# Natural Language Processing

## Partial/Chunk Parsing

## Chapter 13.5 in Jurafsky&Martin

# Sources

- These slides are borrowed from Mike Rosner at University of Malta (http://staff.um.edu.mt/mros1/)
  - This presentation is largely based on the NLTK Chunking Tutorial by Steven Bird, Ewan Klein and Edward Loper version 0.6.3 (2006)

# Two Kinds of Parsing

- ***full parsing*** with formal grammars (HPSG, LFG, TAG, ...) to be compared with robust parsing
- ***chunk parsing*** in the context of tagging (also called partial, shallow, or robust parsing.
- Chunk parsing is an efficient and robust approach to parsing natural language
- It is a popular alternative to full parsing

# Example Chunks

[I begin] [with an intuition] :

[when I read] [a sentence] ,

[I read it] [a chunk] [at a time] .

# Abney (1991): Parsing by Chunks

- These chunks correspond in some way to prosodic patterns.
- … the strongest stresses in the sentence fall one to a chunk, and pauses are most likely to fall between chunks.
- The typical chunk consists of a single content word surrounded by a constellation of function words, matching a fixed template.

# Content and Function Words

[I begin] [with an intuition] :

[when I read] [a sentence] ,

[I read it] [a chunk] [at a time] .

# Inter and Intra Chunk Structure

- Within chunks
  - A simple context-free or finite state grammar is quite adequate to describe the structure of chunks.
- Between chunks within a sentence
  - By contrast, the relationships between chunks are mediated more by lexical selection than by rigid templates.
  - The order in which chunks occur is much more flexible than the order of words within chunks.

# Summary

- Chunks are non-overlapping regions of text
- Usually consisting of a head word (such as a noun) and the adjacent modifiers and function words (such as adjectives and determiners).

# Chunking and Tagging

**Tagging**

- Segmentation – identifying tokens
- Labelling – identifying the correct tag

**Chunk Parsing**

- Segmentation – identifying strings of tokens
- Labelling- identifying the correct chunk type

# Segmentation and Labelling

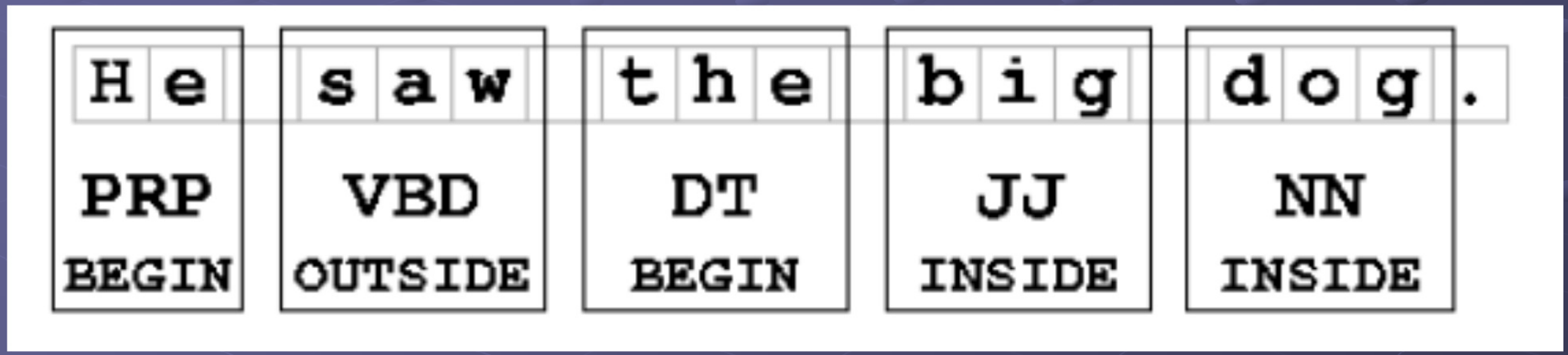
# Chunking vs Full Parsing

- Both can be used to build hierarchical structures
- Chunking involves hierarchies of limited depth (usually 2)
- Complexity of full parsing typically $O(n^3)$, versus O(n) for chunking
- Chunking leaves gaps between chunks
- Chunking can often give imperfect results: *I turned off the spectroroute*

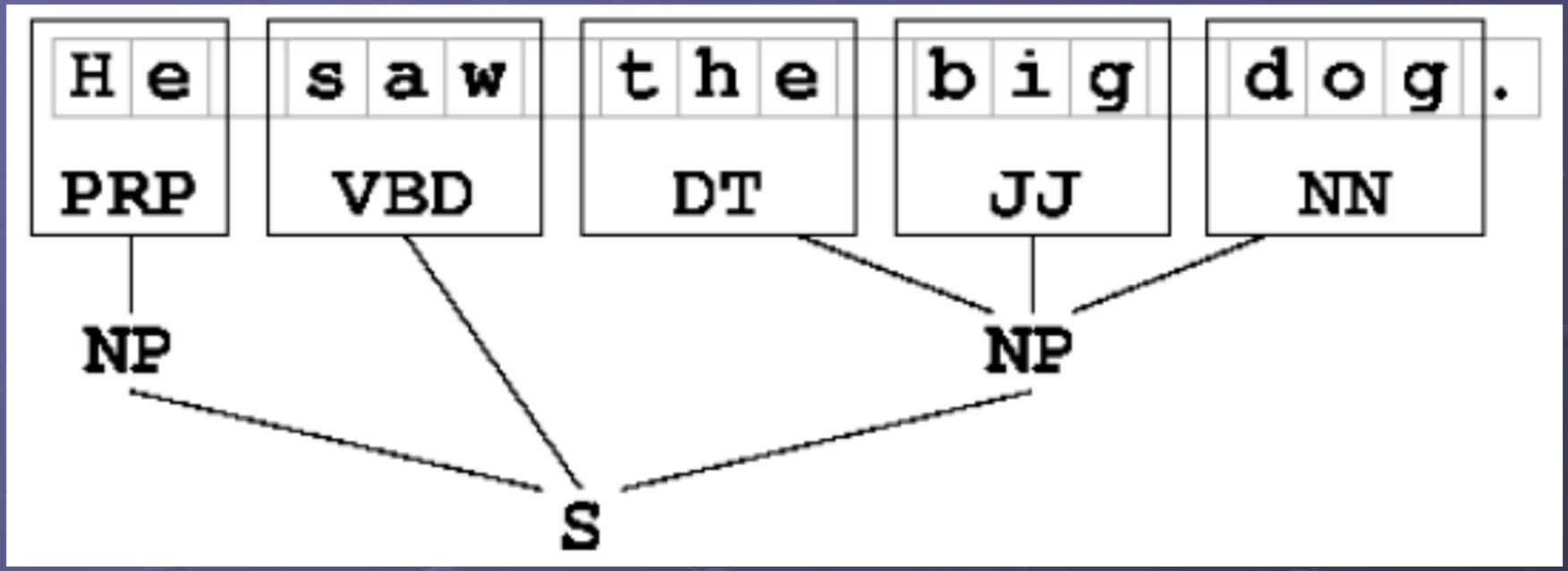
# Representing Chunks
# IOB Tags vs Trees

- Each token is tagged with one of three special chunk tags, INSIDE, OUTSIDE, or BEGIN.
- A token is tagged as BEGIN if it is at the beginning of a chunk, and contained within that chunk.
- Subsequent tokens within the chunk are tagged INSIDE. All other tokens are tagged OUTSIDE.

# Example IOB Tags



# Example Tree



# Chunk Parsing

- A chunk parser finds contiguous, non-overlapping spans of related tokens and groups them together into chunks.
- It then combines these individual chunks together, along with the intervening tokens, to form a ***chunk structure***.
- A ***chunk structure*** is a two-level tree that spans the entire text, and contains both chunks and un-chunked tokens.

# Example Chunk Structure

(S: (NP: 'I')
'saw'
(NP: 'the' 'big' 'dog')
'on'
(NP: 'the' 'hill'))

# Chunking with Regular Expressions

- NLTK-Lite provides a regular expression chunk parser, parse.RegexpChunk to define the kinds of chunk we are interested in, and then to chunk a tagged text.
- The chunk parser begins with a structure in which no tokens are chunked
- Each regular-expression pattern (or chunk rule) is applied in turn, successively updating the chunk structure.
- Once all of the rules have been applied, the resulting chunk structure is returned.

# Tag Strings

- A tag string is a string consisting of tags delimited with angle-brackets, e.g., <DT><JJ><NN><VBD><DT><NN>
- tag strings do not contain any whitespace.

# Chunking with Regular Expressions

- The NLTK chunk parser operates using chunk definitions that are supplied in the form of *tag patterns* which are regular expressions over tags, e.g.

  <DT><JJ>?<NN>
- NB: RE operators come *after* the tag.
- ? means an optional element

# Tag Patterns

- Angle brackets group, so <NN>+ matches one or more repetitions of the tag string <NN>; and <NN|JJ> matches the tag strings <NN> or <JJ>.
- The operator * within angle brackets is a wildcard, so that <NN.*> matches any single tag starting with NN.
- <NN>* matches zero or more repetitions of <NN>

# Another Example

<DT>?<JJ.*>*<NN.*>

- In this case the DT is optional
- It is followed by zero or more instances of <JJ.*>
- <JJ.*> matches *any type* of adjective
- The adjectives are followed by <NN.*>, i.e. *any type* of NN.

# Creating a Chunk Parser in NLTK

- First create one or more rules

```
rule1 =
parse.ChunkRule('<DT|NN>+',
                'Chunk sequences of DT and NN')
```

- Then create the parser

```
chunkparser =
   parse.RegexpChunk([rule1], chunk_node='NP',
   top_node='S')
```

- Note that RegexpChunk has optional second and third arguments that specify the node labels for chunks and for the top-level node, respectively.

# First Match Takes Precedence

- If a tag pattern matches at multiple overlapping locations, the first match takes precedence.
- For example, if we apply a rule that matches two consecutive nouns to a text containing three consecutive nouns, then the first two nouns will be chunked

# Example

```
>>> from nltk_lite import tag
>>> text = "dog/NN cat/NN mouse/NN"
>>> nouns = tag.string2tags(text)
>>> rule = parse.ChunkRule('<NN><NN>', 'Chunk
    two consecutive nouns')
>>> parser = parse.RegexpChunk([rule],
    chunk_node='NP', top_node='S')
>>> parser.parse(nouns)
(S: (NP: ('dog', 'NN') ('cat', 'NN')) ('mouse', 'NN'))
```

# Two Rule Example

```
>>> sent = tag.string2tags("the/DT little/JJ cat/NN
    sat/VBD on/IN the/DT mat/NN")
>>> rule1 = parse.ChunkRule('<DT><JJ><NN>',
    'Chunk det+adj+noun')
>>> rule2 = parse.ChunkRule('<DT|NN>+', 'Chunk
    sequences of NN and DT')
>>> chunkparser = parse.RegexpChunk([rule1,
    rule2], chunk_node='NP', top_node='S')
>>> chunk_tree = chunkparser.parse(sent,
    trace=1)
```

# Trace

- Input:

<DT> <JJ> <NN> <VBD> <IN> <DT> <NN>

- Chunk det+adj+noun:

{<DT> <JJ> <NN>} <VBD> <IN> <DT> <NN>

- Chunk sequences of NN and DT:

{<DT> <JJ> <NN>} <VBD> <IN> {<DT> <NN>}

# Rule Interaction

- When a ChunkRule is applied to a chunking hypothesis, it will only create chunks that do not partially overlap with chunks already in the hypothesis.
- Thus, if we apply these two rules in reverse order, we will get a different result:

# Reverse Order of Rules

- >>> chunkparser = parse.RegexpChunk([rule2, rule1], chunk_node='NP', top_node='S')
- Input:

  <DT> <JJ> <NN> <VBD> <IN> <DT> <NN>
- Chunk sequences of NN and DT:

  {<DT>} <JJ> {<NN>} <VBD> <IN> {<DT> <NN>}
- Chunk det+adj+noun:

  {<DT>} <JJ> {<NN>} <VBD> <IN> {<DT> <NN>}

# Chinking Rules

- Sometimes it is easier to define what we *don't* want to include in a chunk than we *do* want to include.
- Chinking is the process of removing a sequence of tokens from a chunk.
- If the sequence of tokens spans an entire chunk, then the whole chunk is removed; if the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before.
- If the sequence is at the beginning or end of the chunk, these tokens are removed, and a smaller chunk remains.

# Creating Chink Rules

- ChinkRules are created with the ChinkRule constructor,
- >>> chink_rule = parse.ChinkRule('<VBD|IN>+', 'Chink sequences of VBD and IN')
- To show how it works, we first define chunkall_rule = parse.ChunkRule('<.*>+', 'Chunk everything')
- and then put the two together

# Running Chink Rules

```
>>> chunkparser =
    parse.RegexpChunk([chunkall_rule, chink_rule],
    chunk_node='NP', top_node='S')
```

- Input:

  <DT> <JJ> <NN> <VBD> <IN> <DT> <NN>
- Chunk everything:
  {<DT> <JJ> <NN> <VBD> <IN> <DT> <NN>}
- Chink sequences of VBD and IN:
  {<DT> <JJ> <NN>} <VBD> <IN> {<DT> <NN>}

# The Unchunk Rule

- Unchunk rules are very similar to ChinkRule except that it will only remove a chunk if the pattern matches an entire chunk.
- >>> unchunk_rule = parse.UnChunkRule('<NN|DT>+', 'Unchunk sequences of NN and DT')

# Chunkparser with Unchunk Rule

```
>>> chunk_rule =
    parse.ChunkRule('<NN|DT|JJ>+','Chunk
    sequences of NN, JJ, and DT')
>>> chunkparser =
    parse.RegexpChunk([chunk_rule,
    unchunk_rule], chunk_node='NP',
    top_node='S')
```

# Unchunk Parser Trace

- Input:

  <DT> <JJ> <NN> <VBD> <IN> <DT> <NN>
- Chunk sequences of NN, JJ, and DT:

  {<DT> <JJ> <NN>} <VBD> <IN> {<DT> <NN>}
- Unchunk sequences of NN and DT:

  {<DT> <JJ> <NN>} <VBD> <IN> <DT> <NN>

# Merge Rules

- MergeRules are used to merge two contiguous chunks.
- Each MergeRule is parameterized by two tag patterns: a left pattern and a right pattern.
- A MergeRule will merge two contiguous chunks C1 and C2 if the end of C1 matches the left pattern, and the beginning of C2 matches the right pattern.

# Split Rules

- SplitRules are used to split a single chunk into two smaller chunks.
- Each SplitRule is parameterized by two tag patterns: a left pattern and a right pattern.
- A SplitRule will split a chunk at any point p, where the left pattern matches the chunk to the left of p, and the right pattern matches the chunk to the right of p.

# Evaluating Chunk Parsers

- Essentially, evaluation is about comparing the behaviour of a chunk parser against a standard.
- Typically this involves the following phases
  - Save already chunked text
  - Unchunk it
  - Chunk it using chunk parser
  - Compare the result with the original chunked text

# Evaluation Metrics

- Metrics are typically based on the following sets:
  - **guessed**: The set of chunks returned by the chunk parser.
  - **correct**: The correct set of chunks, as defined in the test corpus.
- From these we can define useful measures

# Evaluation Metrics

- Precision:
  - Number of correct chunks guessed by parser / Total number of guessed chunks
- Recall:
  - Number of correct chunks guessed by parser / Total number of correct chunks
- F-Measure: harmonic mean of precision and recall
  - $F_1 = 2PR / (P+R)$