e la fine di una frase. I tag in questione, verranno successivamente utilizzati nella fase di decoding. L'operazione nel penultimo blocco di codice permette la creazione dei bigrammi per calcolare $C(t_{i-1}, t_i)$, **allineando** i tokens della stessa frase come due sequenze differenti:

```
START ita de    meus  exeo      dominium  et  in    tuus  -
ita   de  meus  exeo  dominium  et        in  tuus  trado -

trado sum       potestas .
sum   potestas  .        END
```

Da notare che i valori nelle strutture `emission_counts` e `transitions_counts` contengono le frequenze assolute dei bigrammi (tag, tag) e (tag, token), si procede dunque (qui omesso) a **normalizzare** i suddetti valori ottenendo le *transition* e *emission* probabilities. Come step finale, le emission probabilities vengono aggiunte allo smoother.

3.2.2 Decoding

La fase di decoding è implementata dal metodo `HMMPosTagger.predict(tokens)`. Successiva all'istanziazione delle stutture dati `ViterbiMatrix` e `backpointers` seguono tre steps principali:

Inizializzazione In questa fase viene riempita la prima colonna della matrice con il corrispettivo valore di viterbi:

$$v_0(s) = P(s|\texttt{START}) \cdot P(o_0|s)$$

```
for pos in viterbi_mat.pos_tags:
    emission_prob = self.emission_probs_smoother.get(pos,tokens[0])
    viterbi_value = self.transition_probs[HMMPosTagger.START_TOKEN][pos] *
                    emission_prob

    viterbi_mat.assign(pos, 0, viterbi_value)
...
```

Da notare come il valore dell'emission probability, è ottenuto tramite l'utilizzo dello smoother (`emission_probs_smoother`).

Passo ricorsivo In questa fase vengon riempite le rimantenti colonne della matrice con il valore di viterbi calcolato come:

$$v_t(s) = \max_{s'=1}^{N} v_{t-1}(s) \cdot P(s|s') \cdot P(o_t|s) \tag{3}$$

```
for token_idx, token in enumerate(tokens[1:], start=1):
    for pos in viterbi_mat.pos_tags:
        emission_prob = self.emission_probs_smoother.get(pos,token)

        viterbi_prefix = viterbi_mat.get_prefix_probs(token_idx)
        transition_prefix = [self.transition_probs.get(prefix_tag, 0).get(pos, 0)
                             for prefix_tag in viterbi_mat.pos_tags]

        viterbi_value = (viterbi_prefix * transition_prefix).max() * emission_prob
        viterbi_mat.assign(pos, token_idx, viterbi_value)

        max_tag = (viterbi_prefix * transition_prefix).idxmax()
        backpointers[token_idx-1].append((pos, max_tag)

...
```

Da notare che `viterbi_prefix` e `transition_prefix` sono vettori di dimensione $|TAGSET| \times 1$ e l'operazione `(viterbi_prefix * transition_prefix)` è un moltiplicazione *element-wise*. Inoltre, nella struttura `backpointers` viene memorizzato il tag che massimizza il prodotto. Il tag in questione rappresenta il tag predetto per il token precedente o_{t-1} rispetto al corrente corrente indice t.

Finalizzazione In modo analogo allo step precedente, viene calcolato il valore di viterbi $v_T(\texttt{END})$ secondo l'equazione (3) ma omettendo il termine $P(o_t|s)$ in quanto non avrebbe senso calcolare la probabilità $P(o_T|\texttt{END})$.

```
...

end_token_idx = len(tokens)
viterbi_prefix = viterbi_mat.get_prefix_probs(end_token_idx)
transition_prefix = [self.transition_probs.get(prefix_tag, 0).get(HMMPosTagger.END_TOKEN, 0)

max_tag = (viterbi_prefix * transition_prefix).idxmax()
backpointers[end_token_idx-1].append((HMMPosTagger.END_TOKEN, max_tag))

predicted_tags = HMMPosTagger._reconstruct_path(tokens, backpointers)
```

L'ultima operazione permette di ricostruire la sequenza di tag predetti per ogni token in input. L'operazione è delegata al metodo "privato" `reconstruct_path(tokens, backpointers)` che sfrutta i puntatori memorizzati nella struttura `backpointers` per ripercorrere all'**indietro** la sequenza e ottenere i tag predetti.

Di seguito viene riportato un *Minimal Working Example*:

```
TRAIN_FILE = Path("data/UD_Latin-LLCT/la_llct-ud-train.conllu")
train_tokens, train_tags = preprocess_data(pyconll.load_from_file(TRAIN_LATIN_FILE))

tagger = HMMPosTagger(smother=NounSmoother())
tagger.fit(train_tokens, train_tags)

predictions = tagger.predict("Cogito ergo sum.")
```

con la sequenza predetta in output:

```
[('Cogito', 'NOUN'), ('ergo', 'VERB'), ('sum', 'AUX'), ('.', 'PUNCT')]
```

3.3 Sparsness & Smoothing

Il problema della sparsità dei parametri con l'utilizzo di *n-grammi* come language model è un fenomeno ben noto in letteratura. Il problema emerge in quanto, all'aumentare di n, la copertura richiesta dal training set cresce esponenzialmente, infatti con un vocabolario di dimensione V, bisognerebbe stimare V^n possibili n-uple [3]. Come conseguenza, molte delle probabilità $P(w_n|w_{n-1:1})$ saranno **nulle**. Questa dinamica viene solitamente identificata con il nome di *sparsness*. L'utilizzo di $n >> 1$ risulta chiaramente inaccettabile nella pratica, sopratutto in un contesto di apprendimento statistico **supervisionato** che richiede annotazioni manuali (o semi-manuali); per questo motivo si tende a limitare $n = 2, 3, 4, 5$.

A questo proposito, nel lavoro descritto in questo documento, sono state fatte le seguenti approssimazioni:

- Utilizzo di **bigrammi** come language model.
- La stima delle emission probabilities è basata sui **lemmi** piuttosto che le word-forms, inducendo implicitamente delle classi di equivalenza e di conseguenza, riducendo numero di parametri da stimare. Ad esempio, come può essere inferito dalla Figura 6, le probabilità $P(\text{collaboro}|t), \ldots, P(\text{collaborare}|t)$ sono tutte equivalenti a $P(\text{collaboro}|t)$.

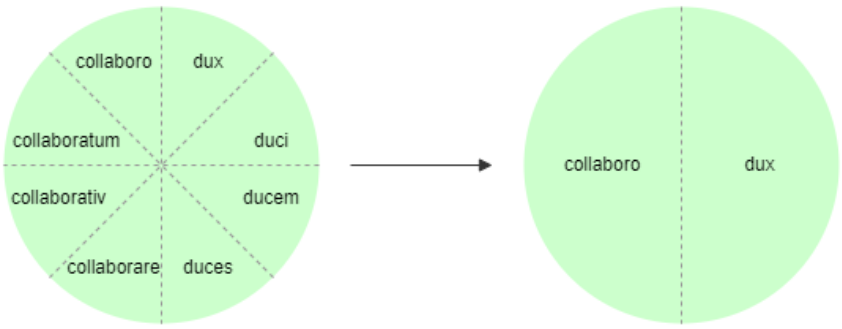

Figura 6: Esempio di come l'utilizzo dei lemmi induce classi di equivalenza.

Un ulteriore problema da tenere in conto è la presenza di parole *Out Of Vocabulary (OOV)* [3], ovvero la possibilità di dover utilizzare l'emission probability associata a tokens non presenti nelle frasi del train set. In questi casi differenti strategie più o meno complesse possono essere utilizzate per ottenere $P(UNK|t)$. Di seguito vengono riportare le strategie utilizzate:

No Smoothing nessuna strategia di smoothing, viene associata una probabilità $P(UNK|t) = \varepsilon$ con $\varepsilon \to 0$

Noun Smoothing strategia in cui la massa di probabilità si concentra unicamente sul tag $NOUN$:

$$P(UNK|t) = \begin{cases} 1.0 & \text{se } t \in \{NOUN\} \\ 0.0 & \text{altrimenti} \end{cases}$$

Noun-Verb Smoothing strategia in cui la massa di probabilità si distribuisce equamente sui tags $NOUN$ e $VERB$:

$$P(UNK|t) = \begin{cases} 0.5 & \text{se } t \in \{NOUN,\ VERB\} \\ 0.0 & \text{altrimenti} \end{cases}$$

Uniform Smoothing strategia in cui la massa di probabilità si distribuisce in modo uniforme sull'intero tagset:

$$P(UNK|t) = \frac{1}{|TAGSET|}$$

Rule Based Smoothing strategia in cui la massa di probabilità si concentra unicamente sul tag individuato dall'applicazione di regole di **pattern matching sintattico**. Nella directory `data` è presente il file `latin_derivational_suffixes_rules.txt` che può essere letto dalla funzione `utils.load_pattern_rules()` e restituisce una lista di regole da passare come argomento in input al costruttore della classe `RuleBaseSmoother`. Le regole (solo per il latino) sono state estratte automaticamente dallo script `generate_latin_rules.py` effettuando un semplice *scraping* dalla seguente risorsa.

Alcuni esempi dimostrativi sono riportati nella sezione "smoothing" nel notebook `esercitazione.ipynb`, le strategie appena descritte sono implementate dalle classi presenti nel *module* `smoothing`. Dal punto di vista implementativo tutti gli smoother estendo una classe base `BaseSmoother`:

```
class BaseSmoother():
    def __init__(self, probs_dict=None, unknow_prob=1e-64):
        self.probs_dict = probs_dict
        self.unk_prob = unknow_prob
        self.known_tokens = set()

    def add_probabilities(self, probs_dict):
        self.probs_dict = probs_dict

        for tokens_probs in probs_dict.values():
            self.known_tokens.update(tokens_probs.keys())

    def get(self, key_from, key_to):
        return self.probs_dict[key_from].get(key_to, self.unk_prob)
```

L'interfaccia della classe offre due metodi:

- `add_probabilities(probs_dict)`: il metodo mantiene traccia del dizionario passato come argomento in input e da esso costruisce un insieme di "parole conosciute", ovvero tutti i token che occorrono almeno una volta nel train set.
- `get(key_from, key_to)`: il metodo implementa la logica della strategia. In primis viene verificato se l'argomento `key_to` è presente o meno nell'insieme di "parole conosciute". In caso negativo, calcola il valore di probabilità in base alla strategia e al POS tag (`key_from`) in input, restituendo il valore in output. In caso contratio, restituisce il valore di emission probability calcolata in fase di learning. Nel caso specifico dello `BaseSmoother`, se il token è sconosciuto, restituisce una probabilità prossima allo 0 (1×10^{-64}).

4 Risultati e Discussione

Per valutare le performance sono stati effettuati due esperimenti: il primo, valuta le differenti strategie di smoothing utilizzando il *dev set*; il secondo, effettuato sul *test set*, è volto a comparare 4 pos tagger basati su modelli differenti:

Random Baseline "dummy" model in cui ogni tag della sequenza dei tokens in input, viene predetto campionando in modo uniforme dal tagset.

Majority Baseline "dummy" model in cui ogni tag della sequenza di token in in input, viene predetto in base al tag più frequente nel training set.

HMM + Uniform modello basato su hidden markov model e Viterbi decoding con strategia di smoothing uniform.

MEMM modello discriminativo basato su HHM e *maximum entropy* models, è stata utilizzata la seguente implementazione.

I risultati dell'accuratezza delle predizioni con differenti strategie di smoothing sono riportati in Tabella 1.

Strategy	LLCT	AGP
No Smoothing	95.02	77.77
Noun	96.93	81.37
Noun-Verb	96.95	81.54
Uniform	97.09	80.77
Rule Based	85.47	-

Tabella 1: Accuracy sul *dev set* per i corpus LLCT e AGP.

In prima analisi, dalla Tabella 1, l'aspetto più evidente è la sistematica decrescita delle performance per il corpus AGP. Le cause potrebbero essere molteplici, probabilmente uno dei fattori imputabili a questa dinamica è la maggiore complessità grammaticale della lingua in questione rispetto al Latino. Inoltre, seppur in misura minore, c'è da constatare che il tagset AGP conta 14 tags contro i 15 del latino; cosa, che seppur riduca l'ambiguità, aumenta la difficoltà del tagging (aspetto già affrontato a lezione con il *Brown Corpus*).

In generale si può apprezzare che l'utilizzo dello smoothing migliora le performance di 2/3 punti percentuali rispetto la baseline (No Smoothing). La motivazione può essere attribuita all'effettiva occorrenza di parole sconosciute all'interno del dev set che possono incidere negativamente sulla baseline. Una possibile metrica per quantificare l'effetto dello smoothing è il *OOV rate* [3], sul dev set LLCT risultano 169 parole sconosciute con un OOV rate del 16% rispetto alle 1708 parole sconosciute e 42% OOV rate del dev set AGP.

L'unica eccezione è costituita per la configurazione con il *Rule Based Tagger*, non presente per il Greco antico in quanto non è stato possibile recuperare risorse lessicali da cui estrapolare delle regole. In questo caso il peggioramento sull'accuracy può essere dovuto a due fattori principali: 1) la mancanza di un effettiva validazione linguistica sulle regole a cura di un esperto; 2) l'utilizzo dei lemmi piuttosto che delle wordforms può incidere negativamente in quanto le regole si basano sui suffissi e questi spesso vengono eliminati dal processo di lemmatizzazione.

I risultati dell'accuratezza delle predizioni con differenti modelli sono riportati in Tabella 2. Le performance sulla *Random Baseline* sono riportate per completezza in quanto tale baseline risulta significativa solo in caso di dataset bilanciato, sicuramente non il caso analizzato come si può osservare dalle Figura 1 e Figura 2. Come si può notare dalla Tabella 2, l'accuracy sul test set dei due tagger *HMM* e *MEMM* sono pressoché identiche alla *Majority Baseline*. Questo risultato non sorprende notevolmente in quanto è ben noto che il *POS Tagging* è un task con un livello di difficoltà limitato rispetto a task più difficili come *Word Sense Disambiguation*, *Question Answering*, *Co-reference resolution*, etc.

Tagger	LLCT (%)	AGP (%)
Random Baseline	6.64	7.26
Majority Baseline	96.84	80.30
HMM + Uniform Smoothing	96.93	81.88
MEMM	98.702	78.732

Tabella 2: Accuracy sul test set per i corpus LLCT e AGP

Un analisi più granulare sugli errori commessi dal modello può essere effettuata dalle matrici di confusione in Figura 7. Nel caso del corpus LLCT (prima matrice, sulla sinistra) si può notare che la maggior parte delle predizioni errate riguardano i tags `NOUN`, di cui 159 predizioni sono erroneamente confuse con `PROPN`; e `AUX`, di cui 202 predizioni sono erroneamente classificate come `VERB`. Caso interessante sono le predizioni sul tag `ADJ` in cui gli errori si distribuiscono in modo più o meno uniforme su tutti i tags. Per quanto riguarda il corpus AGP (seconda matrice, sulla destra), risalta sicuramente l'errore sul tag `PART`, che non viene mai correttamente predetto. Pertanto, dopo avere brevemente investigato le possibili motivazioni di questo errore, è emerso che il tag `PART` è totalmente assente nel test set. Le performance peggiori rispetto al corpus LLCT possono essere in parte spiegate da quest'ultima osservazione ma anche dai non trascurabili errori effettuati sui tag `NOUN` e `VERB`.

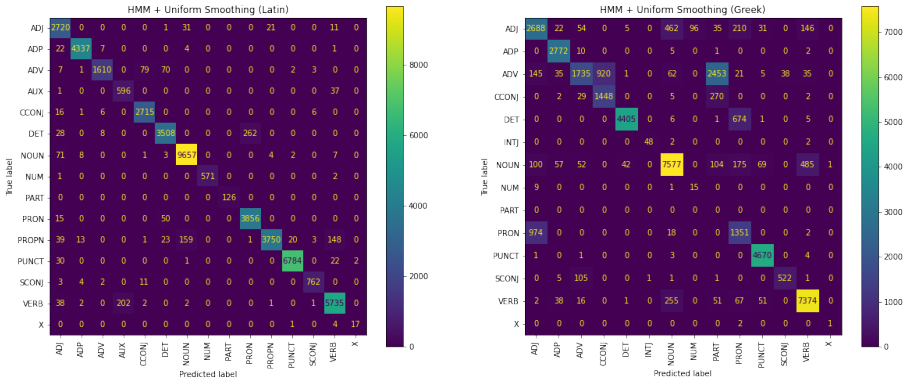

Figura 7: Matrici di confusione

References

[1] Timo Korkiakangas. *Late Latin Charter Treebank 2 (LLCT2), version 1.2*. Version 1.2. Zenodo, Feb. 2020. DOI: 10.5281/zenodo.3633614. URL: https://doi.org/10.5281/zenodo.3633614.

[2] David Bamman. *Late Latin Charter Treebank 2 (LLCT2), version 1.2*. Feb. 2019. URL: https://github.com/PerseusDL/treebank_data.

[3] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Vol. 2. Feb. 2008.