**Natural Language Processing (NLP)**

**Session 6**

1. **Syntactic Ambiguity**:
   1. Generative Grammars: John saw a man with binoculars.
      1. Meaning 1:
         1. S - John VP
         2. VP - tV NP
         3. tV - saw
         4. NP - a man with binoculars
      2. Meaning 2:
         1. S - John VP
         2. VP - tV NP
         3. tV - saw
         4. NP - a man
         5. PP - with binoculars
      3. How can we deal with the syntactic (and therefore semantic/meaning) ambiguity of the parse of ‘john saw a man with binoculars’? We need to assign more than one possible structure!
   2. Syntactic Parsing:
      1. Parsing: The task of assigning a syntactic structure to a sentence.
      2. Applications in NLP
         1. Grammar checking in word processing tools.
         2. Semantic analysis, which has applications to
            1. Question answering
            2. Information extraction
            3. Dialogue systems (intent recognition)
   3. Challenges to Parsing:
      1. Ambiguity
         1. Two kinds:
            1. attachment ambiguity: a constituent can be attached to the parse tree at more than one place.
            2. coordination ambiguity: different sets of phrase can be conjoined by a conjunction, e.g. and/or
      2. Example: “I shot an elephant in my pyjamas.”
         1. “in my pyjamas” can be
            1. part of a NP headed by “an elephant” (the elephant was in my pyjamas!) or
            2. part of a VP headed by “shot”. (I shot the elephant whilst I was wearing my pyjamas)
      3. Attachment Ambiguity:
      4. A tree with a root node (S), then expanding to 2 nodes in the next level (NP and VP), then 1 node for the left node (Pronoun) and two nodes for the right node (Verb and NP). The tree grows asymmetrically to depth 6 in the same manner (one node on the left, two on the right).  
           
         I shot an elephant in my pyjamas.  
         An example of PP-attachment ambiguity.
      5. We saw the Eiffel Tower flying to Paris”
         1. “flying to Paris” is an example of a VP-attachment ambiguity, as it can have:
         2. have Eiffel Tower as subject (the Eiffel Tower flies!) or:
         3. modify the verb “saw” (we saw the Eiffel Tower while we were flying to Paris)   
              
            It does not always have to “make sense” semantically (for any type of ambiguity).
      6. Coordination Ambiguity:  
           
         “Old men and women dance”.
         1. (old men) and women dance.
         2. (old(men and women)) dance.
      7. The combination of these two ambiguities and the fact that the options do not have to make sense semantically, gives rise to many options.
2. **Parsers**:
   1. Different syntactic structures mean different meanings/semantics!
   2. When developing parsers, we must have the ambiguity challenge in mind. We want a parser that generates ALL possible parses.
   3. CFG Parsers
      1. Paradigm: parsing as search
      2. Searching through the space of possible parse trees to find the correct one: one whose root is S and whose leaves are exactly the words in the input sentence.
      3. Classic search algorithms:
         1. Top-Down
         2. Bottom-Up
         3. Dynamic Programming
   4. Top-Down:
      1. Given: a grammar and an input sentence
      2. Start: the root of your parse tree: S
      3. Continue: find all trees that can start with S
         1. Method: look for rules with S on their left-hand side
         2. Repeat for each child
         3. Stop: when the children are exactly the input words  
              
            There is a tree example but here which would take far too long to put here considering there are 140 slides. Please refer to it from the lecture PDF.
   5. Bottom-up:
      1. Start: words of the sentence we are parsing
      2. Continue: find all trees that can start with words
         1. Method: look for rules with words on their right-hand side
         2. Repeat for each child
         3. Stop: a tree with root S  
              
            There is a tree example but here which would take far too long to put here considering there are 140 slides. Please refer to it from the lecture PDF.
   6. Which Method?
      1. Each has their own advantages and disadvantages
      2. The Top-Down will never waste time with trees that cannot result in S.
      3. The Bottom-Up will not waste time with trees that cannot end in the words of input.
3. **CKY Algorithm**:
   1. CKY (or CYK) algorithm was introduced by different people. This version is due to:
      1. Cocke
      2. Kasami
      3. Younger
   2. It is also called “chart parsing”.
   3. It stores constituents and subtrees as it finds them, so they do not need to be reconstructed again.
4. **Chomsky Normal Form**:
   1. To use CKY, one has to work with grammars that are in Chomsky Normal Form (CNF). This is when each rule expands to either of the following forms:
      1. Only: two non-terminals
      2. one single: single terminal
   2. If we work with these grammars have we lost expressive power? Nothing is lost!
      1. Theorem: A CFG and its CNF accept the same language.

|  |  |
| --- | --- |
| **L\_1 grammar** | **L\_1 in CNF** |
| S -> Aux NP VP | S -> X1 VP X1 -> Aux NP S -> X2 PP |
| VP -> NP PP | VP -> X2 PP X2 -> Verb NP |

* 1. Why CNF?
     1. The parse trees are now all binary!
     2. So they can be denoted by 2 dimensional matrices.
        1. More specifically, the upper triangle of an (n+1)\*(n+1) matrix.
        2. Where n is the number of words in the input sentence.

1. **Parsing Algorithm: CKY**:  
     
   How do we fill in the matrix? 1- Index your input sentence by inserting a number before and after each word:  
     
   0 Book 1 the 2 flight 3 through 4 Houston 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Book | The | Flight | through | Houston |
| S, VP, Verb, Nominal, Noun  [0,1] | [0,2] | S, VP, X2   [0,3] | [0,4] | S, VP, X2    [0,5] |
|  | Det  [1,2] | NP  [1,2] | [1,4] | NP  [1,5] |
|  |  | Nominal, Noun  [2,3] | [2,4] | Nominal  [2,5] |
|  |  |  | Prep  [3,4] | PP  [3,5] |
|  |  |  |  | NP, Proper-Noun  [4,5] |

The cell representing the entire input is [0, n].  
  
The main diagonal has the constituencies of the words of the sentence:   
[0,1]Book, [1,2]the, [2,3]flight, [3,4]through, [4,5]Houston.  
  
The second diagonals cover constituencies of combinations of two words:   
[0,2]Book the, [1,3]the flight, [2,4]flight through, [3,5] through Houston.  
  
The third diagonals cover constituencies of combinations of three words:   
[0,3]Book the flight, [1,4]the flight through, [2,5]flight through Houston,  
  
The last diagonal has the constituency of the full sentence:   
[0,5] Book the flight through Houston

* + 1. The matrix is filled in a bottom-up fashion.
       1. fill the matrix a column at a time from the left.
       2. fill each column from bottom to top.
    2. Why?
       1. At each cell, we have all the info we need.
       2. This is how incremental, word-by-word parsing (which is what humans are supposed to do) works.
       3. Each cell (i,j) of matrix is filled with a set of non terminals, representing all constituents from position i to j.
       4. Consider the cell (0,1). Fill it with the non terminals that represent the constituency of the word from position 0 to 1, i.e. “Book”.
       5. “Book” can be an imperative sentence S, it can be a VP (book a seat), or just a Verb (book from home). It can also be part of a group of nouns (book worm)- so a Nominal, or only one Noun.
    3. We have the constituents, but haven’t recorded the structural relation they are in to each other as we parse.   
         
       We need to record these to get the different parses/syntactic trees.
    4. After filling the table, start with the S cell of the table, cell (0,n). In the presence of ambiguity, we want to return all possible parses of S.   
         
       Make each non-terminal point to cells from which it was derived in the whole matrix.   
         
       Now all parses are returned by choosing the S from cell [0,n] and retrieving all of its component constituents from the matrix, by following the pointers.
  1. Examples:
     1. [3,5] points to [3,4] and [4,5].   
          
        This means that the PP in [3,5], i.e. “through Houston” can be generated by Prep NP or Prep Proper-Noun

In this case, it is Prep Proper-Noun.

* + 1. [2,5] points to [2,3] and [3,5].   
         
       This means that the Nominal in [2,5], i.e. “flight through Houston” can be generated by Nominal PP or Noun PP   
         
       In this case, it is Noun PP
    2. and so on … in particular …   
         
       The S1 in [0,5] points to Verb in [0,1] and NP in [1,5], which gives the parse Verb NP (the correct one).   
         
       Book, which flight? the flight that goes through Houston
    3. The S2 in [0,5] points to VP in [0,3] and PP in [3,5], which gives the parse VP PP (which is slightly wrong).   
         
       Book the flight, how? through Houston.
    4. The S3 in [0,5] points to the X2 (Verb NP) in [0,3] and the PP in [3,5]. This gives the parse Verb NP PP (also slightly wrong).   
         
       Book the flight. How? through Houston, e.g. while passing through Houston, etc.

1. **Probabilistic CFGs: parsing and learning:**
   1. The idea behind any statistical method:
      1. Sometimes we cannot have full knowledge about something.
      2. With enough knowledge, we can figure out its probability.
      3. This is better than saying nothing about it.
      4. In the case of parsing:
         1. We want to account for syntactic ambiguity in a weighted way - i.e. to assign different probabilities to different possible parse trees, rather than just pick one tree which fits.
         2. Getting likelihood measures of grammaticality rather than just grammatical vs ungrammatical is more robust (like language modelling).
2. **PCFG:**
   1. Probabilistic Context Free Grammars (PCFG’s) were first introduced by Booth in 1969.
   2. A tuple with four parameters: (N, Sigma, R, S)
      1. N – a set of non-terminal symbols
      2. Sigma – a set of terminal symbols (disjoint from N)
      3. S – a designated start symbol
      4. R – a set of production rules of the form  
           
         A -> beta[p]  
           
         for A and beta as in a CFG: where p is a real value indicating the probability of being used given A.
      5. A PCFG is obtained from a CFG by augmenting each rule with a probability p.
      6. What is this probability: A -> beta[p].
      7. The probability that the given non-terminal A will be expanded to the string beta.
      8. In other words, the conditional probability of the string beta given the non-terminal A.
      9. Other notations:
         1. P(A -> beta)
         2. P(A -> beta given A)
         3. P(RHS given LHS)
      10. All possible expansions of a non-terminal add up to 1:  
            
          Sigma\_{beta} P(A -> beta) = 1
3. **Computations:**
   1. A PCFG assigns a probability to each parse tree T of a sentence S. The tree with the higher probability is said to be the more likely one.
   2. The probability of a tree T of a sentence S is computed as follows:  
        
      P(T intersect S) = product sum of P(RHS given LHS)
   3. This is the product of the probabilities of all the n rules  
        
      LHS -> RHS  
        
      used to expand each of the n non-terminals of T
   4. The most probable parse tree for a given S is chosen as follows:  
        
      P(S) = maximise argument of P(T)
4. **Probabilistic CKY**:
   1. Assign a 3rd dimension to each cell of matrix.
   2. This dimension corresponds to each non-terminal that can be put in the original cell.
   3. The value of the 3rd dimension cells contain a probability of that non-terminal.
   4. So we have an (n+1)\*(n+1)x k matrix.
   5. A cell [i,j,A] therein is the probability of the constituent A that is between positions i to j of the input.
   6. Example: (PCKY parse)

|  |  |  |  |
| --- | --- | --- | --- |
| S -> NP VP | 0.8 | Det -> the | 0.4 |
| NP -> Det N | 0.3 | Det -> a | 0.4 |
| VP -> V NP | 0.1 | N -> meal | 0.01 |
| V -> includes | 0.05 | N -> flight | 0.02 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The | Flight | Includes | A | meal |
| Det .4    [0,1] | NP .3\*.4 \* .02= .0024   [0,2] | [0,3] | [0,4] | S 0.8 \* .000012\* .0024 =.00000002304 [0,5] |
|  | N .02  [1,2] | [2,2] | [1,4] | - |
|  |  | V 0.05  [2,3] | [2,4] | VP 0.2 \*0.05 \* 0.0012 = 0.000012 [2,5] |
|  |  |  | Det 0.4  [3,4] | NP 0.3 \* 0.4 \* 0.01 = 0.0012 [3,5] |
|  |  |  |  | N 0.01  [4,5] |

1. **Evaluating Constituency Parsers**:
   1. Parse Eval:
      1. A metric that measures how much the constituents in the hypothesis parse tree look like the constituents in a gold standard parse tree.
      2. Gold standard: hand annotated corpora, like the Penn treebank.
      3. Recall = (No. correct constituents in the hypothesis parse of s) / (No. constituents in the gold standard parse of s)
      4. Precision = (No. correct constituents in the hypothesis parse of s) / (No. of total constituents in the hypothesis parse of s)
      5. Normal target measure: F-Score
2. Problems with PCFGs:
   1. Independence Assumption: PCFG’s assume rules are applied independent of each other: “the probability of a group of independent events is their multiplication.”

|  |  |  |
| --- | --- | --- |
|  | Pronoum | Non-pronoun |
| Subject | 91% | 9% |
| Object | 34% | 66% |

* + 1. Not correct, e.g. NP’s that are sbj's are far more likely to be pronouns.
    2. Solution: Conditionalise the probabilities on whether the non-terminal is in subject or object position, e.g.:   
       NP\_sbj -> Pronoun [0.91] NP\_obj -> Pronoun [0.34]
  1. Lack of sensitivity to lexical facts,   
     In English it is more likely that PP’s attach themselves to NP’s. This will parse the following correctly:  
       
     “fishermen caught tons of herring”,   
       
     PP “of herring” modifying NP “tons”, via the rules: VP->V NP, NP -> NP PP  
       
     But not the following: “workers dump sacks into bins”  
       
     PP “into bins” modifying VP “dump”, where the rule is: VP->V NP PP  
       
     Solution: lexicalized grammars: VP(dumped) -> VBD(dumped) NP(sacks) PP(into)

1. **Treebanks and Grammar Extraction**:
   1. How could we build a system to learn a grammar automatically?
   2. CFG’s can in principle be used to assign a parse tree to any given sentence.
   3. Given a corpus, we can “annotate” each of its sentences with a parse tree.
   4. A corpus thus annotated is called a Treebank.
   5. Treebanks are widely used in empirical investigations of syntactic phenomena.
   6. How to build a Treebank, manual annotation:
      1. Use linguists to create trees by hand for each sentence.
   7. or, semi-automatic/assisted:
      1. Use an automatic parser (based on manual rules or trained on other sources)
      2. Use linguists to hand-correct the parser
   8. Example: Penn Treebank
   9. Produced from: Brown, ATIS, Wall Street Journal: English
   10. Other languages such as Arabic and Chinese
   11. A treebank is a corpus whose every sentence is paired with its parse tree.
   12. The Penn treebank is a large treebank consisting of the corpora: WallStreet Journal, Brown, ATIS, Switchboard
   13. 1 million terminals, 1 million non-terminals, 17,500 rules
   14. Amongst these are 4,500 rules to expand VPs
   15. Example corpus: Air Traffic Information System (ATIS)
       1. Early example of an application of formal grammar
       2. Automatic reservation of flights for airlines
   16. Given the trees of a treebank, one can extract the rules used to generate the trees. This is called grammar extraction.
2. **What about PCFG’s? How to learn the probabilities of rules**:
   1. Use a treebank:
      1. The probability of each expansion of a non-terminal is obtained by counting the number of times that expansion occurs in the corpus, then normalising it:  
           
         P(alpha -> beta given alpha) = (Count(alpha -> beta)) / (summation of Count(alpha -> gamma))
      2. (For semi-supervised learning) Parse your nonannotated corpus.
         1. For non-ambiguous sentences: increment a counter for every rule in the parse, then normalise.
         2. For ambiguous sentences: work incrementally
            1. assign an equal prob to rules of the parser.
            2. parse a sentence and compute a probability for it, use these to weight the counts and re-estimate the probabilities, until convergence.
3. **CCG Bank** 
   1. A CCG (Combinatorial Categorial Grammar) Bank was put together by Hockenmaier and Steedman in 2007).
   2. It was created by translating CFG trees from the Penn Treebank.
   3. It has 48,934 sentences paired with CCG derivations.
   4. It provides a lexicon of 44,000 words with over 1200 categories.
4. **Shortcomings: Generative and Logical Grammars** 
   1. Generative grammars have been extensively used by linguistics to develop extensive theories of language constructs. But their drawbacks are …
      1. Their phrase structures are not very intuitive to nonlinguists. #
      2. More importantly, they differ hugely from one another for different languages.
         1. For instance for language with a different word order, one has to sit down and develop a whole new set of rules.
5. **Shortcomings: Generative and Logical Grammars**:
   1. In Farsi, the sentence structure is Sbj-Obj-Verb (SOV), as opposed to in English where it is Sbj-Verb-Obj (SVO). So we have to change our S-> NP VP rule to something like:   
        
      S-> NP NP VP   
      where   
      VP -> verb   
      NP -> Nom PP | PP Nom
   2. In languages with a relatively free word order such as Czech, one is in more trouble and has to add a rule for every possible location of a phrase. That is:   
        
      S -> NP VP NP | NP NP VP | VP NP NP   
      1. Given that there may be different possibilities changes within each NP and VP, huge set of rules will be generated for these languages.
   3. Logical grammars suffer from similar problems for languages with different or flexible word order. For instance for Farsi, we have to change the type of the transitive verb from (NP\S)/NP to NP\NP\S.
   4. And for Czech, we have to add many verb types to our lexicon, e.g.   
      (NP\S)/NP,   
      NP\NP\S,   
      S/NP/NP
      1. In this approaches, although the number of rules stays the same (the same 2 cancelation schemata for any language), the lexicon becomes huge.
6. **Dependency Grammars**:
   1. We can overcome some of these problems with dependency grammars.   
        
      Dependency grammars are based on the notion of a dependency relation.
   2. Dependency relations are an extension of the traditional notion of grammatical relation, examples of which are: nsubj, dobj, det, mod…
   3. As such, they are more intuitive. Also, all languages have such relations in them, although may be their order in a sentence differs. But that is not a problem, since dependency grammars do not assume a fixed word order.
   4. A dependency grammar is expressed in terms of a graph, sometimes refined to a tree.
   5. A dependency grammar is expressed in terms of a graph, sometimes refined to a tree.
   6. Formal definitions: The most general definition of dependency grammars is that of a dependency structure: G = (V, A). This is a graph G with:
      1. a set of vertices (nodes) V, which are labelled by the words of sentences we are analysing.
      2. a set of arcs (edges) A, labelled by the dependency relation between the two words it is connecting.
   7. Each language might pose different restrictions on its dependency structures. A most common restriction used by many parsers is assuming that these graphs are trees.
   8. A dependency tree is defined as follows:
      1. There is a single designated root node that has no incoming arcs.
      2. Except for the root node, each vertex has exactly one incoming arc.
      3. There is a unique path from the root node to each vertex in V.  
           
         There is a graph example here which you can access in the lecture PDF.
7. **Projectivity**:
   1. An arc between v and w in a dependency tree is called projective if there is a path between v and every other word that occurs in the sentence between v and w.
   2. A dependency tree is projective if all its arc are.
   3. Projectivity is shown to be equivalent to the absence of crossings of arcs in dependency trees.
   4. It is important because: there is an algorithm than translates generative grammar trees to dependency trees. This algorithm only produces projective trees. So one can say that the nonprojective trees are either not context free or have grammatical mistakes in them.
8. **Dependency Parsing:**
   1. The main difference between a dependency parser and other parsers is that the former has a list of input tokens and a Stack.
   2. The most popular method is transition parsing which uses an algorithm called shift-reduce.
   3. At the shift stage, input tokens are pushed on the stack.
   4. At the reduce stage, the top two elements of the stack are matched against each other, and a decision is made so as whether one is the head of the other in a dependency relation or vice versa.
   5. In the standard approach to transition-based parsing, the operators used to produce new configurations are surprisingly simple and correspond to the intuitive actions one might take in creating a dependency tree by examining the words in a single pass over the input from left to right:
      1. Assign the current word as the head of some previously seen word,
      2. Assign some previously seen word as the head of the current word,
      3. Or postpone doing anything with the current word adding it to a store for later processing.
   6. To make these actions more precise, we’ll create three transition operators that will operate on the top two elements of the stack:
      1. LEFTARC: Assert a head-dependent relation between the word at the top of the stack and the word directly beneath it; remove the lower word from the stack.
      2. RIGHTARC: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack;
      3. SHIFT: Remove the word from the front of the input buffer and push it onto the stack.
   7. At the initial stage the stack is empty.
   8. At the final stage, a set of relations are returned as the parse of the sentence.
   9. At each stage, the parser consults an oracle! It is assumed that the oracle makes the right choice for the reduce step. This is an unlikely assumption.
   10. In reality, supervised machine learning techniques are applied to train a classifier as the oracle. This needs lots of training data.
   11. This algorithm is greedy with no back tracking or correcting mechanism.
   12. The statistical side of dependency parsing comes from a score given by the oracle (a trained classifier, like an SVM).
   13. Regardless of the specific learning approach, this choice can be viewed as assigning a score to all the possible transitions and picking the best one from a given configuration of the parser c:  
         
       hat{T}(c) = maximise argument of Score(t, c)
   14. The oracle is trained on features of the configuration.
   15. These also use a shift-reduce algorithm where instead of a dependancy relations table, we have access to a lexicon.
9. **Dependency Parsing Evaluation** 
   1. Exact match (= S) percentage of correctly parsed sentences
   2. Attachment score (= W) percentage of words that have the correct head
   3. Labelled attachment score for single dependency types (labels):
      1. Precision
      2. Recall
      3. F1 measure
   4. Correct root
      1. percentage of sentences that have the correct root
10. Dependency Treebank
    1. The first dependency banks were produced for morphologically rich languages such as Czech, Hindi and Finnish, e.g. Prague Dependency Treebank for Czech.
    2. Like the CCG case, the English dependency treebanks have been extracted from existing resources such as the Penn Treebank by translating the CFG trees to dependency trees.
    3. Recently they have been extended to conversational telephone speech, weblogs, talk shows and so on in English, Chinese and Arabic.