Università degli Studi di Torino
Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Relazione Progetto

Tecnologie del Linguaggio Naturale
Sviluppo di un PoS Tagger per il Latino e Greco Antico

Docente:
Prof. Alessandro Mazzei

Candidati:
Damiano Gianotti
Roberto Demaria

Sessione Settembre 2021
a.a 2020/2021

Contenuto del Documento

1 Introduzione

1.1 Scope

L'obiettivo del progetto è quello di implementare un PoS Tagger statistico basato su Hidden Markov Model (HMM) con memoria 1, utilizzando la preziosa risorsa *Universal Dependencies Treebank*, comprendente diversi treebank annotati di varie lingue, tra cui il Late Latin Chapter Treebank (LLCT) per il latino e l'Ancient Greek Perseus per il greco antico, che utilizzeremo come corpus alla base dello sviluppo del nostro parser per queste due lingue antiche. Fatto questo, andremo a valutarne la bontà confrontandola rispetto ad due baseline: una puramente statistica e una basata su MEMM.

1.2 Dataset

I dataset, come già accennato, fanno riferimento a due treebank separati, uno per il latino e uno per il greco antico. Il corpus è suddiviso in tre parti per tre compiti differenti: un "training set", un "development set" e un "test set" con le seguenti percentuali: 80% training, 10% validation e 10% testing. Il training set verrà utilizzato nella fase di Learning. Dovremo poi valutare il sistema, usando diverse strategie di smoothing ed in questa fase il development set ci tornerà utile al fine di creare un algoritmo avanzato; infine il "test set" ci consentirà di testare il nostro modello.

I files si presentano in formato *.conllu* e sono costruiti con numerose frasi nella corrispondente lingua, annotate in maniera semi-automatica. Per ogni parola della frase vengono fornite molte informazioni, tra cui, oltre alla word form e al lemma, anche il PoS Tag attribuitogli; nel caso del **Latino**, ad esempio, potrà essere uno dei quindici seguenti:

$$POS = \left\{ \begin{array}{l} ADJ, ADP, ADV, AUX, CCONJ, \\ DET, NOUN, NUM, PART, PRON, \\ PROPN, PUNCT, SCONJ, VERB, X \end{array} \right\} \tag{1}$$

Nonostante `Python` fornisse delle librerie per il parsing automatico di questo tipo di file (`pyconllu`), si è scelto di andare a curare anche questa parte, parsificando i file ad hoc in strutture dati appositamente scelte, per avere un maggiore controllo sul flusso operativo e dimostrare trasparenza su come i dati vengano utilizzati.

1.3 Struttura del Progetto

Il flusso del progetto si può dividere tre macro fasi: una fase iniziale di Modelling, in cui si creano i contatori; una fase di Learning, dove andremo a costruire la probabilità di transizione e la probabilità di emissione; una fase di Decoding per poter avere una sequenza di tags (la più probabile) data una nuova frase in ingresso. A tal fine utilizzeremo una tecnica di programmazione dinamica nota come "Algoritmo di Viterbi" per trovare la migliore sequenza di tags.

L'idea alla base di questo algoritmo è quella di evitare l'esplosione combinatoria, calcolando ricorsivamente una sequenza di tags ottimale (cioè che massimizza la probabilità) da soluzioni ottimali di sotto-problemi, dove si considerano versioni troncate della sequenza di osservazione. In pratica vengono riciclate le sotto-sequenze già calcolate per calcolare la probabilità su ogni nuova sequenza. Infine, con la fase di backpoint, viene ricalcolato il percorso fatto e individuati i tag con la probabilità massima calcolata per ogni parola.

2 Learning

2.1 Funzione di conteggio

```
def create_counters(self, path: str):
    """
    We are using collections.Counter() as inner data structure
    1. Calculating C(ti, wi), n_tags_given_word: counts of tags
    given specific words
    2. Calculating C(ti-1, ti), n_tags_given_tag: counts of tags given tag
    :param path: path of the corpus
    """
    with open(path, 'r') as corpus:
        for line in corpus:
```

Per questa fase di conteggio sono stati utilizzati dei dizionari multidimensionali, all'interno dei quali troviamo per ogni chiave dei `collections.Counter()`, perfetti per questo scopo.

Si tratta, in pratica, di dizionari di Contatori. Ad esempio il conteggio C(ti-1, ti) è strutturato come un dizionario che ha i POS tag come chiavi e come valori un altro dizionario che ha nuovamente i POS tag come chiavi e i conteggi come valori. In questo modo è possibile contare e assegnare un valore a quante volte un tag è preceduto da un altro tag. La matrice di emissione è strutturata in modo molto simile, come un dizionario le cui chiavi sono parole e i cui valori sono, a loro, volta, dizionari. Questi hanno come chiavi i tags e come valori il conteggio di quante volte quella stessa parola sia associata a quel preciso tag. Il metodo lavora sul training set, andando ad aggiornare di volta in volta queste strutture dati.

2.1.1 Aggiunta Tag

Abbiamo ritenuto utile aggiungere ai già esistenti POS tag, due ulteriori etichette:

1. Il primo ('START') è un tag usato per calcolare la probabilità che una parola inizi una frase, e che quindi sia la prima;
2. 'END' analogamente serve per calcolare la probabilità che una parola non sia seguita da nient'altro, ovvero sia alla fine della frase. Per questo verrà aggiunto solo come *value* del `Counter` e non è necessario che sia trattato anche come chiave.

```
# Handling end of phrase
if line == '\n':
    self.n_tags_given_tag[tag].update({'END': 1})
if (...):
    # Handling start of phrase
    if int(line.split('\t')[0]) == 1:
        if 'START' not in self.n_tags_given_tag:
            self.n_tags_given_tag['START'] = Counter()
        self.n_tags_given_tag['START'].update({tag: 1})
```

2.1.2 $\forall t, w | t \in TAG, w \in WORD \rightarrow C(t_i|w_i)$

Ovvero quante volte un tag viene attribuito ad una determinata parola:

```
# Calculating C(ti, wi): counts of tags given specific words
if word not in self.n_tags_given_word:
    self.n_tags_given_word[word] = Counter()
self.n_tags_given_word[word].update({tag: 1})
```

Ad esempio `n_tags_given_tag['Iulius']['NOUN']` fornisce quante volte la parola *Iulius* è stata taggata con *NOUN*.

2.1.3 $\forall t, w | t \in TAG, w \in WORD \rightarrow C(t_{i-1}|t_i)$

```
# Calculating C(ti-1, ti): counts of tags given tag
if int(line.split('\t')[0]) > 1:
    if prev_tag not in self.n_tags_given_tag:
        self.n_tags_given_tag[prev_tag] = Counter()
    self.n_tags_given_tag[prev_tag].update({tag: 1})
```

Allo stesso modo `n_tags_given_tag['ADV']['PRON']` fornisce quante volte una parola taggata con *ADV* apparirà prima di una parola taggata con *PRON*.

2.2 Modello probabilistico

Successivamente, dopo aver terminato la fase di conteggio, possiamo costruire il nostro modello probabilistico, sfruttando, come struttura dati, nuovamente, dei dizionari multi-dimensionali. Le singole probabilità vengono calcolate con una leggera variante rispetto a quanto visto a lezione: esse sono state salvate in forma logaritmica in modo tale da non avere valori troppo piccoli in fase di Decoding, evitando il fenomeno di underflow, e da essere computazionalmente più efficienti nelle fasi successive.

La funzione `calculate_prob` richiama il metodo statico `aux_calculate_prob` che a seconda dei parametri, costruisce la matrice di transizione o quella di emissione, andando a dividere i conteggi precedentemente effettuati, rispettivamente per $C(t_i)$ e $C(t_{i-1})$.

```
@staticmethod
def aux_calculate_prob(n_tags_giv: dict, given_tag=None) -> dict:
```

2.2.1 Probabilità di transizione $P(t_i|ti-1)$

Ovvero la probabilità che compaia un tag t dato un tag precedente t_{i-1}. In questo caso solo il primo parametro corrispondente a `n_tags_given_tag` viene fornito a `aux_calculate_prob`.

```
# Calculate P(ti | ti-1) transizione
p = math.log(n_tags_giv[element][n_tag] / num_occurrence(n_tags_giv[element]))
```

Riprendendo l'esempio `prob_tag_given_tag['ADV']['PRON']` fornisce la *log* probability che una parola taggata con *ADV* appaia prima di una parola taggata con *PRON*.

2.2.2 Probabilità di emissione $P(w_i|t_i)$

Ecco dunque la probabilità che un tag t sia attribuito ad una parola w Ora, invece, entrambi i parametri sono forniti a `aux_calculate_prob` poiché adesso serve anche il dizionario `n_tags_given_word` per effettuare il calcolo.

```
# Calculate P(wi | ti) emissione
p = math.log(n_tags_giv[element][n_tag] / num_occurrence(given_tag[n_tag]))
```

Analogamente `prob_word_given_tag['Iulius']['NOUN']` fornisce la *log* probability che la parola *Iulius* sia stata taggata con *NOUN*.

3 Decoding

L'algoritmo di Viterbi

La fase di decoding, ovvero di applicazione del modello, viene messa in pratica attraverso questo algoritmo, che, attraverso la **programmazione dinamica**, ci consente di evitare di andare a calcolare tutte le possibili combinazioni [word (osservabile), tag (hidden)] per una certa sequenza di parole, che causerebbe un'esplosione combinatoria in base alla lunghezza della frase.

La nostra implementazione prende in entrata due parametri: una frase sotto forma di lista di parole, e un dizionario di supporto, generato nella fase di smoothing, che serve a migliorare la performance dell'algoritmo. I risultati ottenuti saranno la matrice di Viterbi e il POS tag più probabile per le parole di una data frase.

```
def viterbi_algo(self, phrase: list, smoothed_p: dict):
```

Nella prima parte dell'algoritmo si procede all'inizializzazione di alcune variabili locali importanti e al recupero delle probabilità di emissione e transizione.

```
# class variables with more clear and short names
trans_prob: dict = self.prob_tag_given_pred_tag
emission_prob: dict = self.prob_word_given_tag

# 0. local variables
back_pointer = dict()
pos_back_pointer = dict()
viterbi = dict()
maximum_tag: str = '_'  # init as empty space
```

Il dizionario `back_pointer` verrà riempito con i valori massimi della probabilità che verranno poi assegnati ai rispettivi POS tag, mentre `pos_back_pointer` prenderà poi questi tag e li assegnerà alle parole della frase creando, quindi, il risultato finale dell'algoritmo; `viterbi` conterrà la matrice di Viterbi che verrà riempita di volta in volta con le probabilità calcolate e, infine, il `maximum_tag` conterrà per gradi il tag con probabilità massima, calcolata dal metodo `maximum`. L'algoritmo è poi suddiviso in 3 step principali che verranno spiegati in seguito.

3.1 Initialization step

```
for hmm_state in trans_prob.keys():
    # create the rows of the matrix
    viterbi[hmm_state] = list()
    back_pointer[hmm_state] = list()

    # In case we don't find we assign 0 probability
    probability: float = ALMOST_ZERO_P

    if hmm_state in trans_prob['START']:
        first_word = phrase[0]
        if first_word in smoothed_p:
            if hmm_state in smoothed_p[first_word]:
```

```
                probability = trans_prob['START'][hmm_state] +
                smoothed_p[first_word][hmm_state]
        else:
            if hmm_state in emission_prob[first_word]:
                probability = trans_prob['START'][hmm_state] +
                emission_prob[first_word][hmm_state]

    # Update the dictionaries key = pos:probability
    viterbi[hmm_state].append(probability)
    back_pointer[hmm_state].append(maximum_tag)
```

La prima fase dell'algoritmo consiste nell'inizializzare la prima colonna della matrice di Viterbi. Nel caso non esista una entry nella probabilità di transizione per un certo tag, la cella nella matrice corrispondente non viene inizializzata a zero (per via dei logaritmi), ma con un valore costante piccolo `ALMOST_ZERO_P = -99999`.

Diversamente, per ogni tag, la cella viene inizializzata con la somma dei logaritmi della probabilità che quel tag sia preceduto dal tag START (prob. transizione $a_{0,S}$) e della probabilità che alla prima parola della frase sia associato un tale tag (prob. emissione $b_S(O_t)$). Se la parola dovesse essere sconosciuta (non compare nel training set), allora, invece di considerare la probabilità di transizione, verrà utilizzata quella fornita dallo smoothing.

3.2 Recursion step

Segue poi una fase ricorsiva dove vengono riempite tutte le altre colonne della matrice eccetto l'ultima.

```
last_t = 0
for t in range(len(phrase)):
    if t != 0:
        for state in trans_prob.keys():
            maximum_tag, bpointer, m_vit =
            self.maximum(vit=viterbi, word=phrase[t],
            index=t - 1, m_tag=maximum_tag,
            current_state=state, smoothed=smoothed_p)

            # nella colonna attuale viene salvato solo
            # il valore massimo tra tutte quelle appena calcolate.
            viterbi[state].append(round(m_vit, 3))
            back_pointer[state].append(round(bpointer, 3))

        # il tag con la probabilità più alta di essere
        # riferito alla parola precedente
        pos_back_pointer[phrase[t - 1]] = maximum_tag
    # salvo l' indice finale
    if t == len(phrase) - 1:
        last_t = t
```

Per ogni parola e per ogni possibile tag, vengono calcolate tutte le somme (in logaritmo) delle probabilità riferite allo stato di Viterbi precedente più la somma delle probabilità di transizione dallo stato precedente a quello attuale (t_{i-1}, t_i); di questa somma, nella colonna

attuale, viene poi considerato solo il valore massimo tra quelli appena calcolati che viene poi sommato alla probabilità che quella parola sia associata proprio a quello specifico tag. Questa operazione viene effettuata tramite il richiamo della funzione `maximum`; in questo modo andiamo a riempire ogni singola cella, colonna dopo colonna. Al termine del calcolo su una specifica colonna, il `pos_back_pointer` viene aggiornato assegnado il tag più probabile alla parola corrispondente.

3.2.1 La funzione di massimo

Una menzione particolare merita la funzione `maximum` sopracitata.
Come parametri abbiamo rispettivamente la matrice di Viterbi, la parola (colonna) che stiamo analizzando, l'indice della parola precedente (importante per recuperare le probabilità), il massimo tag trovato finora, lo stato corrente che stiamo analizzando e, infine il dizionario di smoothing di supporto.

```
def maximum(self, vit: dict, word: str, index: int,
m_tag: str, current_state: str, smoothed: dict):

    if word in smoothed:
        # We are gonna use the smoothed probabilities in case unknow workds
        emission_p = smoothed
    else:
        emission_p = emission_prob

        for state in trans_prob.keys():
            if current_state in trans_prob[state] and current_state in emission_p[word]:
                # Max function viterbi[s', t-1] + A_s',s
                max_v = vit[state][index] + trans_prob[state][current_state]
                if max_v > m_vit:
                    m_vit = max_v
                # Argmax function viterbi[s', t-1] + A_s',s
                argmax = vit[state][index] + trans_prob[state][current_state]
                if argmax > bpointer:
                    bpointer = argmax
                    m_tag = state

        # Adding B_s(O_t) that is indie from s'
        if current_state in emission_p[word]:
            m_vit += emission_p[word][current_state]

    return m_tag, bpointer, m_vit
```

Inizialmente il metodo controlla se la parola si trova o no nel dizionario di smoothing, dopo di che va a calcolare la probabilità di Viterbi. In seguito verifica se la cella, appena calcolata, possiede una probabilità maggiore dell'attuale massimo e, in caso affermativo, lo prende come nuovo massimo e ne sovrascrive il valore. Discorso analogo per quanto riguarda il backpointer.

La funzione ritornerà il tag più probabile (quello col valore massimo), il back pointer per ricostruire il percorso e la matrice di Viterbi aggiornata.

3.3 Termination step

L'ultimo passaggio è analogo al primo, ma fa riferimento al tag END e serve a completare l'ultima colonna della matrice.

```
m_path: float = ALMOST_ZERO_P
last_tag: str = '_'
for s in trans_prob.keys():
    if s != 'START' and 'END' in trans_prob[s]:
        temp = viterbi[s][last_t] + trans_prob[s]['END']
        if temp > m_path:
            m_path = temp
            last_tag = s
```

In questo caso si prendono tutti i possibili tag, escluso lo START, e per quelli che possono transire nello stato END si calcola la probabilità, salvando tale valore nella variabile `temp`. Il calcolo del massimo viene effettuato *in place* (dato che non compare più la probabilità di emissione): si controlla se l'attuale valore sia maggiore dell'attuale massimo `m_path` e, in caso affermativo, si sovrascrivono i parametri di tag e valore corrispondenti.

4 Smoothing

4.1 Motivazioni

Questa fase è resa necessaria dal fatto che è ovviamente possibile, in fase di test, trovare delle parole che non sono presenti nel training set. Per questa ragione si è reso necessario taggare con una certa probabilità anche queste parole sconosciute, che vanno gestite. A tale scopo sono state create nuove probabilità di emissione, sulla base di diverse assunzioni.

4.2 Quattro alternative possibili

1. $P(unk|NOUN) = 1$
   Ad una parola sconosciuta si attribuisce il tag $NOUN$ con una probabilità massima.

2. $P(unk|NOUN) = P(unk|VERB) = 0.5$
   Ad una parola sconosciuta si attribuisce il tag NOUN e VERB con il 50% di probabilità per ognuno.

3. $P(unk|t_i) = 1/|(PoS)|$
   Ad una parola sconosciuta viene associato ogni tag con probabilità $uniforme$. Ogni possibile tag ha la stessa probabilità di essere associato a quella word form.

4. $P(unk|t_i) = |(PoSUnique)_{t_i}|/|(Unique)|$

   (a) Prese in considerazione l'insieme di word che compaiono solamente una volta $Unique$ all'interno del development set, per ogni tag si conta quante volte è associato ad una unique-word.

   (b) Dunque, per ogni parola sconosciuta unk viene calcolata la probabilità per ogni tag come il rapporto tra il numero totale di occorrenze di tale tag per le unique-word e il modulo di quest'ultime.

4.3 Implementazione

In linea con quanto svolto in precedenza, abbiamo usato il *log* della probabilità, dividendo il tutto in due metodi. Nel primo abbiamo raggruppato le prime 3 tecniche, più semplici.

```
def basic_smooth(path, emission, tags) -> (dict, dict, dict):
    """
    :param path: path to train test file
    :param emission: probability of word given tag
    :param tags: list of possibile tags
    :return: em_noun, em_nn_vb, em_uniform
    """
```

Il secondo invece è interamente dedicato allo smoothing statistico che vedremo più in dettaglio.

```
def statistical_smooth(path_dev, path_test, emission, tags) -> dict:
    """
    :param path_dev: percorso per il dev treebank
    :param path_test: percorso per il test treebank
    :param emission: dizionario contenente le probabilità di emissione
    :param tags: dizionario contenente le probabilità di transizione
    :return: dev_smooth
    """
```

Prendiamo in considerazione il *dev* set e lo analizziamo linea per linea per compiere quanto descritto in (a).

```
    if word not in word_count:
        tag_count_by_word[word] = Counter()
    word_count.update({word: 1})
    tag_count_by_word[word].update({tag: 1})
```

In seguito, per ogni parola univoca andiamo a memorizzare la coppia $key : value$:

- *key*: è uno dei possibili tag di quella word;
- *value*: il numero totale di occorrenze dei tag che compaiono per quella word.

```
    # If the word is singleton
    if word_count[word] == 1:
        tag_count.update({tag: tag_count_by_word[word][tag]})
```

Infine, recuperati i valori dal *test* set, per quelle parole sconosciute (*unknown*) andiamo a calcolare la probabilità logaritmica salvandola nell'apposito dizionario.

```
for tag in tags:
    if tag_count[tag] > 0:
         # P(unk|t_i) = |(PoSUnique)_{t_i}| / |(Unique)|
        dev_smooth[word][tag] = log(tag_count[tag] / num_dev_unk_words)
```

5 Descrizione dei risultati

Utilizzando il test set siamo andati a confrontare il nostro sistema con due baseline: una statistica e una più complessa, basata su Maximum Entropy Markov Models (MEMM) con custom-engineered fetures (credits Gan-Tu).

5.1 Frequency Baseline

Questa funzione rappresenta una possibile implementazione del semplice approccio statistico basato sulle frequenze.

```
def baseline(phrase, n_tags_given_word) -> dict:
```

- se la parola è conosciuta prendiamo il suo tag più frequente;
- se è sconosciuta assumiamo sia un nome ovvero $P(unk|NOUN) = 1$.

```
if word in n_tags_given_word:
    res[word] =
        max(n_tags_given_word[word].items(), key=operator.itemgetter(1))[0]
else:
    res[word] = 'NOUN'
```

Come è intuibile il metodo restituirà un dizionario contenente coppie [word, tag].

5.2 Rating e tabelle

Infine, accenniamo velocemente al metodo `rate` per valutare le prestazioni del nostro sistema: per ogni tecnica di smoothing, esso chiama l'algoritmo di Viterbi su tutte le frasi del test set con corrispettivo dizionario di supporto.

```
pos_backpointer, viterbi = self.viterbi_algo(phrase, metric)
```

Il risultati prodotti possono essere riassunti in queste 3 tabelle:

Strategy	Ancient Greek (%)	Latin (%)
$P(unk\|NOUN) = 1$	73.238	95.664
$P(unk\|NOUN) = P(unk\|VERB) = 0.5$	75.757	95.897
$P(unk\|t_i) = 1/\|(PoS)\|$	71.258	95.997
$P(unk\|t_i) =$ $\|(PoSUnique)_{t_i}\|/\|(Unique)\|$	75.471	96.931
F Baseline	73.525	95.403
MEMM	78.732	98.702

Table 1: **Confronto tra i due treebank**

POS	max_noun	NN_and_VB	uniform	statistical	baseline
ADJ	47.392	47.172	45.964	46.787	46.238
ADP	98.404	98.404	98.334	98.334	98.890
ADV	30.353	30.473	35.525	31.556	26.784
CCONJ	79.795	79.795	73.792	79.795	80.820
DET	94.366	94.240	82.638	92.362	96.285
INTJ	91.429	91.429	91.429	60.000	91.429
NOUN	98.966	78.513	73.609	71.344	99.010
NUM	50.000	50.000	50.000	50.000	75.000
PRON	54.562	55.447	68.556	58.370	59.876
PUNCT	99.407	99.407	99.449	99.407	99.873
SCONJ	80.000	80.312	85.000	80.625	92.188
VERB	47.049	85.971	68.053	93.187	46.969
X	0.000	0.000	0.000	0.000	0.000
Accuracy	73.238	75.757	71.258	75.471	73.525

Table 2: **Accuratezza sui singoli POS: Greco Antico**

POS	max_noun	NN_and_VB	uniform	statistical	baseline
ADJ	94.032	94.032	94.032	94.032	94.240
ADP	99.403	99.403	99.403	99.403	99.724
ADV	93.735	93.735	93.735	93.735	93.120
AUX	87.000	88.000	87.667	88.000	99.000
CCONJ	98.544	98.544	98.613	98.544	97.573
DET	98.056	98.003	97.951	98.056	92.591
NOUN	99.351	98.681	98.255	98.498	99.696
NUM	93.857	93.857	93.857	93.857	88.396
PART	100.000	100.000	100.000	100.000	100.000
PRON	98.299	98.777	98.830	98.777	99.894
PROPN	78.061	78.061	81.957	91.883	78.154
PUNCT	99.582	99.612	99.582	99.612	100.000
SCONJ	96.783	96.783	95.710	96.783	95.979
VERB	92.688	95.295	94.144	93.907	91.672
X	100.000	100.000	100.000	100.000	100.000
Accuracy	95.664	95.896	95.997	96.931	95.403

Table 3: **Accuratezza sui singoli POS: Latino**

6 Conclusioni

I risultati in tabella 1 mostrano che, in generale, sussiste una migliore accuracy sul latino rispetto al greco antico. A nostro avviso, questo fatto è riconducibile a diverse concause. Tra queste sia che il greco antico risulta essere una lingua grammaticalmente più complessa rispetto al latino, sia che il numero di POS usati per il greco antico risulta inferiore (14) rispetto a quello usato per il latino (15). In particolare, come abbiamo visto a lezione nel caso del Brown Corpus, avendo un numero maggiore di tags l'ambiguità diminuisce anche se aumenta la difficoltà del POS tagging.

Inoltre, un'altra possibile ragione che giustifica la performance più bassa per il greco, può facilmente essere legata al fatto che nel test set e nel development set sono presenti solo 13 tags invece dei 14 del training set (dettaglio di cui ci siamo accorti in fase di analisi); per essere più precisi manca il tag *PART* che, infatti, risulta spesso erroneamente classificato, contribuendo in modo significativo al peggioramento dell'accuracy, con circa un migliaio di errori in media. Esempio (*PART* (prediction) | *ADV* (true), 1092).

In generale, come si può constatare, la performance dipende molto da come viene effettuato lo smoothing sulle parole sconosciute. In generale, per quanto riguarda l'HMM, *lo smoothing statistico* è quello che fornisce una accuracy **migliore** su entrambe le lingue, perché è in grado di cogliere maggiormente la struttura linguistica sottostante.

Analizzando più in dettaglio il prospetto 3, relativo all'accuratezza sul latino, si può inoltre vedere come le maggiori difficoltà si abbiano sui tags *PROPN* e *VERB*. Da un'analisi approfondita è possibile osservare che, tra gli errori più comuni, vi è proprio la confusione del tag *PROPN* con *NOUN* e di *VERB* con *AUX* o *NOUN*.

Per quanto riguarda il greco antico (prospetto 2) la performance sui singoli tags dipende pesantemente dalla tecnica di smoothing impiegata. In particolare si evidenzia una certa difficoltà nella classificazione dei nomi (*NOUN*) e dei verbi (*VERB*) che vengono, infatti, spesso confusi. Lo *smoothing statistico* fornisce tuttavia un notevole miglioramento nella classificazione dei verbi, ma fornisce la peggiore performance per i nomi. Qui la migliore performance si ottiene con lo smoothing *NN_and_VB* anche se la *baseline* si avvicina molto. Da notare inoltre che, in generale, la peggiore accuracy si ottiene sul modello *uniforme* (71.3%) indicando che la distribuzione sottostante delle parole sconosciuta è ben lontana dalla semplice uniforme. A riprova, il fatto che *la baseline* e lo smoothing *max_noun* abbiano delle accuracy così alte sui nomi (intono al 99%) indica che, molto probabilmente, la maggior parte delle parole nuove sono dei nomi.

Infine abbiamo visto sperimentalmente che il modello MEMM, *non sviluppato da noi*, ha grande successo per taggare quelle parole sconosciute su cui HMM faticava (circa 82% di correttezza) proprio a causa di:

1. un modello n-gram (aggregato con caratteristiche aggiuntive personalizzate) per le sequenze di parti del discorso (PoS);

2. un modello di distribuzione della likelihood dei tag di parte del discorso per le parole.