Natural Language Processing

- Intro
- Syntax
- Semantics
- Pragmatics

Review

- Intelligence involves:
  - communicating
  - recognizing what we see around us
  - navigating
  - Using our knowledge to draw new conclusions and solve problems.
- So far looked at last of these:
  - knowledge representation and problem solving.
- Time to turn back to first of these.

Natural Language Processing

- Natural language processing concerns:
  - Understanding or generating spoken and written text.
- Examples of NL tasks:
  - understanding questions/instructions posed in natural language “e.g., switch off the computer”.
    - May be spoken or typed
  - Generating natural language responses
  - “skimming” written texts to find information, or to create a summary.
  - .Generating coherent NL documents from some structured information.

Natural Language Processing

- So NL Processing covers
  - understanding and generation
  - short sentences and larger texts
  - spoken and written language.
- We will focus mainly on understanding short written sentences.

Knowledge Required

What knowledge do we (as humans) use to make sense of language?

- Knowledge of how words sound..
  - “cat” == “c” “a” “t”
- Knowledge of how words can be composed into sentences (the grammar).
  - The can sat on the mat OK
  - sat mat can on the NO
- Knowledge of people, events, the world, types of text.
  - Recognizing adverts for what they are.
  - Understanding indirect requests “I don’t quite understand this” as request for help.

Stages of processing

To deal with complexity, can process language in series of stages:

- speech recognition
  - using knowledge of how sounds make up words.
- syntactic analysis
  - using grammar of language to get at sentence structure.
- semantic analysis
  - mapping this to meaning
- pragmatics
  - using world knowledge and context to fill in aspects of meaning.

Syntactic Analysis

- We will focus on syntax.
- How do we *recognize* that a sentence is grammatically correct?
  - The cat sat on the mat. OK
  - On the the sat cat mat. NO.
- More importantly, how to we use knowledge of language structures to assign structure to a sentence (helping in deriving its meaning).
  - *(The large green cat) (sat on (the small mat))*
  - Bracketed bits are meaningful subparts.

Grammars

- Grammars define the legal structures of a language.
- We “parse” a sentence using a grammar to:
  - Determine whether it is grammatical.
  - Assign some useful structure/grouping to the sentence.
- We want the words denoting an object to be grouped together, and words denoting actions to be grouped together.

Syntactic Categories

- Grammars based on each word belonging to a particular category:
  - nouns
  - verbs
  - adjectives
  - adverbs
  - articles/determiners
- The black cat jumps quickly
- article adjective noun verb adverb

Larger groupings

- Noun phrase: sequence of words denoting an object. e.g.,
  - the black cat.
- Verb phrase: sequence of words denoting an action. e.g.,
  - jumps quickly
  - runs after the small dog
  - kicks the small boy with the funny teeth
- Note that verb phrases may contain noun phrases.

Simple NL Grammar

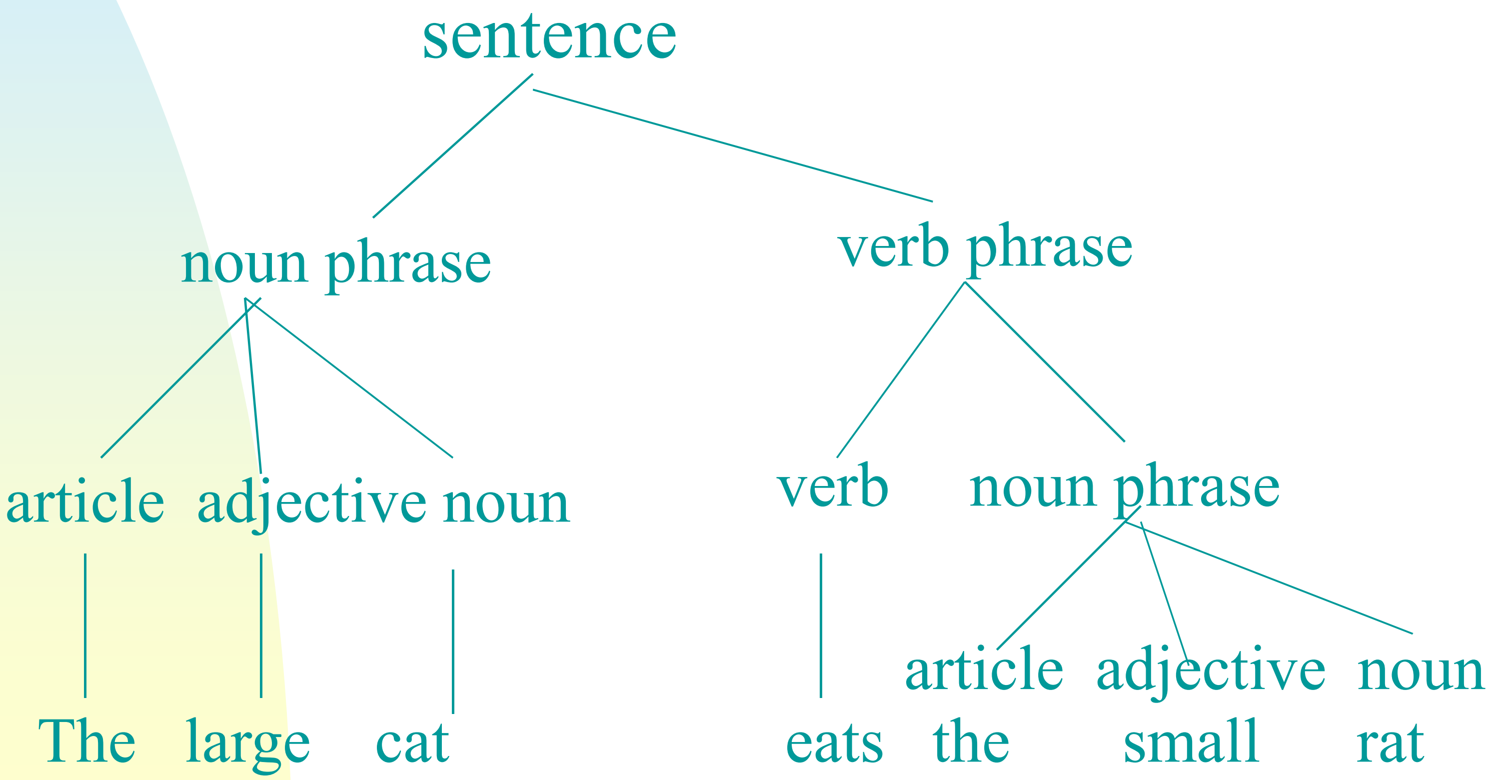
- We can write a simple NL grammar using *phrase structure* rules such as the following:
  - `sentence --> nounPhrase, verbPhrase.`
  - `nounPhrase --> article, adjective, noun.`
  - `verbPhrase --> verb, nounPhrase.`
- This means:
  - a sentence can consist of a noun phrase followed by a verb phrase.
  - A noun phrase can consist of an article, followed by an adjective, followed by a noun.
- Rules define constituent structure.

Parsing

- Using these rules we can determine whether a sentence is legal, and obtain its structure.
- “The large cat eats the small rat”
- This consists of:
  - Noun Phrase: The large cat
  - Verb Phrase: eats the small rat
- The verb phrase in turn consists of:
  - verb: eats
  - Noun Phrase: the small rat

Parse Tree

This structure can be represented as a tree:



Parse Tree

- This tree structure gives you groupings of words. (e.g., the small cat).
- These are meaningful groupings - considering these together helps in working out what the sentence means.

Parsing

- Basic approach is based on *rewriting*.
- To parse a sentence you must be able to “rewrite” the “start” symbol (in this case *sentence*) to the sequence of syntactic categories corresponding to the sentence.
- You can rewrite a symbol using one of the grammar rules if it corresponds to the LHS of a rule. You then just replace it with the symbols in LHS. e.g.,
  - sentence
  - nounPhrase verbPhrase
  - article adjective noun verbPhrase

  etc

A little more on grammars

- Example grammar will ONLY parse sentences of a very restricted form.
- What about:
  - John jumps
  - The man jumps
  - John jumps in the pond
- We need to add extra rules to cover some of these cases

Extended Grammar

  - `sentence --> nounPhrase, verbPhrase.`
  - `nounPhrase --> article, adjective, noun.`
  - `nounPhrase --> article, noun.`
  - `nounPhrase --> properName.`
  - `verbPhrase --> verb, nounPhrase.`
  - `verbPhrase --> verb.`
- (Think how you might handle “in the pond”..)
- Grammar now parses:
  - John jumps the pond.
- And fails to parse ungrammatical ones like:
  - jumps pond John the

NL Grammars

- A good NL grammar should:
  - cover a reasonable subset of natural language.
  - Avoid parsing ungrammatical sentences
    - (or at least, ones that are viewed as not acceptable in the target application).
  - Assign plausible structures to the sentence, where meaningful bits of the sentence are grouped together.
- But.. The role is NOT to check that a sentence is grammatical. By excluding dodgy sentences the grammar is more likely to get the right structure of a sentence.

Prolog's Grammar Notation

- Prolog has a built in grammar notation (definite clause grammars).
- Prolog's built in search mechanism enables sentences to be easily parsed.
- Example:
  - ```
    sentence --> np, vp.
    np --> article, noun.
    vp --> verb.
    article --> [the].
    noun --> [cat].
    verb --> [jumps].
    ```
- Note how dictionary entries (words) are entered just as another rule.
    ```

Example 2

```
sentence --> np, vp.
np --> article, noun.
vp --> verb.
vp --> verb, np.
article --> [the].
article --> [a].
noun --> [cat].
verb --> [jumps].
verb --> [likes].
```

Behind the grammar notation

- Prolog in fact converts these rules into “ordinary” prolog when it reads them in.
  - a --> b, c.
- Converted to:
  - `a(Words, Rest) :- b(Words, Temp), c(Temp, Rest).`
  - Words and Rest are lists. So we have:
    - a list of words can be parsed as constituent a, with Rest words left over.
    - This is ok if the list can be parsed as a “b”, with Temp words left over, and Temp can be parsed as a “c” with Rest left over.
- You may see this underlying form when tracing.

Parsing

- You can parse a sentence using a special "phrase" predicate:
  - ```
    ?- phrase(sentence, [the, cat, jumps]).
    Yes
    ```
- You can even use it to generate all "grammatical" sentences:
  - ```
    ?- phrase(sentence, X).
    X = [the, cat, jumps] ;
    X = [the, cat, jumps, the cat] ;
    ```

Returning the parse tree

- Just knowing whether a sentence parses isn't terribly useful.
- Also want to know the structure of the sentence.
- Simplest way for small examples is to add another argument to all rules that will contain a partly filled in structured object:
  - sentence(s(NP, VP)) --> np(NP), vp(VP).
  - noun(noun(cat)) --> [cat].
- End up with
  - ```
    ?- phrase(sentence(Tree), [the, cat,.. ]).
    Tree = s(np(art(the), noun(cat)),
             vp(verb(jumps)))
    ```
    ```

NL Generation: Templates

- To generate NL from some prolog term, easiest (but inflexible) is to use templates and matching.
- First arg is prolog term, second is list of words.
- ```
  template(carries(P, Obj,From, To),
      [P, carries, the, Obj, from, the, From,
  to, the, To]).
  ?- template(carries(john, wine, kitchen,
  study), Sent).
  Sent = [john, carries, the, wine, from,
  the, kitchen, to, the, study]
  ```
- Can then write out the list of words nicely..
  ```

Other bits of Prolog
1. Commenting

- In prolog can enter comments in two ways:
  - % anything following a percent, on that line
  - /* anything between these delimiters */
- It is conventional to have:
  - comments at start of file (author,purpose, date etc).
  - Comment before each predicate. E.g.,
    - % member(Item, List): checks whether Item
      % is a member of List
  - We also sometimes refer to e.g., member/2 to mean the member predicate with two args.

2. Disjunctions

- In prolog you CAN have disjunctions in body of rule, signalled by “;”.
  - `a(X) :- b(X) ; c(X).`
- But this is equivalent to having two rules.
  - `a(X) :- b(X).`
  - `a(X) :- c(X).`
- Logical equivalence can be demonstrated.
  - $\forall X\ (b(X) \vee c(X)) \Rightarrow a(X)$ *equiv to*
  - $(\forall X\ b(X) \Rightarrow a(X)) \wedge (\forall X\ c(X) \Rightarrow a(X))$
- (But we won’t bother).
- Normally we use two rules.

3. Recursion

Common in Prolog to use recursive rules, where rule “calls” itself. (term in head occurs in body).

```
ancestor(Person, Someone) :-
    parent(Person, Someone).
ancestor(Person, Someone) :-
    parent(Person, Parent),
    ancestor(Parent, Someone).
```

Base case

Recursive case

Someone is your ancestor if they are your parent.
Someone is your ancestor if Parent is your parent, and Parent has Someone as their ancestor.

Execution of recursive rules.

- Consider:
  - parent(joe,jim).
  - parent(jim, fred).
- Goal:
  - ancestor(joe, fred).
- Execution:
  - `Goal matches head of first rule.`
  - `Tries to prove parent(joe, fred). FAILS.`
  - `Backtracks and tries second rule.`
  - `Tries to prove: parent(joe, Parent).`
  - `Succeeds with Parent=jim.`
  - `Tries to prove: ancestor(jim, fred).`
  - `Goal matches head of first rule.`
  - `Tries to prove parent(jim, fred). SUCCEEDS.`

Other Examples of Recursive Rules

- `connected(X, Y) :- touches(X, Y).`
- `connected(X, Z) :- touches(X, Y), connected(Y, Z).`

- `inheritsFrom(X, Y) :- subclass(X, Y).`
- ```
  inheritsFrom(X, Y) :-
        subclass(X, Z), inheritsFrom(Z, Y).
  ```
  ```

Left recursion and infinite loops

- Problems can occur if you write recursive rules where the head of the rule is repeated on the LEFT hand side of the rule body, e.g.,
  - above(X, Y) :- above(X, Z), above(Z, Y).

    Left recursive call

  - above(prolog_book, desk).
  - above(ai_notes, prolog_book).
  - ?- above(ai_notes, desk).
    Yes
    ?- above(desk, ai_notes).
    (doesn't terminate)

4. Arithmetic and Operators

- Prolog has the usual range of Arithmetic Operators (+, -, =, * ..)
- But unless you force it to, it won't evaluate arithmetic expressions. They will simply be prolog structures to be matched.
- ```
  ?-  1+1 = 2.
  No
  ```
- ```
  ?- 1+1 = 1+X.
  X = 1
  ```
- ```
  data(tweety, [legs=2, feathers=yes]).
  ?- data(tweety, Data), member(legs=X, Data).
  X = 2
  ```
  - (Note how we can call several goals from Prolog prompt)
  ```

Arithmetic

- We can force Prolog to evaluate things using the special operator “is”.
  - `?- X is 1 + 1.`
    `X = 2.`
  - `?- Y=1, X is Y + 1.`
    `X is 2.`
- We could therefore write a limbs-counting predicate:
-
```
no_limbs(animal, N) :-
    no_arms(animal, Narms),
    no_legs(animal, Nlegs),
    N is Narms + Nlegs.
```

5. Input and Output

- Prolog has various input and output predicates.
  - write/1 - writes out a prolog term.
  - read/2 - reads in a prolog term.
  - put/1 - writes one character.
  - get/1 - reads one character.
- Example:
  - sayhello :- write(‘Hello’), nl.
- Use with care as weird things happen on backtracking.

6. The CUT

- Serious programs often need to control the backtracking.
- The “!” (cut) symbol is used to do this. We will aim to avoid it, but as an example:
  - a(A) :- b(A), !, c(A).
- Prevents backtracking past the cut.
- Commits to particular solution path.

7. Consulting Programs

- So far used
  - consult(‘filename.pl’). OR
  - consult(filename).
  - [‘filename.pl’]. OR
  - [filename].
- Can also (equivalently) use:
  - [‘filename.pl’]. OR
  - [filename].
    - (Assumes .pl if not mentioned).

8. Negation

- What if you want to say that something is true if something is can NOT be proved.
- Logically we would have:
  □ X □ tall(X) □ short(X)
- In prolog we use the symbol \+ and have rules such as:
- short(X) :- \+ tall(X).
- Some prologs allow the word “not”:
  short(X) :- not tall(X).

Exercises

- Work out what order solutions would be returned given the following program and query:
  - `onTop(prolog_book, desk).`
  - `onTop(ai_notes, prolog_book).`
  - `onTop(timetable, ai_notes).`
  - `onTop(ai_book, desk).`
  - `above(X, Y) :- onTop(X, Y).`
  - `above(X, Y) :- onTop(X, Z), above(Z, Y).`
- ?- above(Object, desk).

Exercises

- What are results of following matches:
  - `[X, Y] = [1, 2].`
  - `[a(X), b(Y)] = [a(1), b(2)].`
  - `[X | Y] = [1, 2].`
  - `[X, Y | Z ] = [1, 2].`
  - `[X, 1] = [2, Y].`
  - `[a | Y] = [[Y], b].`
- How would you use the “member” predicate to find out if there were any items of the form a(X) in a given list?

Summary

- Natural Language Processing covers understanding and generating spoken and written language, from sentences to large texts.
- Focus on understanding sentences. First step is to parse sentence to derive structure.
- Use grammar rules which define constituency structure of language.
- Parse gives tree structure which shows how words are grouped together.

Summary

- Have briefly covered:
  - Grammars in Prolog -
    - Prolog’s search method enables simple parsing.
  - Other aspects of Prolog -
    - but many more..