ANNO ACCADEMICO 2024/2025

Tecnologie del Linguaggio Naturale

WSD e Ngrams

Luca Barra

UNIVERSITÀ
DI TORINO

DIPARTIMENTO DI INFORMATICA

1 Word Sense Disambiguation on SemCor

Il **Word Sense Disambiguation** (WSD) è il task che consiste nell'identificazione di quale senso di una determinata parola sia inteso in una frase o in un contesto.

1.1 Algoritmo di Lesk

L'**algoritmo di Lesk** è uno dei più semplici approcci al WSD task. Si base sulla premessa che parole utilizzate in un determinato contesto probabilmente hanno un senso comune L'algoritmo compara le definizioni della parola che si vuole disambiguare con le parole circostanti in modo da trovare il senso più appropriato. Per questa esercitazione si è scelto di implementatare una versione semplificata dell'algoritmo di Leskin cui il significato di ogni parola è determinato dal senso che ha il maggior overlapping tra la definizione (utilizzando un *thesaurus*, in questo caso **WordNet**) e il suo contesto. Questo approccio considerà ogni parola individualmente indipendentemente dai sensi delle altre parole nel contesto. Tuttavia l'algoritmo di Lesk ha il difetto di dipendere dalla definzione esatta della parola, per cui l'assenza di determinate parole può cambiarne completamente il risultato.

Listing 1.1: Versione semplificata dell'algoritmo di Lesk

```
SynsetPtr simple_lesk(semcor_word word, semcor_phrase context) {
  int pos = semcor_to_wordnet_pos(word.get_pos());
  if (pos == -1)
    return nullptr;

  string search_word =
      word.get_lemma().empty() ? word.get_text() : word.get_lemma();

  SynsetPtr synset = get_synset(search_word, pos);
  if (!synset)
    return nullptr;

  SynsetPtr best_sense = synset;
  int max_overlap = 0;

  SynsetPtr current = synset;
  while (current) {
    string signature;
    if (current->defn)
      signature += string(current->defn);
    int overlap = compute_overlap(signature, context);
```

```
    if (overlap > max_overlap) {
      max_overlap = overlap;
      best_sense = current;
    }

    current = current->nextss;
  }

  return best_sense;
}
```

1.1.1 API di WordNet

Come detto nella sezione precedente si è scelto di utilizzare WordNet come thesaurus. In questa sezione sono elencate le specifiche API C utilizzate:

- `findtheinfo_ds`: restituisce il risultato come una linked list.
- `free_syns`: libera la memoria allocata da `findtheinfo_ds`.
- `wninit`: per aprire il database.

1.2 SemCor

Per testare l'algoritmo si è usato il corpus *SemCor*. Il parsing dei file xml è stato fatto mediante la libreria `pugixml`.

I file avevano la seguente struttura:

- $< p >$
  - $< s >$
    - wf

I valori considerati sono stati:

- pos: il pos tag della parola.
- cmd: valore booleano che indica se la parola abbia un senso (e.g. le stopwork non lo hanno).
- lemma: il lemma della parola.
- wnsn: la chiave del senso della parola in WordNet.

Listing 1.2: Validazione della predizione dell'algoritmo di Lesk tramite confronto con gold sense.

```
int gold_num = stoi(gold_key);
int current_num = 1;

SynsetPtr curr_sense = get_synset(predicted->words[0], pos);
SynsetPtr full_list = curr_sense;
SynsetPtr gold_sense = nullptr;

while (curr_sense) {
  if (current_num == gold_num) {
    gold_sense = curr_sense;
    break;
```

```
  }
curr_sense = curr_sense->nextss;
current_num++;
}
```

1.2.1 Risultati

Il programma è stato eseguito 10 volte e ogni volta si è andati a disambiguare una parola per frase (50 frasi per esecuzione). La media dell'accuratezza è intorno al 60% che è sensato come risultato considerando che il WSD è un task *difficile* e che l'algoritmo di Lesk è molto semplice.

Numero Esecuzione	**Accuratezza (%)**
1	62
2	58
3	50
4	72
5	54
6	50
7	68
8	52
9	64
10	70

Table 1.1: Accuratezza per ciascuna esecuzione

2

Modeling Social and Literary Language with N-grams

Un *n-gram* è una sequenza di n simboli adiacenti in un determinato ordine. In particolare in questa esercitazione ci si è concentrati su n-grams di parole:

- **Bi-grams:** formati da 2 parole.
- **Tri-grams:** formati da 3 parole.

Inoltre si è scelto, per rendere le cose più interessanti, di implementare un rudimentale modello di **temperatura**. La temperatura è un parametro per "controllare" la *randomness* di un modello: una temperatura più bassa rende il modello più conservativo, una temperatura più alta lo rende più creativo.

2.1 Bi-grams

Per estrarre i Bi-grams da un input si è utilizzata una *finestra* di due parole che scorrendo genera un HashMap la cui chiave è la prima parola e la cui entry è a sua volta una HashMap in cui vengono contate le occorrenze di quella coppia di parole.

Listing 2.1: Estrazione dei bi-grams

```
pub fn generate_bigrams(tokens: Vec<String>) -> HashMap<String, HashMap<String, usize>> {
    let mut bigrams: HashMap<String, HashMap<String, usize>> = HashMap::new();

    for window in tokens.windows(2) {
        if let [word1, word2] = window {
            let entry = bigrams.entry(word1.clone()).or_default();
            *entry.entry(word2.clone()).or_insert(0) += 1;
        }
    }

    bigrams
}
```

Dopo che i bi-grams sono stati estratti si può generare il testo. Per generare del testo utilizzando i bi-grams è necessario fornire una parola iniziale, la lunghezza del testo che si vuole generare e un valore per la temperatura. La funzione utilizza un ciclo per generare le parole successive:

1. Per prima cosa colleziona tutte le possibili parole successive.
2. Poi applica la temperatura a ogni parola raccolta creando dei **pesi**.

3. La parola viene generata casualmente. Ovviamente questo è influenzato dalla probabilità che una parola segua un'altra (e.g. una parola con probabilità del 90% avrà più probabilità di essere scelta rispetto a una con solo 5%).

Listing 2.2: Generazione dei bi-grams

```rust
pub fn generate_bigrams_text(
    bigrams: &HashMap<String, HashMap<String, usize>>,
    start_word: &str,
    length: usize,
    temperature: f64,
) -> String {
    let mut rng = rng();
    let mut result = vec![start_word.to_string()];
    let mut current_word = start_word.to_string();

    for _ in 0..length - 1 {
        if let Some(next_words_map) = bigrams.get(&current_word) {
            let words: Vec<_> = next_words_map.keys().cloned().collect();
            let weights: Vec<f64> = apply_temperature(next_words_map, temperature);
            let dist = WeightedIndex::new(&weights).unwrap();
            let next_word = words[dist.sample(&mut rng)].clone();
            result.push(next_word.clone());
            current_word = next_word;
        } else {
            break;
        }
    }

    format_text(&result)
}
```

2.1.1 Letteratura

Utilizzando il testo di Moby dick si sono trovati i seguenti risultati.

Input	Temperatura	Lunghezza	Testo
whale	0.5	30	whale, and if you, and the first time. but the old man, and in it, the whale, and all the whale.
whale	1.0	30	whale. they hint to the clear from all heeding what a sea; iron in china from the body was cast loose - fish yet his strength, you
whale	1.5	30	whale thus trampled with halting, the night going further," muttered ahab gives very heedfully as upon a laugh out for unless the chap with yourself to such

Table 2.1: Testo generato dai bi-grams per diversi valori di temperatura con input "whale".

2.1.2 Twitter

2.2 Tri-grams

Per estrarre i Bi-grams da un input si è utilizzata una *finestra* di due parole che scorrendo genera un HashMap. Di per sé il procedimento è lo stesso utilizzato per i bi-grams, ma questa volta la chiave sono due parole invece che una.

Listing 2.3: Estrazione dei tri-grams

```rust
pub fn generate_trigrams(tokens: Vec<String>) -> HashMap<String, HashMap<String, usize>> {
    let mut trigrams: HashMap<String, HashMap<String, usize>> = HashMap::new();

    for window in tokens.windows(3) {
        if let [word1, word2, word3] = window {
            let key = format!("{} {}", word1, word2);
            let entry = trigrams.entry(key).or_default();
            *entry.entry(word3.clone()).or_insert(0) += 1;
        }
    }

    trigrams
}
```

Dopo che i tri-grams sono stati estratti si può generare il testo. Per generare del testo utilizzando i tri-grams è necessario fornire una coppia di parole iniziale, la lunghezza del testo che si vuole generare e un valore per la temperatura. La funzione utilizza lo stesso procedimento utilizzato per i bi-grams per generare parole successive.

Listing 2.4: Generazione dei tri-grams

```rust
pub fn generate_trigram_text(
    trigrams: &HashMap<String, HashMap<String, usize>>,
    start_phrase: &str,
    length: usize,
    temperature: f64,
) -> String {
    let mut rng = rng();
    let mut result = vec![start_phrase.to_string()];
    let mut current_phrase = start_phrase.to_string();

    for _ in 0..length - 1 {
        if let Some(next_words_map) = trigrams.get(&current_phrase) {
            let words: Vec<_> = next_words_map.keys().cloned().collect();
            let weights: Vec<f64> = apply_temperature(next_words_map, temperature);
            let dist = WeightedIndex::new(&weights).unwrap();
            let next_word = words[dist.sample(&mut rng)].clone();
            result.push(next_word.clone());

            current_phrase = format!(
                "{} {}",
                current_phrase
                    .split_whitespace()
                    .skip(1)
                    .collect::<Vec<&str>>()
                    .join(" "),
                next_word
            );
        } else {
            break;
        }
    }

    format_text(&result)
  }
```

2.2.1 Letteratura

Utilizzando il testo di Moby Dick si sono trovati i seguenti risultati.

Input	Temperatura	Lunghezza	Testo
call me	0.5	25	call me ishmael. tell' em. they were placed in the fishery, yet will i thee, thou grinning whale!" cried
call me	1.0	25	all me a dismal gloom, now wildly elbowed, fifty years ago. poor lazarus there, ahoy! have a purse!" screamed
call me	1.5	25	call me an immortal by brevet. yes, yes, it seems to be served. they look." whereupon planting his feet,

Table 2.2: Testo generato dai tri-grams per diversi valori di temperatura con input "call me".

2.2.2 Twitter