Riassunto Mazzei

Matteo Brunello

1 Morfologia

1.1 Algoritmo di Viterbi

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

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

Il prodotto di queste due parti, ci da la probabilità della sequenza di tag con probabilità massima, per cui possiamo essenzialmente sfruttare questo fatto per costruire passo passo la sequenza, semplicemente selezionando ad ogni passo il tag che massimizza la probabilità (parte 2), e salvando la probabilità 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 è una matrice $T \times N$, dove T è il numero di PoS tags e N è la lunghezza della frase. Il numero di PoS tags T è in realtà $+2$ poichè bisogna aggiungere 2 tags speciali di `START` e `EOS`.

Inizialmente, l'algoritmo inizializza tutte le probabilità della prima colonna e poi, scorrendo per ogni colonna (parola) e per ogni riga (tag), calcola il valore corrispondente della cella (cioè salvandone il massimo calcolato al passo precedente * probabilità di transizione * probabilità di emissione).

Inoltre, in una matrice di `backpointers` memorizza l'indice dell'elemento massimo al passo precedente, in modo da fare backtracking una volta terminato.

1.2 NER Tagging

Il Named Entity Recognition è il task di trovare le Named Entities in un testo, cioè qualsiasi elemento che può 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

Il NER tagging è a sua volta costituito da 2 sottotask:

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

1.2.1 Difficoltà

Le difficoltà del NER tagging sono da ricercarsi in due motivazioni principali:

1. **Segmentazione**: non viene assegnato un tag ad ogni parola, ma è necessario trovare un segmento di più parole da taggare.

2. **Ambiguità**: i nomi propri sono inerentemente 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

È possibile risolvere il problema della segmentazione del NER tagging con il BIO tagging. In questo caso si trasforma il problema di segmentazione in un problema di tagging (parola per parola). L'idea di base è quella di 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`). A questo punto il problema diventa quello di taggare parola per parola, indicando eventualmente se la singola parola rappresenta l'inizio (`I`) o un elemento costituente (`I`) di una Named Entity oppure se non è nessuna delle due (`O`).

Ad esempio, *"The president George Washington"* sarà associato alla seguente sequenza di tag: `O, O, B-Person, I-Person`.

1.3 Differenze tra HMM e MEMM

Le differenze sono su diversi livelli:

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

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

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

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 è il seguente

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

Questa probabilità è 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 probabilità del tag giusto sia massima. Nonostante sia evidente che MEMM sia più flessibile poichè permette di utilizzare features linguistiche arbitrarie, questa flessibilità viene pagata in termini di complessità 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 conteggio delle occorrenze nel corpus, per cui è molto meno oneroso dal punto di vista computazionale e molto meno suscettibile alla stocasticità degli ottimizzatori. Tuttavia, è molto meno flessibile poichè le uniche features linguistiche su cui si basa sono la parola stessa e il tag attuale.

Entrambi i modelli soffrono del problema della *sparseness*, cioè quando si incontrano parole non conosciute. In questo caso si possono impiegare diverse tecniche tra le quali: supporre a priori che la parola sia un nome oppure associare la probabilità degli altri tags. Nel caso dell'HMM è possibile mitigare il problema andando a creare più modelli (unigrammi, bigrammi, trigrammi) e a interpolarli tra loro utilizzando ad esempio i *moltipicatori di Lagrange*.

---

2 Sintassi

2.1 Chomsky e la sua gerarchia

Secondo Chomsky la linguistica può essere divisa in **competence** (cioè la competenza grammaticale) e **performance** (come questa competenza viene utilizzata). Secondo lui il linguaggio naturale (la competence) può essere modellato per mezzo delle *Grammatiche Generative*, cioè sistemi formali di riscrittura ispirati ai lavori di Turing e Post. Formalmente, una grammatica generativa è una quadrupla $\langle \Sigma, V, S, P \rangle$ dove:

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

Queste grammatiche hanno la particolare caratteristica di poter essere interpretate in due modi differenti: sia come modello per rappresentare la *struttura sintattica* di una frase, che come *modello di generazione* di frasi sintatticamente valide. L'ipotesi alla base di questa idea rivoluzionaria è che i vari simboli non terminali modellino i costituenti della frase. Questa formalizzazione del linguaggio naturale, permise ai linguisti di studiare con rigore matematico l'espressività sintattica dei linguaggi. Lo studio di queste grammatiche portò, tra le altre cose, alla teorizzazione di una gerarchia di grammatiche (che successivamente prese il nome omonimo dal suo autore), ognuna con capacità espressive differenti (di seguito riportate con espressività crescente):

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

Dalla scoperta di questa categorizzazione linguistica nacque anche la necessità di stabilire a quale categoria appartenesse il linguaggio umano. Uno dei primi tentativi fu quello di ipotizzare che fosse CF, ma l'ipotesi fu smentita da un controesempio: il Tedesco Svizzero. Tale lingua, infatti, non è CF per via di alcune caratteristiche come la presenza di dipendenze incrociate. 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

Un 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.

Un parser bottom-up, parte dalle foglie (le parole) e regredisce eventualmente le varie regole della grammatica (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 è 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 (combinazione destra e sinistra). Quindi una grammatica di questo tipo consiste in un lessico (parole) associate a delle categorie più eventualmente delle regole di combinazione.

Ad esempio, `ama: (S\NP)/NP` è 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 testa (**head**) e un **dipendente** (modifier, inferior, subordinate).

Differentemente dalle grammatiche generative, questo tipo di grammatiche non hanno valenza generativa, bensì hanno solo l'obiettivo di fungere da **schema di annotazione**. Per stabilire se un sintagma è un head o un dipendente, si utilizzano criteri morfologici, sintattici e semantici guidati quindi dalla linguistica.

2.4.1 Vantaggi

- Generalizzazione tra tutti i linguaggi.
- Trasparenza e semplicità di rappresentazione.
- Identifica le relazioni sintattiche immediatamente.
- Le coppie di dipendenze possono essere buone features per classificatori e fare information extraction.

2.5 Grammatiche a dipendenze: Algoritmi di parsing

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

- **Programmazione Dinamica**: è possibile utilizzare una versione modificata di CKY, ma ha la complessità 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 successiva conversione 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 Ambiguità sintattica

L'ambiguità sintattica (structural ambiguity) nasce dal momento che ci possono essere più alberi di derivazione validi. Due casi noti in cui questo può verificarsi sono:

- **Attachment Ambiguity**: si verifica quando c'è un'ambiguità nella separazione tra una preposizione e la sua clausola. Questo può verificarsi quando una proposizione può essere legata a due o più frasi. Ad esempio: *"Ho parlato col il Professore di matematica nel suo ufficio"* può 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'ambiguità nella loro struttura o posizione. Ad esempio, in *"Ho visto Maria e Paolo baciarsi"* la coordinazione *"e"* può 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 Ambiguità sintattica vs ambiguità semantica

L'ambiguità sintattica è causata da ambiguità inerenti all'interpretazione del posizionamento e delle relazioni che intercorrono tra elementi sintattici. Dal punto di vista formale, si presenta come più alberi sintattici validi per una singola frase.

L'ambiguità semantica è causata dalla molteplicità di significati che possono essere associati alla stessa struttura morfologica di una parola. Ad esempio, nell'Inglese *"Bank"* è un termine *polisemico*, cioé significa che può assumere significati differenti in base al contesto. In questo caso può indicare sia una *banca* di deposito (*"Bank account"*), che il *letto* di un fiume ("*River bank"*).

Nonostante differiscano, entrambe le ambiguità causano lo stesso problema, cioè che può essere associato più di un significato diverso per la stessa frase.

2.8 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.

2.9 Algoritmo di parsing MALT

Caratteristiche:

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

È un algoritmo per la parsificazione di grammatiche a dipendenze. È detto "deterministico" perché ad ogni run, si ottiene un singolo albero di dipendenze, contrariamente a come succede con CKY. L'algoritmo si basa sul concetto di *automa a stati finiti*. Lo stato di questo automa è 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 parte da uno stato iniziale in cui l'input buffer è pieno (contiene tutta la frase) e lo stack è vuoto, per poi apllicare 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 è vuota (sicché conterrà il risultato).

2.9.1 Problematiche

Il primo problema è che le dipendenze che vengono create non sono tipate, ciò significa che l'algoritmo indica semplicemente che esiste una relazione ma senza indicarne il tipo. In questo caso, si potrebbe risolvere creando appositamente delle operazioni del tipo `LEFT_subj`, `RIGHT_subj`, che se eseguite generano la relazione tipata. In questo caso, però, il numero di operazione crescerebbe considerevolmente in relazione al numero di PoS tag diversi ($2n + 1$ operazioni per n PoS tags).

Il secondo problema è dovuto alla scelta dell'oracolo, cioè la scelta dell'algoritmo che ha il compito di decidere l'azione effettiva da eseguire ad ogni iterazione.

2.9.2 Costruzione dell'oracolo

L'oracolo è un Machine Learning Classifier che viene addestrato sui vari stati del programma. In altri termini, l'idea è quella di apprendere un modello di ML che funga da automa a stati finiti, cioè che mappa *Stati* del programma in *Azioni* da eseguire. Per ottenere un modello di questo tipo è necessario stabilire:

- **Features linguistiche significative**: bisogna selezionare le features dello stato che sono più 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 invece delle sequenze di passi dell'algoritmo che hanno generato l'albero (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.

Il classificatore non è nient'altro che un classificatore lineare. Si definisce un vettore di feature $f(c,t)$ che dipende dallo stato c e dalla transizione t. Lo score viene definito come $S(c,t) = w \cdot f(c,t)$, per cui l'algoritmo di apprendimento trova un assegnamento di pesi $\hat{w}$ tale che massimizzi lo score della transizione corretta per tutte le configurazioni c nel training set.

2.10 Algoritmo CKY

Caratteristiche:

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

L'algoritmo CKY è 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'ambiguità strutturale sintattica. L'algoritmo funziona solo su grammatiche in Chomsky Normal Form (cioè grammatiche che possono produrre solo 2 simboli non-terminali o un solo simbolo terminale). L'idea dell'algoritmo si basa sull'intuizione che se esiste una regola $A \to BC$, e A copre la frase 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 si memorizzano per ogni *span* $(i \dots j)$ i non-terminali che coprono tale span, possiamo riutilizzarli senza dover ricalcolare se coprano o meno tale span. In questa maniera è possibile costruire alberi via via sempre più grandi riutilizzando i sottoalberi calcolati ai passi precedenti.

Per poter memorizzare i non-terminali, CKY utilizza una matrice $N \times N$ (con N numero di parole della frase) in cui in ogni posizione (i,j) sono memorizzati tutti i non-terminali delle 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 è ragionevole siccome lo span $j \dots j$ coincide con uno span di lunghezza 1 che è di fatto 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 perché sono state salvate precedentemente nella tabella). Se ciò avviene, allora A copre $i \dots j$, per cui viene inserita nella tablla alla posizione (i,j).

Si noti che la complessità è $O(n^3)$ poichè ci sono 3 cicli `for` innestati:

- Il primo scorre tutte le parole.
- Il secondo scorre tutti i possibili *span* fino alla parola attuale.
- Il terzo scorre tutti i possibili *split* dello span attuale.

2.11 Difetti

- Il caso peggiore e il caso medio coincidono.
- Necessita di una grammatica in CNF.

2.12 Algoritmo CKY (Probabilistico)

Caratteristiche:

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

CKY genera tutti gli alberi possibili, però non si ha modo di decidere quale fra questi sia il più adatto. Secondo questa variante, un albero di derivazione ha associata una probabilità che è il prodotto di tutte le regole che sono state utilizzate nella derivazione. L'algoritmo non differenzia particolarmente da CKY classico, l'unica differenza è che ci sono delle probabilità associate ad ogni regola. Mano a mano che l'algoritmo aggiunge le regole alla tabella, ne salva anche il valore di probabilità. In caso una regola derivi da due sottoalberi, si moltiplica la probabilità della regola per le probabilità associate ai due sottoalberi. Ad esempio, se avessimo la regola $P(VP \to V\ NP) = 0.2$ e i due sottoalberi (rappresentati da non-terminali nelle celle) V e VP, rispettivamente con $P(V) = 0.05$ e $P(VP) = 0.0024$, si inserirà nella casella corrente il corrispondente non-terminale VP con probabilità $P(VP) = 0.2 \cdot 0.05 \cdot 0.0024$.

In questo modo ogni albero finale (rappresentato dal simbolo S) avrà associata una probabilità, per cui per disambiguare sarà sufficiente scegliere quello con il valore massimo.

Le probabilità possono essere stimate a partire da un tree bank nel modo seguente:

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

---

3 Semantica

3.1 Semantica Argomentale

La semantica di cui abbiamo discusso a lezione è la semantica argomentale, cioè ha lo scopo di rappresentare chi ha fatto cosa (struttura predicato argomento), per cui si ignora totalmente la componente temporale. Per modellare questa semantica si utilizza il formalismo della logica del prim'ordine, principalmente perché rappresenta un buon compromesso tra complessità computazionale e semplicità di rappresentazione.

3.2 Principio di composizionalità di Frege

Il principio di composizionalità di Frege è un principio secondo il quale il significato di una frase è funzione del significato delle sue componenti e di come questi significati sono combinati tra loro. Ne deriva che secondo questo principio, una volta ottenuto il significato dei singoli elementi di una frase, è possibile comporli tra di loro in base alle regole di composizione semantica per ottenere il significato della frase intera.

È quindi possibile sviluppare sulla base di questa idea un algoritmo, che prende il nome di *Algoritmo fondamentale della linguistica computazionale*, il quale consiste nei seguenti passaggi:

1. Parsifica la frase ottenendo l'albero sintattico.
2. Determina la semantica per ogni sintagma (*foglie*).
3. Componi la semantica partendo dalle foglie e risalendo alla radice, ottenendo il significato dell'intera frase.

3.3 Semantica di Montague

La semantica di Montague è un formalismo che unifica la rappresentabilità della logica del prim'ordine con la calcolabilità del lambda calcolo. L'introduzione del lambda calcolo serve a superare delle limitazioni intrinseche alla logica del prim'ordine, poichè esso è solamente un formalismo di *rappresentazione* e non di calcolo (non sappresenta *computazioni*). Più precisamente, le due limitazioni principali sono:

1. Non è possibile rappresentare predicati con variabili libere.
2. Non è possibile imporre un ordine preciso di applicazione dei predicati.

Con l'introduzione del lambda calcolo si risolvono entrambi i problemi. A questo punto, le grammatiche vengono annotate con delle lambda astrazioni in cui vengono specificate anche le regole di composizione.

3.4 Rappresentazione di Articoli e Coordinazioni

Per rappresentare articoli all'interno di questo formalismo, è sufficiente permettere l'astrazione anche sui predicati, di seguito sono riportati diversi formalismi per rappresentare i vari articoli:

- *un*: $\lambda P \lambda Q.\exists z(P(z) \land Q(z))$
- *tutti*: $\lambda P \lambda Q.\exists z(P(z) \to Q(z))$
- *nessuno*: $\lambda P \lambda Q.\exists z(P(z) \to \neg Q(z))$
- *e*: $\lambda P \lambda Q \lambda R.(P(R) \land Q(R))$
- *ogni*: $\lambda P \lambda Q(\forall x(P(x) \to Q(x)))$

3.5 Rappresentazione di Avverbi

Per rappresentare avverbi, invece, non possiamo utilizzare l'avverbio come predicato e passargli come argomento l'altro predicato, questo perché si otterrebbe una logica del second'ordine. Ad esempio:

$$sweetly(love(Paolo, Francesca))$$

Si potrebbe invece pensare di aggiungere un argomento al predicato che funga da modificatore

$$love(Paolo, Francesca, sweetly)$$

Ma questa soluzione non è particolarmente elegante. Si decide quindi di *reificare* l'evento, cioè identificare con una variabile un **evento**

$$\exists e \ : \ love(e, Paolo, Francesca) \wedge sweetly(e)$$

Questo modo di rappresentare gli avverbi può anche essere scritto in un'altra maniera detta *Neo-Davidsoniana* in cui essenzialmente vengono generalizzati anche gli argomenti (si normalizzano tutti i predicati in modo che abbiano solo 2 argomenti)

$$\exists \ : \ love(e) \wedge agent(e, Paolo) \wedge patient(e, Francesca) \wedge sweetly(e)$$

Questa formulazione rispetta anche la transitività, e rappresenta inoltre una tipologia di modellazione più sistematica. Mentre in Montague si hanno predicati n-ari in base ai modificatori, nello stile Neo-Davidsoniano si hanno invece solo predicati unari e i modificatori sono esplicitati con i luoghi semantici.

---

4 Natural Language Generation

4.1 Architettura

L'architettura del NLG e' divisa in 3 fasi organizzate in una pipeline (l'input di una e' l'output della precedente):

- Text Planning (Output: Text Plan)
- Sentence Planning (Input: Text Plan, Output: Sentence Plan)
- Linguistic Realization (Input: Sentence Plan, Output: Testo generato)

Questa suddivisione viene fatta poichè ogni fase è composta da diversi task che necessitano sia di *conoscenza di dominio* che di *conoscenza linguistica.*

4.2 Task

I Task del Natural Language Generation sono 7:

- Text Planning
  1. **Content Determination**: viene determinato il contenuto dei messaggi, cioè cosa bisogna generare. Questo contenuto è rappresentato per mezzo di sintagmi oppure per mezzo di concetti e relazioni tra entità del dominio.
  2. **Discourse Planning**: si determina la struttura che relaziona le frasi tra di loro, cioè l'ordine in cui devono essere dette.
- Sentence Planning
  3. **Sentence Aggregation**: i messaggi vengono combinati eventualmente per produrre un flusso più naturale.
  4. **Lexicalization**: vengono determinate le parole da utilizzare per esprimere i concetti e le relazioni del dominio. Inoltre viene impiegata la sintassi in generazione.
  5. **Referring Expression Generation**: il goal di questo task è di bilanciare fluenza con ambiguità. Utilizzando la pragmatica è possibile ottenere economicità nella rappresentazione a patto di introdurre delle ambiguità.
- Linguistic Realization

6. **Morpho-Syntactical Realization**: si utilizzano le regole morfologiche per generare le parole (ad esempio *+ed* per formare il passato) e le regole sintattiche per formare le frasi (ad esempio Verbo prima del Soggetto se forma è interrogativa).
7. **Orthografical Realization**: concerne l'introduzione di lettere maiuscole a inizio frase, punteggiatura e lettere maiuscole per i nomi.

---

5 Dialogue Systems

5.1 Definizione di dialogo

Attività collaborativa **Attività**: motivata dal desiderio di raggiungere un obiettivo **Collaborativa**: ordinata per coordinare i partecipanti

5.2 Features significative del dialogo

Il dialogo ha 4 features significative:

1. **Turn-Based**: i partecipanti fanno a turni durante una conversazione. È essenzialmente un problema di allocazione di una risorsa (il canale comunicativo) a più utilizzatori. Si possono utilizzare alcuni spunti quali il silenzio, l'esitazione o l'intonazione.
2. **Speech-Acts**: i turni sono strutturati in delle azioni comunicative unitarie chiamate *speech acts*. Essi possono essere:
   - **Assertivi**: informa l'interlocutore di qualcosa.
   - **Direttivi**: richiedi all'interlocutore di fare qualcosa.
   - **Commissivi**: informa l'interlocutore che ti impegni a fare qualcosa in futuro.
   - **Espressivi**: informa l'interlocutore sullo stato d'animo di chi sta parlando.
   - **Dichiarativi**: espressioni che indicano un nuovo stato del mondo.
3. **Conversational-Context**: l'interpretazione delle azioni comunicative è subordinata dai contesto della conversazione. Ci sono 3 possibilità:
   - **Non-Sentential Utterances**: costruzioni linguistiche che rendono ambigua l'interpretazione dello speech act.
   - **Conversational Implicatures**: frasi in cui lo speech act può essere determinato solamente attraverso il contesto.
   - **Referring-Expressions**.
4. **Grounding-Signals**: i partecipanti danno dei segnali che hanno/non hanno ricevuto/capito ciò che l'altro interlocutore ha comunicato. Si basa su una collezione di conoscenza e assunzioni in comune che vengono stabilite durante un'interazione.

5.3 Chatbots

Le architetture per i ChatBots possono essere categorizzate in due approcci principali:

- Rule Based.
- Corpus Based.

5.3.1 ELIZA

ELIZA era un chatbot rule-based che simulava uno psicanalista di stampo Rogeriano, che consisteva nel riflettere indietro le dichiarazioni fatte dal paziente. ELIZA utilizzava delle regole che avevano il duplice compito di fare *pattern-matching* e *trasformazione*. Ad esempio la regola:

```
(0 YOU 0 ME) -> (WHAT MAKES YOU THINK I 3 YOU)
```

Faceva pattern matching sul pattern nell'antecedente della regola e produceva l'antecedente in cui 3 è il terzo costituente nel pattern (cioè il bind 0 in questo caso).

Un'altro aspetto importante di ELIZA è quello del ranking. Essenzialmente viene stabilito un rank per una serie di pattern che viene utilizzato poi dall'algoritmo per decidere quale regola attivare.

L'algoritmo sceglie la keyword con il rank maggiore tra tutte le keywords presenti nella frase in input e ne applica la trasformazione corrispondente. In caso non ci siano keywords, allora scatta il `NONE` pattern, cioè delle frasi di placeholder predefinite.

Un'altra caratteristica interessante è la presenza di memoria. Ogni qual volta che la keyword *"My"* è la highest-ranking, seleziona una trasfomazione inserita nella memoria, la applica alla frase e la inserisce in uno stack del discorso. Successivamente, se nessuna keyword fa match, prima di far scattare il pattern `NONE` ritorna il top dello stack del discorso. In questo modo è possibile attingere alla memoria quando non si hanno pattern attivi così da ottenere più informazioni e chiedere altro in relazione a quanto detto precedentemente nell'interazione.

Vengono poi impiegate altre accortezze per evitare di non riutilizzare sempre la stessa regola (utilizzando un semplice contatore).

5.3.2 Corpus Based

I Chatbot basati sui corpus possono essere sviluppati essenzialmente seguendo due approcci:

- Response by **retrieval**: in cui si apprende un modello da un corpus di interazioni. L'output è fisso, cioè non viene generato nuovo testo, ma viene semplicemente ritornata la risposta facendo retrieval dal corpus.
- Response by **generation**: si apprende un language model da un grande corpus di interazioni, per cui la risposta sarà generata.

Il corpus per apprendere questi modelli può provenire da diverse fonti di informazioni come:

- Trascrizioni telefoniche.
- Dialoghi nei film.
- Crowd-working con conversatori umani.

5.4 Architettura dei sistemi di dialogo

L'architettura dei sistemi di dialogo è (come molte delle altre architetture viste) a pipeline. I vari stadi corredati da input/output sono:

1. **Speech Recognition**
   - Input: Segnale acustico
   - Output: Parole
2. **Language Understanding**
   - Input: Parole
   - Output: Dialogue Act (User)
3. **Dialogue Manager**
   - Input: Dialogue Act (User)
   - Output: Dialogue Act (System)
4. **Response Generation**
   - Input: Dialogue Act (System)
   - Output: Parole
5. **Text to Speech Synthesis**
   - Input: Parole
   - Output: Segnale acustico

Di seguito sono discussi nel dettagli alcuni step più importanti

5.4.1 Natural Language Understanding

Si occupa di ottenere una vera e propria rappresentazione semantica di cosa è stato detto. È un task difficile perché bisogna fare i conti con l'ambiguità. Ci sono due approcci possibili:

- **Classico**: si determina il significato finale attraverso 3 step
  1. Analisi Sintattica: si ottiene l'albero di parsificazione della frase.
  2. Interpretazione Semantica: si ottiene una prima interpretazione semantica ad esempio per mezzo della semantica formale di Montague.

3. Analisi Pragmatica: prende in input il significato (FOL) dall'analisi precedente e la conoscenza del mondo e in output restituisce da un atto comunicativo.

- **Moderno**: approccio stile frame/slot-filling.

5.4.2 Frame-Based Dialogue Agents

Sono sistemi di dialogo basati sull'approccio moderno (slot-filling) che nascono con il compito di aiutare e supportare un utente nella soluzione di un determinato task (es. prenotare un biglietto)

La struttura principale di questi sistemi sono i frames. Un frame è un template che contiene slots che possono essere riempiti con delle informazioni. Inoltre, ogni slot è anche associato ad una domanda per avere quella informazione specifica. L'insieme di frames è chiamato anche **ontologia di dominio**.

Il sistema utilizza quindi le domande per poter riempire uno o più slots in base alle risposte dell'utente. Quindi, un frame rappresenta tutta l'informazione di un dominio particolare necessaria a risolvere quel task specifico, e il dialogue manager decide su quale frame bisogna focalizzare il discorso. Per poterlo decidere, il dialogue manager esegue 3 passi principali:

1. Domain Classification: si determina a quale dominio si sta facendo riferimento.
2. Intent Determination: si determina l'intento dell'utente.
3. Slot Filling: estrai i dati significativi dalla frase per riempire gli slots. Questa fase può essere fatta attraverso delle regole di pattern matching.

5.5 Dialogue Managers

In base al diverso utilizzo del dialogue system avremo diversi dialogue managers. Abbiamo quindi le seguenti possibilità:

- Sistemi di dialogo come **front-end** di un sistema backend: forniscono un interfaccia che funge da intermediario tra il backend del sistema.
- Sistemi di dialogo come **agenti**: hanno accesso a delle funzionalità specifiche al task che devono risolvere e supportano l'utente al raggiungimento del task.

Il dialogue manager è composto a sua volta di 2 componenti principali:

- **Dialogue Context Model**: è il modello del discorso, implementato tramite strutture dati apposite.
- **Dialogue Control**: dedide cosa fare dopo nel contesto comunicativo attuale. Impiega algoritmi basati su grafi per governare le dinamiche del dialogo.

5.5.1 Valutazione

I sistemi di dialogo possono essere valutati per mezzo di 2 possibili sistemi di valutazione:

1. Valutazione tramite test non oggettivi.
2. Trindi Tricklist: lista di sedici condizioni si/no per la valutazione oggettiva.

---

6 Esercizi

6.1 Esercizio 1 (Viterbi)

Fare il learning di un modello HMM sul corpus riportato di seguito. Utilizzare poi l'algoritmo di Viterbi per fare il decoding del modello ottenuto sulla frase *"Paolo ama Francesca"*.

- `Paolo/N pesca/V`
- `Giovanni/N ama/V i/D cani/N`
- `Francesca/N ama/N`
- `Una/D pesca/N Francesca/A`

Probabilità di transizione:

- $P(N \mid S_{ini}) = \frac{3}{4}$
- $P(D \mid S_{ini}) = \frac{1}{4}$

- $P(S_{end} \mid V) = \frac{1}{4}$
- $P(S_{end} \mid N) = \frac{1}{2}$
- $P(S_{end} \mid A) = \frac{1}{4}$
- $P(V \mid N) = \frac{2}{6}$
- $P(D \mid V) = \frac{1}{2}$
- $P(A \mid N) = \frac{1}{6}$
- $P(N \mid D) = 1$
- $P(N \mid N) = \frac{1}{6}$

Probabilità di emissione:

- $P(Paolo \mid N) = \frac{1}{6}$
- $P(pesca \mid V) = \frac{1}{2}$
- $P(pesca \mid N) = \frac{1}{6}$
- $P(Giovanni \mid N) = \frac{1}{6}$
- $P(ama \mid V) = \frac{1}{2}$
- $P(ama \mid N) = \frac{1}{6}$
- $P(i \mid D) = \frac{1}{2}$
- $P(cani \mid N) = \frac{1}{6}$
- $P(Francesca \mid N) = \frac{1}{6}$
- $P(Francesca \mid A) = 1$
- $P(Una \mid D) = \frac{1}{2}$

Risultato della codifica:

	S_{ini}	Paolo	ama	Francesca	S_{end}
S_{end}					1/6912
`N`		1/8	1/288	1/10368	
`V`		0	1/48	0	
`D`		0	0	0	
`A`		0	0	1/1728	
S_{ini}	1				

6.2 Esercizio 2 (CKY)

Data la Grammatica in CNF di seguito, eseguire CKY sulla frase *"Paolo ama Francesca dolcemente"*.

- $S \rightarrow NP\ VP$
- $VP \rightarrow VP\ ADV$
- $VP \rightarrow V\ NP$
- $NP \rightarrow Paolo$
- $NP \rightarrow Francesca$
- $V \rightarrow ama$
- $ADV \rightarrow dolcemente$

`NP`		`S`	`S`
	`V`	`VP`	`VP`
		`NP`	
			`Adv`

6.3 Esercizio 3 (MALT)

Si esegua l'algoritmo MALT sulla frase *"Paolo ama Francesca dolcemente"*, indicando per ogni passo l'azione eseguita e lo stato dell'algoritmo.

Azione	Stato
`NONE`	`[root], ["Paolo", "ama", "Francesca", "dolcemente"], []`
`SHIFT`	`[root, "Paolo"], ["ama", "Francesca", "dolcemente"], []`

Azione	Stato
LEFT	[root], ["ama", "Francesca", "dolcemente"], [("ama", "Paolo")]
SHIFT	[root, "ama"], ["Francesca", "dolcemente"], [("ama", "Paolo")]
RIGHT	[root], ["ama", "dolcemente"], [("ama", "Paolo"), ("ama", "Francesca")]
SHIFT	[root, "ama"], ["dolcemente"], [("ama", "Paolo"), ("ama", "Francesca")]
RIGHT	[root], ["ama"], [("ama", "Paolo"), ("ama", "Francesca"), ("ama", "dolcemente")]
RIGHT	[], [root], [("ama", "Paolo"), ("ama", "Francesca"), ("ama", "dolcemente"), (root, "ama")]
SHIFT	[root], [], [("ama", "Paolo"), ("ama", "Francesca"), ("ama", "dolcemente"), (root, "ama")]

6.4 Esercizio 4 (Montague)

6.5 Esercizio 5 (Neo-Davidsonian Style)