PoS_ViterbiHMM

May 11, 2021

PoS tagging per latino e greco: confronto tra baseline e HMM (Viterbi) Francesco Sannicola

Librerie utilizzate:

- csv per la lettura dei corpus
- math per utilizzare la funzione logaritmo
- numpy, esattamente i moduli array, delete, empty. Vedremo una sola struttura utilizzare questi moduli
- time per il monitoraggio e il confronto dei tempi di esecuzione delle due strategie

[1]:
```
from csv import reader as csv_reader
from numpy import array as np_array
from numpy import delete as np_delete
from numpy import empty as np_empty
from math import log
import time
```

- Counter permetterà di contare le occorrenze di uno stesso elemento all'interno di una struttura

[2]:
```
from collections import Counter
```

- word_tokenize sarà utilizzato per estrarre le parole dalle frasi in input
- MWETokenizer permetterà di effettuare un preprocessing ai token (solo nel latino)

[3]:
```
from nltk import word_tokenize
from nltk.tokenize import MWETokenizer
```

Il metodo trainParsing(file) ha come input il file train del corpus.

In output avremo le coppie (parola, tag) lette dal file più le coppie (parola, INIT) e (parola, END). La struttura utilizzata, in questo caso, è la lista.

[4]:
```
def trainParsing(file):

    #w_e tutte le parole finali
    w_e = []
    #w_t tutte le parole con proprio tag più i delimitatori END e INIT
    w_t = []
    # w_s tutte le parole iniziali
```

```
    w_s= []

    w_t.append(('INIT', 'INIT'))
    with open(file) as fd:
        rd = csv_reader(fd, delimiter="\t", quotechar='"')
        i = -1
        for row in rd:
            if len(row) > 3:
                if i == 0:
                    w_s.append((row[1].lower(), 'INIT'))
                    i = 1
                w_t.append((row[1].lower(), row[3].lower()))
                last_str = row[1].lower()
            if len(row) == 0:
                w_e.append((last_str, 'END'))
                w_t.append(('END', 'END'))
                w_t.append(('INIT', 'INIT'))
                i = 0
        w_t.pop()
    return w_t, w_e, w_s
```

Calocolo le coppie (parola, tag) dal file dev.

[5]:
```
def devParsing(file):
    w_t_dev = []
    with open(file) as fd:
        rd = csv_reader(fd, delimiter="\t", quotechar='"')
        for row in rd:
            if len(row) > 3:
                w_t_dev.append((row[1].lower(), row[3].lower()))
    return w_t_dev
```

Parsing del file test: - calcolo coppie (parola, tag) - calcolo e parsificazione (solo per latino) delle frasi su cui testare gli algoritmi

[6]:
```
def testParsing(file):
    query_list = []
    w_t_test = []

    first_char_init = 9

    with open(file) as fd:
        rd = csv_reader(fd, delimiter="\t", quotechar='"')
        i = -1
        for row in rd:
            if len(row) == 1:
                if row[0].startswith('# text'):
```

```
                    if '...' not in row[0]:
                        query_list.append(row[0][first_char_init:(len(row[0]))].
↪lower().replace('.', ' .').replace('·', ' ·'))
                    else:
                        query_list.append(row[0][first_char_init:(len(row[0]))].
↪lower().replace('·', ' .'))

            elif len(row) > 3:
                w_t_test.append((row[1].lower(), row[3].lower()))
    return w_t_test, query_list
```

Percorso relativo dei corpus (test, dev, train).

[7]:
```
greek_train_tree_bank = "./Bank/Greek/grc_perseus-ud-train.conllu"
latin_train_tree_bank = "./Bank/Latin/la_llct-ud-train.conllu"

latin_dev_tree_bank = "./Bank/Latin/la_llct-ud-dev.conllu"
greek_dev_tree_bank = "./Bank/Greek/grc_perseus-ud-dev.conllu"

greek_test_tree_bank = "./Bank/Greek/grc_perseus-ud-test.conllu"
latin_test_tree_bank = "./Bank/Latin/la_llct-ud-test.conllu"
```

Metodo utilizzato più avanti per il quarto metodo di smoothing.

Sono necessari in input le coppie (parola, tag) appartenenti al file ***dev***.

Vengono calcolate e restituite in output: - le parole che appaiono una sola volta - le coppie (parola, tag) per le parole che compaiono una sola volta

[8]:
```
def singleWordDistribution(w_t_dev):
    #obtain tuples word:tag appearing one time
    single_occ_word_tag = dict()
    single_occ_words = list({key:val for key, val in Counter(i[0] for i in␣
↪w_t_dev).items() if val == 1})

    u=0
    for word in single_occ_words:
        for tup in w_t_dev:
            if tup[0] == word:
                single_occ_word_tag[word] = tup[1]
                u+=1
                break

    tag_occ_singleoccWord=Counter(single_occ_word_tag.values())
    return tag_occ_singleoccWord, single_occ_words
```

Attraverso Counter effettuo il conteggio delle occorenze delle coppie (parola, tag) per le tre strutture. Avremo, quindi, un oggetto Counter molto simile ad un dizionario: le coppie (parola, tag) come chiave e il relativo conteggio come valore.

Per ogni tag viene calcolato, inoltre, il numero di volte che appare nel corpus del file train.

[9]:
```
def computeOcc(w_t, w_s, w_e):
    w_t_occ = Counter(w_t)
    w_s_occ = Counter(w_s)
    w_e_occ = Counter(w_e)

    t_occ = Counter([i[1] for i in w_t])

    return w_t_occ, w_s_occ, w_e_occ, t_occ
```

Probabilità di emissione:

$$p(w|t) = \frac{P(t,w)}{P(t)}.$$

Come accennato in precedenza useremo la funzione logaritmo per evitare eventuali underflow dovute a probabilità estremamente basse.

$$log(p(w|t))$$

La funzione logaritmo restituirà sempre un valore minore o uguale a 0. Più l'evento è probabile e più il logaritmo è vicino allo zero. Nel caso che la probabilità fosse uguale a 0 e che la funzione logaritmo non sia applicabile a questo valore, si assegna alla probabilità un valore molto vicino allo zero (nel nostro caso 0.00001)

Dopo una serie di sperimentazioni è stato scelto di applicare il valore assoluto alla funzione logaritmo in modo di lavorare sempre con valori positivi.

$$|log(p(w|t))|$$

[10]:
```
def computeEmission(w_t_occ, t_occ):
    p_emission = dict()
    # prob w given t
    for key, value in w_t_occ.items():
        prob = value / t_occ.get(key[1])
        if prob == 0:
            prob = 0.00001
        if key[1] in p_emission:
            p_emission[key[1]].update({key[0]: abs(log(prob))})
        else:
            p_emission[key[1]] = {key[0]: abs(log(prob))}
    return  p_emission
```

Calcolo probabilità di emissione per le parole iniziali.

[11]:
```
def computeEmissionInit(w_s_occ, t_occ):
    p_emission_init = dict()
    #compute emission probability for initial state
    for key, value in w_s_occ.items():
```

```
        prob = value / t_occ.get(key[1])
        if prob == 0:
            prob = 0.00001
        if key[1] in p_emission_init:
            p_emission_init[key[1]].update({key[0]: abs(log(prob))})
        else:
            p_emission_init[key[1]] = {key[0]: abs(log(prob))}
    return p_emission_init
```

Calcolo probabilità di emissione per le parole finali.

[12]:
```
def computeEmissionEnd(w_e_occ, t_occ):
    p_emission_end = dict()
    #compute emission probability for end state
    for key, value in w_e_occ.items():
        prob = value / t_occ.get(key[1])
        if prob == 0:
            prob = 0.00001
        if key[1] in p_emission_end:
            p_emission_end[key[1]].update({key[0]: abs(log(prob))})
        else:
            p_emission_end[key[1]] = {key[0]: abs(log(prob))}
    return p_emission_end
```

Probabilità di transizione:

$$p(t_1|t) = \frac{P(t, t_1)}{P(t)}.$$

Anche in questo caso consideremo il valore assoluto del logaritmo della probabilità:

$$|log(p(t_1|t))|$$

[13]:
```
def computeTransition(w_t, t_occ):
    p_transition_dict = dict()
    # prob t1 given t
    for t1 in t_occ.keys():
        for t in t_occ.keys():
            count = 0
            for i in range(1, len(w_t)):
                if w_t[i][1] == t1:
                    if w_t[i - 1][1] == t:
                        count += 1
            prob = count / t_occ.get(t)
            if prob == 0:
                prob = 0.00001
            if t in p_transition_dict:
                p_transition_dict[t].update({t1: abs(log(prob))})
            else:
```

```
                p_transition_dict[t] = {t1: abs(log(prob))}
    return p_transition_dict
```

Calcolo tutti i possibili states.

[14]:
```
def getStates(p_transition_dict):
    i = 0
    states = np_empty(len(p_transition_dict) - 2 , dtype=object)
    for key in t_occ.keys():
        if str(key) != 'INIT' and str(key) != 'END':
            states[i] = str(key)
            i += 1
    return states
```

Ottengo tutte le parole presenti nel corpus train.

[15]:
```
def getCorpusWords(p_emission):
    all_words = []
    for value in p_emission.values():
        all_words.extend(list(value.keys()))
    return all_words
```

Una volta popolata la lista contentente i PoS calcolati è possibile andare a calcolare l'accuratezza, PoS corretti ed errati.

[16]:
```
def calculateAccuracy(all_pos, w_t_test):
    right_pos = 0
    wrong_pos = 0
    n = 0

    for word in all_pos:
        if (word[1] == w_t_test[n][1]):
            right_pos +=1
        else :
            wrong_pos +=1
        n += 1

    accuracy = right_pos/(right_pos+wrong_pos)

    print("Right PoS: %s" % right_pos)
    print("Wrong PoS: %s" % wrong_pos)
    print("Accuracy: %s" % accuracy)
```

Controllo se le parole all'interno di *all_pos* e la lista *w_t_test* combaciano esattamente (utile per debugging).

[17]:
```
def checkWrongWords(all_pos, w_t_test):
    same = 0
```

```
    for i in range(0, len(all_pos)):
        if all_pos[i][0] == w_t_test[i][0]:
            same+=1
        else:
            print(i)
```

All'interno del corpus per la lingua latina sono presenti delle parole come *"[adj]"* o *"[–]"* che necessitano un trattamento diverso rispetto alle altre. Le funzioni *.split()* oppure *word_tokenize* di nltk non sono in grado di ottenere il token corretto. Per esempio la singola parola *"[–]"* viene suddivisa in *"[", "–"* e *"]"*

Ci sono possibili soluzioni:

- non considerarle perchè potrebbero essere degli errori o delle annotazioni. Inoltre ne sono presenti in una percentuale veramente ridotta considerando il numero totale di parole e che non influisce più di tanto sull'inferenza
- trovare una metodologia che forzi la corretta tokenizzazione

E' stato deciso di implementare una strategia che permetta la corretta tokenizzazione. La variabile a seguire specifica tutti i token da processare in modo diverso.

[18]:
```
token_to_merge = [
                  ('[', 'adj', ']'), ('[', 'Adj', ']'),
                  ('[', 'adv', ']'), ('[', 'Adv', ']'),
                  ('[', 'aux', ']'), ('[', 'Aux', ']'),
                  ('[', 'cconj', ']'), ('[', 'Cconj', ']'),
                  ('[', 'det', ']'), ('[', 'Det', ']'),
                  ('[', 'init', ']'), ('[', 'Init', ']'),
                  ('[', 'noun', ']'), ('[', 'Noun', ']'),
                  ('[', 'num', ']'), ('[', 'Num', ']'),
                  ('[', 'part', ']'), ('[', 'Part', ']'),
                  ('[', 'pron', ']'),('[', 'Pron', ']'),
                  ('[', 'propn', ']'),('[', 'Propn', ']'),
                  ('[', 'punct', ']'),('[', 'Punct', ']'),
                  ('[', 'sconj', ']'),('[', 'Sconj', ']'),
                  ('[', 'verb', ']'),('[', 'Verb', ']'),
                  ('[', 'x', ']'), ('[', 'X', ']'),
                  ('[', '--', ']'),
                  ('[', 'participle', ']')
                 ]
```

Creo una istanza di oggetto di MWETokenizer passando i token su cui non fare lo split.

[19]:
```
tokenizer = MWETokenizer(token_to_merge)
```

Metodo che consente di stampare la tabella di Viterbi (usato principalmente per debugging).

[20]:
```
def dptable(V):
    yield " ".join(("%10d" % i) for i in range(len(V)))
```

```
    for y in V[0]:
        yield "%.7s: " % y+" ".join("%.7s" % ("%f" % v[y]) for v in V)
```

0.1 Implementazione metodo di Viterbi

0.1.1 Hidden Markov Model e programmazione dinamica.

In input è necessario specificare la tecnica di smoothing:

- 1: consideriamo tutte le parole sconosciute dei *noun*

$$P(unk|NOUN) = 1$$

- 2: consideriamo tutte le parole sconosciute dei *noun* oppure dei *verb* con la stessa probabilità

$$P(unk|NOUN) = 0.5 \; and \; P(unk|VERB) = 0.5$$

- 3: consideriamo le parole sconosciute con una probabilità che dipende dal numero di states (numero di tag)

$$P(unk|t_i) = 1/(number\ of\ PoS\ Tags)$$

- 4: tagghiamo le parole sconosciute secondo la distribuzione di parole che appaiono una sola volta all'interno del *dev corpus* calcolata precedentemente.

$$P(unk|t_i) = (number\ of\ words\ appear\ one\ time\ tagged\ with\ t_i)/(number\ of\ words\ appear\ one\ time)$$

In output avremo la lista *all_pos* contentente tutte le coppie (parola, tag) frutto delle computazioni.

Il primo for effettua un ciclo su tutte le frasi in input prese dal corpus test. Dopodichè si procede nel parsing della frase attraverso il MWETokenizer istanziato precedentemente e che risolverà i problemi relativi allo splitting di parole che in realtà non andrebbero divise.

Possiamo ora iniziare con il cosidetto ***Inizialization step***: il secondo for che vediamo in questo metodo si occupa di inizializzare la matrice di Viterbi (in questo caso chiamata backtrace) con le probabilità di transizione da *INIT* allo stato i-esimo moltiplicata per la probabilità che quella parola sia iniziale. Nel caso non si abbia mai visto quella parola come la prima di una frase allora si moltiplica per un valore abbastanza alto (20).

Da quest'ultima operazione possiamo iniziare a capire che, a differenza dell'algoritmo di Viterbi classico il quale punta nel massimizzare il risultato del prodotto tra probabilità di transizione e di emissione, l'obiettivo ora è minimizzare. Ricordiamo che tutti i valori di probabilità sono stati applicati alla funzione logaritmo e successivamente alla funzione valore assoluto. Così facendo avremo valori alti nel caso di probabilità basse. Perciò il nome delle variabili potrebbe trarre in inganno: non parliamo di probabilità ma di valori alti in caso di probabilità tendendi allo 0 e di valori bassi per probabilità tendenti a 1.

Passiamo ora al ***recursion step***: consideriamo ora dalla seconda parola fino all'ultima. Effettuiamo un ultimo passo di processing della stringa andando a togliere eventuali *"_"* dalle parole dovuto all'esecuzione di MWETokenizer. Anche in questo caso cicliamo su ogni stato e andiamo a popolare la matrice di viterbi andando a considerare il ***minimo*** prodotto tra la matrice di viterbi del livello precedente e la probabilità di transizione ed emissione. Se la parola attualmente in esame è sconosciuta allora procediamo nell'esecuzione di una tecnica di smoothing. Invece, se una parola

è conosciuta, ma non è mai stata taggata con un particolare tag (scopo del except KeyError) allora andremo ad moltiplicare per un valore alto (50).

Il ***termination step*** si occupa di scegliere un tag per l'ultima parola della frase minimizzando il prodotto tra la matrice al livello precendete, la probabilità di transizione dallo stato attuale allo stato finale e la probabilità che quella parola sia finale.

Effettuato il ***termination step***, grazie alla matrice che abbiamo costruito nei passi precedenti, possiamo calcolare i PoS più probabili.

[21]:
```
def ViterbiHMM(p_emission, p_emission_init, p_transition, query_list,␣
↪smoothing_mode, tag_occ_singleoccWord = None, single_occ_words = None):
    all_pos = []
    for query in query_list:

        input_splitted = tokenizer.tokenize(word_tokenize(query))
        T = len(input_splitted)

        # Tracking tables from first observation (Inizialization step)
        backtrace=[{}]
        for i in states:
            try:
                ␣
↪backtrace[0][i]=p_transition['INIT'][i]*p_emission_init['INIT'][input_splitted[0]]
            except KeyError:
                backtrace[0][i]=p_transition['INIT'][i] * 20


        for t in range(1, T):
            input_splitted[t] = input_splitted[t].replace('_', '')
            backtrace.append({})
            for y in states:
                #Termination step
                if t == T - 1:
                    try:
                        (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0]['END'] * p_emission[y][input_splitted[t]], y0) for y0 in␣
↪states)
                    except KeyError:
                        (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0]['END'] * 50, y0) for y0 in states)
                else:
                    if input_splitted[t] not in all_words:

                        if smoothing_mode == 1:
                            # P(unk|NOUN) =1
                            if y == 'noun':
```

```
                                (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * 0.0001 , y0) for y0 in states)
                            else:
                                (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * 50, y0) for y0 in states)

                        elif smoothing_mode == 2:
                            # P(unk|NOUN) =0.5 and P(unk|VERB) = 0.5
                            if y == 'noun':
                                 (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * abs(log(0.5)), y0) for y0 in states)
                            elif y == 'verb':
                                 (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * abs(log(0.5)), y0) for y0 in states)
                            else:
                                (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * 50, y0) for y0 in states)

                        elif smoothing_mode == 3:
                            #P(unk|ti) = 1/#(PoS_TAGs)
                            (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * abs(log(1 / len(states))), y0) for y0 in states)

                        elif smoothing_mode == 4:
                            #Another smoothing technique based on the dev file␣
↪and words which appear one time
                            for tag in states:
                                if y == tag:
                                    try:
                                        p_emission_new_word =␣
↪abs(log(tag_occ_singleoccWord[tag] / len(single_occ_words)))
                                    except ValueError:
                                        p_emission_new_word =  50
                                    (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * p_emission_new_word, y0) for y0 in states)
                                    break
                        else :
                            return
                    else:
                        try:
                            (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * p_emission[y][input_splitted[t]], y0) for y0 in states)
                        except KeyError:
                            (prob, state) = min((backtrace[t-1][y0] *␣
↪p_transition[y0][y] * 50, y0) for y0 in states)
                backtrace[t][y] = prob
```

```
            #for i in dptable(viterbi):
            #   print (i)
            opt=[]
            for j in backtrace:
                for x,y in j.items():
                    if j[x]==min(j.values()):
                        opt.append(x)
        # print ('The PoS are\n'+''
        #       .join(map(''.join, zip([x + '/' for x in input_splitted], [x +␣
↪'\n' for x in opt]))))
        for l in range(0,T):
            all_pos.append([input_splitted[l].replace('_', ''), opt[l]])
    return all_pos
```

0.1.2 Implementazione baseline

Semplice algoritmo basato sul numero di volte che una certa parola, all'interno del train set, appare con un determinato tag. Assegno alle parola in ingresso il tag con cui essa appare più volte all'interno del *train*.

[22]:
```
def Baseline(w_t_occ, query_list):
    all_pos=[]
    for query in query_list:
        input_splitted = tokenizer.tokenize(word_tokenize(query))
        T = len(input_splitted)
        tag_target = 'noun'
        for t in range(0, T):
            max = 0
            for key, value in w_t_occ.items():
                if key[0]==input_splitted[t].replace('_', ''):
                    if max < value:
                        max = value
                        tag_target = key[1]
            all_pos.append([input_splitted[t].replace('_', ''), tag_target])
    return all_pos
```

0.2 Latino

[23]:
```
w_t, w_e, w_s = trainParsing(latin_train_tree_bank)
w_t_dev = devParsing(latin_dev_tree_bank)
w_t_test, query_list = testParsing(latin_test_tree_bank)

tag_occ_singleoccWord, single_occ_words = singleWordDistribution(w_t_dev)

w_t_occ, w_s_occ, w_e_occ, t_occ = computeOcc(w_t, w_s, w_e)

start_time = time.time()
```

```
p_emission = computeEmission(w_t_occ, t_occ)
p_emission_init = computeEmissionInit(w_s_occ, t_occ)
p_emission_end = computeEmissionEnd(w_e_occ, t_occ)

p_transition = computeTransition(w_t, t_occ)
print("--- %s seconds (Calculate probability - latin) ---" % round(time.time() -␣
 ↪start_time, 2))

all_words = getCorpusWords(p_emission)
states = getStates(p_transition)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 1)
print("--- %s seconds (Viterbi, 1st smoothing - latin) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 2)
print("--- %s seconds (Viterbi, 2nd smoothing - latin) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 3)
print("--- %s seconds (Viterbi, 3th smoothing - latin) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 4,␣
 ↪tag_occ_singleoccWord = tag_occ_singleoccWord, single_occ_words =␣
 ↪single_occ_words)
print("--- %s seconds (Viterbi, 4th smoothing - latin) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)
```

```
--- 4.02 seconds (Calculate probability - latin) ---
--- 31.06 seconds (Viterbi, 1st smoothing - latin) ---
Right PoS: 22353
Wrong PoS: 1726
Accuracy: 0.9283192823622244
--- 30.97 seconds (Viterbi, 2nd smoothing - latin) ---
Right PoS: 22393
Wrong PoS: 1686
Accuracy: 0.9299804809169816
--- 31.19 seconds (Viterbi, 3th smoothing - latin) ---
```

```
Right PoS: 22436
Wrong PoS: 1643
Accuracy: 0.9317662693633456
--- 31.11 seconds (Viterbi, 4th smoothing - latin) ---
Right PoS: 22676
Wrong PoS: 1403
Accuracy: 0.9417334606918892
```

Analogamente a quanto fatto con Viterbi, analizzo i tempi di esecuzione della baseline.

[24]:
```
start_time = time.time()
all_pos = Baseline(w_t_occ, query_list)
print("--- %s seconds (Baseline - latin) ---" % round(time.time() - start_time,␣
 ↪2))
calculateAccuracy(all_pos, w_t_test)
```

```
--- 32.59 seconds (Baseline - latin) ---
Right PoS: 22869
Wrong PoS: 1210
Accuracy: 0.949748743718593
```

0.3 Greco

[25]:
```
w_t, w_e, w_s = trainParsing(greek_train_tree_bank)
w_t_dev = devParsing(greek_dev_tree_bank)
w_t_test, query_list = testParsing(greek_test_tree_bank)

tag_occ_singleoccWord, single_occ_words = singleWordDistribution(w_t_dev)

w_t_occ, w_s_occ, w_e_occ, t_occ = computeOcc(w_t, w_s, w_e)

start_time = time.time()

p_emission = computeEmission(w_t_occ, t_occ)
p_emission_init = computeEmissionInit(w_s_occ, t_occ)
p_emission_end = computeEmissionEnd(w_e_occ, t_occ)

p_transition = computeTransition(w_t, t_occ)
print("--- %s seconds (Calculate probability - latin) ---" % round(time.time() -␣
 ↪start_time, 2))


all_words = getCorpusWords(p_emission)
states = getStates(p_transition)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 1)
```

```
print("--- %s seconds (Viterbi, 1st smoothing - greek) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 2)
print("--- %s seconds (Viterbi, 2nd smoothing - greek) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 3)
print("--- %s seconds (Viterbi, 3th smoothing - greek) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)

start_time = time.time()
all_pos = ViterbiHMM(p_emission, p_emission_init, p_transition, query_list, 4,␣
 ↪tag_occ_singleoccWord = tag_occ_singleoccWord, single_occ_words =␣
 ↪single_occ_words)
print("--- %s seconds (Viterbi, 4th smoothing - greek) ---" % round(time.time()␣
 ↪- start_time, 2))
calculateAccuracy(all_pos, w_t_test)
```

```
--- 3.37 seconds (Calculate probability - latin) ---
--- 135.14 seconds (Viterbi, 1st smoothing - greek) ---
Right PoS: 14625
Wrong PoS: 6334
Accuracy: 0.6977909251395582
--- 129.54 seconds (Viterbi, 2nd smoothing - greek) ---
Right PoS: 15020
Wrong PoS: 5939
Accuracy: 0.7166372441433274
--- 129.3 seconds (Viterbi, 3th smoothing - greek) ---
Right PoS: 14633
Wrong PoS: 6326
Accuracy: 0.6981726227396345
--- 129.86 seconds (Viterbi, 4th smoothing - greek) ---
Right PoS: 15062
Wrong PoS: 5897
Accuracy: 0.7186411565437282
```

[26]:
```
start_time = time.time()
all_pos = Baseline(w_t_occ, query_list)
print("--- %s seconds (Baseline - greek) ---" % round(time.time() - start_time,␣
 ↪2))
calculateAccuracy(all_pos, w_t_test)
```

```
--- 138.1 seconds (Baseline - greek) ---
Right PoS: 13541
Wrong PoS: 7418
Accuracy: 0.6460709003292142
```

0.4 Analisi dei risultati

0.4.1 Accuratezza

Latino Per quanto riguarda la prima lingua presa in considerazione, il latino, abbiamo dei risultati interessanti. Le prestazioni in termini di accuratezza registrate dall'algoritmo *baseline* sono assolutamente notevoli: parliamo del ***95%*** delle parole taggate correttamente. L'applicazione del modello di Markov, in particolare dell'algoritmo di Viterbi, non ha portato a delle migliorie significative, anzi, l'utilizzo delle più generiche tecniche di smoothing portano a una diminuzione dell'accuratezza del 2%.

Tabella riassuntiva:

Tecnica di smoothing	Accuratezza	PoS corretti	PoS errati
Baseline	0.94974	22869	1210
Smoothing 1	0.92831	22353	1726
Smoothing 2	0.92998	22393	1686
Smoothing 3	0.93176	22436	1643
Smoothing 4	0.94173	22676	1403

Greco Discorso diverso per quanto riguarda la lingua greca. L'accuratezza di tutte le strategie non è elevata. L'algoritmo base si avvicina al ***65%***. In questo caso l'utilizzo dell'algoritmo di Viterbi ha comportato delle migliorie importanti con un:

- +6% applicando la prima e la terza strategia di smoothing
- +7% con la seconda e la quarta tecnica di smoothing

Nel dettaglio:

Tecnica	Accuratezza	PoS corretti	PoS errati
Baseline	0.64607	13541	7418
Smoothing 1	0.69779	14625	6334
Smoothing 2	0.71663	15020	5939
Smoothing 3	0.69817	14633	6326
Smoothing 4	0.71864	15062	5897

0.4.2 Tempi di esecuzione

Latino La complesità temporale delle varie strategie è allineata: sia la *baseline* che *Viterbi* impiegano all'incirca ***31*** secondi. Nel caso di Viterbi va considerato anche il tempo trascorso per calcolare le probabilità di emissione e di transizione (quest'ultima è calcolata in poco più di 4 secondi).

Tecnica di smoothing	Tempo di esecuzione (s)
Baseline	32.59
Smoothing 1	31.06
Smoothing 2	30.97
Smoothing 3	31.19
Smoothing 4	31.11

Greco L'andamento si ripete anche per il greco: pressochè paragonabili i tempi di esecuzione di *baseline* e *Viterbi* i quali aumentano notevolmente, se confrontati con il latino, a causa dell'incremento del corpus. Anche in questo caso c'è da aggiungere circa 3 secondi per il calcolo delle probabilità di transizione ed emissione necessarie per *Viterbi*.

Tecnica di smoothing	Tempo di esecuzione (s)
Baseline	138.1
Smoothing 1	135.14
Smoothing 2	129.54
Smoothing 3	129.3
Smoothing 4	129.86

0.4.3 Conclusioni

I risultati empirici ottenuti dimostrano che non sempre conviene attuare delle strategie più articolate per risolvere questo tipo di problemi. Il PoS tagger base per il latino, dopo una fase di pre-processing delle parole all'interno del corpus, raggiunte un livello di accuratezza già parecchio elevato. Migliorare questo già ottimo risultato non è scontato. Per il greco, invece, partiamo da una *baseline* decisamente più bassa e che l'algoritmo di Viterbi riesce a migliorare senza troppi problemi. Questo risultato è condizionato dalla presenza di un numero superiore di frasi presenti nel *corpus* train del greco e che consente un più accurato calcolo delle probabilità.

[ ]: