Relazione NER Tagging multilingua con Hidden Markov Model

Giulio Taralli e Ismaila Toure

Appello Luglio 2024

Indice

1 Introduction

Il NER (Named Entity Recognition) tagging multilingua è un tecnica che consiste nell'individuazione di entità come nomi di persone, organizzazioni, luoghi all'interno di un testo. Il NER è molto utile nella NLP in quanto permette di fare sentiment analysis oppure information extraction. All'interno della struttura linguistica a livelli il NER Tagging è situato nella parte superiore del livello morfologico, in quanto ha a che fare sia con il livello morfologico lessicale (ad ogni parola associa un tag), che con il livello sintattico (vogliamo capire se la parola Napoli, nel contesto della frase, si riferisce alla città o alla squadra di calcio).

2 Scopo del progetto

Lo scopo del progetto è quello di implementare un NER Tagging utilizzando l'algoritmo HMM (Hidden Markov Model) valutando il comportamento su 3 dataset distinti di 3 lingue diverse (italiano, spagnolo, inglese) e utilizzando e valutando diverse tecniche di smoothing sulle parole sconosciute. Attraverso l'accuracy sui tag, e la precision e la recall sulle entità predette dall'algoritmo HMM si confrontano questi risultati su due baseline differenti.

3 Scelte implementative

3.1 Oggetti usati

Si è deciso di implementare una classe Quad che rappresenta una quadrupla di un'entità, essa verrà utilizzata durante la fase di valutazione, più precisamente durante il calcolo della precision e della recall sulle entità.

```python
class Quad:
    def __init__(self, entity, indexSentence, indexStart, indexEnd):
        self.entity = entity
        self.indexSentence = indexSentence
        self.indexStart = indexStart
        self.indexEnd = indexEnd

    def __eq__(self, other):
        return (self.entity == other.entity and
                self.indexSentence == other.indexSentence and
                self.indexStart == other.indexStart and
                self.indexEnd == other.indexEnd)

    def __hash__(self):
        return hash((self.entity, self.indexSentence, self.indexStart, self.indexEnd))

    def toString(self):
        return f"Quad(Entity={self.entity}, indexSentence={self.indexSentence}, indexStart={self.indexStart}, indexEnd={self.indexEnd})"
```

Gli attributi della classe Quad sono:

- entity: l'entità della quadrupla
- indexSentence: l'indice della frase a cui la quadrupla fa riferimento
- indexStart: l'indice della parola di partenza della quadrupla all'interno della frase
- indexEnd: l'indice della parola finale della quadrupla all'interno della frase

3.2 Funzione fit

La funzione HMMFit prende in input una lista di liste parsificata attraverso la libreria conllu. All'interno della funzione fit quello si andrà a fare sarà calcolare le due distribuzioni che torneranno utili durante la fase di deconding:

- La probabilità di transizione: $P(t_i \mid t_{i-1})$., nel codice è tagDistribution
- La likelihood (probabilità di verosimiglianza): $P(w_i \mid t_i)$, nel codice è wordTagDistribution

La fase di learning dell'HMM risulta essere un conteggio sulle quali si andranno a calcolare le probabilità:

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

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

Come scelte implementative per l'ottenimento delle probabilità si è deciso di utilizzare le probabilità logaritmiche in quanto potremmo avere delle probabilità molto piccole durante la fase di learning, con il rischio durante la fase di decoding di perdere accuratezza dei valori di probabilità.

- I valori nella scala logaritmica sono tra $(-\infty, 0]$.
- per rappresentare un evento E con P(E) = 0, si è scelto di è scelto di utilizzare una variabile epsilon = 10^{-10}. da inserire all'interno del logaritmo

Un'altra scelta implementativa della fase di learning è quella di utilizzare come strutture dati sia per effettuare il conteggio, sia per i due output della funzione sono i dizionari di dizionari, in quanto:

- Non permettono ripetizioni: essendo delle coppie chiave valore
- Dentro il valore associato ad una chiave sarà presente un sotto-dizionario il quale conterrà altri valori chiavi associati ai conteggi

```
# complete creation of tagDistribution dictionary
# we use the log-probability for better accuracy (since the probability are very small)
# if the probability is 0, then the log-probability will be log(10**-10)
epsilon = 10**-10
for eventTag in countTag.keys():
    tagDistribution[eventTag] = {}
    for condTag in countTag.keys():
        if transitionCount[eventTag][condTag]/countTag[condTag] == 0:
            tagDistribution[eventTag][condTag] = math.log(epsilon)
        else:
            tagDistribution[eventTag][condTag] = math.log(transitionCount[eventTag][condTag]/countTag[condTag])
```

```
# complete creation of wordDistribution dictionary
# we use the log-probability for better accuracy (since the probability are very small)
# if the probability is 0, then the log-probability will be log(10**-10)
for word in wordTagCount.keys():
    wordDistribution[word] = {}

    uniqueTagDistribution.pop("START", None)
    uniqueTagDistribution.pop("END", None)
    if sum(wordTagCount[word][tagLoop] for tagLoop in countTag.keys()) == 1:
        tag = [tagLoop for tagLoop in countTag.keys() if wordTagCount[word][tagLoop] == 1]
        uniqueTagDistribution[tag[0]] += 1

    for tag in countTag.keys():
        if wordTagCount[word][tag]/countTag[tag] == 0:
            wordDistribution[word][tag] = math.log(epsilon)
        else:
            wordDistribution[word][tag] = math.log(wordTagCount[word][tag]/countTag[tag])
```

3.3 Funzione predict

All'interno della funzione predict si richiama, per ogni frase, la funzione viterbiAlgorithm che applica il decoding utilizzando l'algoritmo di viterbi sulla singola frase. In questa funzione si è deciso di non utilizzare i dizionari di dizionari come strutture dati per salvare le parole di una frase di test in quanto può capire di utilizzare le stesse parole durante la frase (esempio: l'uso dell'articolo determinativo, un nome proprio, ecc...) mentre in questo caso si vuole predire il NER Tag di ogni parola all'interno della frase. Si è deciso quindi di utilizzare delle matrici classiche sia per viterbi che per il backpointer.

```
def HMMPredict(test_senteces, tagDistribution, wordDistribution, uniqueTagDistribution, strategySmoothing):
    b_value = 0 # in log probabilities 0 is the neutral element
    solution = []

    for sentence in test_senteces:
        sentenceSolution = viterbiAlgorithm(sentence, tagDistribution, wordDistribution, b_value, uniqueTagDistribution, strategySmoothing)
        # recontruct and add to the solution...
        solution.append(sentenceSolution)

    return solution
```

All'interno della funzione HMMPredict si un'importante variabile: b_value e un parametro importante: strategySmoothing:

- valore neutro che andiamo a sommare nelle probabilità logaritmiche durante la fase di inizializzazione di viterbi. Il suo valore è 0 in quanto nelle probabilità logaritmiche si sommano tra di loro, e nella somma lo 0 è l'elemento neutro.
- rappresenta la strategia di smoothing nel caso in cui si trova una parola sconosciuta, a seconda del valore si applicherà uno smoothing diverso:
  - 0: assegnamo la probabilità log(1) alla parola sconosciuta associata al tag "O"
  - 1: assegnamo la probabilità log(0.5) alla parola sconosciuta associata sia al tag "O" che al tag "MISC)
  - 2: smoothing uniforme: assegnamo log(1/count(NER_TAGs)) alla parola sconosciuta su tutti i NER Tag
  - 3: assegnamo log(1/numero di parole che compaiono una volta sola dato quel tag) su tutti i NER TAG
    - Se non ci sono parole che compaiono una volta sola associate a un particolare NER Tag si assegna la probabilità $\log(10^{-10})$

```python
# strategy can be:
# 0: always "O": P(unk|O) = 1
# 1: always "O" and "MISC": P(unk|O) = P(unk|B-MISC) = 0.5
# 2: uniform: P(unk|tag(i)) = 1 / #(NER_TAGs)
# 3: 1 / # of word's tags appearing only once
def smoothing(strategy, currentTag, totNERtags, uniqueTagDistribution):
    epsilon = 10**-10
    match(strategy):
        case 0:
            if currentTag == "O":
                return math.log(1)
            else:
                return math.log(epsilon)
        case 1:
            if currentTag == "O" or currentTag == "B-MISC":
                return math.log(0.5)
            else:
                return math.log(epsilon)
        case 2:
            return math.log(1 / totNERtags)
        case 3:
            if uniqueTagDistribution[currentTag] != 0:
                return math.log(1 / uniqueTagDistribution[currentTag])
            else:
                math.log(epsilon)
        case _:
            return math.log(epsilon)
```

3.4 Funzione evaluation

Nella funzione evaluation si calcolano tre metriche di valutazione: accuracy, precision e recall. La metrica accuracy valuta parola per parola se il NER Tag predetto è uguale a quello vero presente nel test set.
Una volta calcolata l'accuracy, si passa alla conversione del risultato della predizione e del contenuto del test set in liste di oggetti Quad in modo tale da calcolare precision e recall sulle entità.
Per le metriche precision e recall la questione è leggermente più complessa: nel dibattito attuale non si è ancora arrivati a una definizione precisa universale. Si può implementare una precision e recall lasca: che quindi si conteggia come true positive una quadrupla predetta corretta senza che gli indici di partenza e di fine siano esattamente uguali a quelli dell'entità del test set, oppure si può implementarne una molto restrittiva: che quindi tiene in considerazione tutti gli attributi della classe Quad, e perché possa essere identificata come true positive deve combaciare esattamente con quella del test set.
Come scelta implementativa noi abbiamo identificato:

- il true positive come: la quadrupla predetta identica a quella del test set
- false positive come: una quadrupla predetta che non è presente all'interno della liste delle quadruple del test set
- false negative: una quadrupla del test set che non è presente all'interno della lista di quadruple predette dall'algoritmo HMM

```
truePositive = [pred for pred in predictions if pred.entity == trueClass and
(pred.entity, pred.indexSentence, pred.indexStart, pred.indexEnd) in
dict_testSentences]
    falsePositive = [pred for pred in predictions if pred.entity == trueClass and
(pred.entity, pred.indexSentence, pred.indexStart, pred.indexEnd) not in
dict_testSentences]
    falseNegative = [test for test in test_sentences if test.entity == trueClass
and (test.entity, test.indexSentence, test.indexStart, test.indexEnd) not in
dict_predictions]
```

Per migliorare la complessità temporale, prima di calcolare i valori true positive, false positive e false negative si è deciso di convertire le due liste di quadruple in dizionari di quadruple, ognuna associata a una tupla (entità, indice frase, indice inizio, indice fine) in modo tale che la ricerca di ognuna di queste quadruple sia O(1) e non appesantisca ulteriormente la complessità temporale.

4 Descrizione delle due baseline

Dopo il calcolo delle metriche di valutazione dell'algoritmo HMM, si andrà a confrontarle con le stesse metriche ma usando due algoritmi differenti di NER Tagging:

- Uno molto semplice: che associa a ogni parola il NER Tag più frequente
- Uno più difficile che utilizza l'algoritmo MEMM (Maximum Entropy Markov Model)

Non è stato possibile implementare, e quindi confrontare i risultati dell'HMM con quelli generati dalla baseline più difficile in quanto si sono riscontrati vari problemi dovuti a un codice mal scritto all'interno della libreria fornita nella consegna:

- aggiunto file = open(filename, 'r', encoding='utf-8') alla funzione load_data (riga 175)
- aggiunto un padding ai token (righe 200-202), altrimenti andava in eccezione (riga 208-210)

E nonostante le correzioni, l'esecuzione della libreria ha prodotto un:

- MemoryError: Unable to allocate 32.5 GiB for an array with shape (88400, 266) and data type ¡U371

Per cui siamo di fronte ad una limitazione fisica di memoria con il codice fornito in consegna.

5 Risultati

Si analizzano adesso i vari risultati nei diversi dataset con le varie tecniche di smoothing

-	**Strategy 0**	**Strategy 1**	**Strategy 2**	**Strategy 3**
IT	0.9703928245162362	0.9704311550918976	0.9740821424236423	0.9725744731142953
ES	0.9734944190710567	0.9740023733726852	0.9785739620873418	0.9776456318119519
EN	0.9566880649810036	0.9569613145926522	0.962302783028579	0.9610675450581123

Tabella 1: Accuracy

-	**Baseline facile**
IT	0.9411018762816786
ES	0.9419530842897617
EN	0.931242162789392

Tabella 2: Accuracy baseline

-	IT		ES		EN	
-	precision	recall	precision	recall	precision	recall
MISC	0.0920	0.4411	0.1787	0.5299	0.2305	0.6058
ORG	0.6204	0.6455	0.3588	0.4618	0.4361	0.4952
LOC	0.7336	0.7408	0.7102	0.8230	0.6257	0.6551
PER	0.6838	0.7098	0.6225	0.6739	0.5594	0.5702

Tabella 3: Precision e recall della baseline semplice sui vari dataset.

Con tutte le tecniche di smoothing uitilizzate, le accuracy risultano essere tutte maggiori rispetto all'accuracy della baseline per ogni lingua utilizzata.
Si note che la lingua spagnola risulta essere quella che produce l'accurattezza maggiore, mentre la lingua inglese produce le accurrattezze più piccole. Questo si potrebbe spiegare dal fatto che nell'inglese le parole sono molto più ambigue rispetti alle lingue neo-latine.

Precision e recall si assomigliano su ogni lingua, ciò che li distingue è la strategia di smoothing applicata:

- per qualsiasi strategia, le precison e recall più basse sono sempre sull'entità MISC, perché essendo un tag miscellanea, qualsiasi parola che non sia ORG, LOC, PER oppure O, verrà taggata MISC.
- la strategia 2 (uniforme) e la strategia 3 (distribuzione di probabilità sulla parola che compare una volta sola nel train) si comportano meglio rispetto alle altre producendo i valori più alti.

5.1 Dataset italiano

Strategy 0	**Precision**	**Recall**
MISC	0.6547	0.5352
ORG	0.8376	0.7818
LOC	0.8690	0.7965
PER	0.8038	0.7050

Tabella 4: Strategy 0

Strategy 1	**Precision**	**Recall**
MISC	0.6576	0.5453
ORG	0.8392	0.7018
LOC	0.8688	0.7975
PER	0.8040	0.7049

Tabella 5: Strategy 1

Strategy 2	**Precision**	**Recall**
MISC	0.7895	0.5581
ORG	0.8438	0.7899
LOC	0.8690	0.7973
PER	0.9137	0.8125

Tabella 6: Strategy 2

Strategy 3	**Precision**	**Recall**
MISC	0.6816	0.5533
ORG	0.7075	0.7953
LOC	0.8748	0.8038
PER	0.9158	0.8113

Tabella 7: Strategy 3

5.2 Dataset spagnolo

Strategy 0	**Precision**	**Recall**
MISC	0.7617	0.6639
ORG	0.7984	0.7499
LOC	0.9035	0.8911
PER	0.7872	7246

Tabella 8: Strategy 0

Strategy 1	**Precision**	**Recall**
MISC	0.7614	0.6796
ORG	0.8037	0.7499
LOC	0.9042	0.8911
PER	0.7884	0.7246

Tabella 9: Strategy 1

Strategy 2	**Precision**	**Recall**
MISC	0.7981	0.6931
ORG	0.8191	0.7807
LOC	0.9062	0.8916
PER	0.9253	0.8753

Tabella 10: Strategy 2

Strategy 3	**Precision**	**Recall**
MISC	0.8152	0.7091
ORG	0.6862	0.8196
LOC	0.9179	0.8994
PER	0.9248	0.8758

Tabella 11: Strategy 3

5.3 Dataset inglese

Strategy 0	**Precision**	**Recall**
MISC	0.7058	0.5889
ORG	0.7600	0.6380
LOC	0.8006	0.7086
PER	0.6768	0.5459

Tabella 12: Strategy 0

Strategy 1	**Precision**	**Recall**
MISC	0.7082	0.5953
ORG	0.7610	0.6380
LOC	0.8007	0.7086
PER	0.6745	0.5461

Tabella 13: Strategy 1

Strategy 2	**Precision**	**Recall**
MISC	0.7302	0.6010
ORG	0.7763	0.6540
LOC	0.8181	0.7253
PER	0.8596	0.7098

Tabella 14: Strategy 2

Strategy 3	**Precision**	**Recall**
MISC	0.7411	0.6098
ORG	0.6669	0.6676
LOC	0.8234	0.7283
PER	0.8617	0.7083

Tabella 15: Strategy 3