# Riassunto Mazzei

Matteo Brunello

# 1 Morfologia

## 1.1 Algoritmo di Viterbi

L'algoritmo di Viterbi e' un algoritmo di programmazione dinamica per calcolare in tempo polinomiale la sequenza di tag che massimizza la likelihood. Questo perche' per un approccio naive, calcolare a mano tutte le possibili sequenze richiederebbe un tempo esponenziale. L'idea e' quella che si basa essenzialmente sul fatto che la probabilita' della sequenza di tag $t_1, \ldots, t_n$ che massimizza la verosimiglianza puo' essere divisa essenzialmente in 2 parti:

1. La probabilita' della migliore sequenza di tag $t_1, \ldots, t_{n-1}$.
2. Il massimo prodotto tra la probabilita' di transizione del tag $t_n$, dato $t_{n-1}$ e la probabilita' di osservazione della parola $w_n$ dato $t_n$.

Il prodotto di queste due parti, ci da la probabilita' della sequenza di tag con probabilita' massima, per cui possiamo essenzialmente sfruttare questo fatto per costruire passo passo la sequenza, semplicemente selezionando ad ogni passo il tag che massimizza la probabilita' (parte 2), e salvando la probabilita' massima in un'opportuna struttura dati. In altri termini

$$\begin{aligned} v_t(j) &= \max_{i=1}^{N} v_{t-1} a_{ij} b_j(o_t) \\ &= \max_{i=1}^{N} v_{t-1} P(t_j \mid t_i) P(w_t \mid t_j) \end{aligned}$$

Tale struttura e' una matrice $T \times N$, dove $T$ e' il numero di PoS tags e $N$ e' la lunghezza della frase.

## 1.2 NER Tagging

Il Named Entity Recognition e' il task di trovare le Named Entities in un testo, cioe' qualsiasi elemento che puo' essere riferito con un nome proprio. Alcuni esempi di tag comuni possono essere:

- `LOC` (Location): New York City
- `PER` (Person): Pietro Smusi
- `ORG` (Organization): Stanford University

E' costituito da 2 sottotask:

1. Trovare gli span che costituiscono nomi propri
2. Taggare con i tag corretti questi span

### 1.2.1 Difficolta'

Le difficolta' del NER tagging sono sottolineate da due ragioni principali:

1. **Segmentazione**: In questo caso, non assegnamo un tag ad ogni parola, ma dobbiamo trovare il segmento stesso da taggare
2. **Ambiguita'**: i nomi propri sono molto ambigui, ad esempio lo stesso nome potrebbe essere utilizzato per riferirsi ad organizzazioni, persone e posti diversi (es. Washington).

### 1.2.2 Risoluzione della segmentazione

Possiamo risolvere il problema della segmentazione del NER tagging con il BIO tagging. In questo caso si trasforma il problema andando a introdurre 3 tag:

- `B` - Begin
- `I` - Inside
- `O` - Outside

I tag di `B` (Begin) e `I` (Inside) vengono tipati con il tipo del tag NER (es. `B-Person`), in questo modo si risolve il problema della segmentazione poiche' si ha un tag per ogni parola.

### 1.3 Differenze tra HMM e MEMM

Le differenze sono su diversi livelli:

- **Tipologia di modello**: HMM e' un modello *generativo* mentre MEMM e' *discriminativo*.
- **Fase di learning**: HMM necessita solo dei counts delle occorrenze, MEMM necessita di algoritmi di ottimizzazione che potrebbero anche non convergere mai ad un ottimo locale
- **Informazione utilizzata**: HMM e' in grado di utilizzare come features solamente la parola corrente, MEMM puo' utilizzare qualsiasi feature linguistica booleana

A livello tecnico, le differenze sono anche a livello di probabilita'. Come detto, in HMM si ha un modello generativo, per cui, data una sequenza di tag vogliamo generare la sequenza di parole piu' verosimile. Per far cio' applichiamo Bayes alla formulazione del problema e otteniamo il modello

$$\hat{t}_1^n = \arg\max_{t_1^n} \prod_{i=1}^{n} P(w_i \mid t_i) P(t_i \mid t_{i-1})$$

In MEMM, invece, abbiamo un modello discriminativo per cui non si applica l'ipotesi di Bayes ma semplicemente si applica un'ipotesi di indipendenza e di Markov, per cui il modello risultante e' il seguente

$$\hat{t}_1^n = \arg\max_{t_1^n} \prod_{i=1}^{n} P(t_i \mid t_{i-1}, w_i)$$

Questa probabilita' e' modellata utilizzando un modello di regressione lineare su un template di features linguistiche scelte $\vec{w} \cdot \vec{f}$. L'apprendimento consiste nello scegliere il vettore $\vec{w}$ tale per cui la probabilita' del tag giusto sia massima. Nonostante sia evidente che MEMM sia piu' flessibile poiche' permette di utilizzare features linguistiche arbitrarie, questa flessibilita' viene pagata in termini di complessita' nella fase di learning. Inoltre, alcuni ottimizzatori potrebbero anche non riuscire ad ottenere una configurazione (sub) ottimale per i pesi.

D'atra parte, l'apprendimento di un HMM consiste semplicemente nel fare un count delle occorrenze nel corpus, per cui e' molto meno oneroso e molto meno suscettiible alla stocasticita' degli ottimizzatori. D'altra parte, e' molto meno flessibile poiche' l'unica feature linguistica su cui si basa e' la parola stessa e il tag attuale.

Entrambi i modelli soffrono del problema della *sparseness*, cioe' quando si incontrano parole non conosciute. In questo caso si possono impiegar diverse tecniche tra le quali risaltano: supporre sia un nome oppure associare la probabilita' degli altri tags.

---

## 2 Sintassi

### 2.1 Chomsky e la sua gerarchia

Chomsky differenzia tra la **competence** (cioe' la competenza grammaticale) e la **performance** (come questa competenza viene utilizata nella comunicazione). Secondo lui il linguaggio naturale (la competence) puo' essere modellato per mezzo delle Grammatiche Generative, cioe' dei sistemi formali di riscrittura ispirati a Turing e Post. Formalmente una grammatica generativa e' una quadrupla $\langle \Sigma, V, S, P \rangle$ dove:

- $\Sigma$ e' l'alfabeto
- $V$ e' l'insieme dei simboli non-terminali
- $S$ e' lo *starting symbol*
- $P$ e' un insieme di regole di riscrittura $\theta \to \gamma$

Queste grammatiche sono in grado sia di modellare la struttura sintattica di una frase che di generarla. L'ipotesi e' che i vari simboli non terminali modellino i costituenti della frase. Chomsky individua anche una gerarchia di diverse grammatiche, ognuna con capacita' espressiva differente (espressivita' crescente):

- Linear
- Context-Free
- Context-Sensitive
- Type 0

Una domanda che imperverso' nella ricerca per molto tempo fu quindi quella di stabilire a quale categoria appartenga il linguaggio umano. Inizialmente si ipotizzo' fosse CF, ma il Tedesco Svizzero non lo e', per cui invalidava l'ipotesi. Ci furono quindi diverse proposte a riguardo, che culminarono con l'invenzione delle grammatiche Mildly Context Sensitive. Alcune di queste grammatiche appartenenti a questa categoria degne di nota sono:

- Tree Adjoining Grammars (strutture ad albero e operazioni di adjoining e substitution)
- Head Grammars
- Linear Indexed Grammars
- Combinatory Categorial Grammars (bottom up, categorie di elementi atomici che si combinano attraverso regole di combinazione)

## 2.2 Parser Top Down/Bottom Up

Il parser top-down parte dalla radice $S$ e facendosi guidare dalla grammatica, genera i vari alberi che possono anche non essere compatibli con la frase. Il parser bottom-up, parte dalle foglie (le parole) e regredisce eventualmente le varie regole della grammatica, per cui si fa guidare dalle parole. Ne segue che tutti gli alberi sono compatibili con le parole ma non tutti sono ben formati (hanno $S$ come radice).

## 2.3 Grammatica CCG

Una Combinatory Categorial Grammar e' una grammatica bottom up (si parte dalle parole e si costruisce mano a mano l'albero) dove gli elementi principali sono delle categorie di parole. Queste categorie poi vengono combinate con altre categorie per mezzo di opportune regole di combinazione. Ad esempio, il verbo *amare* e' una categoria che cerca un qualche altro elemento alla sua sinistra (soggetto) e un'altro elemento alla sua destra (complemento oggetto) con cui combinarsi.

## 2.4 Sintassi a Dipendenze

La sintassi a dipendenze postula che la struttura sintattica di una frase consiste di elementi lessicali legati tra loro da relazioni binarie asimmetriche chiamate **dipendenze**. Queste relazioni presuppongono una **head** e un **dipendente** (modifier, inferior, subordinate).

## 2.5 Grammatiche a dipendenze: Algoritmi di parsing

Ci sono diversi algoritmi di parsing per le grammatiche a dipendenze:

- **Programmazione Dinamica**: e' possibile utilizzare una versione modificata di CKY, ma ha la complessita' computazionale pari a $O(n^5)$.
- **Graph Algorithms**: si crea un MST per la frase in un grafo in cui le parole sono i nodi e gli archi sono le dipendenze. I vari archi sono pesati e questi pesi vengono appresi da un ML classifier.
- **Parsing a costituenti + Conversione**: parsing di una grammatica con un algoritmo di parsing a costituenti noto e converto successivamente il risultato in formato a dipendenze secondo delle tabelle di percolazione derivate dalla teoria X-Bar.
- **Parsing Deterministico**: parsing che ad ogni run ritorna un singolo albero di parsing, guidato da Machine Learning Classifiers che prendono scelte greedy.
- **Constraint Satisfaction**: vengono eliminate tutte le possibili dipendenze che non soddisfano determinati vincoli.

## 2.6 Ambiguita' sintattica: PP Attachment & Coordination Ambiguity

L'ambiguita' sintattica (structural ambiguity) nasce dal momento che ci possono essere piu' alberi di derivazione validi. Due casi degni di nota sono:

- **Attachment Ambiguity**: si verifica quando c'e' un'ambiguita' nella separazione tra una preposizione e la sua clausola. Questo puo' verificarsi quando una proposizione puo' essere legata a due o piu' frasi. Ad esempio: *"Ho parlato col il Professore di matematica nel suo ufficio"* puo' voler dire *"Ho parlato di matematica con il Professore nel suo ufficio"* oppure *"Ho parlato con il professore di matematica nel suo ufficio"*
- **Coordination ambiguity**: si verifica quando una frase contiene una serie di elementi o frasi coordinate che possono essere interpretati in modi diversi a causa dell'ambiguita' nella loro struttura o posizione. Ad esempio, in *"Ho visto Maria e Paolo baciarsi"* la coordinazione *"e"* puo' essere interpretata come *"Ho visto Maria e ho visto Paolo baciarsi"* (anche in momenti diversi) oppure *"Ho visto Maria e Paolo baciarsi"* (nello stesso momento)

## 2.7 Algoritmo di parsing TUP

Caratteristiche:

- **Grammatica**: Dipendenze.
- **Algoritmo**: Bottom-Up, Depth-First.
- **Oracolo**: Rule-Based.

Algoritmo per la parsificazione di grammatiche a dipendenze basato su 3 passaggi: Chunking, Coordination e VerbSub-Cat.

## 2.8 Algoritmo di parsing MALT

Caratteristiche:

- **Grammatica**: Dipendenze.
- **Algoritmo**: Bottom-Up, Depth-First.
- **Oracolo**: Probabilistico.

E' un algoritmo per la parsificazione di grammatiche a dipendenze. E' detto "deterministico" perche' ad ogni run, si ottiene un singolo albero di dipendenze, contrariamente a come succede con CKY. L'algoritmo si basa su il concetto di automa. Lo stato di questo automa e' composto da:

- Input buffer: contiene le parole mancanti da analizzare.
- Stack: contiene le parole attualmente analizzate.
- Dependency relations: contiene le dipendenze tra le parole create fino al momento attuale.

Su questo stato si possono applicare 3 operazioni principali:

- `SHIFT`: prende la prossima parola dalla lista (rimuovendola) e la inserisce in cima allo stack.
- `LEFT`: fa pop della parola ($b$) sullo stack ($b = pop(stack)$) e crea una dipendenza $(a, b)$ con la prossima parola della lista ($a$).
- `RIGHT`: fa pop della parola ($b$) sullo stack ($b = pop(stack)$), crea una dipendenza $(b, a)$ con la prossima parola della lista ($a$), rimuove $a$ dalla lista e inserisce al suo posto $b$.

L'algoritmo pare da uno stato iniziale in cui l'input buffer e' pieno (contiene tutta la frase) e lo stack vuoto, e applica ad ogni passo l'operazione che viene suggerita dall'oracolo fino a quando non si raggiunge uno stato finale in cui l'input buffer e lo stack sono vuoti e la dependency relations non e' vuota.

### 2.8.1 Problematiche

Il primo problema e' che le dipendenze che vengono create non sono tipate. In questo caso, si potrebbe risolvere creando appositamente delle operazioni del tipo `LEFT_subj`, `RIGHT_subj`, per cui quando vengono eseguite creano la relazione tipata. In questo caso pero' il numero di operazione crescerebbe considerevolmente in relazione al numero di PoS tag diversi ($2n + 1$ operazioni per $n$ PoS tags).

Il secondo problema e' dovuto alla scelta dell'oracolo, cioe' la scelta dell'algoritmo che sceglie effettivamente l'azione da eseguire.

### 2.8.2 Costruzione dell'oracolo

L'oracolo e' un Machine Learning Classifier che viene addestrato sui vari stati del programma. Bisogna inoltre stabilire:

- **Features linguistiche significative**: bisogna selezionare le features dello stato che sono piu' significative ai fini della classificazione. Tipiche features possono riguardare le *posizioni* nello stato e gli *attributi* di alcune parole.
- **Dataset**: costruito per mezzo del Dependency Tree Bank, facendo reverse engineering degli alberi e ottenendo dei passi dell'algoritmo (si apprende su "storie" di esecuzione di azioni)
- **Algoritmo di training**: che deve far apprendere i pesi che vanno a massimizzare lo score della transizione corretta per tutte le configurazioni nel training set

## 2.9 Algoritmo CKY

Caratteristiche:

- **Grammatica**: Chomsky Normal Form (CF).
- **Algoritmo**: Bottom-Up, Dynamic Programming.
- **Oracolo**: Rule-Based.

L'algoritmo CKY e' un algoritmo di parsing dinamico e bottom up che calcola tutti i possibili alberi di parsing di una frase in tempo $O(n^3)$, controllando di fatto l'esplosione combinatoria dovuta all'ambiguita' strutturale sintattica. L'algoritmo funziona solo su grammatiche in Chomsky Normal Form (cioe' grammatiche che possono produrre solo 2 simboli non-terminali o solo un simbolo terminale). L'idea dell'algoritmo si basa sull'intuizione che se esiste una regola $A \to BC$, e $A$ copre dalla posizione $i$ alla posizione $j$, allora $\exists k \ : \ i < k < j$ in cui $B$ copre $i \dots k$ e $C$ copre $k \dots j$. Se noi memorizziamo per ogni *span* le regole che coprono tale span, possiamo riutilizzarle senza doverle ricalcolare.

CKY utilizza quindi una matrice $N \times N$ (con $N$ numero di parole della frase) in cui, nella posizione $(i, j)$ vengono memorizzate le regole che coprono lo span $i \dots j$. L'algoritmo, considerando la parola $j$-esima, inserisce inizialmente nella matrice alla posizione $(j, j)$ tutte le regole che coprono tale parola. Questo e' appunto ragionevole siccome lo span $j \dots j$ coincide con la singola parola che si sta considerando.

Successivamente, l'algoritmo considera tutti i possibili span dall'inizio della parola (0) fino a $j$ e per ogni span tutti i possibili split $k$. Per ognuno di essi, ci si chiede se esiste una regola $A \to BC$ tale per cui $B$ copre lo span $i \dots k$ e $C$ copre lo span $k \dots j$ (lo sappiamo perche' sono state salvate precedentemente nella tabella). Se cio' avviene, allora $A$ copre $i \dots j$, per cui viene inserita nella tablla alla posizione $(i, j)$.

Si noti che la complessita' e' $O(n^3)$ poiche' ci sono 3 cicli `for` innestati:

1. Scorre tutte le parole
2. Scorre tutti i possibili *span* fino alla parola attuale
3. Scorre tutti i possibili *split* dello span attuale

## 2.10 Difetti

- Il caso peggiore e il caso medio coincidono
- Esige una grammatica in CNF

## 2.11 Algoritmo CKY (Probabilistico)

Caratteristiche:

- **Grammatica**: Probabilistic CF.
- **Algoritmo**: Bottom-Up, Dynamic Programming.
- **Oracolo**: Probabilistic.

CKY genera tutti gli alberi possibili, pero' non si ha modo di decidere quale fra questi sia il piu' adatto. Secondo questa variante, un albero di derivazione ha associata una probabilita' che e' il prodotto di tutte le regole che sono state utilizzate nella derivazione. Essenzialmente si va a costruire l'albero piu' probabile andando a selezionare la regola di derivazione piu' probabile. Per fare cio' viene definita una distribuzione di probabilita' sulle regole della grammatica. Ogni non terminale e' associato ad una probabilita' tale per cui:

- $A \rightarrow \beta[p]$ con $p \in [0,1]$
- $\sum_{\beta} P(A \rightarrow \beta) = 1$

Le probabilita' vengono calcolate a partire da un tree bank nel modo seguente

$$P(\alpha \rightarrow \beta \mid \alpha) = \frac{Count(\alpha \rightarrow \beta)}{Count(\alpha)}$$

---

# 3 Semantica

## 3.1 A cosa serve il Lambda Calcolo?

## 3.2 Come rappresentare articoli e sostantivi nel lambda calcolo?

## 3.3 Semantica formale e composizionale con esercizio (Montague)

## 3.4 Mi parli della Semantica di Montague

---

# 4 Natural Language Generation

## 4.1 Quali sono le differenze tra Referencing Expression e Lessicalizzazione?

## 4.2 Perche' l'architettura dell'NLG e' divisa in 3 fasi?

---

# 5 Dialogue Systems

---

# 6 Miscellanee

## 6.1 Qual'e' la differenza tra ambiguita' sintattica e ambiguita' semantica?