Tommaso Comelli
Matricola 957268

Relazione Laboratio TLN – Mazzei

L'esercizio scelto è il numero 1, ovvero la realizzazione di un PoS Tagger per lingue morte (latino e greco). Il codice è stato scritto in Python.

I. Librerie
II. Strutture Dati
III. Struttura Codice e Algoritmi
IV. Conclusioni

I. Librerie

Le librerie utilizzate sono:

I. Pathlib
II. Numpy
III. Pyconll

I. Pathlib: libreria utilizzata per gestire i percorsi file all'interno del codice.
II. Numpy: libreria utilizzata per facilitare l'utilizzo di array multidimensionali e matrici, arricchito con delle funzioni per poter operare in modo efficiente.
III. Pyconll: libreria utilizzata per lavorare in modo facile con il formato CoNLL-U.

II. Strutture Dati

Le pricipali strutture dati utilizzate sono:

I. Dizionario
II. Matrice
III. Array

I. Dizionario
Il dizionario è stata la struttura dati più usata nel codice per molteplici aspetti: è veloce sia nel memorizzare sia nel recuperare i dati senza dover scannerizzare l'intera collezione di dati, le chiavi del dizionario possiedono un'unica referenza e sono associabili a dei valori che sono mutabili.
Un esempio è il dizionario *eprob = {' SoS -> I': 0.13, 'EoS -> You': 0.04}* in cui vengono salvate le probabilità di emissione: la prima chiave unica dell'esempio è composta dal pos_tag "SoS", seguita da una freccia, seguita dalla parola "I"; il valore associato alla chiave è 0.13.
Una delle funzioni più usate per costruire i dizionari è *add_to_hash()* che dato in input un dizionario e una chiave inserisce la chiave nel dizionario e se la chiave esiste già incrementa di uno il suo valore.

```
# aggiunge una una chiave ad una hash oppure se esiste già incrementa il suo valore
def add_to_hash(hash, key):
    if key not in hash.keys():
        hash.update({key: 1})
    else:
        hash[key] += 1
```

II. Matrice

La matrice è essenziale per il codice perché, a differenza del dizionario, abbiamo bisogno di mantenere i dati ben ordinati ed è la stessa struttura che viene utilizzata da Viterbi che sfrutteremo per trovare la migliore sequenza di pos_tag nelle frasi analizzate.
Nel codice la matrice *mat* viene create tramite la libreria numpy (as np), le sue dimensioni sono composte come righe dal numero di pos_tag, come colonne la lunghezza della frase studiata più i due caratteri speciali SoS e EoS.

```
mat = np.zeros(shape=(pos_len, s_length + special_char))
```

III. Array

Quest'ultima struttura dati è molto efficiente, semplice e veloce, inoltre ci permette come la matrice di tenere conto dell'ordine dei dati e di operare su di essi.
Un esempio è l'array *backtrace* che, inizializzato sempre tramite la libreria numpy, contiene una sequenza di pos_tag di una frase ed infatti la sua grandezza è determinata dalla lunghezza della frase più i due caratteri speciali SoS e EoS.

```
backtrace = np.empty(shape=(s_length + special_char), dtype=np.dtype('U5'))
backtrace[0] = sos
```

III. Struttura Codice e Algoritmi

Le classi più importanti nel codice sono:

I. prob.py
II. viterbiAlg.py
III. baselineAlg.py
IV. smooth.py

I. prob.py

Questa classe è utilizzata per la maggiorparte dei calcoli delle probabilità che sono necessari agli algoritmi: sia quello di Viterbi, sia quello di baseline.
Fra le funzioni più importanti ci sono *e_prob()*, *t_prob_fin()* e *most_used_tag()*.
In *e_prob()* viene calcolata la probabilità di emissione: per ogni token di ogni frase nel corpus si crea una stringa *name* contenente pos_tag e forma del token e si aggiunge al dizionario eprob tramite *add_to_hash()*, successivamente ogni chiave del dizionario si divide per la frequenza del suo tag che è salvata nel dizionario *n_pos* (il quale tiene traccia delle coppie (pos, frequenza_pos)).

```
# calcola le emission probability e le ritorna tramite dizionario
def e_prob(corpus):

    # la prima parte della funzione calcola le occorrenze delle varie combinazioni tag -> parola
    for sentence in corpus:
        for token in sentence:
            name = token.upos + arrow + token.form
            add_to_hash(eprob, name)

    # divide i value di eprob per il corrispettivo value di countpos, caratterizzato dallo stesso tag
    for name in eprob:
        # salva in una str la prima parte della key di eprob cioè il tag
        key_pos = name.split(space, 1)[0]
        if key_pos in n_pos.keys():
            eprob[name] /= n_pos[key_pos]

    return eprob
```

La funzione *t_prob_fin()* come operazioni è simile concettualmente a *e_prob()* ma al posto di calcolare la probabilità di emissione calcola quella di transizione.
Nel primo ciclo viene calcolato il numero di volte che al pos_tag *i-1* segua il pos_tag *i*.
Nel secondo ciclo invece si ricava la probabilità di transizione vera e propria dividendo per la frequenza dei pos_tag presenti nel corpus. Alla fine della funzione viene restituito il dizionario *t_prob*.

```
# calcola le transition probability e le ritorna tramite dizionario
def t_prob_fin(corpus):
    num_eos = 0
    for sentence in corpus:
        length = sentence.__len__()
        for token in sentence:
            # se il token non è l'ultimo allora o lo salva se è nuovo o incrementa il tag i-1 -> tag i
            if length > 1:
                next_token = sentence.__getitem__(int(token.id))
                nameprob = token.upos + arrow + next_token.upos
                add_to_hash(tprob, nameprob)

            else:
                num_eos += 1
                nameprob = token.upos + arrow + eos
                add_to_hash(tprob, nameprob)

            length -= 1

    # calcola tutte le probabilità di transizione tag i-1 -> tag i
    for name in tprob:
        # salva in una str la seconda parte della key di tprob cioè il tag dopo la freccia
        key_eos = name.split(arrow, 1)[1]
        key_sos = name.split(arrow, 1)[0]
        if key_eos == eos:
            tprob[name] /= num_eos

        elif key_sos != sos:
            tprob[name] /= n_pos[key_sos]

    return tprob
```

Infine la funzione *most_used_tag()* serve per contare i tag più usati per ogni parola: il primo ciclo serve per contare le frequenze delle combinazioni (parola / tag), il secondo per dividere le frequenze (parola / tag) per il numero di occorrenze della parola considerata, l'ultimo per associare a ogni parola il suo tag più frequente.

```
copy_dict = cprob.copy()
# trova la parola con il tag più frequente per una ciascuna parola
for name in cprob:
    key_name = name.split(slash, 1)[0]
    key_pos = name.split(slash, 1)[1]
    for copy in copy_dict:
        key_copy_name = copy.split(slash, 1)[0]
        if key_copy_name == key_name and cprob[name] >= copy_dict[copy]:
            frequent_tag_word.update({key_name: key_pos})
return frequent_tag_word
```

ultimo ciclo della funzione *most_used_tag()*

II. viterbiAlg.py

Questa classe possiede tre diverse funzioni: *get_eprob()*, *get_tprob()* e *viterbi()*.
Le prime due funzioni sono simili e servono a ricavare dagli omonimi dizionari la probabilità di emissione per una determinata coppia di (parola, tag) e quella di transizione per una determinata coppia di (tag i-1, tag).

```
# se non trova la probabilità ritorna la save_prob altrimenti ritorna quella corretta
def get_eprob(pos, token):
    name = pos + arrow + token
    if name not in eprob:
        return save_prob
    return eprob[name]


# se non trova la probabilità ritorna la save_prob altrimenti ritorna quella corretta
def get_tprob(tag, oldtag):
    name = oldtag + arrow + tag
    if name not in tprob:
        return save_prob
    return tprob[name]
```

La funzione *viterbi()* riceve in input un corpus che viene analizzato nel primo ciclo frase per frase. Vediamo l'inizializzazione di alcune strutture dati che abbiamo già descritto prima come *mat* e *backtrace*, altre nuove come *toker_arr* che rappresenta un array di token lungo quanto la lunghezza della frase più i due caratteri speciali SoS e EoS. Da notare che *toker_arr* viene inizializzato subito con tutti le parole presenti nella frase. Da notare sia il parametro *max_col* che indica la probabilità massima della colonna precedente, sia il parametro *index_max_col* che rappresenta l'indice in cui si trova la probabilità massima di una colonna.

```
def viterbi(corpus):

    for sentence in corpus:

        s_length = sentence.__len__()
        mat = np.zeros(shape=(pos_len, s_length + special_char))
        token_arr = np.empty(shape=(s_length + special_char), dtype=np.dtype('U20'))
        count = 0
        token_arr[count] = sos
        backtrace = np.empty(shape=(s_length + special_char), dtype=np.dtype('U5'))
        backtrace[0] = sos
        max_col = 1
        # corrisponde a sos nel tagset, quindi è il valora iniziale
        index_max_col = 0

        # inizializzo l'array di token
        for token in sentence:
            count += 1
            token_arr[count] = token.form

        token_arr[count + 1] = eos
```

inizializzazione viterbi()

All'interno del primo ciclo troviamo il cuore dell'algoritmo di Viterbi: inizialmente si procede ad operare colonna per colonna e in ogni colonna riga per riga.
La prima colonna, trattata con il primo if, ha bisogno solo della probabilità di emissione che viene ricavata tramite la funzione *get_eprob()* dando in input il pos_tag della riga e la parola della colonna, questa probabilità è salvata nella casella della matrice corrispondente.
Per le colonne successive avrò bisogno di considerare, tramite l'else, anche la probabilità di trasmissione quindi memorizzerò in *old_tag* il pos_tag nella casella della colonna precedente con la probabilità più alta.
Il parametro *temp_prob* memorizza la probabilità da inserire nella casella della matrice e corrisponde al massimo della colonna precedente moltiplicato per mille (per evitare che il numero della probabilità fosse troppo basso infatti grazie a questo l'accuracy migliora di circa 1%), moltiplicato per *e_prob* e *t_prob*.
L'if poco dopo serve nel caso ci sia un numero negativo, nel caso ci sia nella casella della matrice rimane lo zero dell'inizializzazione.
L'ultimo if aggiorna l'*index_max_col*, il *backtrace* e *max_col* tranne per la prima colonna.
Infine viene chiamata in causa la classe accuracyViterbi.py, in particolare la funzione *save_num()* che mantiene il conto delle parole classificate in modo corretto.

```
for col in range(token_arr.__len__()):
    for row in range(pos_len):
        # per la prima colonna salva nella matrice tutte le probabilità di emissione
        # (non calcolo quella di transizione perché nella prima colonna non c'è)
        if col == 0:
            e_prob = get_eprob(pos_array[row], token_arr[col])
            mat[row, col] = e_prob

        # per le successive colonne controllo le righe della colonna precedente
        # e salvo la massima prob per ogni casella della colonna attuale
        else:
            e_prob = get_eprob(pos_array[row], token_arr[col])
            old_tag = pos_array[index_max_col]
            t_prob = get_tprob(pos_array[row], old_tag)
            temp_prob = max_col * 1000 * float(e_prob) * float(t_prob)

            if temp_prob > mat[row, col]:
                mat[row, col] = temp_prob

    # calcolo il max della colonna e salvo il pos del max nel backtrace
    if col != 0:
        index_max_col = mat.argmax(axis=0)[col]
        backtrace[col] = pos_array[index_max_col]
        max_col = mat[index_max_col, col]

for i in range(backtrace.__len__()):
    print(token_arr[i] + space + backtrace[i])

accuracyViterbi.save_num(backtrace, sentence)
```

III. baselineAlg.py

Questa classe utilizza un algoritmo molto semplice per classificare le parole ai pos_tag: nel primo passo usa la funzione *most_used_tag()*, descritta in precedenza e appartenente alla classe prob.py, che ritorna un dizionario con una coppia (parola, pos_tag) in cui il pos_tag è il pos_tag più frequente per quella parola.
Dopo abbiamo due cicli for che analizzano ogni parola di ogni frase del corpus e controllano se la parola è presente nel dizionario o meno. Se non è presente utilizza una tecnica di smoothing per decidere il pos_tag, altrimenti prende il pos_tag dal dizionario. Come ultimo passaggio, prima di stampare l'accuracy, si aggiorna un dizionario che rappresenta in che modo sono state "taggate" le parole.

```
# assegna il tag a ogni token in base al più frequente tag di quella parola
def baseline_tagger(corpusx):
    freq_dict = most_used_tag(corpus)
    for sentence in corpusx:
        for token in sentence:

            if token.form not in freq_dict:
                pos = smooth.simple_smooth_bis()
            else:
                pos = freq_dict[token.form]

            pos_tagged.update({token.form: pos})

    accuracyBaseline.calc_accuracy(pos_tagged, file_pathdevgreek)
```

IV. smooth.py

Quest'ultima classe possiede sei funzioni che rappresentano vari tipi di smoothing che possono essere usati nel pos_tagging.
La funzione *simple_smooth()* assume che il pos corretto sia "noun", la *simple_smooth_bis()* invece ritorna il pos "noun" o quello "verb" con il 50% di probabilità per ognuno come se fosse un testa o croce. Queste due funzioni sono applicabili solo per l'algoritmo baseline in questa versione ma ne sono state create di identiche per l'algoritmo di Viterbi.
La funzione *smooth_ntag()* ritorna invece una probabilità calcolata dividendo uno per tutti i pos_tag.

```
# ritorna sempre NOUN
def simple_smooth():
    return "NOUN"


# ritorna al 50% NOUN e 50% VERB
def simple_smooth_bis():
    if random.randint(0, 1) < 1:
        return "NOUN"
    else:
        return "VERB"
```

```
# ritorna la probabilità 1/n_tag
def smooth_ntag():
    pos_array = get_tags()
    l_tag = len(pos_array)
    prob = 1 / l_tag
    return prob
```

L’ultima funzione è la più complicata delle sei, riceve in input un corpus e un tag da considerare. Il primo ciclo aggiunge al dizionario *one_word_dict* le parole che compaiono nel corpus una sola volta, il secondo salva nello stesso dizionario i tag di queste parole, il terzo conta le frequenze dei tag di *one_word_dict* e le salva in *percentage_one_word*, infine l’ultimo divide le frequenze per il totale.

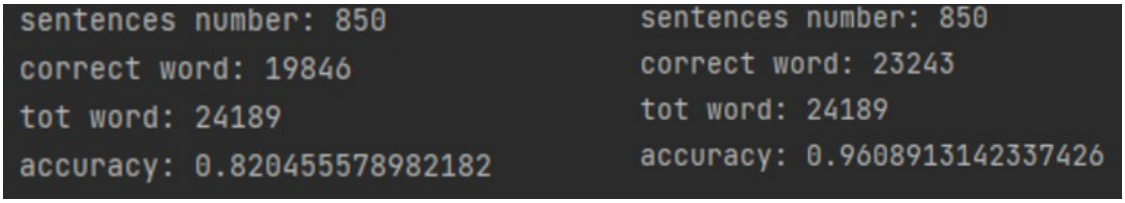
```
# controllo quante parole compaiono una sola volta nel devtest e ritorno la percentuale del tag in input
def smooth_dev(dev_corpus, tag):

    one_word_dict = {}
    percentage_one_word = {}
    word_dict = count_name(dev_corpus)

    # aggiunge nel dizionario one_word tutte le parole che hanno occorrenza uno
    for word in word_dict:
        if word_dict[word] == 1:
            one_word_dict.update({word: ''})

    # salva i tag delle parole con occorrenza uno
    for sentence in dev_corpus:
        for token in sentence:
            if token.form in one_word_dict:
                one_word_dict[token.form] = token.upos

    # conta quanti tag ci sono fra le parole che compaiono una volta sola e il loro totale
    for word in one_word_dict:

        if one_word_dict[word] not in percentage_one_word:
            percentage_one_word.update({one_word_dict[word]: 1})
        else:
            percentage_one_word[word] += 1

    # divide il numero di ciascun tag per il totale
    for pos in one_word_dict:
        percentage_one_word[pos] /= len(one_word_dict)

    return percentage_one_word[tag]
```

Altre Classi

Le altre classi sono: start.py che avvia il codice, accuracyViterbi.py e accuracyBaseline.py che entrambe stampano l’accuratezza uno per l’algoritmo di Viterbi, l’altro per la baseline.

IV. Conclusioni

Il PoS_Tagger funziona su entrambi le lingue antiche anche se la precisione è decisamente migliore per il latino come si vede nella prima figura rispetto al greco che è nella seconda immagine.
Le performance dell'algoritmo di Viterbi migliorano notevolmente rispetto alla baseline che per essere il più semplice possibile si fonda solo sul concetto di pos_tag più frequentemente associato a una parola.
Le varie tecniche di smoothing sono abbastanza semplici e non hanno per questo portato troppo beneficio soprattutto per l'algoritmo di Viterbi in cui la probabilità di emissione per una parola sconosciuta è stata trattata assegnandole una probabilità molto bassa indifferentemente dal pos_tag preso in considerazione. Questa mi è sembrata la scelta da prendere in quanto i risultati sono stati migliori.



Latino (baseline e Viterbi)

Greco (baseline e Viterbi)