Direct Translate

Emanuele Gentiletti Alessandro Caputo

Introduzione

La consegna del progetto richiedeva l'implementazione di due componenti:

- Un PoS tagger basato su Hidden Markov Model e algoritmo di Viterbi
- Un traduttore direct da inglese a italiano, funzionante sulle seguenti frasi:
  - "Il droide nero poi abbassa la maschera e l'elmetto di Vader sulla sua testa."
  - "Questi non sono i droidi che stai cercando."
  - "I tuoi amici possono fuggire, ma tu sei condannato."

Seguendo la struttura del progetto, articoliamo la seguente relazione in due parti, una dedicata al PoS tagger e una al traduttore. L'implementazione del progetto è realizzata in Python, con l'ausilio delle seguenti librerie:

- `pyconll`: Parsing dei treebank
- `pyyaml`: Parsing di file yaml, usati per rappresentare il lessico nella fase di traduzione
- `toolz`: Utilità varie per la programmazione funzionale.

PoS Tagger

Addestramento

L'implementazione del PoS tagger è articolata nelle fasi di addestramento e di implementazione del tagger. Per l'addestramento, abbiamo fatto uso di `dict` nidificati per salvare le matrici di probabilità. L'Hidden Markov Model nella sua interezza è contenuto nella seguente classe:

```
class HMM(NamedTuple):
    transitions: Dict[str, Dict[str, float]]
    emissions: Dict[str, Dict[str, float]]
```

`transitions` contiene le probabilità di transizione. Per leggere la probabilità di transizione da `"ADJ"` a `"NOUN"` si accede così al dizionario:

```
transitions["NOUN"]["ADJ"]
```

Si noti che i PoS nell'accesso sono invertiti rispetto a quello che ci si aspetterebbe naturalmente. Alla fine dell'addestramento andiamo a trasporre manualmente la matrice, in modo che la disposizione rispecchi più naturalmente i pattern di accesso che usiamo nell'implementazione di Viterbi.

Di fatto, andando ad accedere a `transitions["NOUN"]`, il risultato che otteniamo è un `dict` così strutturato: $\{\texttt{key} : P(\texttt{"NOUN"}|\texttt{key})\}$

Le probabilità di transizione sono calcolate conteggiando quante volte ogni transizione avviene all'interno del treebank, in `dict` annidati con la seguente struttura:

```
transitions_counts = {
  "NOUN": {
    "NOUN": 2567,
    "ADJ": 1278,
    ...
  },
  "ADJ": { ... }
}
```

Dove l'elemento `transition_counts[k1][k2]` contiene il numero di volte in cui nel treebank avviene la transizione da `k1` a `k2`.

Per convertire i conteggi in distribuzioni di probabilità condizionate, usiamo la seguente funzione, che applichiamo a ogni dizionario annidato:

```
def div_by_total_log(counts: dict):
    denom = log(sum(counts.values()))
    return {k: log(v) - denom for k, v in counts.items()}
```

Ogni dizionario `transition_counts[k1]` contiene tutti i conteggi delle transizioni che iniziano con `k1`. Sommando tutti i conteggi, otteniamo il numero totale di transizioni che iniziano con `k1` nel treebank. Chiamiamo quindi questo valore `denom`, e dividiamo ogni valore in `transition_counts[k1]` per `denom`. Il risultato che otteniamo sarà la distribuzione di probabilità $P(\texttt{k2}|\texttt{k1})$.

Le probabilità sono rappresentate in forma logaritmica, per minimizzare l'errore dovuto a valori float troppo piccoli. Per cui, invece di andare a dividere, convertiamo entrambi i valori nei loro logaritmi e li andiamo a sottrarre.

Per le probabilità di emissione, la procedura è del tutto analoga. Il dizionario in questo caso ha struttura $\texttt{emissions[tok][pos]} = P(\texttt{tok}|\texttt{pos})$.

Nell'addestramento provvediamo anche a effetturare lo smoothing per le probabilità di transizione ed emissione. Nelle probabilità di transizione andiamo ad agire direttamente sui conteggi: se non ci sono transizioni da A a B, andiamo ad assegnare `transition_counts[A][B] = 1`.

Per le probabilità di emissione, andiamo invece a calcolare la distribuzione dei PoS tag per le parole che appaiono una sola volta all'interno del treebank.

TODO

La procedura di training per le transizioni può essere descritta concisamente dal seguente statement:

```
transitions = pipe(
    training_set,              # il treebank
    transition_counts,         # conta le transizioni
    smooth_transitions,        # esegui smoothing sui conteggi
    valmap(div_by_total_log),  # dividi ogni dict interno per il totale
    transpose,                 # trasponi la matrice
)
```

La funzione `pipe` chiama le funzioni passate come argomenti una dopo l'altra, passando come argomento il risultato della funzione precedente. `pipe(arg, do_a, do_b, do_c)` restituisce `do_c(do_b(do_a(arg)))`.

In modo simile, la procedura per le probabilità di emissione è la seguente:

```
emissions = pipe(
    training_set,
    emission_counts,
    valmap(div_by_total_log),
    transpose
)
smoothing = pipe(
    dev_set,
    smoothing_counts,
    div_by_total_log
)
emissions = dict_with_missing(emissions, smoothing)
```

Dove `dict_with_missing` crea un `dict` che restituisce il secondo argomento (`smoothing`) quando si tenta di accedere a una chiave mancante (una parola non contenuta nel treebank).

La procedura completa di training è nel file `tagger/hmm.py`. Per ottenere un'istanza di HMM contente i dati di training, è sufficiente chiamare la funzione `hmm_ud_english()`.

Tagger

L'implementazione di Viterbi è basata fortemente su una funzione d'utilità, `merge_with`:

```
def merge_with(fn, d1, d2):
    return {key: fn(d1[key], d2[key]) for key in d1.keys() & d2.keys()}
```

La funzione accetta come argomenti una funzione `fn` e due dizionari `d1` e `d2`. Per ogni chiave in comune dei dizionari `k`, la funzione calcola `fn(d1[k], d2[k])`.

Quindi ad esempio:

```
from operator import add

merge_with(add, {'a': 1, 'b': 2, 'c': 2}, {'a': 2, 'b': 2})
# out: {'a': 3, 'b': 4}
```

Questo permette di effettuare operazioni analogamente a come verrebbero effettuate su vettori, ma usando etichette esplicite per identificare i singoli componenti del vettore invece degli indici. La funzione, inoltre, scarta dal risultato i valori che i dizionari non hanno in comune. Nell'implementazione, sfruttiamo questa proprietà per evitare alcuni calcoli inutili.

La classe in cui viene implementato l'algoritmo è la seguente. L'implementazione è contenuta nel file `tagger/viterbi_tagger.py`.

```
class ViterbiTagger:
    def __init__(self, hmm: HMM):
        self.hmm = hmm

    def pos_tags(self, tokens: List[str]) -> List[str]:
        transitions, emissions = self.hmm

        # Mantiene in memoria solo l'ultima colonna invece di tutta la matrice.
        viterbi = merge_with(add, get_row(transitions, "Q0"), emissions[tokens[0]])
        backptr = []

        for token in tokens[1:]:
            viterbi, next_backptr = self._next_col(viterbi, token)
            backptr.append(next_backptr)

        viterbi = merge_with(add, viterbi, transitions["Qf"])
        path_start = max(viterbi.keys(), key=lambda k: viterbi[k])
        return retrace_path(backptr, path_start)
```

La classe viene inizializzata con un Hidden Markov Model già addestrato. L'algoritmo è contenuto nel metodo `pos_tags`, che riceve una lista di parole e restituisce una lista di rispettivi PoS tag.

Nell'implementazione, si fa spesso uso dell'operazione `merge_with(add, x, y)`. Il senso da attribuire a questa operazione è sempre la moltiplicazione di vettori di probabilità (dove la moltiplicazione è commutata in addizione perché lavoriamo con logaritmi). Nella seguente spiegazione, mi prendo la libertà di abbreviare l'espressione con `x * y` per chiarezza.

La prima operazione effettuata è inizializzare la prima colonna della matrice `viterbi`.

```
viterbi = get_row(transitions, "Q0") * emissions[tokens[0]]
```

`Q0` è uno pseudo PoS tag, che rappresenta l'inizio della frase. Ad esempio, $P(\texttt{ADJ}|\texttt{Q0})$ rappresenta la probabilità che ci sia un aggettivo all'inizio di una frase.

`get_row(transitions, "Q0")` restituisce la riga di valori corrispondenti a `"Q0"` (dove in `transitions[k1][k2]`, la colonna è considerata etichettata `k1` e la riga `k2`).

Per cui, per ogni PoS tag `k`:

$$\texttt{get\_row(transitions, "Q0")[k]} = P(\texttt{k}|\texttt{Q0})$$

Nel caso di `emissions[tokens[0]]`, possiamo invece dire che:

$$\texttt{emissions[tokens[0]][k]} = P(\texttt{k}|\texttt{tokens[0]})$$

Possiamo quindi dare il seguente significato all'operazione complessiva:

$$\texttt{viterbi[k]} = P(\texttt{k}|\texttt{Q0}) \cdot P(\texttt{k}|\texttt{tokens[0]})$$

```
    def _next_col(self, last_col, token):
        transitions, emissions = self.hmm

        viterbi = {}
        backptr = {}

        for pos in emissions[token].keys():
            paths_to_pos = merge_with(add, last_col, transitions[pos])
            backptr[pos], viterbi[pos] = max(paths_to_pos.items(), key=lambda it: it[1])

        viterbi = merge_with(add, viterbi, emissions[token])
        return viterbi, backptr
```