APAN PS5430
Applied Text & Natural Language Analytics
Week 3: Basic Text Processing

Javid Huseynov, Ph.D.
Thursday, February 6, 2020

COLUMBIA UNIVERSITY

Week 3 Agenda

- Text Pre-Processing & Noise Removal
  - Regular Expressions
  - Context-Free Grammars
  - Sentence Segmentation
  - Tokenization, Stemming & Lemmatization
  - Part-of-speech Tagging
  - Shallow Parsing/Chunking
  - Dependency vs Constituency
  - Dependency Parsing

NLP Pipeline Tasks

TEXT

Basic Text Processing

- Regular Expressions
- Tokenization Segmentation
- Stemming Lemmatization
- Part-of-Speech Tagging

Information Extraction

- Named Entity Recognition
- Named Entity Disambiguation
- Coreference Resolution
- Relationship Extraction

Meaning Reconstruction & Language Understanding

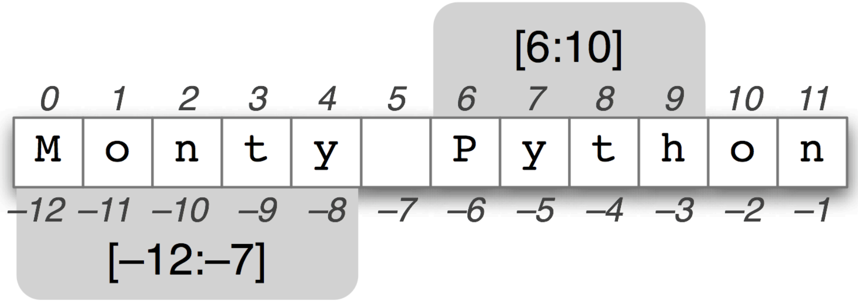
- Sentiment Analysis
- Semantic Analysis Topic Modeling
- Question Answering
- Machine Translation

KNOWLEDGE

Strings

- Fundamental text data types in Python and other programming languages
- Defined within single, double or triple quotes
- Support addition/concatenation, multiplication, regular expressions, etc.

Method	Functionality
`s.find(t)`	index of first instance of string `t` inside `s` (`-1` if not found)
`s.rfind(t)`	index of last instance of string `t` inside `s` (`-1` if not found)
`s.index(t)`	like `s.find(t)` except it raises `ValueError` if not found
`s.rindex(t)`	like `s.rfind(t)` except it raises `ValueError` if not found
`s.join(text)`	combine the words of the text into a string using `s` as the glue
`s.split(t)`	split `s` into a list wherever a `t` is found (whitespace by default)
`s.splitlines()`	split `s` into a list of strings, one per line
`s.lower()`	a lowercased version of the string `s`
`s.upper()`	an uppercased version of the string `s`
`s.title()`	a titlecased version of the string `s`
`s.strip()`	a copy of `s` without leading or trailing whitespace
`s.replace(t, u)`	replace instances of `t` with `u` inside `s`



Text Patterns

- Patterns or regular expressions are useful for processing textual data
  - Given string ***S***, is ***S*** a member of the set defined by pattern ***P***
  - Some common string patterns:

String Literal	Pattern
“123-45-6789”	Social Security Number
“999-999-9999”	Phone number
“ps5430@columbia.edu”	Email address
“09/17/2019”	Date
“1234 1234 1234 1234”	Credit Card Number

Regular Expressions

- A formal language for specifying text strings
  - Used as the first model for any text processing
  - Used as features in machine learning classifiers
  - Used for capturing generalizations
  - Python *re* library [import re]

- Find all instances of the word "*the*" in a text
  the – misses capitalized examples
  [tT]he – Incorrectly returns *other* or *theology*
  [^a-zA-Z][tT]he[^a-zA-Z]
- False Positives: matching **there,then,other**
- False Negatives: not matching **The**

Pattern	Matches	Examples
`[A-Z]`	An upper case letter	`Drenched Blossoms`
`[a-z]`	A lower case letter	`my beans were impatient`
`[0-9]`	A single digit	`Chapter 1: Down the Rabbit Hole`
`[^A-Z]`	Not an upper case letter	`Oyfn pripetchik`
`[^Ss]`	Neither 'S' nor 's'	`I have no exquisite reason"`
`[^e^]`	Neither e nor ^	`Look here`
`a^b`	The pattern a carat b	`Look up a^b now`

Most Common Uses

Some Key Functionality

- Searching a string
  - `re.search` and `re.match`
- Substitute a substring
  - `re.sub`
- Break a string into pieces
  - `re.split`
- Finding a string
  - `re.findall`

This expression…	matches this…	but not this…
`^(the cat).+`	the cat runs	see the cat run
`.+(the cat)$`	watch the cat	the cat eats

Special Characters

- Alternative: `|`
- Grouping: `()`
- Quantification: `?*+{m,n}`
- Anchors: `^$`
- Meta-characters: `.[][-][^]`
- Character classes: `\d\D\w\W`…

If ***A*** and ***B*** are both patterns:

- ***AB***: Pattern A followed by the pattern B
- ***A***| ***B***: Either pattern A or pattern B
- ***A****: Zero or more repetitions of A
- ***A***+: One or more repetitions of A
- **A**?: Zero or one occurrence of A

Regex cheat sheet

Special characters

\	escape special characters
.	matches any character
^	matches beginning of string
$	matches end of string
[5b-d]	matches any chars '5', 'b', 'c' or 'd'
[^a-c6]	matches any char except 'a', 'b', 'c' or '6'
R|S	matches either regex R or regex S
()	creates a capture group and indicates precedence

Quantifiers

*	0 or more (append ? for non-greedy)
+	1 or more (append ? for non-greedy)
?	0 or 1 (append ? for non-greedy)
{m}	exactly m m occurrences
{m, n}	from m to n . m defaults to 0, n to infinity
{m, n}?	from m to n , as few as possible

Special sequences

\A	start of string
\b	matches empty string at word boundary (between \w and \W)
\B	matches empty string not at word boundary
\d	digit
\D	non-digit
\s	whitespace: [\t\n\r\f\v]
\S	non-whitespace
\w	alphanumeric: [0-9a-zA-Z_]
\W	non-alphanumeric
\Z	end of string
\g<id>	matches a previously defined group

Special sequences

(?iLmsux)	matches empty string, sets re.X flags
(?:...)	non-capturing version of regular parentheses
(?P...)	matches whatever matched previously named group
(?P=)	digit
(?#...)	a comment; ignored
(?=...)	lookahead assertion: matches without consuming
(?!...)	negative lookahead assertion
(?<=...)	lookbehind assertion: matches if preceded
(?<!...)	negative lookbehind assertion
(?(id)yes|no)	match 'yes' if group 'id' matched, else 'no'

Python Regex builder: https://pythex.org/

Useful reference on regular expressions: http://users.cs.cf.ac.uk/Dave.Marshall/Internet/NEWS/regexp.html

Context Free Grammar (CFG)

- A grammar is a set of rules for putting strings together forming a language, consists of:
  - a set of **variables** (or non-terminals), one of which is the start variable;
  - a set of **terminals**;
  - a list of **productions (rules)**
- Example: $\mathbf{0}^n\mathbf{1}^n$ $S \to 0S1$
  $S \to \varepsilon$
- Language is context-free if it's generated by CFG

- Context Free Grammar is a 4-tuple (V, Σ, S, P) where:
  - V - finite set of variables
  - Σ - finite alphabet of terminals
  - S - start variable;
  - P - finite set of rules of the form $V \to (V \cup \Sigma)^*$
- Every string generated by grammar must fit its description (**consistency**)
- Every description is generated by the grammar (**completeness**)

Example: A silly language CFG

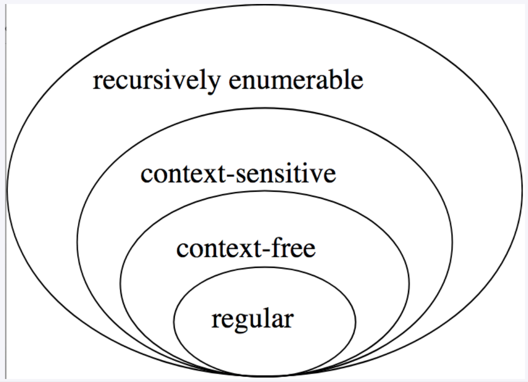
- Assume a CFG that generated sentences composed of noun- and verb-phrases according to the following rules:
    - S → NP VP
        - NP → the N
        - VP → V NP
    - V → sings | eats
    - N → cat | song |canary
- CFG generates "*the canary sings the song*" and "*the song eats the cat*"
- CFG generates all "legal" sentences, not just the meaningful ones

Chomsky hierarchy of grammars

Grammar	Languages	Automaton	Production rules (constraints)*	Examples[3]
Type-0	Recursively enumerable	Turing machine	$\alpha A \beta \rightarrow \gamma$	$L = \{w \mid w$ describes a terminating Turing machine$\}$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \alpha$	$L = \{a^n b^n \mid n > 0\}$
Type-3	Regular	Finite state automaton	$A \rightarrow \mathrm{a}$ and $A \rightarrow \mathrm{a}B$	$L = \{a^n \mid n \geq 0\}$

* Meaning of symbols:

- a = terminal
- A, B = non-terminal
- α, β, γ = string of terminals and/or non-terminals
  - α, β = maybe empty
  - γ = never empty



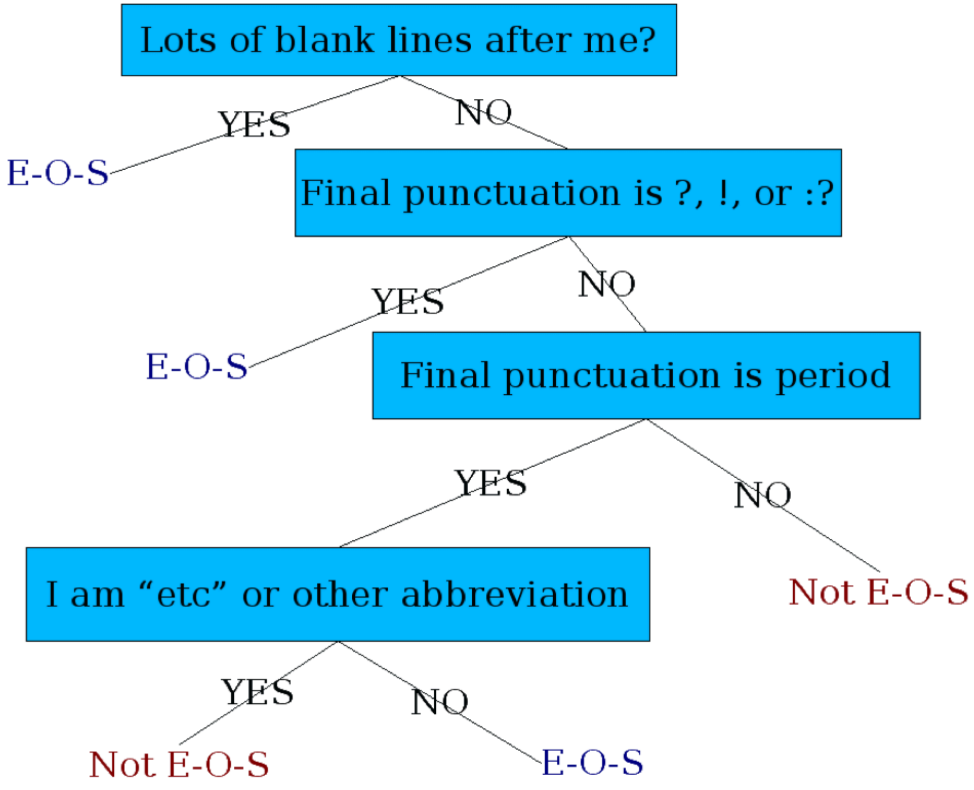
- All programming languages can be defined by CFG
- Natural languages are not context-free

Sentence Segmentation

- !, ? are relatively unambiguous
- Period "." is quite ambiguous
  - Sentence boundary
  - Abbreviations like Inc. or Dr.
  - Numbers like .02% or 4.3

- Build a binary classifier
  - Looks at a "."
  - Decides EndOfSentence/NotEndOfSentence
  - Classifiers: hand-written rules, regular expressions, or machine-learning

Which machine learning algorithm would be applicable to this task?



Tokenization

- **Tokenization** is the task of chopping up a character sequence up into *tokens*, i.e. words, *n*-grams, sentences
- ***n*-gram** is a contiguous sequence of *n* items from a given sequence of text
- **Stop words** are the most common words that are of little value in search query

- To be, or not to be: To | be | , | or | not | to | be
- Unigrams: to | be | or | not | to | be
- Bigrams: to be | or not | to be
- Trigrams: to be or | not to be
- Stop words: to | be | or | not

```
>>> from nltk.tokenize import sent_tokenize, word_tokenize
>>> sent_tokenize(s)
['Good muffins cost $3.88\nin New York.', 'Please buy me\ntwo of them.', 'Thanks.']
>>> [word_tokenize(t) for t in sent_tokenize(s)]
[['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York', '.'],
['Please', 'buy', 'me', 'two', 'of', 'them', '.'], ['Thanks', '.']]
```

Stemming & Lemmatization

- **Stemming** – reducing inflected or derived words to their base form
- **Lemmatization** – grouping together different inflected forms of a word
- Both are **linguistic morphology terms** with the same goal but different approaches

Stemming

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter_stemmer = PorterStemmer()
>>> porter_stemmer.stem('maximum')
u'maximum'
>>> porter_stemmer.stem('presumably')
u'presum'
>>> porter_stemmer.stem('multiply')
u'multipli'
```

Lemmatization

```
>>> from nltk.stem import WordNetLemmatizer
>>> wordnet_lemmatizer = WordNetLemmatizer()
>>> wordnet_lemmatizer.lemmatize('dogs')
u'dog'
>>> wordnet_lemmatizer.lemmatize('churches')
u'church'
>>> wordnet_lemmatizer.lemmatize('aardwolves')
u'aardwolf'
```

```
>>> wordnet_lemmatizer.lemmatize('are')
'are'
>>> wordnet_lemmatizer.lemmatize('is')
'is'
```

```
>>> wordnet_lemmatizer.lemmatize('is', pos='v')
u'be'
>>> wordnet_lemmatizer.lemmatize('are', pos='v')
u'be'
```

Exercise: Lexicon normalization

1. This list of tokens below represent:

   `['I like', 'the APAN', 'Text and', 'Natural Language', 'Analytics class']`

   a) Unigrams
   b) Stop words
   c) Bigrams

2. True or False: **Stop words** are the least common words that appear in a language.
3. What **strings** would this code return?

```
from nltk.stem import WordNetLemmatizer

wordnet_lemmatizer = WordNetLemmatizer()
print(wordnet_lemmatizer.lemmatize('hacks', pos='v'))
print(wordnet_lemmatizer.lemmatize('hackers', pos='n'))
```

> ______________________________

Part-of-Speech (POS) Tagging

- Words that somehow 'behave' alike:
  - Appear in similar contexts
  - Perform similar functions in sentences
  - Undergo similar transformations
- ~9 traditional word classes of parts of speech
  - Noun, verb, adjective, preposition, adverb, article, interjection, pronoun, conjunction

N	noun	chair, bandwidth, pacing
V	verb	study, debate, munch
ADJ	adjective	purple, tall, ridiculous
ADV	adverb	unfortunately, slowly
P	preposition	of, by, to
PRO	pronoun	I, me, mine
DET	determiner	the, a, that, those

Part-of-Speech (POS) Tagging

POS Tagging, aka grammatical tagging or word-category disambiguation, is the process of marking a word in a text with a particular part of speech, based on both its definition and context

- Brown Corpus (Brown University)
- Hidden Markov Models (probability of co-occurrence)
- Dynamic Programming
- Unsupervised taggers (by induction)
- Machine Learning (SVM, Max Entropy, kNN)

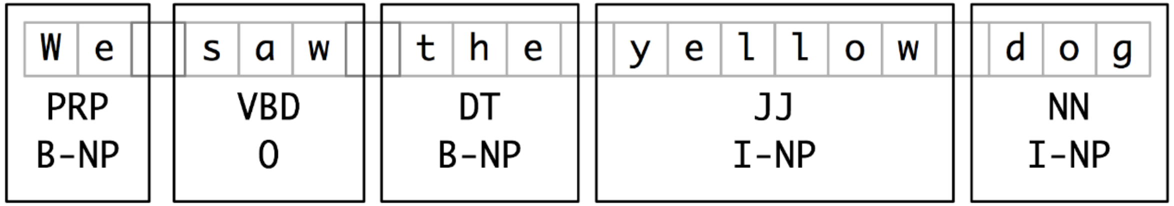
```
>>> import nltk
>>> text = nltk.word_tokenize("Dive into NLTK: Part-of-speech tagging and POS Tagger")
>>> text
['Dive', 'into', 'NLTK', ':', 'Part-of-speech', 'tagging', 'and', 'POS', 'Tagger']
>>> nltk.pos_tag(text)
[('Dive', 'JJ'), ('into', 'IN'), ('NLTK', 'NNP'), (':', ':'), ('Part-of-speech', 'JJ'), ('tagging', 'NN'),
('and', 'CC'), ('POS', 'NNP'), ('Tagger', 'NNP')]
```

- Penn TreeBank Tagset (WSJ corpus)

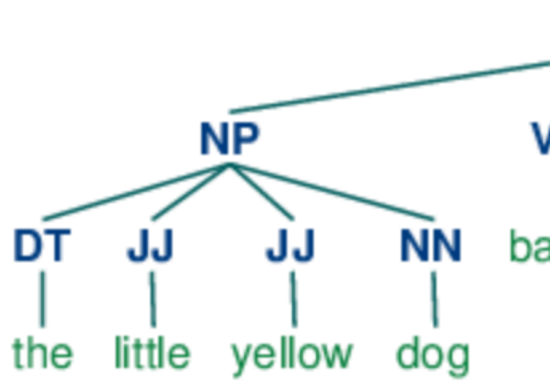
Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	*and, but, or*	SYM	symbol	+,%, &
CD	cardinal number	*one, two*	TO	"to"	*to*
DT	determiner	*a, the*	UH	interjection	*ah, oops*
EX	existential 'there'	*there*	VB	verb base form	*eat*
FW	foreign word	*mea culpa*	VBD	verb past tense	*ate*
IN	preposition/sub-conj	*of, in, by*	VBG	verb gerund	*eating*
JJ	adjective	*yellow*	VBN	verb past participle	*eaten*
JJR	adj., comparative	*bigger*	VBP	verb non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	VBZ	verb 3sg pres	*eats*
LS	list item marker	*1, 2, One*	WDT	wh-determiner	*which, that*
MD	modal	*can, should*	WP	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama*	WP$	possessive wh-	*whose*
NNS	noun, plural	*llamas*	WRB	wh-adverb	*how, where*
NNP	proper noun, sing.	*IBM*	$	dollar sign	$
NNPS	proper noun, plural	*Carolinas*	#	pound sign	#
PDT	predeterminer	*all, both*	"	left quote	' or "
POS	possessive ending	*'s*	"	right quote	' or "
PRP	personal pronoun	*I, you, he*	(	left parenthesis	[, (, {, <
PRP$	possessive pronoun	*your, one's*	)	right parenthesis	],), }, >
RB	adverb	*quickly, never*	,	comma	,
RBR	adverb, comparative	*faster*	.	sentence-final punc	. ! ?
RBS	adverb, superlative	*fastest*	:	mid-sentence punc	: ; ... – -
RP	particle	*up, off*			

Chunking / Shallow Parsing

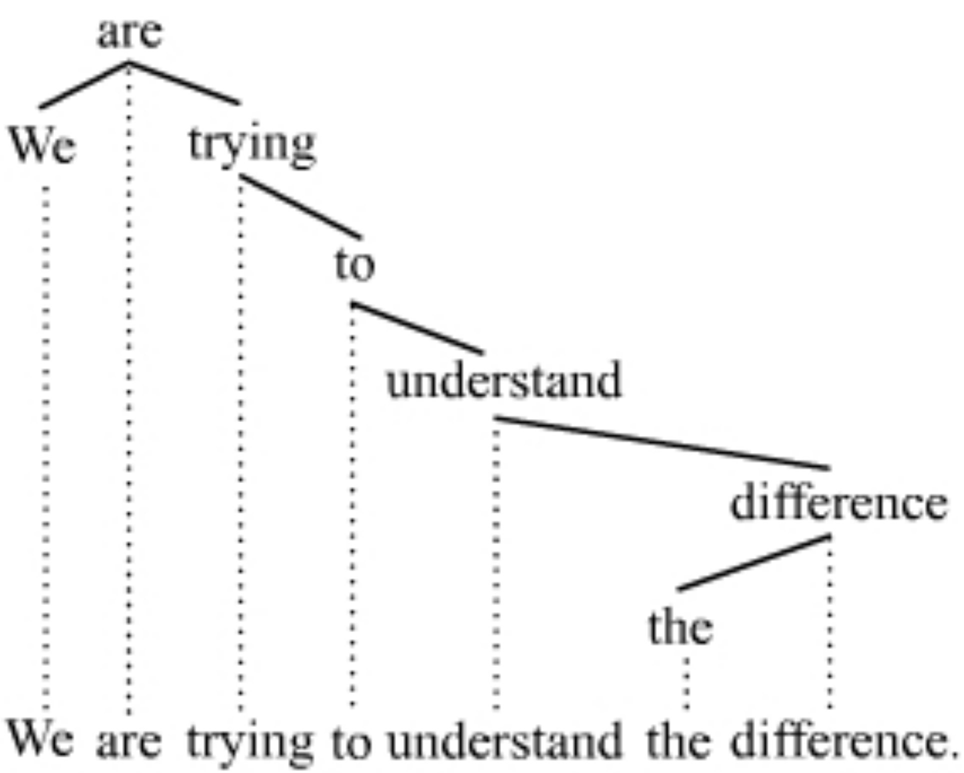
- Segment and label multi-token sequences using regular expressions and POS tags



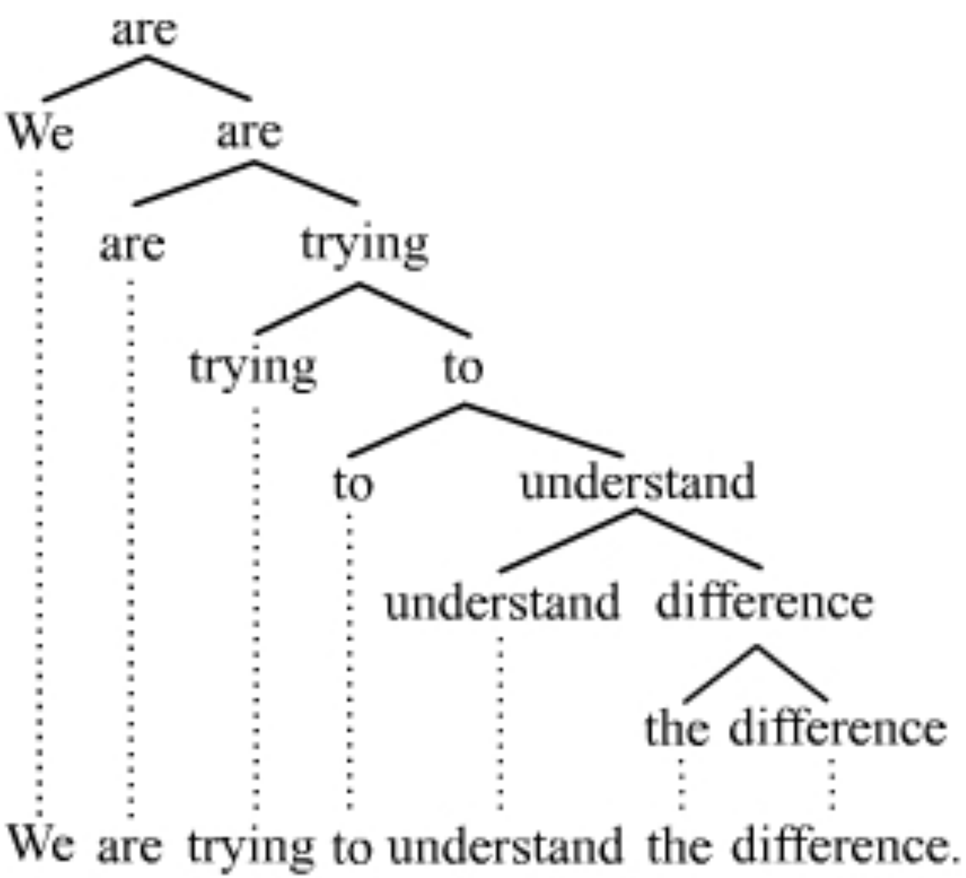

```
>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ❶
... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]

>>> grammar = "NP: {<DT>?<JJ>*<NN>}" ❷

>>> cp = nltk.RegexpParser(grammar) ❸
>>> result = cp.parse(sentence) ❹
>>> print(result) ❺
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
>>> result.draw() ❻
```



Dependency vs Constituency



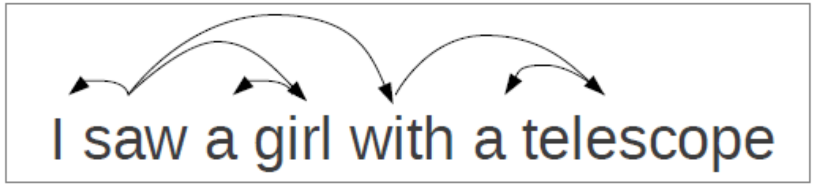
Dependency



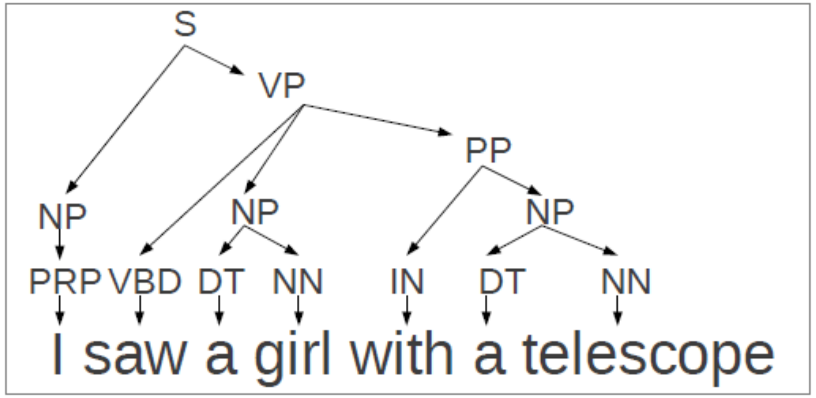
Constituency (BPS)

Types of parsing & dependencies

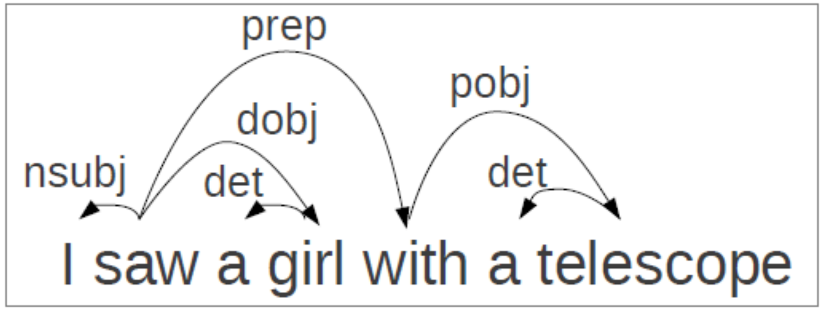
- **Dependency:** focuses on relations between words



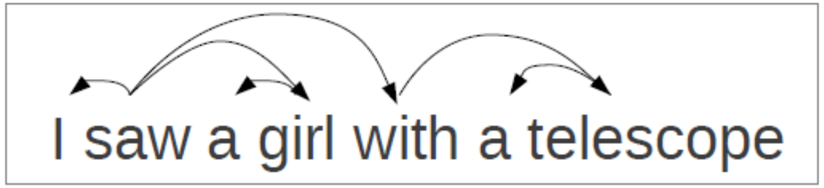
- **Phrase structure:** focuses on identifying phrases and their recursive structure



- **Typed:** Label indicating relationship between words

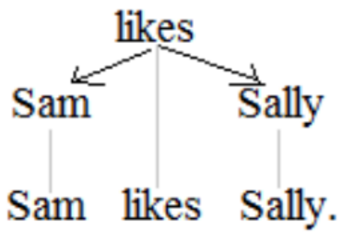


- **Untyped:** Only which words depend

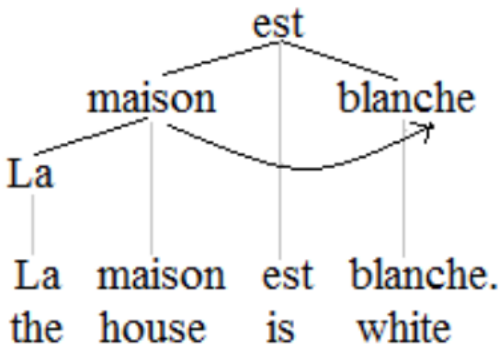


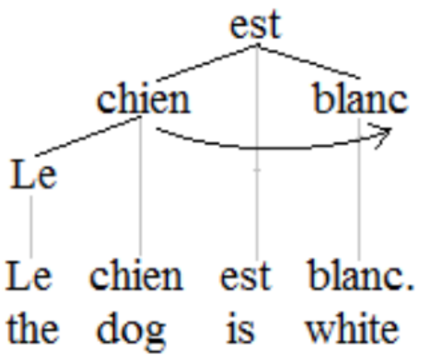
Dependency Types

- Semantic Dependency
    - Predicates and their arguments

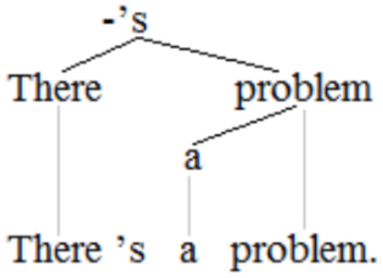


- Morphological Dependency
    - Between words or parts of words





- Prosodic Dependency
    - Syntactically independent *clitic*, depends on host (‘ll, ‘s)



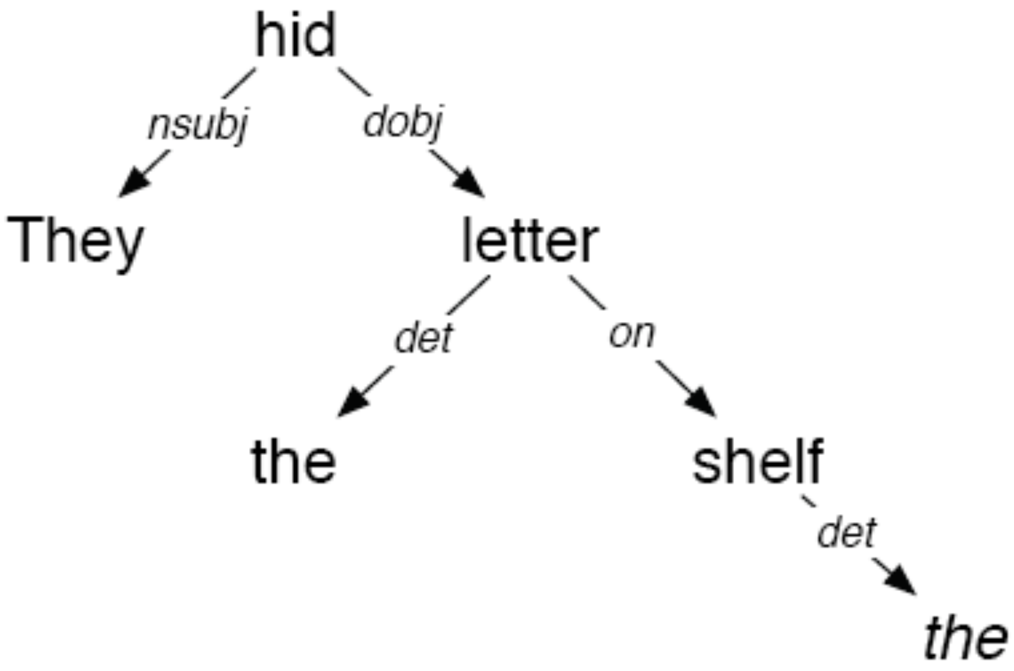
CFG vs dependency grammar

- CFG-style phrase-structure grammars the main focus is on *constituents*
- In a dependency grammar framework, a parse is a tree where
  - the nodes stand for the words in an utterance
  - Links between words are dependency relations between pairs of words.
  - Relations may be typed (labeled), or not.

- Dependency grammar parsing:
- Given a dependency grammar G and an input string $x \in \Sigma^*$, derive some or all of the dependency graphs y assigned to x by G.
- Dependency text parsing:
  - Given a text $T = (x_1, \ldots, x_n)$, derive the correct dependency graph y_i for every sentence $x_i \in T$.
- Text parsing may be grammar-driven or not

Dependency relations & parsing

Argument Dependencies	Description
nsubj	nominal subject
csubj	clausal subject
dobj	direct object
iobj	indirect object
pobj	object of preposition
Modifier Dependencies	**Description**
tmod	temporal modifier
appos	appositional modifier
det	determiner
prep	prepositional modifier



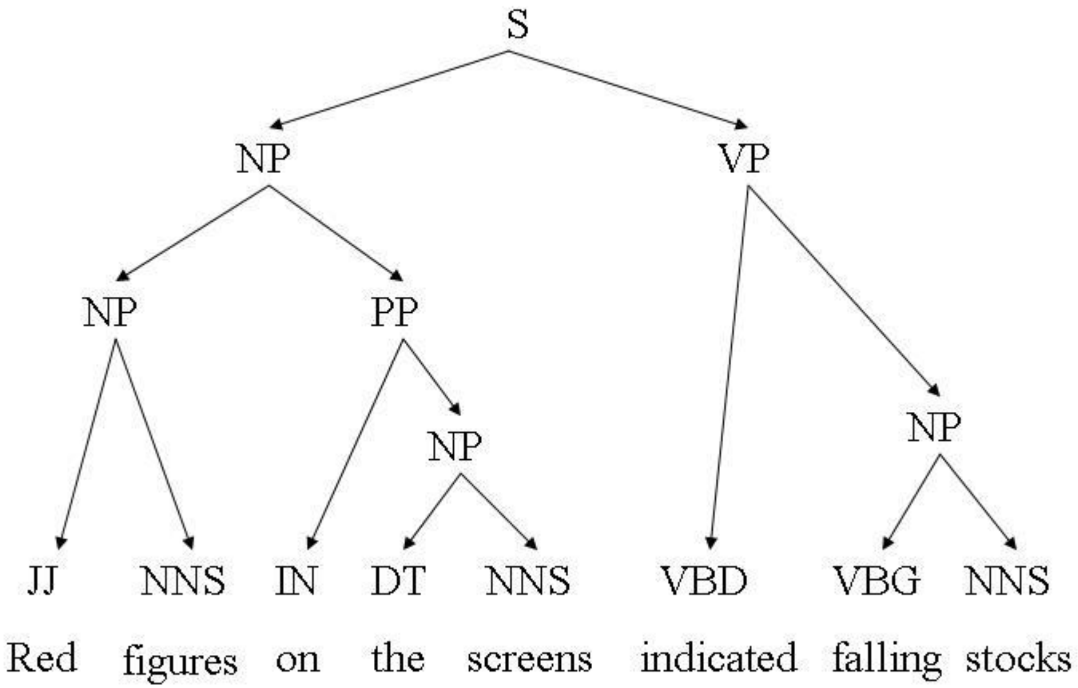
They hid the letter on the shelf

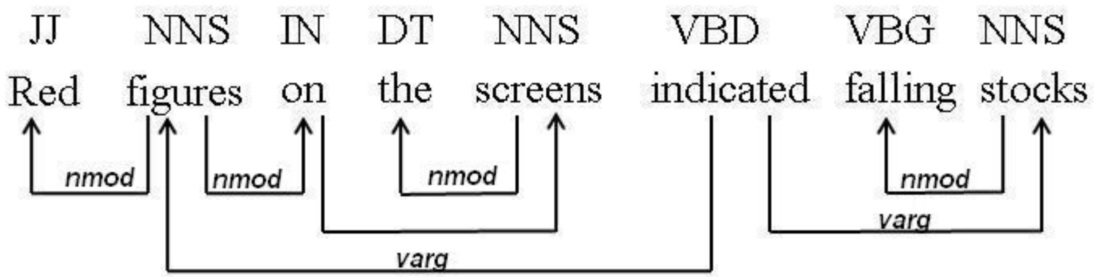
Dependency Parsing advantages and approaches

- Dependency parsing advantages over full phrase-structure parsing
  - Deals well with free word order languages where the constituent structure is quite fluid
  - Faster than CFG-based parsers
  - Dependency structure often captures the syntactic relations needed by later applications
  - CFG-based approaches often extract this same information from trees anyway.

- There are two modern approaches to dependency parsing
  - **Optimization-based approaches** that search a space of trees for the tree that *best* matches some criteria
  - **Shift-reduce approaches** that greedily take actions based on the current word and state.

Dependency vs Phrase structures





Dependency structures explicitly represent
- Head-dependent relations (directed arcs)
- Functional categories (arc labels)
- Possibly some structural categories (parts-of-speech)

Phrase structure explicitly represent
- Phrases (non-terminal nodes)
- Structural categories (non-terminal labels)
- Possibly some functional categories (grammatical functions)

Dependency parser categories & Covington's algorithms

- Dependency based parsers can be broadly categorized into:
  - Grammar driven approaches
    - Parsing using grammars.
  - Data driven approaches
    - Parsing by training on annotated/un-annotated data.
- These approaches are not mutually exclusive

- Incremental parsing in $O(n^2)$ time by trying to link each new word to each preceding one [*Covington 2001*]:
  - PARSE($x = (w_1, \ldots, w_n)$)
    1. **for** $i = 1$ **up to** n
    2. **for** $j = i - 1$ **down to** 1
    3. LINK(w_i , w_j)