UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Un algoritmo per l'addestramento di un analizzatore sintattico proiettivo utilizzando dipendenze non proiettive

LAUREANDO

Luca Schiappadini

RELATORE

Giorgio Satta

ANNO ACCADEMICO 2015/2016

Ringraziamenti

Ringrazio innanzitutto il mio relatore prof. Giorgio Satta, che con la sua presenza costante mi ha permesso di superare tutte le difficoltà incontrate durante la scrittura di questa Tesi.

Un altro ringraziamento va al dott. Francesco Sartorio per avermi dato la possibilità di utilizzare il software da lui sviluppato per la sua tesi di dottorato, oltre che per i consigli fornitomi nello sviluppo e nell'implementazione degli algoritmi.

Voglio infine ringraziare tutte le persone che mi sono state vicino in questi mesi di lavoro e, più in generale, negli ultimi cinque anni di studi. Per primi i miei genitori, che mi hanno sempre supportato e che mi hanno permesso di frequentare questa Università. Poi tutti i miei amici, che, al contrario, mi hanno sempre sopportato ed aiutato a superare i momenti più bui.

Sommario

1. Introduzione

L'elaborazione del linguaggio naturale [1] costituisce uno dei campi dell'informatica in più rapida evoluzione e con le più grandi implicazioni pratiche. Ha come obiettivo, infatti, quello di dotare i computer di conoscenze linguistiche sufficienti ad interagire con gli utenti finali nel modo a loro più naturale, ossia tramite un discorso di senso compiuto.

Questa Tesi si concentra su un problema specifico dell'analisi sintattica tramite grammatiche alle dipendenze [4]. Per poter comprendere delle informazioni fornite in un linguaggio naturale, infatti, un elaboratore deve compiere una serie di analisi in successione: tra queste l'analisi sintattica ricava degli alberi che rappresentano come le parole di una frase si relazionano l'una con l'altra secondo le regole espresse in una grammatica. Nel caso delle grammatiche alle dipendenze, questi alberi si possono dividere, a seconda di una proprietà definita sui loro archi, in due grandi classi, gli alberi proiettivi e quelli non proiettivi.

Negli ultimi anni, i sistemi basati su transizioni [9] sono diventati i più diffusi per la realizzazione di questo tipo di analisi. Costruiscono l'albero in modo incrementale e basano il loro funzionamento sulle predizioni restituite da un modello che viene addestrato sulle frasi contenute in un corpus, ossia un insieme di frasi annotate con l'analisi sintattica corretta. Non tutti questi algoritmi possono costruire alberi non proiettivi: questa limitazione è pesante per tutte quelle lingue in cui le frasi non proiettive sono comuni, ad esempio il tedesco, ma allo stesso tempo porta alla realizzazione di algoritmi più robusti e veloci.

L'obiettivo di questa Tesi è duplice. Innanzitutto si vuole verificare se utilizzare frasi non proiettive nella fase di addestramento del modello per un analizzatore sintattico proiettivo permetta di realizzare un parser con una accuratezza maggiore. Successivamente viene analizzato un nuovo metodo per consentire l'utilizzo di frasi con dipendenze non proiettive all'interno di parser capaci di produrre solo dipendenze proiettive, confrontandolo con altre tecniche esistenti [17] [19].

La Tesi è organizzata, quindi, come segue:

- Capitolo 2, in cui viene brevemente introdotto il campo della Human Language Tecnology.
- Capitolo 3, in cui viene fornita una panoramica sull'analisi sintattica basata su una grammatica alle dipendenze, analizzando in particolare il parser basato su transizioni con oracolo dinamico sul quale è basato l'algoritmo proposto in questa Tesi.

- Capitolo 4, in cui sono esposti i diversi modi con cui è possibile utilizzare dipendenze non proiettive nell'addestramento di analizzatori sintattici proiettivi che saranno poi confrontati nei test.
- Capitolo 5, in cui sono riportati e commentati i risultati degli esperimenti svolti.
- Appendice A, in cui viene illustrato il setup degli esperimenti in modo più dettagliato rispetto al Capitolo 5.

2. Human Language Tecnology

La *Human Language Tecnology* [1], conosciuta anche come *Natural Language Processing* o *Computational Linguistics*, è una disciplina che racchiude una serie di campi comuni alla linguistica generativa, all'informatica, alla logica e alle scienze cognitive che si occupa dell'interazione tra i computer e le informazioni rappresentate in una linguaggio naturale, ad esempio l'italiano o l'inglese. Il suo scopo è, quindi, la creazione di sistemi informatici con conoscenze linguistiche tali da consentire loro, ad esempio, di interagire con una persona in modo naturale tramite un discorso, di aiutare un umano in compiti "linguistici" quali la traduzione da una lingua all'altra o di estrarre automaticamente informazioni da testi o altre forme multimediali. Tra le applicazioni all'avanguardia più conosciute al pubblico si possono citare:

- Google Translator, un programma che consente di effettuare traduzioni tra qualsiasi coppia di lingue supportate dal sistema.
- Siri, un software prodotto da Apple che svolge la funzione di assistente personale, usando queste tecniche per rispondere a domande dell'utente, fare raccomandazioni e compiere azioni, ad esempio effettuare una prenotazione, delegandole a dei *web services*.
- Wolfram|Alpha, un cosiddetto motore computazione di conoscenza, ossia un sistema che interpreta le parole chiave inserite dall'utente e propone direttamente la soluzione ricavata da un'elaborazione dei dati nella sua base di conoscenza, invece di offrire una serie di collegamenti ipertestuali come un motore di ricerca tradizionale.
- Knowledge Graph, una funzione aggiunta nel 2012 al motore di ricerca Google che fornisce informazioni dettagliate sull'oggetto della ricerca oltre alla lista di collegamenti ipertestuali. È concettualmente simile a Wolfram|Alpha, in quanto i suoi risultati sono ricavati da un'analisi semantica sui contenuti acquisiti da molte fonti diverse presenti in rete.
- Graph Search, un motore di ricerca semantico presente fino al 2014 nella versione inglese di Facebook. Restituiva i risultati delle *query* in linguaggio naturale dopo aver effettuato un'elaborazione su una banca dati costruita soprattutto a partire dalle informazioni ricavate dai profili degli utenti.
- Watson, un sistema di intelligenza artificiale sviluppato dalla IBM in grado di rispondere a domande espresse in lingua naturale. È diventato famoso per aver sconfitto i campioni del quiz televisivo statunitense *Jeopardy!*, mentre i piani futuri

dell'azienda prevedono una sua applicazione come sistema di supporto delle decisioni, ad esempio nel campo delle diagnosi mediche.

Il percorso che ha portato alla realizzazione di questi sistemi è stato molto lungo e coincide con lo sviluppo dell'informatica e si è innestato sugli studi linguistici iniziati nei secoli precedenti. Nel 1950 Alan Turing propose un celebre test per determinare l'intelligenza di un elaboratore che prevede l'interazione tra il computer e un umano tramite una conversazione in un linguaggio naturale: se la conversazione con la macchina risulta indistinguibile da quella con un individuo l'elaboratore può esser definito intelligente [2]. Negli stessi anni venne definita la teoria dei linguaggi formali, che costituì per diversi decenni l'unico approccio all'elaborazione del linguaggio naturale. L'approccio stocastico in questo periodo era infatti limitato solo al campo della digitalizzazione dell'informazione, ad esempio il riconoscimento ottico dei caratteri. L'esempio più rappresentativo dei sistemi di quest'era è ELIZA [3], un programma di interazione testuale la cui applicazione più famosa simulava uno psicoterapista rogersiano e che funzionava grazie a poche regole grammaticali e ad un sistema di *pattern matching*. Molte persone messe a contatto con esso di rifiutavano di credere che fosse un programma anche dopo che aver spiegato loro il suo funzionamento.

Verso la fine degli anni ottanta avvenne una rivoluzione nel settore: l'aumento della potenza computazionale portò all'introduzione di tecniche di *machine learning*, superando l'approccio precedente che si basava per lo più su regole scritte a mano. Questo fu possibile anche grazie alla creazione dei primi *corpus*, insieme di documenti annotati con i risultati delle varie analisi linguistiche su cui è possibile effettuare delle analisi statistiche e, quindi, un apprendimento. La maggiore accuratezza dei sistemi realizzati con queste tecniche portò ad un grande aumento dell'interesse nel campo e, quindi, alla realizzazione di sistemi sempre più complessi, come quelli che sono stati portati come esempio nella pagina precedente.

L'elaborazione del linguaggio naturale si suddivide in varie fasi, eseguite di norma in cascata, ognuna delle quali richiede uno specifico livello di conoscenza linguistica:

1. Riconoscimento dei simboli, in cui, nel caso la fonte non sia digitalizzata, si associano ad ogni grafema o suono la sua rappresentazione usata nell'elaborazione, di solito un carattere codificato secondo lo standard Unicode.
2. Analisi grammaticale, o *part-of-speech tagging*, ossia la scomposizione della frase in parole che sono ricondotte ad una categoria grammaticale (in italiano nome, verbo, aggettivo, articolo, pronome, preposizione, congiunzione, avverbio e interiezione) e analizzate secondo la loro specifica caratteristica morfologica (ad esempio per un nome sono indicati genere, numero, tipo etc.).

3. Analisi sintattica, o *parsing*, in cui dalle parole, insieme alle informazioni ottenute al punto precedente, sono ricavati degli alberi che indicano come sono relazionate l'una all'altra secondo le regole di una grammatica.
4. Analisi semantica, in cui si assegna un significato alla frase collegando gli elementi linguistici fin qui ottenuti a componenti cognitive che ne consentano la comprensione.
5. Analisi pragmatica, in cui si analizza il significato della frase in relazione al contesto in cui è stata espressa, con particolare attenzione a chi l'ha pronunciata e agli obiettivi che vuole ottenere.
6. Analisi del discorso, in cui si analizzano unità linguistiche più complesse di una singola frase, ad esempio una discussione tra due persone.

Questa suddivisione si comprende meglio tramite l'analisi di un esempio ormai classico, tratto dal film "2001: Odissea nello Spazio" di Stanley Kubrik. Nella pellicola HAL-9000 è un computer in grado di dialogare correttamente con gli esseri umani presenti sulla nave spaziale di cui costituisce il sistema principale, mentre David è un membro dell'equipaggio.

David: Apri la saracinesca esterna, HAL.
HAL-9000: Mi spiace David, purtroppo non posso farlo.

In questo caso il sistema di elaborazione del linguaggio naturale di HAL-9000 procede in un'analisi che comprende tutti i livelli elencati in precedenza:

1. Riconoscimento dei simboli: HAL analizza il segnale audio per ricavare le singole lettere e, quindi, la sequenza delle parole.
2. Analisi grammaticale: HAL deve essere in grado di rispondere con la coniugazione giusta, dicendo *"posso"* e non *"puoi"*, così come deve essere in grado di riconoscere che *"Apri"* è un verbo all'imperativo alla seconda persona singolare.
3. Analisi sintattica: HAL deve sapere che *"la saracinesca esterna"* è un sintagma nominale complemento oggetto di *"Apri"* e che la frase è corretta.
4. Analisi semantica: HAL deve sapere, ad esempio, cos'è una saracinesca e cosa vuol dire aprire qualcosa.
5. Analisi pragmatica: HAL deve saper rispondere cortesemente a David. Notare come nel film HAL non sia impossibilitato ad aprire la saracinesca esterna, ma abbia deciso autonomamente, in un impeto di paranoia, di uccidere tutto l'equipaggio: la cortesia non sarebbe dunque strettamente necessaria.
6. Analisi del discorso: HAL risponde *"farlo"* riferendosi ad una frase precedente nel discorso, andando quindi oltre la singola frase.

Come si nota da questo esempio, la più grande difficoltà nel Natural Language Processing risiede nell'ambiguità intrinseca ai linguaggi naturali, che è presente in tutti i livelli dell'analisi. Al contrario di un essere umano, un elaboratore può infatti riconoscere e generare direttamente solo linguaggi formali.

Questa Tesi tratta un problema specifico che si presenta all'interno della fase di parsing. Le altre fasi, quindi, non saranno trattate nei capitoli che seguono e, nel caso del riconoscimento dei simboli e dell'analisi grammaticale, si suppone che l'analisi relativa sia già stata svolta sull'input in modo accurato.

3. Dependency Parsing

L’analisi sintattica tramite metodi basati su una grammatica alle dipendenze [4], o *dependency grammar*, è diventata sempre più popolare negli ultimi anni rispetto ad altri approcci a causa della sua maggiore affidabilità con un grande numero di lingue, soprattutto quelle con un ordine libero delle parole, e per la sua codifica trasparente delle relazioni tra le parole, caratteristica che risulta utile in certe applicazioni come l'*Information Extraction*. La relazione predicato-argomento tra le varie parole, chiamata dipendenza, costituisce infatti l'idea centrale di questa tecnica e il loro insieme per una determinata frase crea un grafo che connette tutte le parole che la compongono.

Dal punto di vista storico bisogna notare come l'uso del concetto di dipendenza nell'analisi sintattica preceda di molto la sua applicazione all'interno del Natural Language Processing: venne infatti usato anche nel Medioevo da molti grammatici per l’analisi delle lingue classiche o di quelle slave. Il punto di partenza della teoria moderna della grammatica alle dipendenze viene però considerato il lavoro del linguista francese Lucien Tesnière [5].

Questo capitolo è un'introduzione all'analisi sintattica basata su una grammatica alle dipendenze, detta anche *dependency parsing*, ossia la tecnica che mappa una frase in un albero alle dipendenze. Si possono suddividere questi algoritmi in due gruppi, quelli *data-driven*, in cui si utilizzano tecniche di machine learning per fare l'analisi di nuove frasi, e quelli *grammar-based*, che si basano su una grammatica formale. Questi approcci sono comunque ortogonali e si possono avere soluzioni che ricadono in entrambe le classificazioni.

All'inizio del capitolo sono presentate, quindi, una serie di definizioni e proprietà riguardanti gli alberi alle dipendenze, per poi passare, dopo una panoramica sulle varie tecniche di analisi, all'esame degli algoritmi di parsing data-driven basati sulle transizioni, esaminando in sezioni dedicate sia la fase di apprendimento del modello sia quella di parsing vero e proprio.

3.1. Alberi alle Dipendenze

Nella Figura 3.1 si può vedere un esempio di albero alle dipendenze per una frase molto semplice. Si nota come le parole siano collegate da archi che rappresentano la relazione di dipendenza tramite una freccia, che parte dalla testa, in questo caso *"parlò"*, e termina sul dipendente, in questo caso *"Luca"*. Ad esso è associata anche un'etichetta, *"SBJ"*, che indica il tipo di dipendenza, in questo caso *"soggetto"*. L'insieme di questi archi forma un albero che descrive la struttura sintattica della frase. Si noti, inoltre, come all'inizio della frase sia stata inserita una parola speciale, *"-ROOT-"*, che viene usata per semplificare sia le definizioni che l'implementazione degli algoritmi.

Si procede ora con una presentazione formale degli alberi alle dipendenze.

Definizione 3.1. Si definisce una frase S come una sequenza di *token* $S = w_0 w_1 w_2 \dots w_n$, dove $w_0 = -ROOT-$.

Ogni token è una parola, una cifra o un simbolo diverso della frase che viene estratto in una fase precedente a quella di parsing. Questa operazione di norma è molto semplice, in quanto nella maggior parte dei casi i token corrispondono alla divisione in parole nella frase. Alcune lingue, però, hanno bisogno di un *pre-processing* più approfondito (ad esempio in inglese la parola *"don't"* produce due token, *"do"* e *"n't"*) e questo può causare l'introduzione di rumore nei dati. Ciò nonostante, nel seguito i termini *parola* e *token* saranno usati in modo intercambiabile. Si noti, infine, come w_0 corrisponda sempre al token artificiale -ROOT-, inserito per tre motivi: per semplificare alcune definizioni, perché è utile avere tutti gli alberi radicati nello stesso nodo e per ottenere un grafo connesso nel caso in cui le dipendenze di una frase siano partizionate in un insieme di alberi indipendenti.

Definizione 3.2. Sia $R = \{r_1, \dots, r_m\}$ un insieme finito di possibili tipi di relazioni di dipendenza sintattica tra due parole nella frase. Un tipo di relazione $r_i \in R$ viene detto anche etichetta d'arco o etichetta di dipendenza.

Definizione 3.3. Dato un insieme di relazioni R si definisce arco di dipendenza, o relazione di dipendenza, a un collegamento binario e asimmetrico tra un token sintatticamente subordinato, detto anche dipendente o figlio, e un altro token da cui questo dipende, detto anche testa o padre. Viene rappresentato con una tupla $a = (w_i, r_k, w_j)$, in

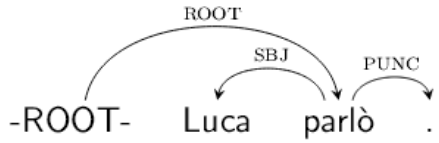


Figura 3.1. Semplice esempio di albero alle dipendenze.

cui w_i e w_j sono rispettivamente la testa e il dipendente e $r_k \in R$ è l'etichetta d'arco, oppure con la notazione analoga $w_i \underset{r_k}{\rightarrow} w_j$.

In alcune applicazioni e in alcune definizioni il tipo di dipendenza non è d'interesse, essendo sufficiente conoscere l'esistenza dell'arco tra due parole. In questi casi verrà usata la notazione semplificata $w_i \rightarrow w_j$, mentre la notazione $w_i \rightarrow^* w_k$ indica che w_k discende da w_i dopo un numero maggiore o uguale a zero di dipendenze.

Definizione 3.4. Dato un insieme di relazioni R e una frase $S = w_0 w_1 w_2 \dots w_n$ si definisce un albero alle dipendenze $T = (V,\ A)$ un albero diretto e ordinato per cui valgono le seguenti proprietà:

1. $V = \{w_0,\ w_1,\ w_2,\ \dots,\ w_n\}$ è l'insieme dei nodi.
2. $A \subset V \times R \times V$ è l'insieme degli archi.
3. $w_0 = -ROOT-$ è la radice dell'albero.

Si ricorda che un albero ordinato è un albero per il quale è specificato un ordinamento per i figli di ogni nodo: nel caso degli alberi alle dipendenze l'ordinamento corrisponde alla disposizione delle parole nella frase.

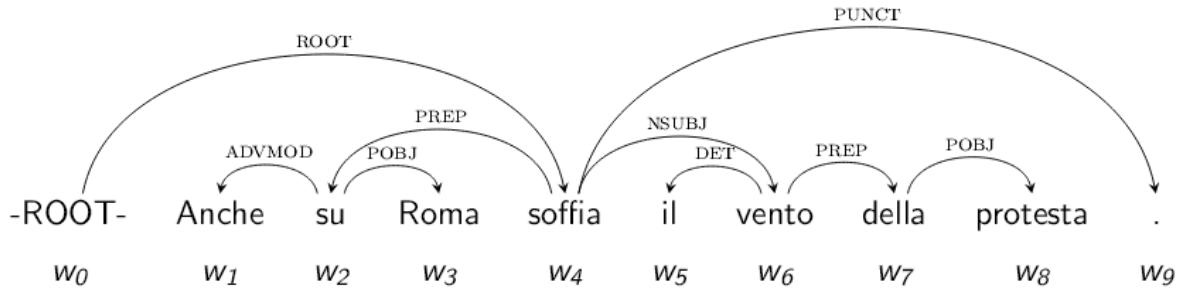


Figura 3.2. Esempio di albero alle dipendenze.

Esempio 3.1. Nella Figura 3.2 si può vedere un esempio di albero alle dipendenze più complesso rispetto a quello riportato nella Figura 3.1; tutti gli esempi riportati nel seguito sono estratti dall'*Italian Stanford Dependency Treebank* [6]. L'albero è nella forma indicata dalla Definizione 3.4; in particolare:

V = {-ROOT, Anche, su, Roma, soffia, il, vento, della, protesta, .}

A = {(-ROOT-, ROOT, soffia), (su, ADVMOD, Anche), (su, POBJ, Roma), (soffia, PREP, su), (soffia, NSUBJ, vento), (vento, DET, il), (vento, PREP, della), (della, POBJ, protesta), (soffia, PUNCT, .)}

Proprietà 3.5. Dato un albero alle dipendenze $T = (V,\ A)$ $\forall\, w_j \in V \setminus \{-ROOT-\}\ \exists!\ a \in A : w_i \to w_j,\ w_i \in V$, ossia ogni token nella frase ha una e una sola testa.

Proprietà 3.6. Dato un albero alle dipendenze $T = (V,\ A)$, se $(w_i,\ r_k,\ w_j) \in A$ allora $\nexists(w_i, r', w_j)$ con $r' \neq r_k$, ossia può esistere solo un arco con una specifica etichetta che collega due nodi.

Proprietà 3.7. Dato un albero alle dipendenze $T = (V,\ A)$, un nodo $w' \in V$ è la radice di un sotto albero $T' = (V',\ A')$. L'insieme dei nodi V' è una sottosequenza della frase *S* e viene chiamato *span* di w'.

Definizione 3.8. Dato un albero alle dipendenze $T = (V,\ A)$ e un nodo $w' \in V$, il *gap-degree* di w' è pari al numero di sottostringhe che costituiscono lo span di w' meno 1. Il gap-degree di un albero è il massimo gap-degree dei suoi nodi.

Si ricava dalla definizione, quindi, che se lo span è composto da una sola sottostringa di *S* il gap-degree è 0, se è composto da due sottostringhe il gap-degree è 1 e così via.

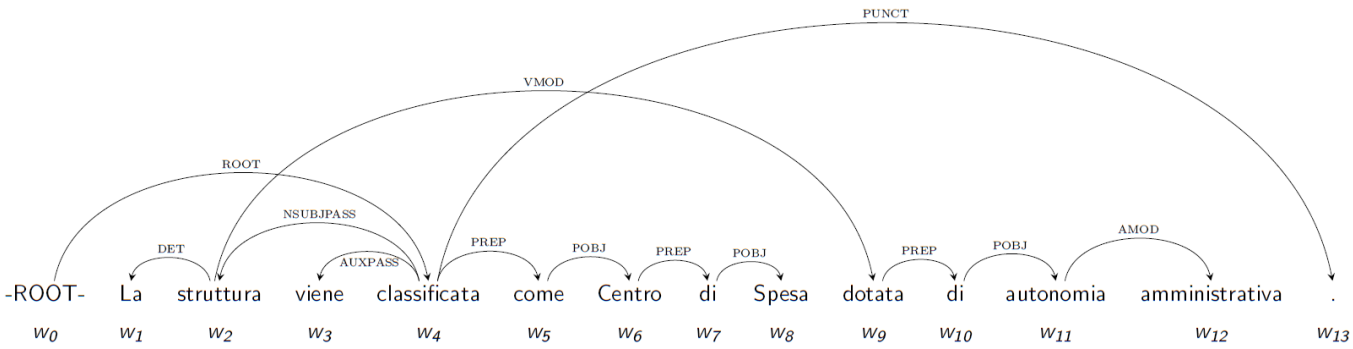
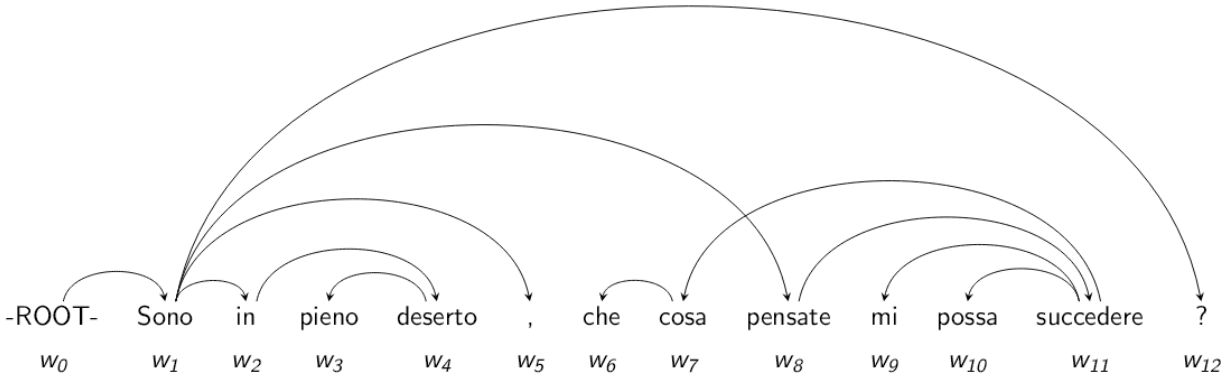


Figura 3.3. Esempio di albero alle dipendenze non proiettivo.

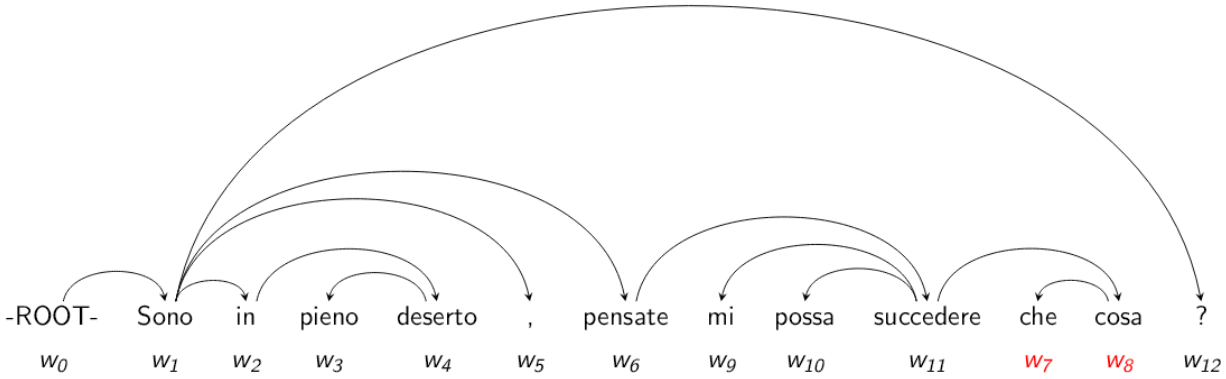
Esempio 3.2. Per comprendere meglio la Proprietà 3.7 e la Definizione 3.8 si consideri la frase contenuta nella Figura 3.3. Lo span di w_9 è costituito dalla sottosequenza $S_1 = w_9w_{10}w_{11}w_{12}$ e w_9 ha gap-degree pari a 0, lo span di w_2 è costituito dalla sottosequenza $S_2 = w_1w_2w_9w_{10}w_{11}w_{12}$ e w_2 ha gap-degree pari a 1 e infine lo span di w_4 è costituito dalla sottosequenza $S_3 = w_1w_2w_3w_4w_5w_6w_7w_8w_9w_{10}w_{11}w_{12}w_{13}$ e w_4 ha gap-degree 0 mentre l'albero con radice w_4 ha gap-degree 1.

3.1.1. Proiettività

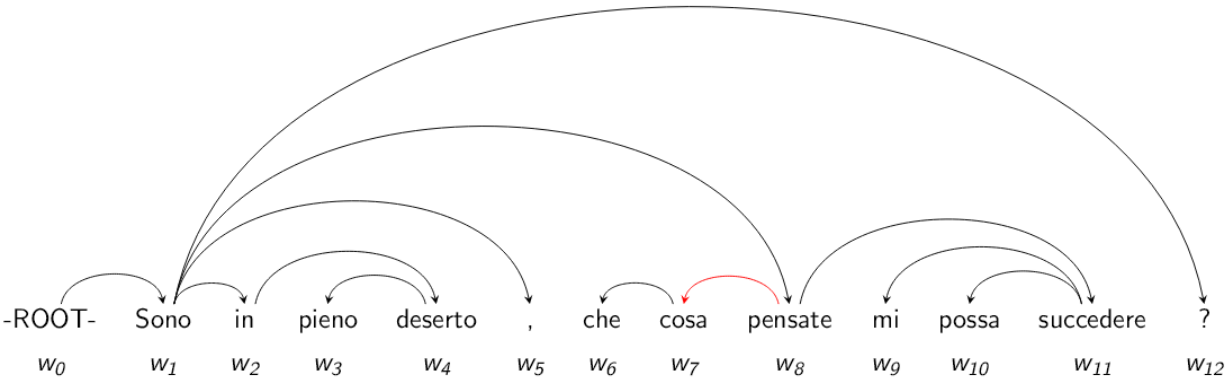
Definizione 3.9. Dato un albero alle dipendenze $T = (V,\ A)$, un arco $(w_i,\ r,\ w_j) \in A$ si dice proiettivo (*projective*) se $w_i \rightarrow^* w_k$ per ogni nodo di indice $i < k < j$ quando $i < j$ oppure $j < k < i$ quando $j < i$, ossia se esiste un cammino diretto tra la testa dell'arco e ogni nodo nell'intervallo tra testa e dipendente.



(a) Albero alle dipendenze non proiettivo.



(b) Albero alle dipendenze proiettivo ottenuto spostando le parole w_7 e w_8.



(c) Albero alle dipendenze proiettivo ottenuto sostituendo l'arco $w_{11} \rightarrow w_7$ con l'arco $w_8 \rightarrow w_7$.

Figura 3.4. Esempio di albero alle dipendenze non proiettivo (a) trasformato in albero alle dipendenze proiettivo tramite riordinamento (b) o tramite trasformazione degli archi (c).

Definizione 3.10. Un albero alle dipendenze $T = (V,\ A)$ è proiettivo se tutti i suoi archi $a \in A$ sono proiettivi o, equivalentemente, se tutti i suoi nodi hanno gap-degree 0. In caso contrario si dice che è non proiettivo (*non-projective*).

Nella Figura 3.4(a) si vede di un albero alle dipendenze con la rappresentazione usuale, in cui gli archi sono disegnati nel semipiano sovrastante le parole della frase. Nel caso sia stato inserito il nodo -ROOT- un albero è non-projective se viola la condizione di planarità, ossia nel caso in cui abbia almeno due archi che si incrociano: nella Figura 3.4(a) sono gli archi $w_1 \rightarrow w_8$ e $w_{11} \rightarrow w_7$.

Come si può vedere sempre nella Figura 3.4, esistono due modi diversi per trasformare un albero non proiettivo in proiettivo. Nel primo i token della frase vengono riordinati (Figura 3.4(b)) mentre nel secondo alcuni archi dell'albero vengono cambiati (Figura 3.4(c)). In entrambi i casi le modifiche sono riportate in rosso nel disegno. La seconda tecnica può essere utilizzata efficacemente per consentire l'elaborazione di frasi non proiettive da parte di algoritmi di parsing sviluppati per frasi proiettive, come si vedrà più in dettaglio nella Sezione 4.1.

La proiettività di un albero alle dipendenze costituisce, infatti, la sua caratteristica più importante, che suddivide gli algoritmi di parsing in due gruppi: quelli che sono in grado di elaborare frasi non-projective e quelli che, invece, non le considerano. Un algoritmo del secondo tipo ha dei vantaggi in termini di prestazioni, in quanto la sua maggiore semplicità consente di ottenere tempi di esecuzione inferiori, una maggiore robustezza e, a volte, anche una maggiore accuratezza. Questo può essere un buon compromesso per alcune lingue, come l'italiano o l'inglese, in cui le frasi non proiettive sono rare, mentre può creare qualche problema in quelle lingue come il ceco in cui queste frasi sono oltre il 20% o in quelle come il tedesco in cui non c'è un ordinamento fisso delle parole.

3.2. Dependency Parsing

Come accennato nell'introduzione a questo Capitolo, il dependency parsing consiste nel *task* di mappare una frase in un albero alle dipendenze che rappresenta in modo corretto le relazioni sintattiche tra le parole che la compongono. Esistono due approcci principali per la risoluzione di questo problema:

1. data-driven, che usa tecniche di machine learning su di un corpus, ossia un insieme di frasi annotate con il loro albero alle dipendenze sintatticamente corretto, per apprendere un modello tramite il quale effettuare il parsing. Gli algoritmi usati in questa Tesi sono di questo tipo.

2. grammar-based, in cui viene definita una grammatica formale per descrivere un linguaggio formale per riconoscere le frasi di ingresso.

L'approccio data-driven, così come è stato appena descritto, è un tipico esempio di apprendimento supervisionato e si compone, quindi, di due problemi distinti, ossia l'apprendimento di un modello di parsing da un insieme rappresentativo di frasi annotate e il parsing vero e proprio, che consiste nell'applicazione del modello appreso nella prima fase per svolgere l'analisi di una frase. Gli algoritmi di questo tipo differiscono, quindi, per il modello di parsing scelto, per l'algoritmo di apprendimento e per l'algoritmo di parsing. Formalmente:

Definizione 3.11. Un modello di dependency parsing è una tupla $M = (\Gamma,\ \lambda,\ \mathrm{h})$, dove Γ è un insieme di vincoli che definisce lo spazio degli alberi alle dipendenze ammessi per una singola frase, λ è un insieme di parametri (anche nullo) e h è un algoritmo di parsing.

L'insieme Γ è specifico al formalismo usato e può limitarsi a restringere lo spazio di ricerca ai soli alberi alle dipendenze oppure può imporre dei limiti più stringenti, ad esempio il rispetto di una grammatica formale.

La fase di apprendimento, specifica dei sistemi data-driven, punta a definire l'insieme dei parametri λ tramite l'addestramento su un *training set* $\mathcal{D}$ composto da coppie di frasi e i loro rispettivi alberi alle dipendenze, $\mathcal{D} = \{(S_d,\ T_d)\}_{d=1}^{|\mathcal{D}|}$.

Una volta ottenuti un insieme di vincoli e un insieme di parametri bisogna fissare un algoritmo di parsing h, ossia un algoritmo che, data una frase *S*, restituisca un albero alle dipendenze $T_S = h(\Gamma,\ \lambda,\ \mathrm{S})$, o un errore nel caso in cui la frase non rispetti i vincoli. L'applicazione di questo algoritmo costituisce la fase di parsing.

I metodi data-driven si possono suddividere a loro volta in due grandi classi: quelli basati sulle transizioni (*transition-based*) e quelli basati sui grafi (*graph-based*). Nei metodi basati sulle transizioni (spiegati più in dettaglio nella Sezione 3.3) si definisce un sistema di transizioni, detto anche modello di stato, per ricavare l'albero alle dipendenze. La fase di apprendimento consiste nel ricavare un modello per predire la transizione di stato successiva, data la storia dell'elaborazione fino quel momento, mentre quella di parsing consiste nel ricavare, dato un modello, la sequenza di transizioni ottima per la frase di ingresso. Gli algoritmi di questo tipo sono simili a quelli per il parsing *shift-reduce* per le grammatiche libere dal contesto.

I metodi basati su grafi, invece, definiscono per ogni frase uno spazio di ricerca costituito da tutti gli alberi di parsing possibili per quell'input. In questo caso il problema dell'apprendimento consiste nell'assegnare un punteggio ai possibili alberi alle dipendenze

associati alla frase, mentre il problema del parsing consiste nel trovare, dato un modello, l'albero alle dipendenze con punteggio più alto associato alla frase d'ingresso. I parser basati su questo metodo hanno un'accuratezza maggiore rispetto a quelli basati sulle transizioni in quanto esplorano completamente lo spazio di ricerca associato ad una frase, mentre i secondi procedono usualmente secondo una strategia *greedy*. Hanno però lo svantaggio di avere una complessità asintotica maggiore: data una frase di ingresso composta da *n* token, un parser transition-based ha complessità $O(n)$ (oppure quadratica in qualche caso particolare [7]) mentre un parser graph-based ha complessità $O(n^3)$. Questo li rende inadatti all'utilizzo in applicazioni che necessitano di una grande mole di dati, come quelle che analizzano milioni di pagine web per costruire basi di conoscenza (vedi Capitolo 2).

I sistemi data-driven assumono generalmente che tutte le frasi di ingresso siano valide, e cercano dunque di restituire sempre un albero alle dipendenze, per quanto improbabile possa essere dal punto di vista sintattico. I metodi grammar-based, al contrario, facendo uso di una grammatica formale accettano solo un sottoinsieme ristretto di tutte le possibili frasi. Nella maggior parte di questi algoritmi non è presente una fase di apprendimento e, per fare un'analogia, si può dire che la grammatica su cui si basano questi metodi corrisponde al modello ricavato nella fase di apprendimento degli algoritmi data-driven. In alcuni casi, però, la grammatica non è definita a mano ma ricavata a partire da dati linguistici, combinando quindi i due approcci.

Anche i metodi grammar-based si possono suddividere in due grandi classi, chiamate *context-free* e *constraint-based*, che, rispettivamente, riusano algoritmi sviluppati per le grammatiche libere dal contesto (risultando quindi simili agli algoritmi usati nei sistemi data-driven) e che vedono il problema di parsing come un problema di soddisfacimento di vincoli, in cui la grammatica è definita come una serie di vincoli su tutti gli alberi alle dipendenze possibili per una frase.

3.3. Transition-Based Parsing

I dependency parser basati sulle transizioni sono degli algoritmi che compiono l'analisi sintattica di una frase in ingresso processandola in modo incrementale: l'input viene letto da sinistra e destra e l'albero alle dipendenze associato viene dunque costruito passo dopo passo. Come si è visto nella Sezione 3.2, questi sistemi sono composti da due componenti principali, un modello addestrato su un *dataset* di alberi alle dipendenze sintatticamente corretti e un algoritmo di parsing non deterministico che crea l'albero prendendo, per ogni token ricavato dalla frase in ingresso, una decisione in base al modello statistico. Questo approccio è stato usato per la prima volta in [8], mentre in [9] si può trovare una panoramica generale sull'argomento.

Un sistema di transizioni è una macchina astratta composta da un insieme di configurazioni, o stati, e da transizioni tra le configurazioni. Un esempio di sistema di transizioni sono gli automi a stati finiti, che sono costituiti da un insieme di stati atomici e da transizioni definite sugli stati e i simboli di ingresso e che accettano una stringa solo se esiste una sequenza di transizioni dallo stato iniziale ad uno di quelli finali. Nel caso di dependency parser basati sulle transizioni le configurazioni sono strutture complesse e le transizioni corrispondono ai passi per derivare l'albero alle dipendenze: una sequenza di transizioni valide identifica, quindi, l'albero sintattico associato alla frase.

Definizione 3.12. Data una frase $S = w_0 w_1 w_2 \dots w_n$ con un insieme di nodi V_S, si definisce configurazione la tupla $c = (\sigma,\ \beta,\ A)$ dove:

1. σ viene chiamato stack ed è costituito da una sottosequenza anche non contigua di S.
2. β viene chiamato buffer ed è costituito da un suffisso di S.
3. A è un insieme di archi di dipendenza già costruiti, con $A \subset V_S \times R \times V_S$.

Una configurazione rappresenta, dunque, un'analisi parziale della frase d'ingresso, in cui le parole nello stack sono già state parzialmente processate, le parole nel buffer sono la porzione rimanente dell'input e gli archi in *A* costituiscono una porzione dell'albero alle dipendenze.

Definizione 3.13. Data una frase $S = w_0 w_1 w_2 \dots w_n$ si definisce

1. Configurazione iniziale la configurazione $c_0 = ([\,], [w_0 w_1 w_2 \dots w_n],\ \emptyset)$
2. Configurazione finale la configurazione $c_f = \left([w_0], [\,],\ A_f\right)$, dove $\left|A_f\right| = n$.

Definizione 3.14. Si definisce transizione un operatore τ che mappa una configurazione *c* in una configurazione *c'*. Per indicarla si usano le due notazioni equivalenti $c \vdash_\tau c'$ oppure $c' = \tau(c)$. Un insieme di transizioni si indica con T.

Una transizione modifica una configurazione muovendo o togliendo i token nello stack e nel buffer e/o creando un nuovo arco. Non è detto che tutte le transizioni siano sempre applicabili: ognuna di esse ha infatti delle precondizioni che possono non essere rispettate da certe configurazioni. Di norma una transizione riguarda solo gli elementi in cima allo stack e i primi elementi del buffer: per questo motivo sono utili le notazioni $\sigma|w_i|w_j$ e $w_k|\beta$ in cui w_i e w_j sono i due elementi in cima ad uno stack generico mentre w_k è il primo elemento di un buffer generico.

Il sistema di transizioni così definito non è deterministico poiché di norma per ogni configurazione c'è più di un tipo di transizione applicabile (un'eccezione si ha con una configurazione con stack vuoto come quella iniziale). Per poter effettuare un parsing

deterministico bisogna quindi avere un procedimento per decidere in modo consistente qual è la transizione corretta.

Definizione 3.15. Data una configurazione *c* si definisce oracolo la funzione $o(c) = \tau$, dove τ è la transizione corretta secondo qualche criterio.

Nella pratica un oracolo perfetto è impossibile da costruire: per questo motivo, tramite tecniche di machine learning, viene costruito un modello che lo approssima.

Definizione 3.16. Data una configurazione c_i si definisce derivazione, o computazione, per una configurazione c_j la sequenza di transizioni $d = \tau_1\tau_2 \ldots \tau_k$ tale che $c_i \vdash_{\tau_1} c_{i+1} \vdash_{\tau_2} \ldots \vdash_{\tau_k} c_j$. Se tale sequenza esiste si dice che c_j è raggiungibile da c_i.

Definizione 3.17. Una sequenza di transizioni per una frase *S* è una derivazione in cui $c_i = c_0$ e $c_j = c_f$. Un albero alle dipendenze T_S è raggiungibile se esiste una sequenza di transizioni che lo genera.

Proprietà 3.18. Una sequenza di transizioni definisce una foresta *F* di alberi alle dipendenze proiettivi. Per trasformare *F* in un unico albero alle dipendenze *T* basta aggiungere tanti archi $(-ROOT-,\ r,\ w_i)$, con una *r* appropriata, quante sono le radici w_i di alberi in *F*.

Definizione 3.19. Due derivazioni *d* e *d'* si dicono equivalenti se, quando applicate alla stessa configurazione c_i, producono la stessa configurazione c_j. L'ambiguità che questa equivalenza genera nel parsing viene detta ambiguità spuria.

Un algoritmo di parsing con ambiguità spurie può avere, quindi, molte derivazioni diverse che raggiungono lo stesso albero alle dipendenze T_S. Questo costituisce un problema: il modello statistico viene definito sulle derivazioni delle strutture sintattiche della grammatica e, se si hanno derivazioni diverse per uno stesso elemento, la probabilità della struttura finale è data dalla probabilità marginale di tutte le derivazioni. Per questo motivo si configurano gli algoritmi in modo tale che scelgano deterministicamente una derivazione, detta canonica, tra tutte quelle possibili. Formalmente:

Definizione 3.20. Dato un albero alle dipendenze $T_S = (V_S,\ A_S)$ e una configurazione c_i per cui esiste una derivazione $c_i \vdash_{\tau_i} c_{i+1} \vdash_{\tau_{i+1}} \ldots \vdash_{\tau_k} c_f = \left([\ w_0], [\ ],\ A_f\right)$, l'oracolo statico è una funzione $staticOracle(T_S, c_i) = \tau_i$ che restituisce una transizione della derivazione canonica, che segue cioè i seguenti principi:

1. Tra due transizione è preferita quella che crea un arco.
2. Se molte transizioni creano un arco viene preferita quella che ha la distanza minima tra la testa e il dipendente.

3. Tra due transizioni è preferita quella che riduce lo stack.

Come si vedrà nella Sezione 3.5, è possibile definire altri tipi di oracolo, uno non deterministico che sfrutta l'ambiguità spuria e uno dinamico che evita la propagazione degli errori.

L'Algoritmo 3.1 illustra il generico algoritmo di dependency parsing basato sulle transizioni. Data una frase in ingresso viene creata la configurazione iniziale e si entra in un ciclo, in cui ad ogni iterazione viene applicata la transizione giudicata migliore dal modello, ottenendo così una nuova configurazione fino a giungere alla configurazione finale. La sequenza di transizioni così ottenuta costituisce l'albero alle dipendenze che viene restituito dal parser.

Gli algoritmi descritti in seguito si rifanno a questo modello e differiscono tra loro solo per la tecnica con cui è realizzato modello e per l'insieme T delle transizioni. Il tipo delle transizioni determina infatti la copertura dell'algoritmo, ossia se può costruire solo archi proiettivi oppure anche archi non proiettivi, così come il tipo di strategia impiegato nella costruzione degli archi, che può essere *bottom-up* nel caso in cui tutte le dipendenze che hanno un generico nodo w_i come testa sono costruite prima di quella in cui è dipendente o *top-down* se avviene il contrario.

Si noti inoltre che nel caso di parsing *labeled*, cioè nel caso in cui siano di interesse i tipi di dipendenza associati ad ogni arco, bisogna sostituire ogni transizione con tante transizioni quanti sono i diversi tipi di dipendenza: in questo modo ognuna di esse crea un arco con un'etichetta specifica. Ad esempio, supponendo di avere un insieme di relazioni $R = \{r_1, r_2, r_3\}$ e un insieme di tipi di transizioni $\mathrm{T} = \{\tau_1, \tau_2\}$, l'insieme delle transizioni possibili diventerà $\mathrm{T}_R = \{\tau_{1,r_1}, \tau_{2,r_1}, \tau_{1,r_2}, \tau_{2,r_2}, \tau_{1,r_3}, \tau_{2,r_3}\}$.

Algoritmo 3.1. Algoritmo di parsing basato sulle transizioni generico.

Input: $S = w_0 w_1 w_2 \dots w_n$
Output: albero alle dipendenze T_S
1: $c = (\sigma, \beta, A) = ([\,], [w_0 w_1 w_2 \dots w_n], \emptyset)$
2: **while** $|\sigma| > 1 \vee |\beta| > 0$ **do**
3: $\mathrm{T}' \leftarrow \emptyset$
4: **for each** τ in T **do**
5: **if** applicable(τ, c) **then**
6: $\mathrm{T}' \leftarrow \mathrm{T}' \cup \{\tau\}$
7: $\tau \leftarrow$model.giveBestTransition(T', c)
8: $c =$ apply(τ, c)
9: $V_S = \{w_0, w_1, \dots, w_n\}$
10: **return** $T_S = (V_S, A)$

Transizione	$c = (\sigma, \beta, A)$	$c' = (\sigma', \beta', A')$	Precondizioni
LEFT-ARC	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_j, \beta, A \cup \{w_j, r, w_i\})$	$w_i \neq -ROOT-$
RIGHT-ARC	$(\sigma\|w_i\|w_j, \beta, A)$	$(\sigma\|w_i, \beta, A \cup \{w_i, r, w_j\})$	
SHIFT	$(\sigma, w_i\|\beta, A)$	$(\sigma\|w_i, \beta, A)$	

Tabella 3.1. Transizioni nell'algoritmo Arc-Standard.

3.3.1. Algoritmo Arc-Standard

L'Algoritmo Arc-Standard è uno degli algoritmi di parsing basati sulle transizioni più semplici e usati ed è stato introdotto per la prima volta in [10]. Nella Tabella 3.1 sono riportate le transizioni possibili: due, LEFT-ARC e RIGHT-ARC, creano un arco tra due elementi adiacenti in cima allo stack mentre la terza, SHIFT, sposta un token dal buffer allo stack. Quindi, data una frase lunga *n*, i parser di questo tipo compiono *2n - 1* transizioni per ricavare un albero alle dipendenze: tutti i nodi devono essere spostati nello stack e per ogni nodo esclusa la radice deve esser creato un arco.

Questo algoritmo crea solo alberi proiettivi e implementa una strategia puramente bottom-up: quando viene creato un nuovo arco il nodo dipendente viene rimosso dallo stack e non può più essere usato in altre relazioni di dipendenza.

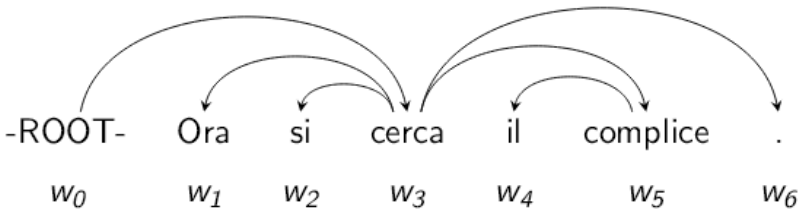


Figura 3.5. Esempio di albero alle dipendenze.

Esempio 3.3. Per la frase riportata nella Figura 3.5 la derivazione canonica secondo l'algortimo Arc-Standard è:

> *d* = SHIFT(0), SHIFT(1), SHIFT(2), LEFT-ARC(2, 3), LEFT-ARC(1, 3), SHIFT(4), SHIFT(5), LEFT-ARC(4, 5), RIGHT-ARC(3, 5), SHIFT(6), RIGHT-ARC(3, 6), RIGHT-ARC(0, 3)

mentre una tra le altre possibili derivazioni è:

> *d'* = SHIFT(0), SHIFT(1), SHIFT(2), LEFT-ARC(2, 3), SHIFT(4), SHIFT(5), LEFT-ARC(4, 5), RIGHT-ARC(3, 5), LEFT-ARC(1, 3), SHIFT(6), RIGHT-ARC(3, 6), RIGHT-ARC(0, 3)

Transizione	$\boldsymbol{c = (\sigma, \beta, A)}$	$\boldsymbol{c' = (\sigma', \beta', A')}$	**Precondizioni**
LEFT-ARC$_1$	$(\sigma \vert w_i \vert w_j, \beta, A)$	$(\sigma \vert w_j, \beta, A \cup \{w_j, r, w_i\})$	$w_i \neq -ROOT-$
RIGHT-ARC$_1$	$(\sigma \vert w_i \vert w_j, \beta, A)$	$(\sigma \vert w_i, \beta, A \cup \{w_i, r, w_j\})$	
LEFT-ARC$_2$	$(\sigma \vert w_i \vert w_j \vert w_k, \beta, A)$	$(\sigma \vert w_j \vert w_k, \beta, A \cup \{w_k, r, w_i\})$	$w_i \neq -ROOT-$
RIGHT-ARC$_2$	$(\sigma \vert w_i \vert w_j \vert w_k, \beta, A)$	$(\sigma \vert w_i \vert w_j, \beta, A \cup \{w_i, r, w_k\})$	
SHIFT	$(\sigma, w_i \vert \beta, A)$	$(\sigma \vert w_i, \beta, A)$	

Tabella 3.2. Transizioni nell'algoritmo di Attardi.

SI noti come LEFT-ARC(i, j) crei un arco da j a i mentre RIGHT-ARC(i, j) crei un arco da i a j. Le due derivazioni iniziano a differenziarsi a partire dalla quarta transizione, in cui in *d'* viene preferita uno SHIFT rispetto ad un LEFT-ARC.

3.3.2. Algoritmo di Attardi

L'algoritmo di Attardi [11] è il primo algoritmo di parsing basato sulle transizioni progettato per poter elaborare frasi non-projective. Per rispettare lo schema generale dato nell'Algoritmo 3.1 bisogna però considerarne una versione semplificata: per garantire la copertura di tutte le frasi non-projective bisognerebbe, infatti, dotare le configurazioni di uno stack ausiliario. Non è semplice definire una proprietà per indicare in modo analitico quali frasi non proiettive siano ammissibili con questa versione, che sono comunque un insieme considerevole delle frasi non proiettive.

Nella Tabella 3.2 si vedono i cinque tipi di transizione di questo algoritmo. SHIFT, LEFT-ARC$_1$ e RIGHT-ARC$_1$ sono le stesse transizioni dell'Arc-Standard mentre LEFT-ARC$_2$ e RIGHT-ARC$_2$ sono usate per costruire archi non proiettivi: queste due, infatti, creano un arco tra il primo e il terzo elemento dello stack senza che siano adiacenti.

3.4. Addestramento del Modello

Come si è visto nella Sezione 3.3, in particolare nell'Algoritmo 3.1 e nelle considerazioni relative alla Definizione 3.15, l'algoritmo di parsing basato sulle transizioni non è deterministico ed è compito del modello scegliere quale transizione applicare tra tutte quelle possibili per procedere nella costruzione dell'albero alle dipendenze.

Il modello è un classificatore che mappa una configurazione *c* in una transizione *τ*, e costituisce un problema standard nel machine learning. Nel caso di apprendimento supervisionato è necessario un training set di frasi annotato con gli alberi alle dipendenze corretti, così come descritto nella Sezione 3.2, che deve essere scomposto in coppie (istanza, classe) per essere analizzato dagli algoritmi di apprendimento: in questo caso le istanze sono le configurazioni, rappresentate secondo delle *features* appropriate (vedi Sezione 3.4.1),

mentre la classe è la transizione associata. Gli algoritmi di classificazioni più usati sono i classificatori lineari, come il percettrone (vedi Sezione 3.4.2) e la regressione logistica, e le reti neurali.

Per l'addestramento del modello si possono usare due approcci diversi, l'apprendimento *standard* e quello *on-line*. Nell'apprendimento standard per ogni frase *S* si estrae la derivazione canonica $d = \tau_0\tau_1 \ldots \tau_k$ che raggiunge il rispettivo albero alle dipendenze corretto $T_S = (V_S, A_S)$. Le coppie (c_0, τ_0), (c_1, τ_1), … , (c_k, τ_k), costituite da una configurazione e la transizione relativa, formano gli elementi di addestramento (*training samples*) ricavati da ogni frase in ingresso: in questo modo l'intero training set viene scomposto in un insieme di coppie. A questo punto ognuna di esse viene considerata indipendentemente dalle altre e si applica l'algoritmo di classificazione scelto, nel quale l'insieme delle classi coincide con quello delle transizioni possibili. Il principale vantaggio di questa tecnica è la facilità nel provare diversi algoritmi grazie alla disponibilità immediata di tutti i training samples, mentre il principale svantaggio consiste nella perdita delle informazioni su come si è giunti ad una determinata configurazione, visto che ogni elemento di addestramento è considerato in modo indipendente.

Nell'apprendimento on-line il modello viene aggiornato continuamente dopo l'acquisizione di ogni training sample. Viene infatti usata una variante dell'Algoritmo 3.1 per generare le istanze, ossia le configurazioni, in ordine sequenziale e un oracolo per ottenere le classi corrette, ossia le transizioni. L'Algoritmo 3.2 aggiorna dunque il modello solo nel caso in cui la sua predizione sulla transizione successiva sia sbagliata, ossia se non coincide con quella restituita dall'oracolo.

Algoritmo 3.2. Algoritmo di addestramento on-line.

Input: $trainingSet = \{(s_1, T_1), \ldots, (s_m, T_m)\}$
Output: modello

```
1:  model ← newModel(T)
2:  for each s_i, T_i in trainingSet do
3:      c = (σ, β, A) = ([ ], s, ∅)
4:      while |σ| > 1 ∨ |β| > 0 do
5:          T' ← ∅
6:          for each τ in T do
7:              if applicable(τ, c) then
8:                  T' ← T' ∪ {τ}
9:          τ ← model.giveBestTransition(T', c)
10:         τ_0 ← oracle(T_i, c)
11:         if τ ≠ τ_0 then
12:             model.update(T', c , τ_0)
13:             goto line 2
14:         c ← apply(τ, c)
15: return model
```

Dopo l'aggiornamento del modello si possono scegliere tre strategie diverse:

1. *Early update*, come si vede nell'Algoritmo 3.2 alla riga 13. Si basa sull'idea che se l'algoritmo fallisce nel raggiungere una configurazione non è significativo procedere con l'elaborazione di quella frase. Viene data, quindi, maggiore importanza alle transazioni iniziali.
2. *Aggressive update*, in cui l'istruzione nella riga 13 dell'algoritmo 3.2 viene sostituita dall'istruzione *goto line 4*. In questo modo, invece di saltare la frase, si continua ad aggiornare il modello fino ad ottenere la predizione corretta. Questo metodo ha lo svantaggio di provocare numerosi aggiornamenti con continue oscillazioni dei parametri del classificatore.
3. *Correct and go on*, in cui l'istruzione nella riga 13 dell'algoritmo 3.2 viene sostituita dall'istruzione $\tau \leftarrow \tau_0$: dopo l'aggiornamento del modello la transizione sbagliata viene corretta con quella data dall'oracolo e l'elaborazione della frase viene continuata.

Il vantaggio principale di questa tecnica sta nel fatto che viene mantenuta la sequenzialità delle derivazioni tramite l'applicazione di una strategia del tutto simile a quella del parsing (si noti infatti la somiglianza con l'Algoritmo 3.1). Il fatto di avere a disposizione solo un training sample per volta costituisce lo svantaggio principale: questo limita infatti la scelta dell'algoritmo (il più usato è il percettrone, vedi Sezione 3.4.2) e impone di iterare l'addestramento più volte per fare convergere i parametri a valori stabili.

3.4.1. Rappresentazione delle Features

L'addestramento di ogni tipologia di classificatore viene effettuato grazie a coppie (istanza, classe): nel caso di dependency parsing basato sulle transizioni un'istanza è una configurazione e una classe una transizione. La configurazione ha però una struttura molto complessa, essendo composta da uno stack, un buffer e un insieme di archi: l'insieme delle configurazioni possibili è praticamente infinito. Per questo è necessario introdurre una funzione Φ che mappi una configurazione c in un vettore di features m-dimensionale, $\Phi(c) = \boldsymbol{v}$. La funzione che estrae ogni singola *feature* è la combinazione di due funzioni distinte, una che estrae uno specifico nodo dalla configurazione e una che estrae l'attributo desiderato.

Teoricamente queste features potrebbero essere costituite da attributi arbitrari ricavati dalla configurazione. Generalmente sono però estratte da alcune caratteristiche di pochi token della configurazione, ad esempio quelli in cima allo stack e i primi del buffer. Queste caratteristiche sono sia informazioni di natura morfologica e grammaticale ricavate nelle fasi precedenti all'analisi sintattica (vedi Capitolo 2), ad esempio il lemma e la categoria

Posizione Token	Attributi				
	Forma	**Lemma**	**Categoria**	**Caratteristiche**	**Arco**
$\sigma(0)$	x	x	x	x	
$\sigma(1)$			x		
$lc(\sigma(0))$					x
$rc(\sigma(0))$					x
$\beta(0)$	x	x	x	x	
$\beta(1)$	x		x		
$\beta(2)$			x		
$\beta(3)$			x		
$lc(\beta(0))$					x
$rc(\beta(0))$					x

Tabella 3.3. Esempio di insieme di features per un algoritmo di parsing basato sulle transizioni. Le righe rappresentano i token considerati, le colonne i tipi di attributo e le celle con una x le features effettivamente estratte.

grammaticale, sia informazioni sugli archi già costruiti, ad esempio sui dipendenti dei nodi in cima allo stack o sui dipendenti delle parole del buffer, ricavati dall'albero alle dipendenze associato alla frase. La tipologia delle features scelte dipende dall'algoritmo di apprendimento e può influenzare in maniera significativa le prestazioni del classificatore.

Nella Tabella 3.3 si può vedere un esempio di features considerate da una classificatore: secondo questo schema da ogni configurazione ne vengono estratte 17. Nello specifico, la posizione del token indica da quale parola vengono estratti gli attributi, dove σ indica lo stack, β il buffer e le due funzioni *lc* e *rc* estraggono rispettivamente il nodo dipendente più a sinistra e più a destra del nodo a cui sono applicate. Tra gli attributi *Forma* indica la parola nella frase, *Lemma* la sua forma base, *Categoria* la sua categoria grammaticale, *Caratteristiche* altre categoria morfologiche come il genere o il numero, e *Arco* il tipo di relazione presente tra il nodo e il suo genitore. Si noti come in alcune posizioni potrebbe non esserci alcun nodo (ad esempio, se nel buffer ci sono solo due parole, $\beta(2)$ e $\beta(3)$ non sono definite): in questo caso si inserisce un valore speciale *null*.

I sistemi allo stato dell'arte [12] usano un insieme di features più complesso di questo, che comprende attributi come la distanza tra token particolari o ricavati dopo due estrazioni di nodo successive (ad esempio sul padre del padre del primo nodo del buffer).

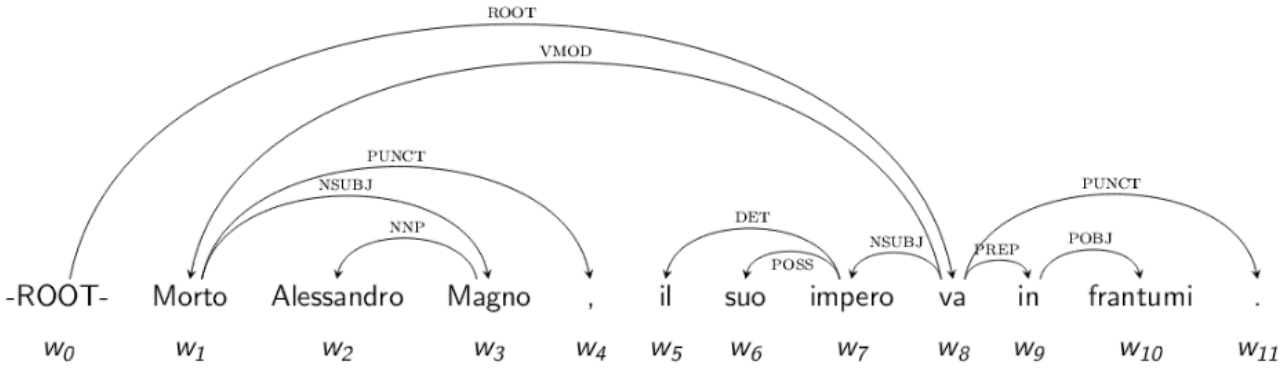


Figura 3.6. Esempio di albero alle dipendenze.

Esempio 3.4. Si consideri la frase riportata nella Figura 3.6; in particolare si consideri che l'elaborazione abbia proceduto fino ad ottenere la configurazione $c' = (\sigma, \beta, A)$, dove:

σ = {-ROOT-, Morto, impero}

β = {va, in, frantumi, .}

A = {(Magno, NNP, Alessandro), (Morto, NSUBJ, Magno), (Morto, PUNCT, ,), (impero, DET, il), (impero, POSS, suo)}

Le features per questa configurazione, estratte secondo lo schema riportato nella Tabella 3.3, sono riportate nella Tabella 3.4.

			Attributi		
Posizione Token	**Forma**	**Lemma**	**Categoria**	**Caratteristiche**	**Arco**
$\sigma(0)$	impero	impero	Nome	Maschile, Singolare	
$\sigma(1)$			Verbo		
$lc(\sigma(0))$					DET
$rc(\sigma(0))$					POSS
$\beta(0)$	va	andare	Verbo	Terza Persona Singolare, Indicativo, Presente	
$\beta(1)$	in		Preposizione		
$\beta(2)$			Nome		
$\beta(3)$			Punteggiatura		
$lc(\beta(0))$					VMOD
$rc(\beta(0))$					PUNCT

Tabella 3.4. Insieme di features per la configurazione nell'Esempio 3.4.

3.4.2. Percettrone

L'apprendimento supervisionato punta a ricavare una funzione che mappi un input $x \in \mathcal{X}$ in un output $y \in \mathcal{Y}$; di norma coincide con la classificazione binaria, in cui l'output corrisponde a due sole classi, cioè in cui $\mathcal{Y} = \{-1, +1\}$. Come visto nella Sezione 3.4.1 questa decisione viene presa in base alle informazioni contenute in un vettore di features derivato tramite la funzione $\Phi(x) : \mathcal{X} \to \mathcal{F} = \mathbb{R}^m$, che mappa una istanza dell'input in uno spazio m-dimensionale. Ipotizzando che i dati siano linearmente separabili, si può parametrizzare la funzione di classificazione in base ad un vettore di pesi $\boldsymbol{w} \in \mathbb{R}^m$ e ad uno scalare $b \in \mathbb{R}$, detto *bias*: si ottiene dunque la funzione $f(x, \boldsymbol{w}, b) = \boldsymbol{w}^T \Phi(x) + b = \sum_d w_d \Phi(x)_d + b$. La decisione viene presa in base al segno di *f*: se positivo la classe sarà *+1*, se negativo *-1*. Con questa ipotesi l'apprendimento si riduce, quindi, alla ricerca di buoni valori per $\boldsymbol{w}$ e b.

Il percettrone, introdotto in [13], apprende il vettore dei pesi e il bias on-line, ossia processando il training set un elemento alla volta, come visto nella Sezione 3.4. Ad ogni passo l'algoritmo si assicura che i parametri classifichino correttamente il training sample: in caso contrario vengono spostati più vicino ad esso. L'algoritmo itera sugli stessi dati fino a quando non vengono più apportati miglioramenti o fino al raggiungimento di un numero massimo di iterazioni. Questo procedimento converge sempre, ma è stato dimostrato che produce una scarsa generalizzazione, ossia delle scarse prestazioni sui dati non incontrati nella fase di addestramento. Questo problema viene risolto restituendo un vettore dei pesi che è la media di tutti i vettori ricavati durante l'elaborazione.

Nell'Algoritmo 3.3 è riportata un'implementazione efficiente di questo algoritmo, così come descritta in [14]. Invece di sommare le componenti del vettore dei pesi e il bias ad ogni iterazione per poi calcolarne la media, come richiederebbe un approccio banale, questi vengono sommati solo quando si compie un errore. Notare come nella riga 6 una previsione

Algoritmo 3.3. Algoritmo del percettrone (Averaged Perceptron).

Input: $trainingSet = \{(x_1, y_1), \ldots, (x_m, y_m)\}, iterMax$
Output: $(\boldsymbol{w}, b)$

1: $\boldsymbol{w_0} \leftarrow \langle 0, \ldots, 0 \rangle, b_0 \leftarrow 0$
2: $\boldsymbol{w_a} \leftarrow \langle 0, \ldots, 0 \rangle, b_a \leftarrow 0$
3: $c \leftarrow 1$
4: **for** i **from** 1 **to** iterMax **do**
5: **for** n **from** 1 **to** m **do**
6: **if** $y_n[\boldsymbol{w_0}^T \Phi(x_n) + b_0] \leq 0$ **then**
7: $\boldsymbol{w_0} \leftarrow \boldsymbol{w_0} + y_n \Phi(x_n), b_0 \leftarrow b_0 + y_n$
8: $\boldsymbol{w_a} \leftarrow \boldsymbol{w_a} + c y_n \Phi(x_n), b_a \leftarrow b_a + c y_n$
9: $c \leftarrow c + 1$
10: **return** $\left(\boldsymbol{w_0} - {\boldsymbol{w_a}}/{c}, b_0 - {b_a}/{c}\right)$

venga considerata sbagliata se ha un segno diverso dalla classe associata all'input, e solo in quel caso i parametri vengano corretti.

Nel caso del parsing le classi che deve prevedere l'algoritmo non sono due: il loro insieme corrisponde infatti a quello delle possibili transizioni, con $\mathcal{Y} = \{\overline{y}_1, \dots, \overline{y}_t\}$. In questo caso non si parla più di classificazione binaria ma multiclasse, ma è possibile adattare l'algoritmo secondo l'approccio cosiddetto *uno-contro-tutti*.

Innanzitutto è necessario raggruppare le classi in t coppie del tipo $\mathcal{Y}_1 = (\overline{y}_1, \{\overline{y}_2, \dots, \overline{y}_t\})$, $\mathcal{Y}_2 = (\overline{y}_2, \{\overline{y}_1, \overline{y}_3, \dots, \overline{y}_t\}), \dots, \mathcal{Y}_t = (\overline{y}_t, \{\overline{y}_1, \dots, \overline{y}_{t-1}\})$. Bisogna poi definire *t* coppie $(\boldsymbol{w}, b)$ diverse, ognuna associata ad uno di questi insiemi di classi $\mathcal{Y}_k$: si ottengono così t classificatori binari diversi, ognuno in grado di predire se un dato input appartiene ad una specifica classe, cioè quella che nella formulazione binaria corrisponde alla classe positiva. La funzione *f* usata per compiere la predizione viene dunque modificata in $f(x) = argmax_k(\boldsymbol{w}_k^T \Phi(x) + b_k)$: la classe *k* predetta sarà quella per il quale verrà restituito un valore più alto. È necessario modificare anche il modo in cui i parametri sono aggiornati: nel caso in cui la classe predetta $\hat{y}$ sia diversa da quella corretta y bisogna abbassare i pesi associati alla risposta sbagliata e alzare quelli associati alla risposta giusta, $\boldsymbol{w}_{\boldsymbol{y}} \leftarrow \boldsymbol{w}_{\boldsymbol{y}} + y\,\Phi(x)$ e $\boldsymbol{w}_{\hat{y}} \leftarrow \boldsymbol{w}_{\hat{y}} - \hat{y}\,\Phi(x)$ (per il bias si procede in modo analogo).

Un altro problema nell'utilizzo di questi algoritmi nell'ambito del parsing è dovuto al fatto che gli attributi estratti dalle configurazioni raramente sono ti tipo numerico, così come richiesto dall'algoritmo, ma categorici, ossia di tipo testuale definito su un insieme predefinito: ad esempio nel caso di un attributo di tipo *Lemma* (vedi Sezione 3.4.1) l'insieme è costituito dall'intero dizionario della lingua. È necessario effettuare, quindi, una trasformazione tra i due domini, che spesso risulta nella creazione di tanti attributi binari distinti quanti sono i possibili valori di uno di questi attributi.

3.5. Oracolo Dinamico

Gli oracoli, introdotti nella Definizione 3.15, sono essenziali nel caso in cui si usi un algoritmo on-line per l'apprendimento del modello (vedi Sezione 3.4) per verificare se la sua predizione sia giusta o sbagliata e decidere, quindi, se procedere con l'aggiornamento del modello stesso. Ne esistono di tre tipi: statici (vedi Definizione 3.20), non deterministici e dinamici.

Data una frase e l'albero alle dipendenze associato, gli oracoli statici seguono solo la derivazione canonica. In questo modo si introduce, però, un vincolo non necessario nell'addestramento, in quanto l'obiettivo di questa fase è approssimare un oracolo che

raggiunga l'albero alle dipendenze corretto, senza limitarsi ad una sola derivazione tra tutte quelle possibili. Per questo motivo sono stati introdotti gli oracoli non deterministici [15], che prendono in considerazione tutte le possibili derivazioni che raggiungono l'albero alle dipendenze corretto.

Definizione 3.21. Dato un albero alle dipendenze $T_S = (V_S,\ A_S)$ e una configurazione c_i per cui esiste almeno una derivazione da cui T_S è raggiungibile, l'oracolo non deterministico è una funzione $nonDeterministicOracle(T_S, c_i) = \{\tau_1, \dots, \tau_k\}$ che restituisce un insieme di transizioni, ognuna delle quali inizia una derivazione diversa che raggiunge T_S.

Per usare questo oracolo, l'Algoritmo 3.2 deve essere leggermente modificato per considerare il fatto che l'oracolo non deterministico restituisce un insieme di transizioni. Inoltre, nel caso in cui si utilizzasse un approccio *correct and go on*, è necessario sceglierne una per correggere la predizione sbagliata del modello: di norma si preferisce quella che massimizza il punteggio restituito dal modello.

Gli oracoli non deterministici non risolvono tutti i problemi associati agli oracoli statici. La strategia greedy impiegata dai parser basati sulle transizioni è, infatti, affetta dal problema della propagazione degli errori. L'Algoritmo 3.2 addestra il modello solo su configurazioni da cui l'albero alle dipendenze corretto è sempre raggiungibile. Se il parser commette un errore si ritrova in una configurazione che è improbabile abbia caratteristiche riconducibili a quelle che ha incontrato della fase di apprendimento. In questa situazione il modello deve, dunque, classificare configurazioni non incontrate in precedenza, e nel farlo commetterà con grande probabilità una serie di errori. Per ovviare a questo problema sono stati introdotti gli oracoli dinamici [15], in cui il parser viene lasciato sbagliare anche durante l'addestramento, in modo da addestrare il modello anche su configurazioni non raggiungibili seguendo solo le derivazioni che terminano nell'albero corretto.

Se si ammettono degli errori durante l'addestramento, l'albero alle dipendenze corretto T_S, chiamato nel seguito albero gold (*gold tree*) T_G, può non essere più raggiungibile durante l'elaborazione della frase. In questo caso, se si suppone che tutti gli archi abbiano la stessa importanza, l'obiettivo del parser diventa il raggiungimento di un albero con il numero minimo di archi sbagliati rispetto l'albero gold. Per questo motivo si definiscono due funzioni di *loss* tramite le quali è possibile confrontare rispettivamente alberi e configurazioni con l'albero alle dipendenze corretto.

Definizione 3.22. Dato un albero alle dipendenze $T = (V_S,\ A)$, si definisce loss di T rispetto all'albero gold $T_G = (V_S,\ A_G)$ la cardinalità dell'insieme differenza tra A e A_G, $\mathcal{L}(T,\ T_G) = |A \setminus A_G|$.

Definizione 3.23. Data una configurazione *c*, si definisce loss di *c* rispetto l'albero gold $T_G = (V_S,\ A_G)$ la loss minima tra tutti gli alberi raggiungibili da *c*, $\mathcal{L}(c,\ T_G) = min_{T \in \mathcal{D}(c)}\big(\mathcal{L}(T,\ T_G)\big)$, dove $\mathcal{D}(c)$ indica l'insieme di tutti gli alberi alle dipendenze raggiungibili da *c*.

Proprietà 3.24. Data una derivazione $c_0 \vdash_{\tau_0} c_1 \vdash_{\tau_1} \ldots \vdash_{\tau_k} c_f$ che raggiunge un albero alle dipendenze *T*, la funzione di loss è monotona non decrescente, ossia $\mathcal{L}(c_0) = 0 \leq \mathcal{L}(c_1) \leq \mathcal{L}(c_2) \leq \cdots \leq \mathcal{L}\big(c_f\big) = \mathcal{L}(T)$.

Questa proprietà deriva dalla natura incrementale del parser alle dipendenze, in cui un arco una volta creato non può più essere modificato o eliminato.

Definizione 3.25. Dato un albero gold $T_G = (V_S,\ A_G)$, si definisce costo di una transizione $c_i \vdash_\tau c_{i+1}$ la differenza tra le loss delle due configurazioni, $\mathcal{C}(\tau, c_i, T_G) = \mathcal{L}(c_{i+1}, T_G) - \mathcal{L}(c_i, T_G)$.

Usando questa definizione di costo è possibile confrontare transizioni in configurazioni in cui l'albero gold non è raggiungibile. In questo modo si definisce l'oracolo dinamico come la funzione che restituisce un insieme di transizioni a costo zero.

Definizione 3.26. Dato un albero gold $T_G = (V_G,\ A_G)$ e una configurazione c_i per cui esiste almeno una derivazione da cui T_G è raggiungibile, l'oracolo dinamico è una funzione $dynamicOracle(T_G, c_i) = \{\tau_i \mid \mathcal{C}(\tau_i, c_i, T_G) = 0\}$ che restituisce un insieme di transizioni, ognuna delle quali inizia una derivazione diversa che raggiunge T_S con la stessa loss.

Nell'Algoritmo 3.3 si può vedere l'algoritmo generale di addestramento del modello modificato nel caso in cui si utilizzi un oracolo dinamico. Si noti come, per consentire

Algoritmo 3.3. Algoritmo di addestramento on-line con oracolo dinamico.

Input: $trainingSet = \{(s_1,\ T_1),\ \ldots,\ (s_m,\ T_m)\}$
Output: modello

```
model ← newModel(T)
for each s_i, T_i in trainingSet do
    c = (σ, β, A) = ([ ], s, ∅)
    while |σ| > 1 ∨ |β| > 0 do
        T' ← ∅
        for each τ in T do
            if applicable(τ, c) then
                T' ← T' ∪ {τ}
        τ ← model.giveBestTransition(T', c)
        T_0 ← dynamicOracle(T_i, c)
        if τ ∉ T_0 then
            model.update(T', c , T_0)
        c ← apply(τ, c)
    return model
```

l'esplorazione degli errori, non venga adottata nessuna delle strategie descritte nel seguito dell'Algoritmo 3.2 e la transizioni venga, quindi, applicata direttamente alla configurazione anche se sbagliata.

3.5.1. Oracolo Dinamico per l'Algoritmo Arc-Standard

Data una configurazione generica, il calcolo della loss non è immediato: per svolgerlo bisogna basarsi su proprietà analitiche degli alberi alle dipendenze o su una simulazione di tutte le derivazioni dell'algoritmo a partire da quella configurazione. Nel caso dell'Algoritmo Arc-Standard (vedi Sezione 3.3.1) si usa il secondo approccio [16].

Nell'Algoritmo 3.4 è illustrato, dunque, lo schema generico dell'implementazione dell'oracolo dinamico per l'Algoritmo Arc-Standard. Ogni transizione ammissibile viene innanzitutto applicata alla configurazione in ingresso per ottenere una nuova configurazione. Da quest'ultima si ricava un buffer di dimensione ridotta, con il quale viene calcolata la loss secondo l'Algoritmo 3.5. Il procedimento per il suo calcolo è, quindi, diviso in due parti: la riduzione del buffer e il calcolo vero e proprio della loss.

Data una configurazione, nella fase di riduzione vengono estratti i più grandi sottoalberi di T_G che hanno il loro span completamente in β. Le loro radici sono messe in una lista, chiamata buffer ridotto, disposte secondo l'ordine con cui appaiono nella frase.

Definizione 3.27. Dato un albero gold $T_G = (V_G,\ A_G)$ e una configurazione $c = (\sigma,\ \beta,\ A)$, si definisce buffer ridotto β_R una sottosequenza di β composta da token che sono radici di alberi *T* che soddisfano le seguenti proprietà:

1. *T* è un sottoalbero di T_G il cui span è interamente compreso in β.
2. *T* è bottom-up completo per T_G, ossia ogni nodo *w* in *T* diverso dalla radice non ha dipendenti in σ.
3. *T* è massimale per T_G, ossia qualunque sottoalbero *T'* in T_G di cui *T* è a sua volta sottoalbero viola le condizioni precedenti.

Algoritmo 3.4. Algoritmo dell'oracolo dinamico per l'algoritmo Arc-Standard.

Input: configurazione $c = (\sigma, \beta, A)$, insieme di transizioni ammissibili T'
Output: insieme di transizioni con loss pari a zero T_0

$\mathrm{T}_0 \leftarrow \emptyset$
for each τ in T **do**
  $c' = (\sigma', \beta', A') \leftarrow \mathrm{apply}(\tau, c)$
  $\beta_R \leftarrow reduce(\beta')$
  $l \leftarrow computeLoss(\sigma', \beta_R)$
  if $l = 0$ **then**
    $\mathrm{T}_0 \leftarrow \mathrm{T}_0 \cup \{\tau\}$
return T_0

Un modo alternativo e meno efficiente per ricavare il buffer ridotto rispetto l'implementazione diretta della definizione è rappresentato dall'esecuzione sul buffer dell'Algoritmo Arc-Standard con oracolo statico, che produce una foresta di sottoalberi di T_G che rispetta la Definizione 3.27.

Questa riduzione, inoltre, non è essenziale per il calcolo della loss, ma viene eseguita per ridurne il tempo di elaborazione: β_R può essere visto, infatti, come il risultato di un pre-processing di β, ottenuto applicando tutte le transizioni che generano sottoalberi di T_G e che possono essere eseguite indipendentemente dallo stack.

Nella seconda fase vengono simulate, tramite tecniche di programmazione dinamica, tutte le possibili computazioni dell'Algoritmo Arc-Standard realizzabili a partire dallo stack e dal buffer ridotto. Nello spazio di ricerca così ottenuto è compreso almeno un albero alle dipendenze raggiungibile dalla configurazione considerata con loss minima.

L'Algoritmo 3.5 si basa su una matrice di dimensione $|\sigma| \times |\beta_R|$, in cui ogni elemento è un dizionario da intero a intero: ogni $\boldsymbol{A}[i,j](h)$ memorizza infatti la loss minima degli alberi alle dipendenze radicati in *h* che possono essere ottenuti combinando i token dell'insieme costituito dagli *i* elementi in cima allo stack e dai primi *j* elementi del buffer ridotto, insieme indicato nello pseudocodice come $\Delta_{i,j} = \{\sigma[k] \mid k \in [1,i]\} \cup \{\beta_R[k] \mid k \in [1,j]\}$. Per semplificare la scrittura dell'algoritmo si suppone che l'elemento in cima allo stack sia stato copiato nella prima posizione del buffer ridotto.

All'inizio dell'algoritmo (riga 1) l'elemento $\boldsymbol{A}[1,1](\sigma[1])$ viene inizializzato con la somma della loss di tutti gli alberi già costruiti nello stack, corrispondente alla loss della configurazione. Successivamente si inizia una visita su tutte le antidiagonali della matrice in

Algoritmo 3.5. Algoritmo per il calcolo della loss per l'algoritmo Arc-Standard.

Input: stack σ, buffer ridotto β_R
Output: loss minima

1: $\boldsymbol{A}[1,1](\sigma[1]) \leftarrow \sum_{i \in [1,|\sigma|]} \mathcal{L}(T(\sigma[i]), T_G)$
2: **for** d **from** 1 **to** $|\sigma| + |\beta_R| - 1$ **do**
3: **for** j **from** $max\{1,\ d - |\sigma| + 1\}$ **to** $min\{d, |\beta_R|\}$ **do**
4: $i \leftarrow d - j + 1$
5: **if** $i < |\sigma|$ **then**
6: **for each** $h \in \Delta_{i,j}$ **do**
7: $\boldsymbol{A}[i+1,j](h) \leftarrow min\{\boldsymbol{A}[i+1,j](h), \boldsymbol{A}[i,j](h) + \delta_G(h \rightarrow \sigma[i+1])\}$
8: $\boldsymbol{A}[i+1,j](\sigma[i+1]) \leftarrow min\{\boldsymbol{A}[i+1,j](\sigma[i+1]), \boldsymbol{A}[i,j](h) + \delta_G(\sigma[i+1] \rightarrow h)\}$
9: **if** $j < |\beta_R|$ **then**
10: **for each** $h \in \Delta_{i,j}$ **do**
11: $\boldsymbol{A}[i,j+1](h) \leftarrow min\{\boldsymbol{A}[i,j+1](h), \boldsymbol{A}[i,j](h) + \delta_G(h \rightarrow \beta_R[j+1])\}$
12: $\boldsymbol{A}[i,j+1](\beta_R[j+1]) \leftarrow min\{\boldsymbol{A}[i,j+1](\sigma[i+1]), \boldsymbol{A}[i,j](h) + \delta_G(\beta_R[j+1] \rightarrow h)\}$
13: **return** $\boldsymbol{A}[|\sigma|, |\beta_R|](0)$

ordine ascendente. Per ogni elemento $\boldsymbol{A}[i,j]$ sono considerate due possibili espansioni, una con gli elementi dello stack (righe 5-8) e l'altra con quelli del buffer ridotto (righe 9-12), tramite la costruzione sia di un LEFT-ARC che di un RIGHT-ARC. Nel caso in cui il sottoalbero risultante abbia loss inferiore a quella degli alberi ottenuti fino quel momento, il valore salvato nel dizionario viene sostituito con quello appena calcolato. La funzione δ_G viene usata per controllare se il nuovo arco creato è presente nell'albero gold: formalmente viene definita come $\delta_G(i \to j) = \begin{cases} 0 & se\ (i \to j \in A_G) \\ 1 & altrimenti \end{cases}$.

L'algoritmo restituisce, infine, il valore contenuto in $\boldsymbol{A}[|\sigma|, |\beta_R|](0)$, ossia la loss dell'albero risultante dalla combinazione di tutti i nodi nello stack e nel buffer ridotto e radicato nel nodo -ROOT-.

Questo algoritmo ha una complessità computazionale pari a $O(n^3)$, dove *n* è la lunghezza della frase. La riduzione del buffer, infatti, può essere svolta in tempo al più lineare considerando il caso in cui venga effettuata tramite l'applicazione al buffer dell'Algoritmo Arc-Standard con oracolo statico. Durante il calcolo della loss ogni elemento del dizionario $\boldsymbol{A}[i,j]$ viene aggiornato un numero di volte che non dipende dall'input con un'operazione effettuata in modo costante. Si conclude, dunque, che la complessità dell'algoritmo in termini di tempo è pari a $O\big(|\sigma| \times |\beta_R| \times (|\sigma| + |\beta_R|)\big) = O(n^3)$, in quanto tutte queste quantità sono limitate superiormente da *n*.

4. Dipendenze Non Proiettive in Parser Proiettivi

Come è stato accennato nella Sezione 3.1.1, gli algoritmi di parsing alle dipendenze possono essere suddivisi in due gruppi, quelli che nella fase di addestramento possono elaborare le frasi con dipendenze non proiettive e quelli che, invece, sono costretti a non considerarle. Questi ultimi algoritmi sono più semplici e, quindi, robusti rispetto ai primi: ciò li porta ad avere buone prestazioni su quelle lingue in cui frasi non proiettive sono rare, ossia in cui costituiscono il 5-10% del corpus, mentre su quelle lingue in cui sono comuni, in cui rappresentano il 15-25% del corpus, raggiungono un'accuratezza non ottimale.

Nonostante questo, la differenza delle prestazioni degli algoritmi proiettivi rispetto a quelle teoricamente raggiungibili su lingue altamente non proiettive è nell'ordine di un punto di accuratezza. La principale metrica di valutazione degli algoritmi, infatti, è il numero di parole che dipendono dal nodo corretto (vedi Sezione 5.1) mentre la percentuale di archi non proiettivi in questi linguaggi è bassa, ossia circa il 2% del totale, sebbene sia grande il numero di frasi in cui sono distribuiti. È importante quindi considerare queste frasi anche perché non facendolo si esclude a priori la possibilità di raggiungere l'accuratezza massima della tecnica in esame.

In letteratura si è cercato di superare la suddivisione tra i due tipi di algoritmi cogliendo i vantaggi di entrambi, ossia la semplicità e la robustezza di quelli sviluppati per frasi proiettive e il pieno utilizzo dei corpus consentito dagli algoritmi sviluppati per frasi non proiettive. Si possono applicare diversi approcci: il primo, illustrato nella Sezione 4.1, prevede la trasformazione degli alberi alle dipendenze associati alle frasi in una fase di pre-processing, eventualmente abbinata ad una trasformazione inversa dell'output del parser per ricostruire la struttura non proiettiva corretta. Un altro approccio, illustrato nella Sezione 4.2, prevede di effettuare la trasformazione da alberi non proiettivi ad alberi proiettivi durante l'addestramento, in cui il parser tenta di raggiungere uno tra gli albero alle dipendenze a loss minima.

4.1. Trasformazione delle Frasi

Un primo approccio per l'elaborazione di frasi non proiettive con algoritmi proiettivi consiste nella trasformazione degli alberi alle dipendenze loro associati in modo da eliminare la non proiettività. Tra le varie tecniche disponibili in letteratura ne sono illustrate due, riportate con i nomi degli autori: nella Sezione 4.1.1 l'algoritmo di Nivre-Nilsson e nella Sezione 4.1.2 l'algoritmo di Goldberg.

Entrambi i metodi producono un albero alle dipendenze proiettivo con loss minima rispetto a quello non proiettivo dato in ingresso, ma differiscono nel modo in cui lo generano. L'algoritmo di Nivre-Nilsson considera infatti trasformazioni che mantengono gli archi del nuovo albero vicini a quelli corretti, ossia spostandone la testa il minimo necessario rispetto la sua posizione originaria, mentre lo spazio di ricerca dell'algoritmo di Goldberg comprende tutti gli alberi proiettivi a loss minima, anche quelli ottenuti con spostamenti considerevoli della testa degli archi.

4.1.1 Algoritmo di Nivre-Nilsson

L'algoritmo descritto in [17] è diviso in due fasi, una che si colloca nella fase di addestramento del parser e una che si colloca nella fase di parsing vero e proprio. La prima consiste in un pre-processing del training set in cui si elimina la non proiettività tramite un numero minimo di operazioni, dette *lift*. Le etichette degli archi interessati da questa operazione, inoltre, sono modificate per includere le informazioni sui cambiamenti effettuati. In questo modo il parser viene addestrato ad assegnare etichette che contengono informazioni riguardanti i lift, utile nella parte successiva dell'algoritmo, che consiste in un post-processing dell'output dell'algoritmo di parsing, in cui le informazioni contenute nelle etichette sono sfruttate per ricostruire la struttura non proiettiva corretta. Per questo motivo, questa tecnica è detta anche parsing pseudo-proiettivo.

Come osservato in [18], un albero non proiettivo può essere sempre trasformato in un albero proiettivo effettuando un certo numero di operazioni di lift su ogni arco non proiettivo, ossia sostituendo la testa di ognuno di questi archi con la testa del nodo padre (l'operazione può essere anche ricorsiva sullo stesso arco). Formalmente:

Definizione 4.1. Dato un arco $w_i \rightarrow w_j$, se $w_k \rightarrow w_i$ si definisce la funzione lift come $lift(w_i \rightarrow w_j) = w_k \rightarrow w_j$.

Per non perdere troppe informazioni sulle relazioni tra le parole, si è interessati a conservare il più possibile la struttura originaria dell'albero alle dipendenze, ossia ad eseguire il minor numero possibile di lift. Nell'Algoritmo 4.1 questo viene assicurato

Algoritmo 4.1. Algoritmo di Nivre-Nilsson (lift).

Input: albero alle dipendenze $T = (V,\ A)$
Output: albero alle dipendenze proiettivo $T' = (V, A')$
1: $A' \leftarrow A$
2: **while** $isProjective(T)$ **do**
3: $\quad a \leftarrow smallestNonProjectiveArc(A')$
4: $\quad A' \leftarrow (A' - \{a\}) \cup lift(a)$
5: **return** $T' = (V, A')$

eseguendo, fino a quando l'albero alle dipendenze in esame non è stato trasformato in uno proiettivo, un lift alla volta sull'arco restituito dalla funzione *smallestNonProjectiveArc*, che individua la dipendenza non proiettiva con distanza minore tra testa e dipendente (in caso di parità procede da sinistra a destra).

Nel caso in cui si fosse interessati solo a rendere proiettivo il training set, ossia solo alla fase di pre-processing, l'Algoritmo 4.1 è sufficiente, anche se si ottiene un'accuratezza leggermente inferiore rispetto al metodo completo [17].

Nel caso in cui, dopo il parsing, si voglia ricostruire la struttura non proiettiva corretta, è necessario estendere l'insieme R dei tipi di relazione per memorizzare le informazioni sui lift effettuati, in modo da poterli invertire. Si posso scegliere diversi schemi per codificare i dati necessari negli archi. Quello a cui corrisponde l'accuratezza più elevata prevede l'associazione ad ogni arco su cui è stato fatto un lift di un'etichetta data dalla giustapposizione tra la sua etichetta e quella esistente nell'albero di partenza tra la testa dell'arco e la sua testa (che, dopo l'operazione di lift, sarà la nuova testa dell'arco) e ad ogni arco nel percorso di lift di una etichetta modificata in modo tale da indicare come sia stato effettuato in lift lungo quegli archi, senza però altre informazioni.

4.1.2 Algoritmo di Goldberg

L'algoritmo di Goldberg è concettualmente simile all'Algoritmo 3.5 (vedi Sezione 3.5.1) per il calcolo della loss per l'Algoritmo Arc-Standard. Infatti, per trasformare una frase da non proiettiva a proiettiva, questo algoritmo ricava tutti i possibili alberi proiettivi raggiungibili e ne sceglie uno tra quelli aventi loss minima rispetto l'albero gold.

Un algoritmo simili potrebbe essere ottenuto eseguendo due modifiche all'Algoritmo 3.5. Come prima cosa, sarebbe necessario salvare in ogni *entry* del dizionario costituente gli elementi della matrice dei *backpointer*, ossia dei riferimenti che permettano di ricavare da quali sottoalberi si è ottenuto l'albero a loss minima la cui radice costituisce la chiave dell'elemento del dizionario. In questo modo, partendo da una delle radici degli alberi con loss minima e visitando ricorsivamente i nodi fino ad arrivare agli elementi dello stack e del buffe ridotto, è possibile ricavare un albero proiettivo a loss minima. Oltre a questo è necessario modificare l'algoritmo per permettere combinazioni tra gli elementi nel buffer ridotto (analogamente a quanto viene fatto nell'algoritmo proposto nella Sezione 4.2). Con questi due cambiamenti, a partire da una frase iniziale $S = w_0 w_1 w_2 \dots w_n$ con un albero gold T_G, ponendo $\sigma = \emptyset$ e $\beta_R = w_0 w_1 w_2 \dots w_n$, si può ottenere un albero alle dipendenze con loss minima.

Questa soluzione ha, però, un tempo di esecuzione troppo elevato. Infatti la complessità asintotica è equivalente a quella dell'Algoritmo 4.2 esposto nel seguito, che ha complessità $O(n^5)$, dove n è la lunghezza della frase che, in questo caso, viene inserita senza alcuna riduzione o pre-processing nel buffer ridotto. L'aumento rispetto all'Algoritmo 3.5 è causato dalla necessità di combinare elementi adiacenti arbitrari. Per ovviare a questo problema l'algoritmo di Goldberg usa una variante dell'algoritmo proposto in [19] che ha una complessità $O(n^3)$. Si noti come non sia presente in letteratura una pubblicazione che descriva questa specifica implementazione, ma come l'algoritmo sia lo stesso conosciuto alla comunità scientifica.

Una spiegazione dettagliata del funzionamento dell'algoritmo è complessa, ma per ricavare la complessità asintotica è sufficiente fare una breve considerazione. Per calcolare la loss minima l'algoritmo ricava tutti gli alberi raggiungibili a partire dalla configurazione data in input, combinando tutte le coppie di sottoalberi adiacenti a partire da quelli con span minore. Si può rappresentare uno di questi sottoalberi generici di span dal nodo *i* al nodo *k* e di radice *h* tramite la notazione $[i, h, k]$. In questo modo, per ottenere un generico albero $[i, h', j]$, si devono calcolare tutte le possibili combinazioni di $[i, h, k]$ e $[k, d, j]$, dove $h' \in \{h, d\}$.

In questo algoritmo gli elementi della matrice usata rappresentano però alberi di tipo $[i, h, h]$ o $[h, h, k]$, in cui cioè la radice coincide con uno dei nodi ai margini dello span. In questo modo l'algoritmo dovrà calcolare un numero maggiore di combinazioni (infatti un albero $[i, h, k]$ sarà rappresentato da due alberi $[i, h, h]$ e $[h, h, k]$ e, quindi, ogni transizione che lo interessa sarà costituita da due combinazioni in successione dell'algoritmo) ma ognuna di esse avrà solo tre gradi di libertà, ognuno dei quali è limitato superiormente da *n*. In questo modo si ricava come l'Algoritmo di Goldberg abbia una complessità di $O(n^3)$. Una considerazione analoga per l'Algoritmo 4.2 troverebbe come per esso ci siano cinque gradi di libertà nelle combinazioni, confermando la complessità di $O(n^5)$ ricavata analiticamente.

4.2. Algoritmo Proposto

Un approccio alternativo a quanto proposto nella Sezione 4.1 prevede di effettuare la trasformazione da un albero alle dipendenze non proiettivo ad un albero proiettivo direttamente durante la fase di apprendimento del modello. Per farlo si abbina ad un algoritmo di parsing già sviluppato un oracolo dinamico (vedi Sezione 3.5) creato appositamente per consentire l'approssimazione delle strutture non proiettive. Si noti come questa soluzione non possa produrre alberi alle dipendenze non proiettivi, in quanto le transizioni utilizzate sono quelle dell'algoritmo proiettivo scelto e non è possibile modificare gli archi come fatto dall'Algoritmo di Nivre-Nilsson.

Questa soluzione ha un grande vantaggio rispetto agli approcci precedenti nel caso in cui anche per loro si utilizzi un oracolo dinamico. Infatti, trasformando le frasi non proiettive durante la fase di pre-processing, si sostituisce l'albero gold T_G con un nuovo albero T_G' a loss minima che l'algoritmo cercherà di apprendere al posto di T_G. L'approccio proposto, invece, considera contemporaneamente tutti gli alberi a loss minima: il suo spazio di ricerca è più ampio.

L'algoritmo proposto in questa Tesi è, quindi, una variante dell'algoritmo per il calcolo della loss per l'oracolo dinamico dell'Algoritmo Arc-Standard presentato nella Sezione 3.5.1. Segue lo schema illustrato nell'Algoritmo 3.4: è composto, infatti, da due parti, la riduzione del buffer, utilizzata per diminuire il tempo di esecuzione, e il calcolo vero e proprio della loss.

Data una configurazione, la fase di riduzione è simile a quella del caso proiettivo: l'unica differenza è, infatti, il vincolo che i sottoalberi le cui radici costituiscono il buffer siano proiettivi. Formalmente:

Definizione 4.2. Dato un albero gold $T_G = (V_G, A_G)$ e una configurazione $c = (\sigma, \beta, A)$, si definisce buffer ridotto β_R per l'algoritmo proposto una sottosequenza di β composta da token che sono radici di alberi T che soddisfano le seguenti proprietà:

1. T è un sottoalbero proiettivo di T_G il cui span è interamente compreso in β.
2. T è bottom-up completo per T_G, ossia ogni nodo w in T diverso dalla sua radice non ha dipendenti in σ.
3. T è massimale per T_G, ossia qualunque albero T' di cui T è sottoalbero in T_G viola le condizioni precedenti.

In questo caso, eseguire l'Algoritmo Arc-Standard con un oracolo statico sul buffer è un modo efficiente per ricavare il buffer ridotto β_R.

Nella seconda fase, illustrata nell'Algoritmo 4.2, vengono simulate, tramite tecniche di programmazione dinamica, tutte le possibili computazioni dell'Algoritmo Arc-Standard a partire dallo stack e dal buffer ridotto. L'algoritmo usa una matrice quadrata di lato $|\sigma| + |\beta_R|$ per salvare le computazioni intermedie. Ogni elemento di coordinate [i, j] della matrice è un dizionario da interi ad interi: $\boldsymbol{M}[i,j](h)$ è quindi pari alla loss dell'albero con span dal nodo i al nodo j e di radice h. Si noti come, nella spiegazione che segue, gli indici riportati per indicare i nodi si riferiscano sempre alla posizione della parola nel lato della matrice, ossia nella giustapposizione dello stack e del buffer ridotto, e non alla sua posizione della frase.

Questa matrice comprende al suo interno la matrice usata per le simulazioni dell'Algoritmo 3.5, che corrisponde alla sottomatrice di dimensione $|\sigma| \times |\beta_R|$ posta nell'angolo superiore destro della matrice. La modifica realizzata dall'algoritmo proposto

Algoritmo 4.2. Algoritmo proposto per il calcolo della loss.

Input: stack σ, buffer ridotto β_R
Output: loss minima

1: **for** i **from** 1 **to** $|\sigma| + |\beta_R|$ **do**
2: $\boldsymbol{M}[i,i](i) = loss(i)$
3: **for** d **from** 1 **to** $|\sigma| + |\beta_R| - 1$ **do**
4: **for** i **from** $max\{|\sigma| - d, 1\}$ **to** $|\sigma| + |\beta_R| - 1$ **do**
5: $j = i + d$
6: **if** $j > |\sigma| + |\beta_R| - 1$ **then**
7: **break**
8: $L \leftarrow \emptyset$
9: **if** $i \leq |\sigma|$ **then**
10: $L \leftarrow \{i\} \cup \{|\sigma|, \dots, j-1\}$
11: **else**
12: $L \leftarrow \{i, \dots, j-1\}$
13: **for each** *l* **in** *L* **do**
14: $r = l + 1$
15: **for** *h* **from** *i* **to** *l* **do**
16: **for** h' **from** *r* **to** *j* **do**
17: $\boldsymbol{M}[i,j](h) = min\{\boldsymbol{M}[i,j](h), \boldsymbol{M}[i,l](h) + \boldsymbol{M}[r,j](h') + \delta_G(h \rightarrow h')\}$
18: $\boldsymbol{M}[i,j](h') = min\{\boldsymbol{M}[i,j](h'), \boldsymbol{M}[i,l](h) + \boldsymbol{M}[r,j](h') + \delta_G(h' \rightarrow h)\}$
19: **return** $\boldsymbol{M}[1, |\sigma| + |\beta_R|](1)$

consiste proprio nella possibilità di combinare gli alberi che costituiscono il buffer ridotto, possibilità esclusa nell'Algoritmo 3.5 perché violerebbe la Definizione 3.27, e la dimensione maggiore della matrice utilizzata è necessaria per poter effettuare queste operazioni.

È grazie a queste combinazioni, infatti, che vengono eliminati gli archi non proiettivi. Gli alberi le cui radici costituiscono lo stack sono proiettivi per costruzione, essendo l'Arc-Standard un algoritmo che costruisce solo alberi proiettivi, mentre gli alberi le cui radici costituiscono il buffer ridotto sono proiettivi per la Definizione 4.2. Questi alberi potrebbero però essere dipendenti di archi non proiettivi: le derivazioni simulate, costituite solo da RIGHT-ARC e LEFT-ARC, forzano l'algoritmo a cercare di raggiungere l'albero di loss minima tra tutti quelli proiettivi raggiungibili dalla configurazione corrente.

L'Algoritmo 4.2 come prima cosa (righe 1-2) inizializza gli elementi nella diagonale principale con la loss dell'albero che li ha come radice: per ogni elemento della diagonale esiste solo un valore nel dizionario (un albero di span da *i* a *i* può avere solo il nodo *i* come radice) che corrisponde all'i-esimo elemento nella giustapposizione tra stack e buffer ridotto. Bisogna notare come tutti gli alberi nel buffer ridotto abbiano loss pari a zero per definizione, essendo costruiti da archi presenti nell'albero gold.

Successivamente sono visitate tutte le superdiagonali della matrice in ordine ascendente, partendo da quella posta subito sopra alla diagonale principale fino all'ultima, costituita solo dall'elemento $\boldsymbol{M}[1, |\sigma| + |\beta_R|]$. Per ognuna di esse sono visitati solo gli elementi con indice di colonna $j > |\sigma| - 1$: gli elementi nello stack, escluso quello in cima,

non possono essere combinati direttamente tra loro, ma solo tramite il coinvolgimento anche di elementi del buffer. A seconda della posizione nella diagonale viene poi individuato il tipo di combinazione da simulare (righe 8-12), ossia se tra nodi e/o alberi con span solo nel buffer ridotto o se anche con elementi dello stack. Nel secondo caso (righe 9-10), infatti, bisogna consentire la combinazione anche con l'elemento nella diagonale principale che rappresenta un nodo nello stack, mentre non bisogna accedere agli elementi della matrice che corrispondono a combinazioni tra due suoi nodi.

Nelle righe 13-18, infine, viene calcolata la loss degli alberi ottenuti combinando tutte le coppie di sottoalberi il cui span corrisponde agli insiemi definiti dagli intervalli [i, l] e [r, j], dove $i \leq l < r \leq j, r = l + 1$ e $|\sigma| \in [i, j]$. Sono considerate tutte le possibili radici dei sottoalberi (righe 15-16) e tutte le transizioni dell'Algoritmo Arc-Standard che creano un arco, ossia RIGHT-ARC (riga 17) e LEFT-ARC (riga 18). La funzione δ_G, definita come nella Sezione 3.5.1, viene usata per controllare se il nuovo arco creato è presente nell'albero gold, e restituisce 1 in caso contrario.

La complessità computazionale dell'algoritmo si ricava facilmente. Indicando con *n* la lunghezza della frase d'ingresso e usando l'Algoritmo Arc-Standard con oracolo statico, il buffer ridotto può essere calcolato in tempo $O(n)$.

Nel calcolo vero e proprio della loss, ogni elemento sopra la diagonale maggiore (esclusi quelli corrispondenti a combinazioni di elementi dello stack) viene visitato una sola volta. Ogni visita, corrispondente alle righe 13-18, ha un costo, misurato in termini di numero di accessi al dizionario associato alla cella della matrice, pari a

$$\sum_{l=i}^{j-1} \sum_{h=i}^{l} \sum_{h'=l+1}^{j} 1 = -\frac{1}{6}(i-j-2)(i-j-1)(i-j) = O(d^3)$$

in cui si è utilizzata la relazione $j = i + d$ e si è limitato superiormente l'intervallo di valori nella sommatoria esterna. Considerando che ogni aggiornamento dei valori del dizionario può essere eseguito in tempo costante e che il lato della matrice e la sua diagonale sono lunghi $|\sigma| + |\beta_R| = O(n)$, la complessità totale dell'algoritmo proposto è $O(n^5)$. In pratica, comunque, il tempo di esecuzione è inferiore, in quanto nella maggior parte delle configurazioni $|\sigma| + |\beta_R| \ll n$. In termini di spazio la complessità è $O(n^3)$, in quanto nel dizionario costituente ogni elemento della matrice ci possono essere $O(n)$ entry e le dimensioni della matrice stessa sono proporzionali alla lunghezza della frase.

Si noti, in ogni caso, come questa complessità sia relativa alla sola fase di addestramento del modello: durante la fase di parsing vero e proprio l'oracolo non viene utilizzato e la complessità è, quindi, $O(n)$, così come accennato nella Sezione 3.2. Su un sistema di potenza medio bassa (sistema operativo Linux, processore dual core 3.5 GHz, 8GB

di RAM) questo si traduce nella possibilità di fare l'analisi sintattica di circa 80 frasi al secondo.

È importante, infine, fare una considerazione sull'accuratezza massima raggiungibile con la tecnica qui esposta. L'algoritmo proposto realizza un nuovo oracolo per l'Algoritmo Arc-Standard: il parser risultante potrà produrre, quindi, sempre e solo alberi alle dipendenze proiettive. Questo significa che, con riferimento alle principali metriche di valutazione utilizzate (vedi Sezione 5.1), l'Exact Match per le frasi non proiettive sarà sempre pari a 0, in quanto l'algoritmo punta a costruire la migliore approssimazione proiettiva della frase in ingresso. L'Attachment Score, invece, avrà un limite superiore pari a

$$AS_{max} = 100 - \sum_{i=1}^{|\mathcal{D}|} \frac{min\{loss(c_0, T_{G,i})\}}{tokens(i)}$$

dove $\mathcal{D}$ è il dataset in esame e la funzione *tokens(i)* restituisce il numero di token dell'i-esima frase; si noti come l'elemento della sommatoria valga 0 in caso di frasi proiettive, essendo la loss minima posta al numeratore uguale a 0. Il limite superiore è dato, quindi, dalla somma del rapporto tra le loss minime delle frasi non proiettive e la loro lunghezza. Nelle frasi non proiettive, infatti, il parser dovrà sbagliare obbligatoriamente un numero di archi pari alla loss minima, dovendo sempre sostituire quel numero di archi con altrettanti archi proiettivi.

5. Risultati Sperimentali

In questo capitolo si presentano i risultati degli esperimenti condotti per verificare quale, tra le varie tecniche descritte nel Capitolo 4 per poter considerare alberi alle dipendenze non proiettivi nell'addestramento di algoritmi di parsing proiettivi, sia la migliore.

I test sono stati condotti utilizzando il software sviluppato dal dott. Sartorio per la sua tesi di dottorato [20]. Questo programma realizza un parser alle dipendenze basato sulle transizioni (vedi Sezione 3.3) che usa come algoritmo di apprendimento un percettrone e come features quelle descritte in [12]. È stato scritto principalmente in Python: solo il codice del percettrone è stato scritto in C per ragioni di efficienza. Per questa Tesi il software è stato esteso per includere l'implementazione dell'algoritmo per l'oracolo dinamico descritto nella Sezione 4.2, con il quale sono stati condotti tutti gli esperimenti.

Nella Sezione 5.1 sono illustrate le metriche di valutazione dei test, mentre nella Sezione 5.2 si illustrano le caratteristiche principali del corpus utilizzato. Nella Sezione 5.3 si riportano i dati degli esperimenti che confrontano le due tecniche presentate nella Sezione 4.1 che prevedono un pre-processing del training set con l'algoritmo proposto nella Sezione 4.2 e con l'Algoritmo Arc-Standard, che viene usato come *baseline*. Nella Sezione 5.4, infine, sono esposti alcuni commenti più approfonditi sui risultati sperimentali. Per una panoramica più dettagliata sul *setup* degli esperimenti si rimanda a quanto scritto nell'Appendice A.

5.1. Metriche di Valutazione

Il metodo standard per valutare un parser consiste nel sottoporgli l'elaborazione di un *test set* per poi confrontare l'output con i risultati corretti e ricavarne, quindi, l'accuratezza in modo sperimentale. Nel caso del parsing alle dipendenze può essere espressa con diverse metriche:

1. *Exact match*, ossia la percentuale di frasi per le quali il parsing è stato eseguito in modo corretto.
2. *Attachment score*, ossia la percentuale di parole che hanno la testa corretta. Questa metrica è possibile grazie alla Proprietà 3.5, per la quale ogni nodo ha una e una sola testa.
3. *Precision* e *Recall*, usate nel caso in cui si vogliano valutare i tipi di relazione presi singolarmente. Corrispondono alle metriche usate normalmente nella valutazione dei

modelli: la precision è la percentuale di dipendenze di un tipo specifico che erano corrette nell'output del parser, mentre la recall è la percentuale di dipendenze di un tipo specifico nel test set che il parser ha classificato correttamente.

Tutte queste metriche possono essere *unlabeled*, ossia si considerano solo le teste delle relazioni per verificare che sia corretta, o *labeled*, ossia si considerano sia le teste che il tipo di relazione.

Le due metriche più utilizzate in letteratura, e usate anche per riportare i risultati dei test nella Sezione 5.3, sono il *labeled attachment score* (LAS) e *l'unlabeled attachment score* (UAS).

5.2. Struttura del Corpus

Per lo svolgimento degli esperimenti sono stati usati i dataset di alcune lingue compresi all'interno dei due dataset multi linguistici sviluppati per la *shared task* delle *Conference on Computational Natural Language Learning* (CoNLL) del 2006 [21] e del 2007 [22]. In entrambe queste conferenze venne fornito ai partecipanti un dataset, composto da una decina di lingue, per confrontare le varie tecniche di parsing alle dipendenze con apprendimento basato su metodi data-driven. Il dataset per ogni lingua è stato ricavato da corpus già esistenti, i cui formati sono stati convertiti in uno unitario. Ogni dataset è già suddiviso in un training set e in un test set e ha già assegnate le informazioni corrette derivate dall'analisi grammaticale.

La Tabella 5.1 riporta alcune informazioni notevoli sui dataset delle lingue utilizzate. In particolare si può vedere come i test siano stati condotti sia su lingue con un'alta percentuale di frasi non proiettive (ceco, tedesco, portoghese e ungherese) sia su lingue in cui la non proiettività è un caso particolare (inglese e italiano). Si noti, inoltre, come i test set

	Ceco	**Tedesco**	**Portoghese**	**Ungherese**	**Inglese**	**Italiano**
Dataset	CoNLL '07	CoNLL '06	CoNLL '06	CoNLL '07	CoNLL '07	CoNLL '07
Training Set						
Token (in migliaia)	432	700	207	132	447	71
Frasi	25364	39216	9071	6034	18577	3110
% Archi Non-Proj.	1.9	2.3	1.3	2.9	0.3	0.5
% Frasi Non-Proj.	23.2	27.8	18.9	26.4	6.7	7.4
Test Set						
Token	4724	5694	5867	7344	5003	5096
Frasi	286	357	288	390	214	249
% Frasi Non-Proj.	22.7	12.3	17	23.6	8.4	5.22

Tabella 5.1. Informazioni sui dataset utilizzati negli esperimenti.

siano piccoli, tutti nell'ordine di 250-400 frasi. Le dimensioni ridotte potrebbero portare a valutazioni errate durante l'analisi dei risultati, in quanto un'accuratezza leggermente inferiore o superiore al previsto dipenderebbe da pochissime frasi. È consigliabile, quindi, non soffermarsi sui risultati di una singola lingua ma guardarli nel loro complesso.

5.3. Risultati dei Test

L'obiettivo degli esperimenti riportati in questa Sezione consiste nel verificare quale tra i metodi descritti nel Capitolo 4 per l'utilizzo di dipendenze non proiettive in parser proiettivi sia il migliore e, in generale, se considerare frasi non proiettive produca un miglioramento nell'accuratezza del parser.

Per questo motivo sono state effettuate quattro diverse serie di test per ogni lingua, tutte condotte usando come oracolo quello descritto nella Sezione 4.2. L'algoritmo proposto, infatti, è una generalizzazione dell'oracolo dinamico per l'Algoritmo Arc-Standard descritto nella Sezione 3.5.1, che funziona solo con frasi proiettive. Nel caso in cui l'algoritmo proposto venga usato solo con frasi proiettive l'unica differenza tra i due sta nel tempo di esecuzione: l'algoritmo proposto risulta più lento in quanto deve simulare un numero maggiore di transizioni, cioè quelle corrispondenti alle combinazioni di elementi nel buffer ridotto. In dettaglio, sono stati condotti test sui seguenti algoritmi:

1. Arc-Standard, in cui durante l'addestramento non si sono considerate le frasi non proiettive. Viene usato come baseline su cui confrontare i risultati degli altri metodi.
2. Trasformazione di Goldberg, in cui le frasi non proiettive nel training set sono state trasformate in una fase di pre-processing tramite l'algoritmo relativo (vedi Sezione 4.1.2).
3. Trasformazione di Nivre-Nilsson, in cui le frasi non proiettive nel training set sono state trasformate in una fase di pre-processing tramite l'algoritmo relativo (vedi Sezione 4.1.1). Viene utilizzata una variante dell'algoritmo che non prevede la codifica delle informazioni sugli archi non proiettivi modificati, e quindi non consente la ricostruzione dell'albero non proiettivo corretto in una fase di post-processing.
4. Algoritmo proposto, in cui i training set sono usati completamente e senza alcun pre-processing.

Tra le informazioni notevoli riguardanti il setup degli esperimenti (per una descrizione più dettagliata si veda l'Appendice A), bisogna considerare come i risultati riportati nel seguito siano il valore medio di cinque test ripetuti ordinando ogni volta le frasi nel training set in modo diverso. A causa del metodo di apprendimento del percettrone, infatti, una loro diversa disposizione può comportare una differenza nell'accuratezza anche

	Ceco	Tedesco	Portoghese	Ungherese	Inglese	Italiano
Unlabeled Attachment Score (UAS)						
Arc-Standard	80,74	88,16	85,36	78,79	87,52	84,33
Goldberg	80,77	89,02	**85,61**	78,57	87,63	84,49
Nivre-Nilsson	81,36	89,05	85,56	**79,39**	87,82	84,41
Algoritmo Proposto	**81,44**	**89,33**	85,53	79,34	**87,84**	**84,71**
Labeled Attachment Score (LAS)						
Arc-Standard	72,49	86,16	81,97	69,76	86,65	80,22
Goldberg	73,02	87,16	**82,44**	70,66	86,73	80,56
Nivre-Nilsson	73,49	87,08	82,43	**71,30**	**86,93**	80,42
Algoritmo Proposto	**73,51**	**87,44**	82,42	71,19	86,92	**80,70**

Tabella 5.2. Risultati degli esperimenti. Per ogni lingua lo score maggiore è evidenziato in grassetto.

di un punto tra un esperimento e l'altro. La fase di addestramento del modello, inoltre, è consistita sempre di 20 iterazioni dell'algoritmo, numero in cui i risultati convergono ad un valore stabile con tutte le tecniche in esame. Infine, i token costituiti da simboli di punteggiatura sono stati considerati al fine del calcolo dell'accuratezza.

Nella Tabella 5.2 sono riportati i risultati degli esperimenti condotti sui dataset descritti nella Sezione 5.2. Le sei lingue si possono dividere in due gruppi: quelle in cui le frasi con associati alberi alle dipendenze non proiettivi costituiscono una percentuale notevole del corpus (ceco, tedesco, portoghese e ungherese) e quelle in cui sono una percentuale trascurabile (inglese e italiano). Sono state scelte lingue con caratteristiche diverse per vedere se, al variare del metodo con cui sono elaborate le frasi non proiettive, si possono rilevare comportamenti diversi tra le due classi di linguaggi. Nel seguito, inoltre, due risultati saranno considerati equivalenti se differiscono per meno di 0.1 e chiaramente diversi se la loro differenza è maggiore di 0.2.

Innanzitutto si noti come, per le lingue fortemente non proiettive, ci sia un aumento significativo dell'accuratezza rispetto a quella dell'Arc-Standard usando una delle tecniche che consente l'elaborazione anche di dipendenze non proiettive; le lingue proiettive presentano un aumento dell'accuratezza più contenuto ma comunque rilevante, essendo di circa 0.3 punti di accuratezza. Unica eccezione è il portoghese, per il quale l'aumento nell'accuratezza, seppur presente, è più limitato: visti i risultati ottenuti con le altre lingue la causa di questo comportamento risiede probabilmente in qualche caratteristica del test set, in particolare la sua dimensione ridotta. È dunque confermata l'idea secondo la quale sia importante utilizzare le frasi non proiettive durante la fase di addestramento del modello, anche per quanto riguarda lingue non fortemente proiettive.

Si confrontano ora le tre tecniche a coppie per stabilire quale sia la migliore:

- Algoritmo di Nivre-Nilsson vs Algoritmo di Goldberg. Paragonando le due tecniche che prevedono un pre-processing della frase si nota come l'Algoritmo di Nivre-Nilsson sia il migliore: nei casi in cui presenta un'accuratezza minore, infatti, la differenza tra i due metodi è inferiore a 0.1 mentre per il ceco, l'ungherese e l'inglese si nota uno scostamento significativo. Questo è probabilmente dovuto al fatto che gli archi trasformati dall'Algoritmo di Nivre-Nilsson sono sempre molti simili a quelli non proiettivi originali, rendendo quindi più semplice l'elaborazione di statistiche ragionevoli da parte del modello, mentre quelli modificati dall'Algoritmo di Goldberg possono risultare anche molto distanti dagli archi non proiettivi di partenza.
- Algoritmo di Nivre-Nilsson vs Algoritmo Proposto. Da questo confronto emerge come i due algoritmi abbiano, per quattro lingue su sei, delle prestazioni sostanzialmente equivalenti, con l'algoritmo proposto che presenta un'accuratezza maggiore solo nel tedesco e nell'italiano. La vicinanza dei risultati è dovuta probabilmente a qualche aspetto linguistico non intuitivo, oltre che all'efficacia delle operazioni di lift nel mantenere una struttura sintattica significativa dal punto di vista statistico.
- Algoritmo di Goldberg vs Algoritmo Proposto. Da questo confronto si ricava, invece, come l'algoritmo proposto abbia un'accuratezza maggiore in cinque lingue su sei, mentre solo nel portoghese i due metodi risultano equivalenti.

Dall'analisi dei risultati si ricava, quindi, come l'algoritmo proposto sia migliore dei metodi analizzati già presenti in letteratura. Inoltre, il confronto tra l'Algoritmo di Nivre-Nilsson e quello di Goldberg, mai effettuato in precedenza, si risolve a favore del primo.

5.4. Commenti

In questa Sezione sono riportati i risultati di alcuni test svolti per approfondire alcuni aspetti degli esperimenti riportati nella Sezione 5.3.

Prima della scrittura di questa Tesi ci si aspettava, infatti, un maggiore distacco tra l'algoritmo proposto ed entrambi gli algoritmi che trasformano le frasi rispetto a quello verificato sperimentalmente. Nella fase di sviluppo inoltre, in cui i test non sono stati condotti sul test set ma su una porzione del training set (vedi Sezione A.1), le accuratezze erano ancora più vicine di quelle riportate nella Tabella 5.2.

Nella Figura 5.1 si può vedere il risultato di un esperimento condotto sul dataset del ceco, nel quale si osserva il cambiamento dell'accuratezza al variare del numero massimo di errori consentiti durante la fase di addestramento. Come visto nella Sezione 3.5, nel caso in cui le predizioni dell'oracolo dinamico e del modello siano diverse, viene scelta sempre

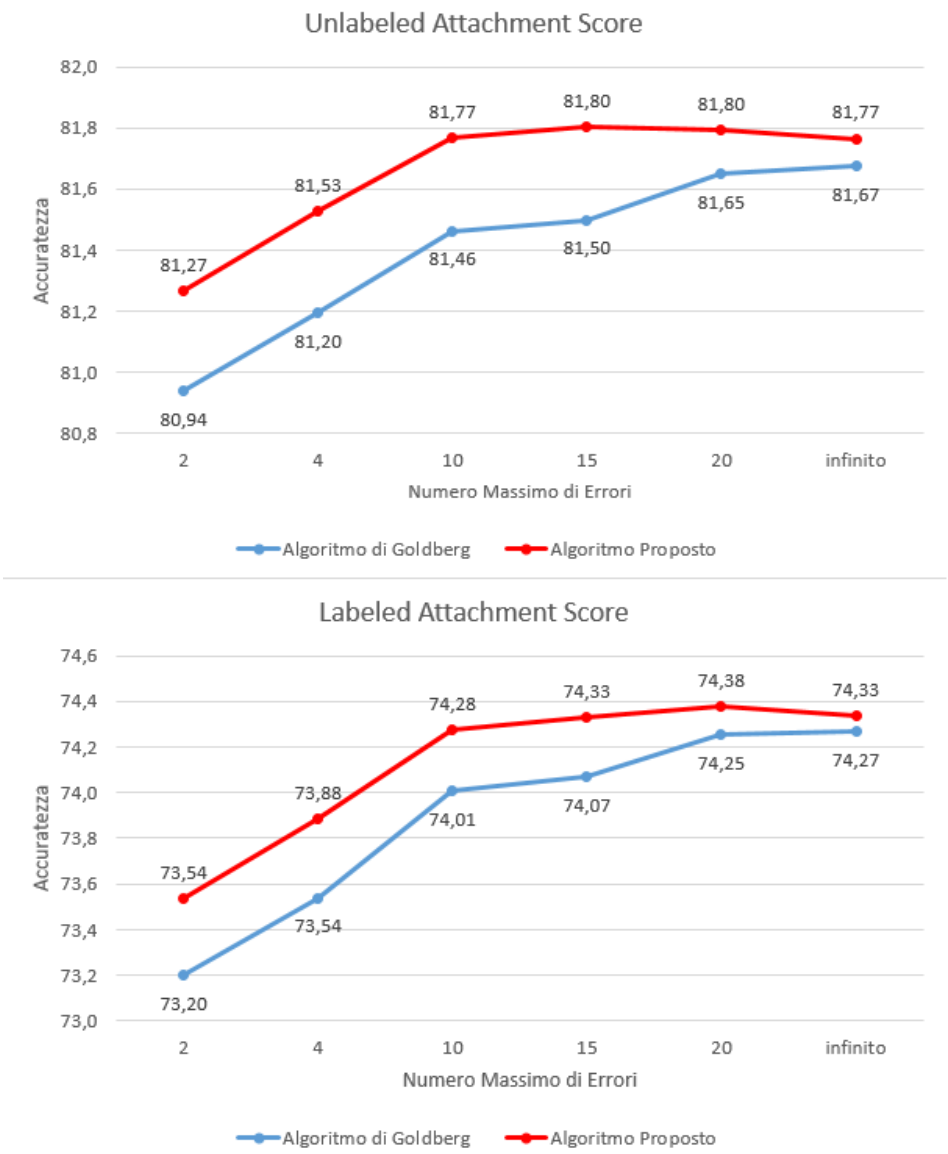
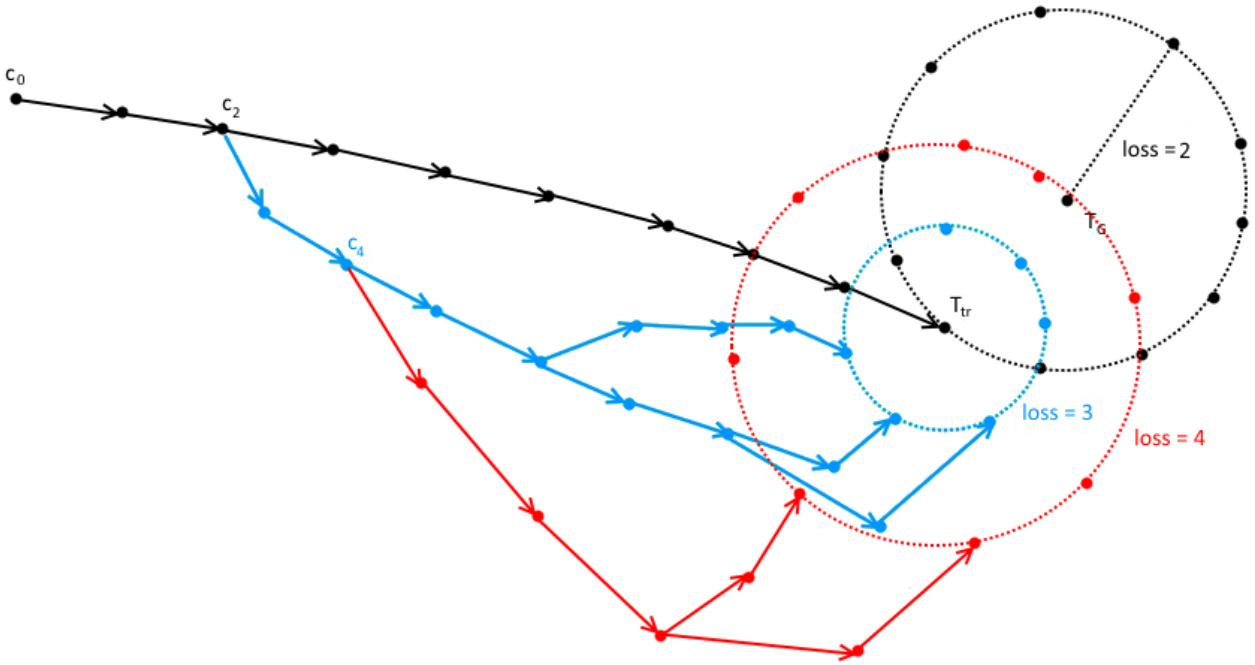


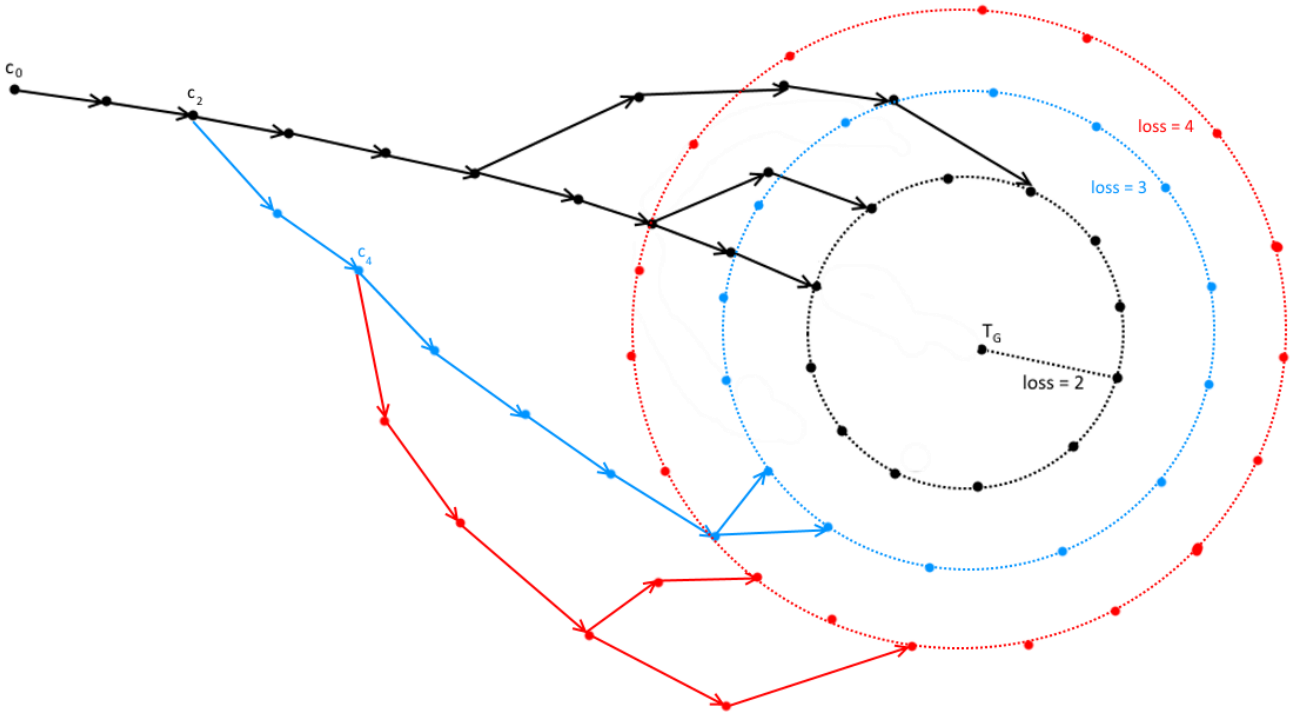
Figura 5.1. Evoluzione dell'accuratezza misurata sul ceco, partizionando il training set in un development set (80% delle frasi) e un test set (20% delle frasi), al variare del numero massimo degli errori consentiti al parser durante la fase di addestramento.

quella indicata dal secondo, in modo da addestrarlo anche su configurazioni sbagliate. È però possibile limitare il numero di volte in cui il parser adotta questo comportamento, per sostituirlo, dopo una certa soglia, con una strategia correct and go on (vedi Sezione 3.4). Nello svolgimento di questo esperimento il training set originale è stato partizionato in modo da poterne usare l'80% come training set e il restante 20% come test set. In questo modo il test set risulta più significativo dal punto di vista statistico.

Nella Figura 5.1, in particolare nel grafico relativo al Labeled Attachment Score, si osserva come all'aumentare del numero massimo di errori l'accuratezza dell'Algoritmo di Goldberg e dell'algoritmo proposto siano sempre più vicine. È ragionevole ipotizzare che un



(a) Evoluzione nel caso dell'Algoritmo di Goldberg (o, analogamente, di Nivre-Nilsson).



(b) Evoluzione nel caso dell'algoritmo proposto.

Figura 5.2. Sono poste a confronto le elaborazioni di una frase non proiettiva con albero gold T_G e loss minima pari a 2 usando un algoritmo di trasformazione (a) o l'algoritmo proposto (b). Per semplicità si usa una rappresentazione bidimensionale dello spazio di ricerca. Nei disegni T_{tr} è l'albero scelto dall'algoritmo di trasformazione come albero proiettivo obiettivo, i punti corrispondono a delle configurazioni, le frecce a delle transizioni, una serie di frecce ad una derivazione, un cerchio ad un insieme di configurazioni con la stessa loss.

esperimento analogo condotto con l'Algoritmo di Nivre-Nilsson avrebbe prodotto dei risultati equivalenti.

Questi risultati confermano per via sperimentale un'ipotesi sulla struttura dello spazio di ricerca di questi algoritmi, formulata per spiegare la somiglianza delle accuratezze tra l'algoritmo proposto e i metodi preesistenti. Data una frase non proiettiva, infatti, gli algoritmi di Goldberg e di Nivre-Nilsson forzano il modello ad imparare una derivazione che raggiunge l'albero alle dipendenze a loss minima ottenuto dalla loro applicazione, mentre l'algoritmo proposto cerca di raggiungere uno qualunque degli alberi a loss minima: inizialmente, quindi, lo spazio di ricerca di quest'ultimo è più ricco degli altri. In entrambi i casi, però, man mano che durante l'addestramento il modello compie delle predizioni sbagliate, la loss minima aumenta e così il numero di alberi alle dipendenze raggiungibili. Se il numero degli errori effettuati è molto maggiore della loss minima iniziale i due spazi di ricerca diventano molto simili, per coincidere nel caso in cui l'algoritmo proposto abbia seguito la derivazione che produce l'albero ottenuto con l'algoritmo di trasformazione e sbagliato, rispetto a T_G, gli archi richiesti per poterlo raggiungere. Per questo motivo i tre algoritmi considerati hanno un'accuratezza più simile tra loro di quanto era stato previsto.

Nella Figura 5.2 si può vedere un esempio di questa teoria. Sono poste a confronto due elaborazioni di una frase non proiettiva utilizzando una tecnica che la trasforma in una fase di pre-processing (Figura 5.2(a)) e l'algoritmo proposto (Figura 5.2(b)): le frecce rappresentano delle transizioni tra i punti che rappresentano delle configurazioni. Lo spazio di ricerca degli algoritmi, che è probabilmente uno spazio n-dimensionale dove n è il numero di token della frase, viene rappresentato per semplicità in modo bidimensionale.

Nella figura si nota come, seguendo le derivazioni nere, l'algoritmo proposto possa raggiungere dallo stato iniziale più di un albero alle dipendenze a loss minima, mentre gli Algoritmo di Nivre-Nilsson e di Goldberg possano raggiungere solo quello ottenuto durante il pre-processing. Non appena viene compiuto un errore, rappresentato nel disegno da un cambio di colore, le due elaborazioni iniziano a somigliarsi: da questo momento in poi, infatti, anche gli algoritmi di trasformazione hanno la possibilità di raggiungere più di un albero. Se gli errori compiuti durante l'addestramento sono molti di più rispetto la loss minima, i due spazi di ricerca, qui rappresentati come circonferenze di configurazioni aventi la stessa loss, diventano molto simili.

Successivamente, nella Tabella 5.3, si può osservare l'accuratezza scorporata per archi proiettivi e non proiettivi. Si sottolinea come, per un errore durante l'acquisizione dei dati, i risultati riportati siano stati ricavati da una media di quattro test e di cinque come fatto nella Sezione 5.3. Si consideri inoltre che a causa delle dimensioni ridotte dei test set

	Ceco		Tedesco		Portoghese		Ungherese		Inglese		Italiano	
	P	**NP**	**P**	**NP**	**P**	**NP**	**P**	**NP**	**P**	**NP**	**P**	**NP**
Unlabeled Attachment Score (UAS)												
Arc-Standard	81,45	76,08	88,49	83,25	86,26	**70,63**	80,98	51,76	87,74	83,58	85,77	**60,04**
Goldberg	81,36	76,25	89,52	82,40	86,48	70,22	80,66	**52,33**	87,80	83,37	85,97	59,14
Nivre-Nilsson	82,03	76,33	89,63	81,88	**86,50**	69,23	**81,73**	51,32	**88,04**	**84,55**	85,84	59,77
Alg. Proposto	**82,14**	**76,95**	**89,64**	**83,96**	86,39	69,89	81,46	52,24	88,02	84,01	**86,23**	59,50
Labeled Attachment Score (LAS)												
Arc-Standard	72,11	75,25	86,39	82,99	82,69	**69,99**	71,29	51,23	86,83	83,58	81,39	**60,04**
Goldberg	72,57	75,62	87,51	82,40	**83,20**	69,97	72,14	51,72	86,87	83,37	81,76	59,14
Nivre-Nilsson	73,10	75,71	87,52	81,88	83,19	69,23	**72,97**	50,97	**87,10**	**84,55**	81,64	59,77
Alg. Proposto	**73,12**	**76,29**	**87,62**	**83,83**	83,10	69,64	72,67	**51,85**	87,07	84,01	**81,97**	59,52

Tabella 5.3. Risultati degli esperimenti scorporati per archi proiettivi (P) e non proiettivi (NP).

una differenza di un punto per le frasi non proiettive può essere dovuto a solo un paio di dipendenze.

Dal confronto tra le accuratezza degli archi proiettivi si nota, innanzitutto, come utilizzare anche le frasi non proiettive durante l'addestramento aumenti la precisione nella costruzione delle dipendenze proiettive. Al contrario, questo non è sempre vero per gli archi non proiettivi: nel portoghese e nell'italiano l'Arc-Standard ha, infatti, un'accuratezza leggermente migliore di tutti gli altri metodi. Se il risultato dell'italiano non ha molta rilevanza, visto il numero molto basso di frasi non proiettive, il risultato del portoghese è più difficile da spiegare, ma in linea con quanto riportato nella Tabella 5.2. Questo fa pensare, infatti, che ci sia qualche caratteristica particolare del test set, ad esempio un elevato numero di parole nuove, che limita l'aumento dell'accuratezza. Per avere delle riposte definitive bisognerebbe, però, svolgere delle analisi più approfondite. In 11 casi su 24 si nota, infine, una perfetta uguaglianza tra UAS e LAS nel caso di archi non proiettivi: questo sembra suggerire come, per questo tipo di dipendenze, ricavare la relazione giusta sia difficile quanto ricavare l'etichetta esatta o come manchino informazioni sufficienti per ricavarle.

Si è svolto, infine, un confronto tra i risultati ottenuti usando l'algoritmo proposto con quelli ottenuti con l'Algoritmo di Attardi (vedi Sezione 3.3.2), un algoritmo non proiettivo. I dati sperimentali per quest'ultimo sono tratti da [20] e sono direttamente confrontabili con quelli ottenuti in questa Tesi in quanto i test sono stati eseguiti sugli stessi dataset e con gli stessi parametri.

L'algoritmo proposto ha un'accuratezza migliore in cinque lingue su sei, con un distacco significativo per quanto riguarda il tedesco e l'ungherese, mentre solo nel ceco l'Algoritmo di Attardi ha un'accuratezza migliore. Facendo una media tra le accuratezze riportate si vede come l'algoritmo proposto abbia un'accuratezza totale di (84.70, 80.36),

	Ceco	Tedesco	Portoghese	Ungherese	Inglese	Italiano
Unlabeled Attachment Score (UAS)						
Algoritmo Proposto	81,44	**89,33**	**85,53**	**79,34**	**87,84**	**84,71**
Attardi	**82.08**	88.86	85.36	76.72	87.38	84.43
Labeled Attachment Score (LAS)						
Algoritmo Proposto	73,51	**87,44**	**82,42**	**71,19**	**86,92**	**80,70**
Attardi	**74.44**	86.94	82.10	68.14	86.40	80.45

Tabella 5.4. Confronto tra i risultati degli esperimenti ottenuti usando l'algoritmo proposto e l'algoritmo di Attardi, presi da [20]. Per ogni lingua lo score maggiore è evidenziato in grassetto.

rispettivamente UAS e LAS, mentre l'Algoritmo di Attardi raggiunge un'accuratezza di (84.14, 79.75), dimostrando anche in questo modo la maggior potenza del primo.

L'oracolo dinamico sviluppato in questa Tesi, capace di utilizzare tutti i dati disponibili nella fase di addestramento, consente quindi all'Algoritmo Arc-Standard di raggiungere risultati su lingue non proiettive paragonabili, se non superiori, a quelli raggiunti da algoritmi in grado di produrre alberi non proiettivi. Gran parte della differenza nell'accuratezza tra le due tipologie di algoritmi era data, quindi, dalla capacità degli algoritmi non proiettivi di utilizzare completamente le informazioni contenute nel dataset, e non dal fatto che questi potessero costruire degli alberi sintattici corretti e fossero, dunque, più adatti all'analisi di lingue con alta percentuale di frasi non proiettive.

6. Conclusioni

In questa Tesi si è discusso di come consentire l'utilizzo di alberi alle dipendenze non proiettivi nell'addestramento di parser che possono produrre solo alberi proiettivi. Due metodi già presenti in letteratura, l'Algoritmo di Nivre-Nilsson e l'Algoritmo di Goldberg, i quali trasformano le frasi non proiettive in frasi proiettive in una fase di pre-processing, sono stati confrontati con un nuovo algoritmo che realizza un nuovo tipo di oracolo dinamico per l'Algoritmo Arc-Standard di complessità $O(n^5)$, in grado di elaborare le frasi non proiettive trasformandole direttamente durante l'esecuzione, senza bisogno di nessuna operazione preliminare. Si noti come questo lavoro sia anche il primo in cui i due algoritmi scelti come termini di paragone sono stati confrontati tra loro.

Si sono svolti esperimenti su sei lingue diverse, quattro delle quali con una grande percentuale di frasi non proiettive. Dai loro risultati è emerso come l'utilizzo delle informazioni contenute nelle frasi non proiettive provochi un incremento sensibile nell'accuratezza rispetto all'Algoritmo Arc-Standard, dimostrando come non fosse corretto trascurarle durante l'addestramento. Un aumento dell'accuratezza si nota anche per quelle lingue che non hanno un'alta percentuale di frasi proiettive. Confrontando in dettaglio le varie tecniche utilizzate, l'algoritmo proposto risulta di poco migliore rispetto all'Algoritmo di Nivre-Nilsson e decisamente migliore di quello di Goldberg, che si è dimostrato il peggiore anche nel confronto con quest'ultimo.

L'algoritmo proposto raggiunge, inoltre, un'accuratezza superiore a quella dell'Algoritmo di Attardi, in grado di produrre dipendenze non proiettive. Si prova in questo modo come buona parte del distacco tra gli algoritmo proiettivi e quelli non proiettivi fosse determinato dall'uso incompleto dei dati da parte dei primi, e non da un intrinseco vantaggio dei secondi. Bisogna considerare, comunque, come solo i secondi producano direttamente degli alberi alle dipendenze non proiettivi, mentre le tecniche analizzate in questa Tesi producono solo alberi alle dipendenze proiettivi. È comunque un fatto molto importante, in quanto rende possibile l'analisi di lingue fortemente non proiettive avendo sia un algoritmo robusto che un'accuratezza accettabile.

Il lavoro iniziato in questa Tesi può essere continuato seguendo strade diverse. Innanzitutto il parser realizzato può essere ulteriormente affinato, ad esempio cercando se sia possibile estrarre nuove features per ottenere una migliore accuratezza nella costruzione degli archi non proiettivi o indagando se sia possibile, per le frasi non proiettive, introdurre delle informazioni aggiuntive nell'albero alle dipendenze prodotto per ricavare la sua

struttura corretta, così come fatto nell'Algoritmo di Nivre-Nilsson. Un altro obiettivo potrebbe essere l'ottimizzazione dell'algoritmo proposto, con l'obiettivo di ridurre i tempi di addestramento del modello: si può infatti intervenire sia sull'implementazione attuale (è probabilmente possibile ridurre la complessità asintotica ad $O(n^3)$ adattando opportunamente l'algoritmo in [19]) sia verificando le prestazioni di una sua implementazione parallela. Sarebbe interessante, infine, realizzare degli oracoli dinamici analoghi a quello proposto in questa Tesi anche per altri algoritmi di parsing proiettivi, come l'Arc-Eager [23], per verificare quale raggiunga l'accuratezza più alta.

7. Bibliografia

[1] Daniel Jurafsky, James H. Martin. **Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition**. Prentice Hall PTR, 2006.

[2] Alan M. Turing. **Computing machinery and intelligence**. *Mind*. Volume 59, pagg. 433-460. 1950.

[3] Joseph Weizenbaum. **ELIZA – A computer program for the study of natural language communication between man and machine**. *Communications of the ACM.* Volume 9(1), pagg. 36–45. 1966.

[4] Sandra Kübler, Ryan McDonald, Joakim Nivre. **Dependency Parsing**. Morgan and Claypool, 2009.

[5] Lucien Tesnière. **Elements de syntaxe structurale**. Edizioni Klincksieck, 1959.

[6] Cristina Bosco, Simonetta Montemagni, Maria Simi. **Converting Italian Treebanks: Towards an Italian Stanford Dependency Treebank**. *7th Linguistic Annotation Workshop & Interoperability with Discourse*. 2013.

[7] Joakim Nivre. **Non-projective Dependency Parsing in Expected Linear Time**. *Proceedings of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. Vol. 1, pagg. 351-359, 2009.

[8] Taku Kudo, Yuji Matsumoto. **Japanese dependency analysis using cascaded chunking**. *Proceedings of the 6th conference on Natural language learning*. Volume 20, pagg. 1-7. Association for Computational Linguistics, 2002.

[9] Joakim Nivre. **Algorithms for deterministic incremental dependency parsing**. *Computational Linguistics*. Volume 34(4), pagg. 513-553. 2008.

[10] Hiroyasu Yamada, Yuji Matsumoto**. Statistical dependency analysis with support vector machines**. *Proceedings of the 8th International Workshop on Parsing Technologies*. Pagg. 195-206, 2003.

[11] Giuseppe Attardi. **Experiments with a multilanguage non-projective dependency parser**. *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*. Pagg. 166-170, 2006.

[12] Yue Zhang, Joakim Nivre. **Transition-based dependency parsing with rich nonlocal features**. *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Pagg. 188-193, 2011.

[13] Frank Rosenblatt. **The perceptron: A probabilistic model for information storage and organization in the brain**. *Psychological Review*. Volume 65, pagg. 386-408, 1958

[14] Hal Daumé III. **Practical Structured Learning Techniques for Natural Language Processing**. Tesi di dottorato, University of Southern California, Los Angeles. 2006.

[15] Yoav Goldberg, Joakim Nivre. **A dynamic oracle for arc-eager dependency parsing**. *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*. Pagg. 959-976, 2012.

[16] Yoav Goldberg, Francesco Sartorio, Giorgio Satta. **A Tabular Method for Dynamic Oracles in Transition-Based Parsing**. *Transactions of the Association for Computational Linguistics*. Pagg. 119-130, 2014.

[17] Joakim Nivre, Jens Nilsson. **Pseudo-Projective Dependency Parsing**. *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. Pagg. 99-106, 2005.

[18] Sylvain Kahane, Alexis Nasr, Owen Rambow. **Pseudoprojectivity: A polynomially parsable non-projective dependency grammar**. *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (ACL '98)*. Pagg. 646-652, 1998.

[19] Jason Eisner. **Bilexical grammars and a cubic-time probabilistic parser**. *Proceedings of the 4th International Workshop on Parsing Technologies*. Pagg. 54-65, 1997.

[20] Francesco Sartorio. **Improvements in Transition Based Systems for Dependency Parsing**. Tesi di dottorato, Università degli Studi di Padova, 2014.

[21] Sabine Buchholz, Erwin Marsi. **CoNLL-X shared task on multilingual dependency parsing**. *Proceedings of the Tenth Conference on Computational Natural Language Learning*. Pagg. 149-164, 2006

[22] Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, Deniz Yuret. **The CoNLL 2007 Shared Task on Dependency Parsing**. *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*. Pagg. 915-932, 2007.

[23] Joakim Nivre. **An efficient algorithm for projective dependency parsing**. *Proceedings of the 8th International Workshop on Parsing Technologies*. Pagg. 149–160, 2003.

Appendice A. Dettagli sugli Esperimenti

Questa Appendice è una panoramica più approfondita sugli esperimenti svolti rispetto a quanto già scritto nel Capitolo 5 ed è rivolta a coloro i quali volessero continuare o controllare il lavoro iniziato in questa Tesi oppure volessero svolgere delle ricerche analoghe partendo dallo stesso software utilizzato [20]. È divisa in due sezioni: nella prima viene descritto più in dettaglio il setup degli esperimenti, già accennato nella Sezione 5.3, mentre nella seconda sono riportate delle considerazioni sull'integrazione del codice scritto per questa tesi nel software esistente.

A.1. Setup Sperimentale

Gli esperimenti sono stati svolti sui dataset preparati per le shared task delle *Conference on Computational Natural Language Learning* (CoNLL) del 2006 [21] e del 2007 [22]. Tutti condividono un formato comune, divenuto uno standard di fatto per il parsing alle dipendenze. Ogni dataset è diviso in due file di testo, uno per il training set ed uno per il test set, in cui ogni frase è separata da una riga vuota ed è costituita da uno o più token, ognuno posto su una riga diversa. Ogni token è composto dai seguenti 10 campi, separati da una tabulazione:

1. ID, ossia un contatore del token che inizia da 1 per ogni nuova frase.
2. FORM, ossia la parola o il simbolo di punteggiatura costituente il singolo token.
3. LEMMA, ossia il lemma della parola presente in FORM, o un underscore se non disponibile.
4. CPOSTAG, ossia un *tag* grezzo rappresentante la categoria grammaticale (part-of-speech). L'insieme dei tag varia da linguaggio a linguaggio.
5. POSTAG, ossia un tag rappresentante la categoria grammaticale in modo più accurato di CPOSTAG, oppure uguale ad esso se non disponibile.
6. FEATS, ossia un insieme non ordinato di features sintattiche o morfologiche in cui gli elementi sono separati da una sbarra verticale, o un underscore se non disponibile.
7. HEAD, ossia la testa del token corrente espresso come valore del campo ID o come 0 se il nodo non dipende da nessun altro.
8. DEPREL, ossia il tipo di relazione con il nodo di cui la parola è dipendente. L'insieme dei tipi varia da linguaggio a linguaggio.
9. PHEAD e
10. PDEPREL, campi deprecati e contenenti un underscore.

Si noti, inoltre, come solo gli esperimenti finali, ossia quelli i cui risultati sono stati riportati in questa Tesi, sono stati condotti usando come test set quello predefinito. Durante lo sviluppo dell'algoritmo, infatti, si è partizionato il training set in due, ottenendo un training set di sviluppo, formato dall'80% delle frasi del training set originario, e un *development set*, composto dal restante 20%. In questo modo si evita di correggere l'algoritmo o di settare dei parametri (in questo caso solo il numero di iterazioni, ma ce ne potrebbero essere altri come la posizione del nodo -ROOT- nella frase o il numero massimo di errori nella fase di addestramento) in modo da ottenere prestazioni migliori sul test set, che viene quindi utilizzato solo per la verifica finale dei risultati ottenuti.

Un approccio migliore prevedrebbe di effettuare i test in questa facendo una *k-fold cross-validation*: in questo caso il training set viene partizionato in *k* parti uguali e i risultati si ottengono dalla media dei risultati di k test distinti, ognuno dei quali utilizza una parte diversa come development set e le $k-1$ rimanenti come training set di sviluppo.

Usare la k-fold cross-validation consente, infatti, di evitare il problema che viene risolto in fase sperimentale svolgendo più volte gli stessi test con un training set ordinato in modo sempre diverso. Infatti il percettrone, così come altre tecniche di apprendimento automatico, è sensibile all'ordine con cui sono forniti i dati in ingresso. Fare la media di test diversi consente di ottenere dei risultati più accurati, visto che la differenza tra due test con *seed* di randomizzazione diversi può raggiungere l'1%. Cinque seed diversi, così come fatto in questa Tesi, è un numero ragionevole per eliminare questo rumore.

Un altro valore che influisce sull'accuratezza finale è il numero di iterazioni con il quale viene addestrato il modello. Questo numero viene ricavato, dopo un'attenta analisi dell'output del parser, individuando un'iterazione nella quale l'accuratezza converge ad un valore stabile, ossia nella quale già da diverse iterazioni oscilla attorno ad un valore medio scostandosi al più dello 0.2-0.3%. La scelta di questo numero è importante: un numero troppo piccolo può, infatti, produrre dei modelli instabili e le cui accuratezze non sono significative mentre un numero troppo grande può produrre lo stesso effetto per via del fenomeno dell'*overfitting*, oltre che allungare inutilmente il tempo di addestramento. Si noti come questo numero potrebbe essere diverso da algoritmo ad algoritmo e da tecnica a tecnica. In questa Tesi si è scelto di svolgere 20 iterazioni per ogni test in quanto l'algoritmo usato era sempre lo stesso e non si notavano differenze significative nella convergenza delle accuratezze delle varie tecniche. Si noti come, nel caso di modelli per parser alle dipendenze, non dovrebbero mai essere necessarie più di una trentina di iterazioni per arrivare alla convergenza.

Come si è visto nella Sezione 3.4.1, l'insieme delle features estratte da ogni configurazione può influire sull'accuratezza dell'algoritmo di parsing nell'ordine dello 0.5%.

In questa tesi si sono usate le features elencate in [12], adattate per l'Algoritmo Arc-Standard, che erano già presenti nel software utilizzato. Non è detto, però, che l'aggiunta di qualche nuova features, soprattutto se pensata specificatamente per fornire informazioni aggiuntive nel caso di archi non proiettivi, non possa produrre un miglioramento dell'accuratezza.

Dal punto di vista pratico, infine, la maggior parte degli esperimenti sono stati condotti sul cluster del Dipartimento di Ingegneria dell'Informazione dell'Università degli Studi di Padova. Il suo utilizzo ha consentito lo svolgimento di 10 test contemporaneamente, rendendo quindi possibile lo realizzazione per tempo di tutte le analisi. Come esempio di uno dei casi peggiori, scegliendo cioè un training set grande e altamente non proiettivo, una singola iterazione dell'algoritmo proposto in questa Tesi sul dataset del ceco (25364 frasi) viene conclusa in circa 30 minuti su un sistema di potenza medio bassa (sistema operativo Linux, processore dual core 3.5 GHz, 8GB di RAM), mentre i tempi di esecuzione sul cluster sono maggiori per via della condivisione delle risorse con altri utenti. Non è da escludere, però, che un'implementazione parallela del software utilizzato abbinata ad un computer di fascia alta possa ridurre i tempi di esecuzione fino a rendere l'esecuzione in locale più veloce rispetto al cluster del Dipartimento.

A.2. Considerazioni sull'Implementazione

Il software realizzato dal dott. Sartorio è un parser alle dipendenze basato sulle transizioni (vedi Sezione 3.3), il cui modello è un percettrone (vedi Sezione 3.4.1), in particolare l'implementazione descritta in [14]. Come già detto, il codice è scritto quasi completamente in Python e in modo estremamente modulare, in modo da consentire facilmente lo svolgimento di esperimenti con nuovi algoritmi. Manca purtroppo di documentazione, ma una buona preparazione teorica (quanto esposto nel Capitolo 3 è più che sufficiente) e del linguaggio di programmazione usato sono più che adeguati per la comprensione del codice.

Più in dettaglio il codice sorgente è diviso nei seguenti file:

- `commons.py` e `costants.py` contengono alcuni funzioni di utilizzo generale.
- `corpus.py` contiene le classi necessarie per l'acquisizione dei dataset da file, per la struttura dati rappresentante una frase e per il salvataggio dell'output, ossia degli alberi alle dipendenze ricavate in fase di parsing.
- `featureTemplates.py` contiene una funzione per chiamare il metodo appropriato di estrazione delle features, contenuto in un file nella cartella `features/`.

- `model.py` è l'interfaccia con il codice del percettrone, scritto in linguaggio C, contenuto nella cartella `perceptron/`.
- `output.py` viene utilizzato per ricavare l'accuratezza del parsing.
- `parser.py` è il file principale da lanciare da riga di comando che elabora i parametri passati come input e che avvia il parser vero e proprio contenuto in `transitionBasedParser.py`.
- `parsingAlgorithmABC.py` contiene le *Abstract Base Class* comuni a tutti gli algoritmi di parsing basati su transizioni. Notare come sia necessario sovrascrivere alcune funzioni di queste classi per implementare un nuovo algoritmo.
- `transitionBasedParser.py` contiene l'implementazione dell'Algoritmo 3.1 per l'addestramento e del parser per la fase di test.

A questi file si devono poi aggiungere i file contenenti le implementazioni degli algoritmi. Per inserirne uno nuovo è necessario:

1. Implementare in modo appropriato le classi `Action` e `ConfigurationABC` contenute in `parsingAlgorithmABC.py`.
2. Implementare, per ogni oracolo desiderato, la classe `OracleABC`, sovrascrivendo solo il metodo `give_cost`. Ad un metodo di parsing posso essere associati più tipi di oracoli, così come visto nella Sezione 3.5.
3. Includere il nuovo algoritmo in `parser.py`, in modo che sia selezionabile da riga di comando e che vengano caricate le classi definite ai due punti precedenti.
4. Impostare il tipo di transizioni in `commons.py` e il file in cui sono contenute le specifiche delle features in `featureTemplates.py`.

Fino a quando si rimane dentro lo schema di parsing basato sulle transizioni definito nell'Algoritmo 3.1 non è necessario modificare nessuna altra parte del codice per aggiungere un nuovo algoritmo.

Si noti come la maggior parte delle funzionalità usate nei test svolti in questa Tesi fossero già implementate, con l'esclusione della possibilità di calcolare l'accuratezza in modo disgiunto tra archi proiettivi e non proiettivi e di limitare il numero massimo di errori compiuti durante l'addestramento del modello. Tra le funzionalità non utilizzate ci sono la possibilità di calcolare l'Exact Match (vedi Sezione 5.1), di scegliere la posizione del nodo -ROOT-, di implementare durante l'addestramento una modalità correct-and-go-on (vedi Sezione 3.4) dopo un numero fissato di iterazioni o con una certa probabilità per ogni singola transizione, di escludere la punteggiatura dal calcolo dell'accuratezza e di modificare l'insieme delle features scelte.