Text Preprocessing (II)

Text and Web Mining (H6751)

WKW School of Communication and Information

Word Sense Disambiguation

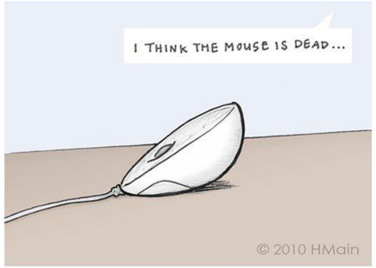
- English words are very often **ambiguous** as to their meaning or reference. [**Homonym**, **polysemy** – word with multiple meanings depending on context.

- For the example "bore," one cannot tell without context even with POS - if the word is referring to a person—"he is a bore"—or a reference to a hole, as in "the bore is not large enough."
  - "bore" is tagged as **NN** (Noun, singular or mass).
  - e.g., may have following features: bore:sense1(**person**), bore:sense2(**hole**)

Word Sense Disambiguation

- Word Sense Disambiguation – identifying the correct sense or semantics of a word based on its usage (**context**)

I killed the mouse while working on my report.



OR

- There are no algorithms that can completely disambiguate a text. Sometimes, humans can't too!

- How can we perform word sense disambiguation with limited context?

Word Sense Disambiguation

- Let's disambiguate "**bank**" in this sentence:

  *"The **bank** can guarantee deposits will eventually cover future tuition costs because it invests in adjustable-rate mortgage securities."*

- Given the following two WordNet senses:

$bank^1$	Gloss:	a financial institution that accepts deposits and channels the money into lending activities
	Examples:	"he cashed a check at the bank", "that bank holds the mortgage on my home"
$bank^2$	Gloss:	sloping land (especially the slope beside a body of water)
	Examples:	"they pulled the canoe up on the bank", "he sat on the bank of the river and watched the currents"

The Simplified Lesk algorithm

Choose sense with **most word overlap** between gloss and context
(not counting function words – word that expresses grammatical or structural relationship with other words in a sentence)

"The **bank** can guarantee deposits will eventually cover future tuition costs because it invests in adjustable-rate mortgage securities."

bank[1]	Gloss:	a financial institution that accepts deposits and channels the money into lending activities
	Examples:	"he cashed a check at the bank", "that bank holds the mortgage on my home"
bank[2]	Gloss:	sloping land (especially the slope beside a body of water)
	Examples:	"they pulled the canoe up on the bank", "he sat on the bank of the river and watched the currents"

Word Sense Disambiguation

- Performs the classic Lesk algorithm for Word Sense Disambiguation (http://www.nltk.org/howto/wsd.html).
  - Given an ambiguous word and the context in which the word occurs, Lesk returns a **Synset** with the highest number of overlapping words between the context sentence and different definitions from each Synset.

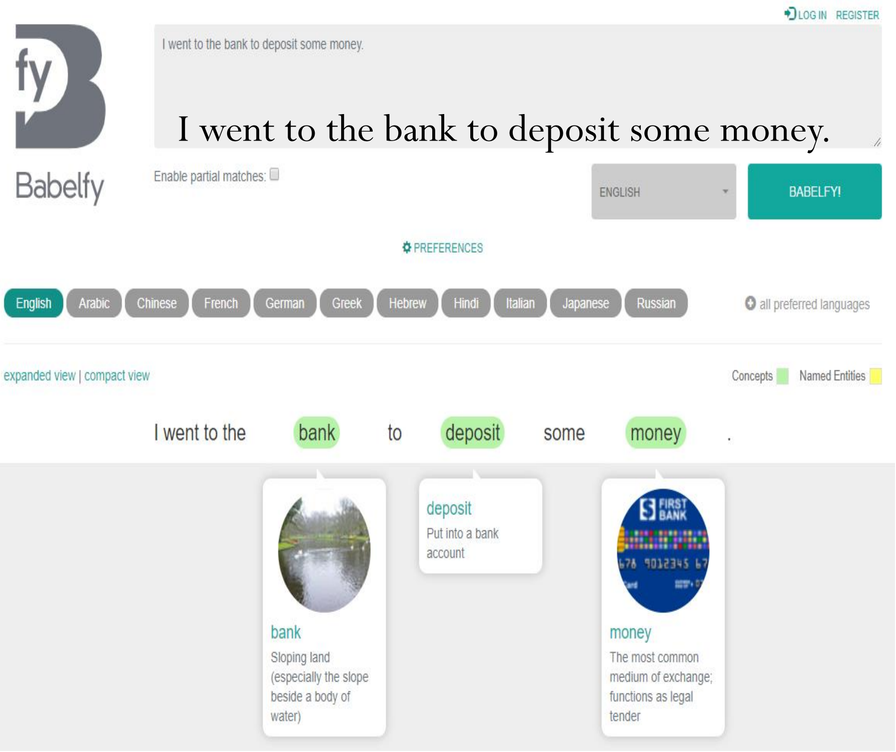
```
>>> from nltk.wsd import lesk
>>> sent = ['I', 'went', 'to', 'the', 'bank', 'to', 'deposit', 'money', '.']

>>> print(lesk(sent, 'bank', 'n'))
Synset('savings_bank.n.02')

>>> print(lesk(sent, 'bank'))
Synset('savings_bank.n.02')
```

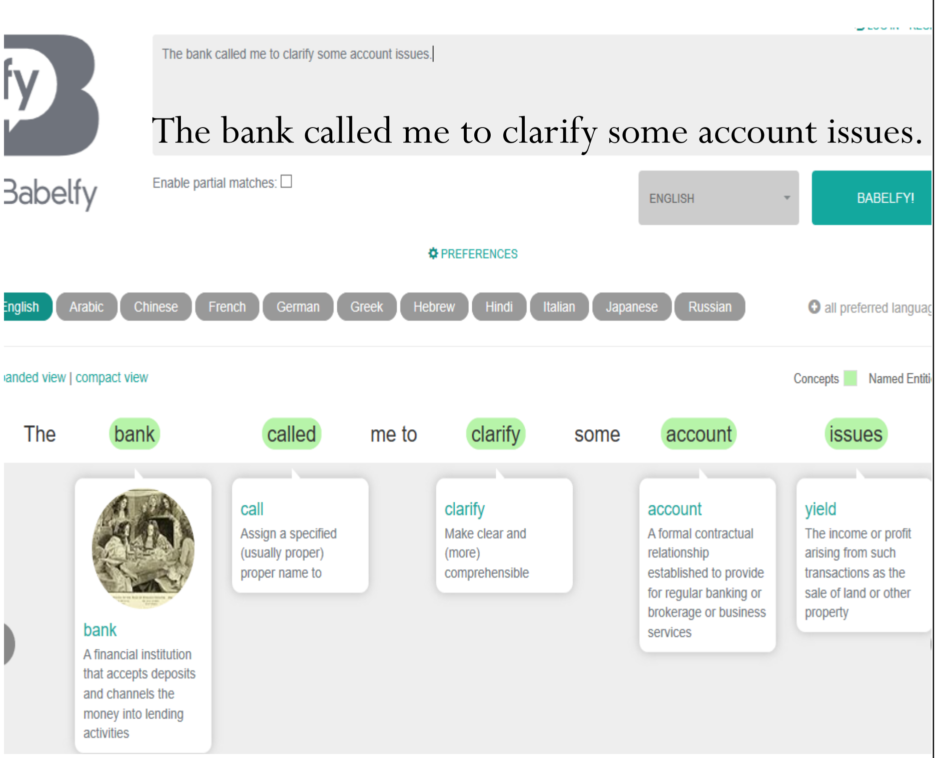
```
from nltk.corpus import wordnet as wn
for ss in wn.synsets('bank'):
    print(ss, ss.definition())
```

```
Synset('bank.n.01') sloping land (especially the slope beside a body of water)
Synset('depository_financial_institution.n.01') a financial institution that accepts deposits and channels the money into lending activities
Synset('bank.n.03') a long ridge or pile
Synset('bank.n.04') an arrangement of similar objects in a row or in tiers
Synset('bank.n.05') a supply or stock held in reserve for future use (especially in emergencies)
Synset('bank.n.06') the funds held by a gambling house or the dealer in some gambling games
Synset('bank.n.07') a slope in the turn of a road or track; the outside is higher than the inside in order to reduce the effects of centrifugal force
Synset('savings_bank.n.02') a container (usually with a slot in the top) for keeping money at home
Synset('bank.n.09') a building in which the business of banking transacted
Synset('bank.n.10') a flight maneuver; aircraft tips laterally about its longitudinal axis (especially in turning)
Synset('bank.v.01') tip laterally
Synset('bank.v.02') enclose with a bank
Synset('bank.v.03') do business with a bank or keep an account at a bank
Synset('bank.v.04') act as the banker in a game or in gambling
Synset('bank.v.05') be in the banking business
Synset('deposit.v.02') put into a bank account
Synset('bank.v.07') cover with ashes so to control the rate of burning
Synset('trust.v.01') have confidence or faith in
```

Word Sense Disambiguation

- Uses **Babelfy** for Word Sense Disambiguation (WSD) http://babelfy.org/).
  - Considered as a state-of-the-art system based on the BabelNet multilingual semantic network for multilingual Word Sense Disambiguation and Entity Linking.





- Unless a project (e.g. Q&A system, Information Retrieval) requires word sense disambiguation, it is best to proceed without such a step.

BeautifulSoup

- It's difficult to decipher textual content in web page due to unnecessary HTML tags.

```
sample = """<h1>Title Goes Here</h1>

<b>Bolded Text</b>
<i>Italicized Text</i>

<img src="this should all be gone"/>
<a href="this will be gone, too">But this will still be here!</a>

I run. He ran. She is running. Will they stop running?
I talked. She was talking. They talked to them about running. Who ran
```

- The BeautifulSoup library provides functions to remove tags with ease.

```
def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

sample = strip_html(sample)
print(sample)
```

```
Title Goes Here
Bolded Text
Italicized Text

But this will still be here!

I run. He ran. She is running. Will they stop running?
I talked. She was talking. They talked to them about running. Who ran to the ta
lking runner?
```

Expanding Contractions

- Contractions are shortened versions of words. Avoided in formal writing but used quite extensively in informal communications.
- Problem for text mining – contractions represent same meaning of contracted words. Need for normalization.
- **Contraction Maps** to match corresponding versions of words. "**Contractions**" library available in Python.

```
CONTRACTION_MAP = {
    "ain't": "is not",
    "aren't": "are not",
    "can't": "cannot",
    "can't've": "cannot have",
      .
      .
      .
    "you'll've": "you will have",
    "you're": "you are",
    "you've": "you have"
```

```
!pip install contractions
import contractions

text = "I can't go to the movies. We don't want to buy the books."

sample = contractions.fix(text)  # e.g., can't -> cannot; don't -> do not
print(sample)
```

```
I can not go to the movies. We do not want to buy the books.
```

Regular expressions

- Regular expressions (aka regexes) create string patterns and use them for searching and substituting specific pattern matches in text.
  - How to search for Woodchuck, woodchuck, Woodchucks, woodchucks

Operator	Function
^	matches the start of the string. E.g., **^The** matches string that start with **The**
$	matches the end of the string. E.g., **end$** matches string that ends with **end**
Braces { }	Indicate range of preceding occurrences. E.g., **ab**{2} **=> abb, ab**{2,3} **=> abb, abbb**
Square Bracket [...]	matches any **one** of the set of characters in [ ]. E.g., **[ab]** matches a or b. Caret ^ appears first in [ ] negates pattern. **[^ab]** => matches anything except a or b
Parentheses ( )	captures **all** the characters within ( ) as a group. E.g., **(the)** matches "**the**" in "**the**n"

Pattern	Matches
`[wW]oodchuck`	Woodchuck, woodchuck
`(456)`	**456** in 123**456**7890

Regular expressions

Representation Operators

Operator	Function
.	matches a single character. E.g., **a.e** matches **ate**, **a1e, a@e**
\w	matches a letter or digit or underbar => [a-zA-Z0-9_]. **\W** matches any non **\w**
\s	matches a single whitespace character. **\S** matches non-whitespace character.
\d	matches a single digit => [0-9]

Repetition Operators

<table>
<tr><th>Operator</th><th>Function</th></tr>
<tr><td>*</td><td>matches zero or more cases of the previous mentioned regex before the * symbol in the pattern. E.g., ab* matches “ac”, “abc”, “abbc”</td></tr>
<tr><td>+</td><td>matches one or more cases of the previous mentioned regex before the + symbol in the pattern. E.g., ab+ matches “ab”, “abbc” but not “ac”</td></tr>
<tr><td>?</td><td>matches zero or one case of the previous mentioned regex before the ? symbol in the pattern. E.g., ab? matches “ac”, “abc” but not “abbc”</td></tr>
<tr><td>|</td><td>OR operator. E.g., a|b matches a or b</td></tr>
</table>

Regular Expressions

- The "**re**" Python module is used for string searching and manipulation
  - **re.match(pattern, string):** This method is used to match pattern at the **beginning** of string.

```
>> re.match('C', 'IceCream').group()
>> None
```

```
>> re.match('C', 'Cake').group()
>> C
```

  - **re.search():** This method is used to match patterns occurring at **any position** in the string.

```
>> re.search('cookie', 'Cake and cookie').group()
>> 'cookie'
```

  - **re.findall():** This method returns all non-overlapping matches of the specified regex pattern in the string.

```
email_address = "Please contact us at: support@abc.com, sales@abc.com
>> results  = re.findall('[\w]+@[\w.]+', email_address )
```

matches letter

Returns ["support@abc.com", sales@abc.com].

Part-of-Speech Tagging

- If the text mining goal is specific, say recognizing names of **people**, **places**, and **organizations**, it is usually desirable to perform additional linguistic analyses of the text and extract more sophisticated features.
  - E.g., San Francisco: San/NNP Francisco/NNP (NNP: Proper noun, singular)
- In English, some analyses may use as few as six or seven categories and others nearly one hundred.
- Most English grammars would have as a minimum **noun, verb, adjective, adverb, preposition,** and **conjunction**.
- POS can be used for **feature reduction**, e.g., use only verb, adjective, and adverb for sentiment classification.
- Distribution of POS can be used for author, gender, and document genre (formal vs. informal) classification.

Part-of-Speech Tagging

- A set of 36 categories, constructed from the Wall Street Journal corpus is used in the **Penn Tree Bank**
- A **tree bank** is a parsed text corpus that annotates sentence structure, such as POS and phrases.
- Almost all POS taggers have been trained on the Wall Street Journal corpus available from LDC (Linguistic Data Consortium, www.ldc.upenn.edu)

Number	Tag	Description
1.	CC	Coordinating conjunction
2.	CD	Cardinal number
3.	DT	Determiner
4.	EX	Existential *there*
5.	FW	Foreign word
6.	IN	Preposition or subordinating conjunction
7.	JJ	Adjective
8.	JJR	Adjective, comparative
9.	JJS	Adjective, superlative
10.	LS	List item marker
11.	MD	Modal
12.	NN	Noun, singular or mass
13.	NNS	Noun, plural
14.	NNP	Proper noun, singular
15.	NNPS	Proper noun, plural
16.	PDT	Predeterminer
17.	POS	Possessive ending
18.	PRP	Personal pronoun
19.	PRP$	Possessive pronoun
20.	RB	Adverb
21.	RBR	Adverb, comparative
22.	RBS	Adverb, superlative
23.	RP	Particle
24.	SYM	Symbol
25.	TO	*to*
26.	UH	Interjection
27.	VB	Verb, base form
28.	VBD	Verb, past tense
29.	VBG	Verb, gerund or present participle
30.	VBN	Verb, past participle
31.	VBP	Verb, non-3rd person singular present
32.	VBZ	Verb, 3rd person singular present
33.	WDT	Wh-determiner
34.	WP	Wh-pronoun
35.	WP$	Possessive wh-pronoun
36.	WRB	Wh-adverb

Part-of-Speech Tagging

- **The Stanford Parser:** a statistical parser
  An implementation in Java: http://nlp.stanford.edu/software/lex-parser.shtml

- **Online Stanford Parser**
  http://nlp.stanford.edu:8080/parser/

Note:
PRP$: Possessive pronoun
RB: Adverb
VBZ: Verb, 3rd person singular present

Phrase Recognition

- Once the tokens of a sentence have been assigned **POS tags**, the next step is to group individual tokens (one being the main or head word) into units, generally called **phrases**.
- **Noun phrases** – act as a subject or object to a verb. Consist of a **noun** or pronoun [**head word**], and **dependent** words before or after the head. [e.g., "the moon"]
- **Verb phrases** – consist of a main verb [MV] alone, or a main verb plus any modal [MO] and/or auxiliary [AUX] verbs.

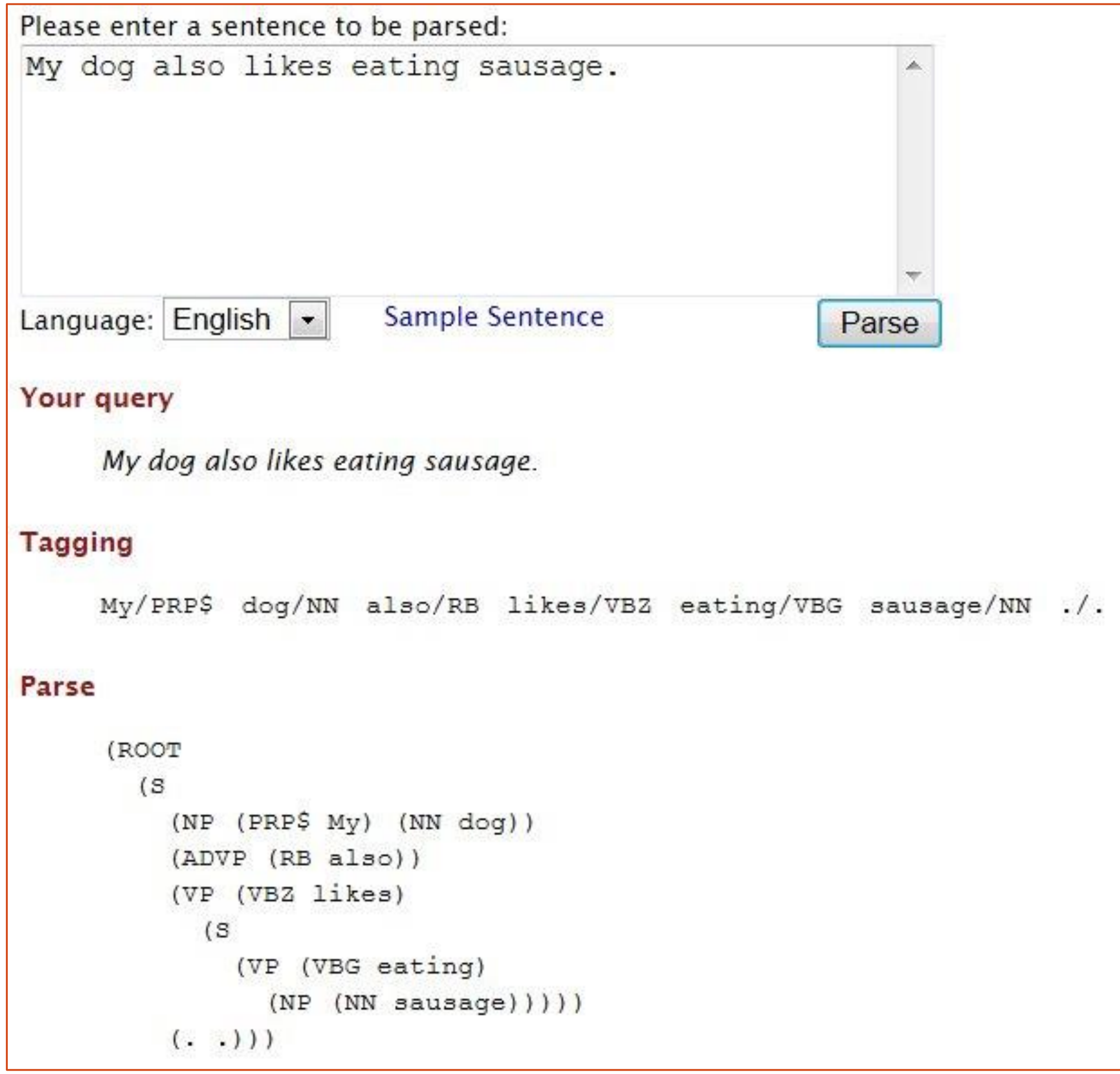
[MV]	[MO] [AUX] [AUX] [MV]
We all laughed.	Tony might have been waiting outside for you

- **Prepositional phrases –** consists of a preposition [head] (e.g., to, with), its object (noun/pronoun), and object modifiers (article/adjective). Article – determiner that precedes a noun.

  e.g., The boy ***with her*** is her son

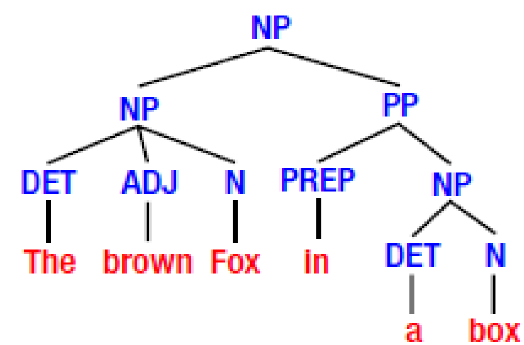
Phrase Recognition

- **The Stanford Parser: online parser** http://nlp.stanford.edu:8080/parser/



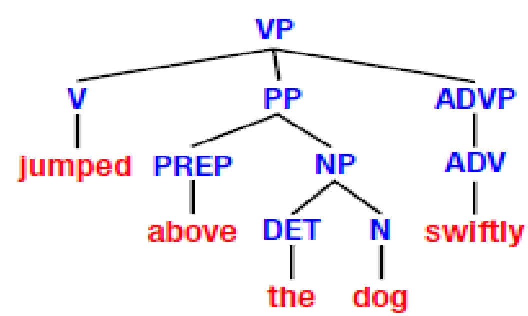
Parsing (Phrase Structure Rules)

- Parsing - step of producing **a full parse of a sentence** based on Phrase Structure Rules
  - Generic rule denotes binary division for a sentence or a clause [S → AB], structure S consists of constituents A and B with the order A followed by B.
  - S → NP VP => sentence or clause divided into the subject (NP) and predicate (VP)
  - NP → [DET][ADJ]**N**[PP] => Noun as the head word optional Determinants, Adjectives, and Prepositional

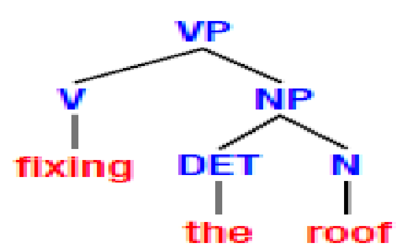


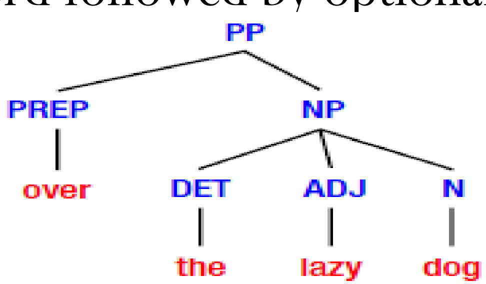
  - VP → **V** [VP][NP][PP][ADJP][ADVP] => Verb head

word followed by optional VP, NP, PP, Adjective Phrase or Adverbial Phrase.



  - PP → PREP [NP] => preposition as head word followed by optional NP

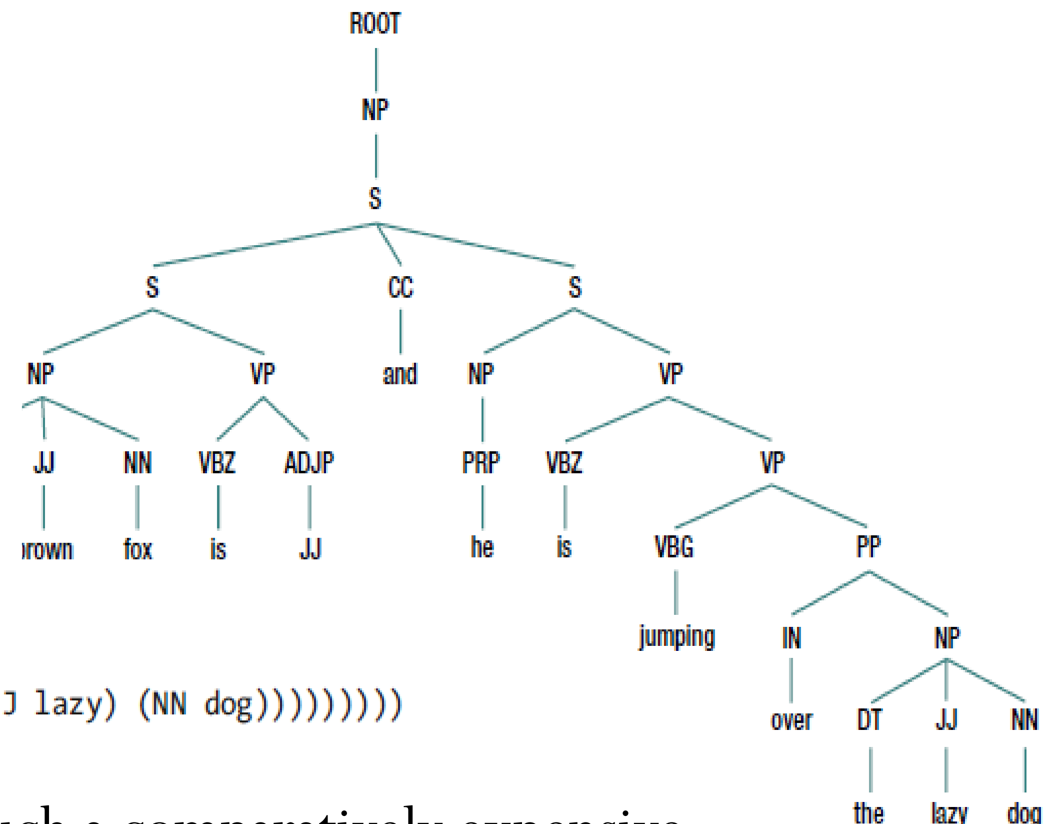




Parsing (Phrase Structure Rules)

Sentence – The brown fox is quick and he is jumping over the lazy dog.

```
(ROOT
  (NP
    (S
      (S
        (NP (DT The) (JJ brown) (NN fox))
        (VP (VBZ is) (ADJP (JJ quick))))
      (CC and)
      (S
        (NP (PRP he))
        (VP
          (VBZ is)
          (VP
            (VBG jumping)
            (PP (IN over) (NP (DT the) (JJ lazy) (NN dog))))))))
```



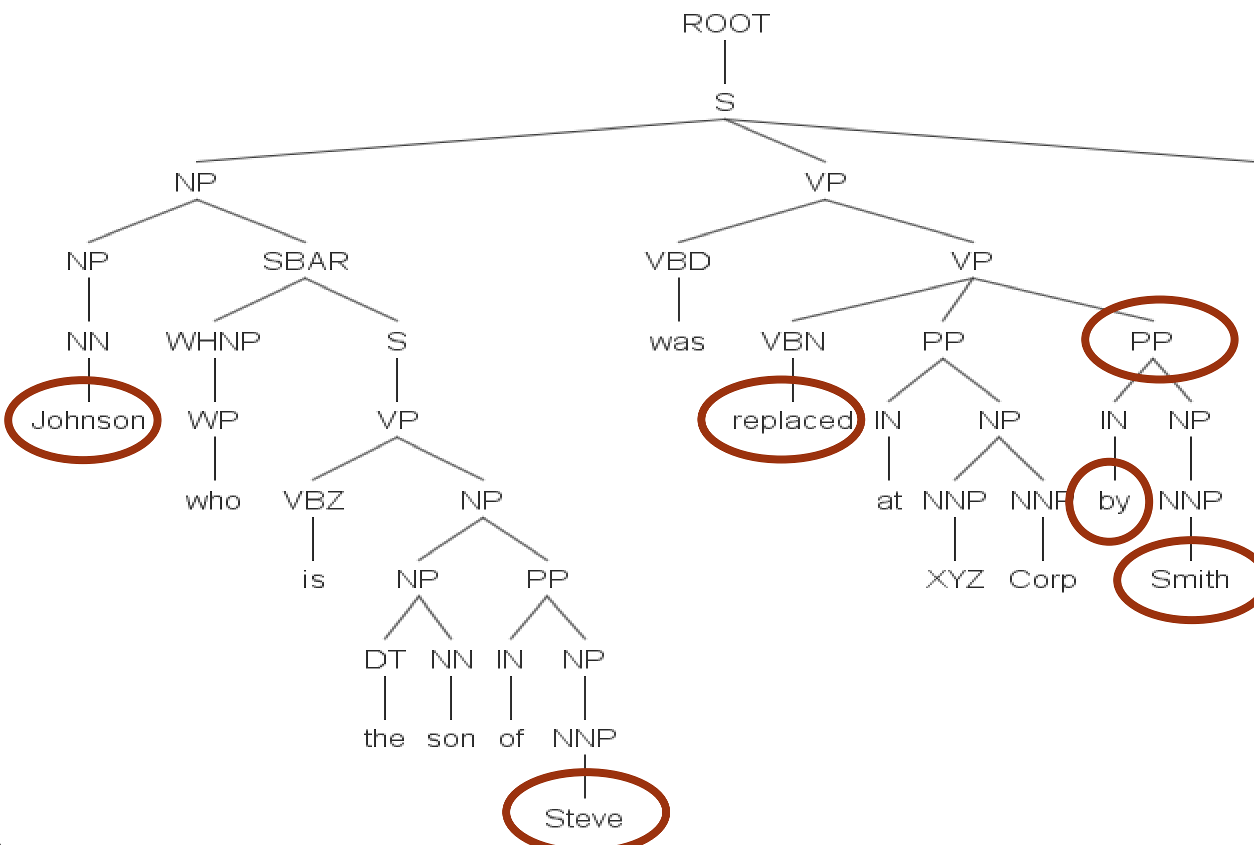
- The reason for considering such a comparatively expensive process is that it provides **detailed syntactic relationships information** that phrase identification cannot provide.

Parsing

- Consider the sentence "Johnson who is the son of Steve was replaced at XYZ Corp by Smith."

Phrase structure tree

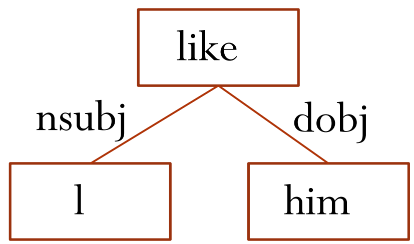
Using the Parse Tree, machine can infer that Johnson was replaced by Smith (correct) and not Steve was replaced by Smith (wrong).

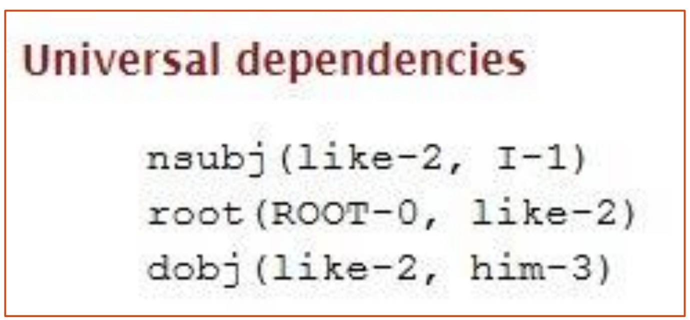



```
(ROOT
  (S
    (NP
      (NP (NNP Johnson))
      (SBAR
        (WHNP (WP who))
        (S
          (VP (VBZ is)
            (NP
              (NP (DT the) (NN son))
              (PP (IN of)
                (NP (NNP Steve))))))))
    (VP (VBD was)
      (VP (VBN replaced)
        (PP (IN at)
          (NP (NNP XYZ) (NNP Corp)))
        (PP (IN by)
          (NP (NNP Smith)))))
    (. .)))
```

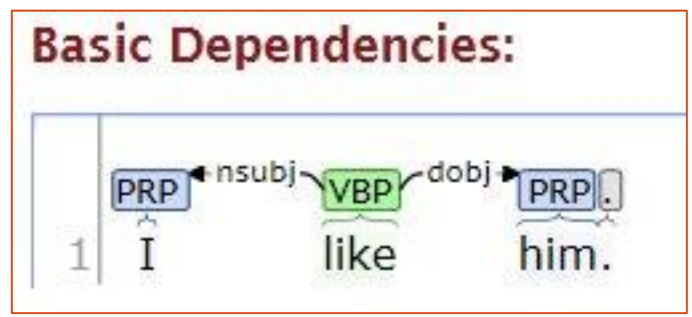
Parsing - how to process a Parse Tree?

- **Universal dependencies** (i.e. grammatical relations; evolved out of Stanford Dependencies) from Stanford Parser: "I like him."





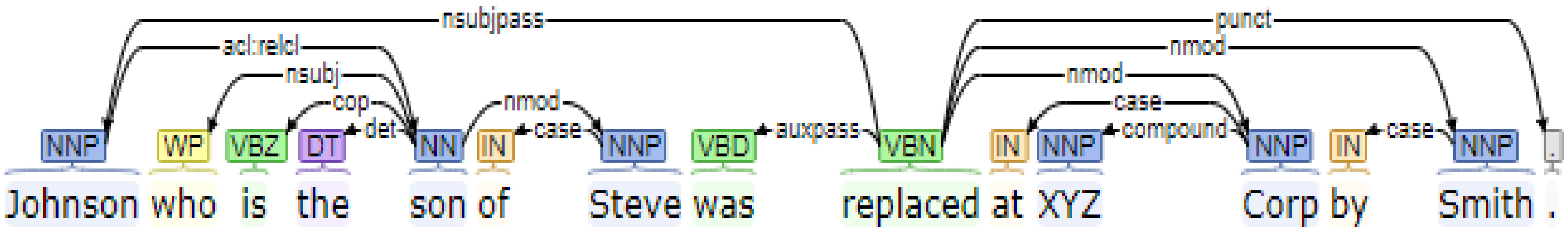
Output from Stanford Parser



Output from Standford CoreNLP

Parsing

- Universal dependencies (i.e. grammatical relations) from Standford CoreNLP: "Johnson who is the son of Steve was replaced at XYZ Corp by Smith."




```
(ROOT
  (S
    (NP
      (NP (NNP Johnson))
      (SBAR
        (WHNP (WP who))
        (S
          (VP (VBZ is)
            (NP
              (NP (DT the) (NN son))
              (PP (IN of)
                (NP (NNP Steve))))))))
    (VP (VBD was)
      (VP (VBN replaced)
        (PP (IN at)
          (NP (NNP XYZ) (NNP Corp)))
        (PP (IN by)
          (NP (NNP Smith)))))
    (. .)))
```

```
nsubjpass(replaced-9, Johnson-1)
nsubj(son-5, who-2)
cop(son-5, is-3)
det(son-5, the-4)
acl:relcl(Johnson-1, son-5)
case(Steve-7, of-6)
nmod(son-5, Steve-7)
auxpass(replaced-9, was-8)
root(ROOT-0, replaced-9)
case(Corp-12, at-10)
compound(Corp-12, XYZ-11)
nmod(replaced-9, Corp-12)
case(Smith-14, by-13)
nmod(replaced-9, Smith-14)
```

Vector Generation for Prediction

- Without any deep analysis of the linguistic content of the documents, we can describe **each document by features that represent the most frequent tokens**.
- Each row is a document, and each column represents a feature.
- Thus, a cell in the spreadsheet is a measurement of a feature (corresponding to the column) for a document (corresponding to a row).

DocID	Apple	Bear	Durian	…	Zoo	Animal?
1	0	3	0	0	2	1
2	1	0	2	0	0	0
…						

- Dictionary (or feature) reduction techniques
  - Local dictionary, removing Stopwords, Frequent words, Feature selection, and Token reduction (stemming and synonyms)

Bag of Words Model

- A vector space model represents unstructured text (or any other data) as numeric vectors, such that each dimension of the vector is a specific feature/attribute.
- BOW represents each document as a numeric vector where each dimension is a specific word from the corpus and the value could be its frequency in the document, occurrence (denoted by 1 or 0), or even weighted values.
- Represented literally as a bag of its own words, disregarding word order, sequences, and grammar.

Bag of Words Example

Document 1

The quick brown fox jumped over the lazy dog's back.

Document 2

Now is the time for all good men to come to the aid of their party.

Term	Document 1	Document 2
aid	0	1
all	0	1
back	1	0
brown	1	0
come	0	1
dog	1	0
fox	1	0
good	0	1
jump	1	0
lazy	1	0
men	0	1
now	0	1
over	1	0
party	0	1
quick	1	0
their	0	1
time	0	1

Stopword List

for
is
of
the
to

CountVectorizer

- The CountVectorizer class can **produce a bag-of-words representation** from a string or file. (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)

class sklearn.feature_extraction.text.CountVectorizer(*input='content'*, *encoding='utf8'*, *decode_error='strict'*, *strip_accents=None*, ***lowercase=True***, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, ***token_pattern='(?u)\b\w\w+\b'***, *ngram_range=(1, 1)*, *analyzer='word'*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.int64'>*)

- By default, CountVectorizer converts the characters in the documents to **lowercase**, and **tokenizes** the documents using a regular expression that splits strings on whitespace and extracts sequences of characters that are two or more characters in length.

CountVectorizer

- The documents in the corpus can be represented by **feature vectors**:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
>>> 'UNC played Duke in basketball',
>>> 'Duke lost the basketball game'
>>> ]
>>> vectorizer = CountVectorizer()
>>> print vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
[[1 1 0 1 0 1 0 1]
[1 1 1 0 1 0 1 0]]
{u'duke': 1, u'basketball': 0, u'lost': 4, u'played': 5, u'game': 2, u'unc': 7,
u'in': 3, u'the': 6}
```

	basketball	duke	game	in	lost	played	the	unc
D1	1	1	0	1	0	1	0	1
D2	1	1	1	0	1	0	1	0

Bag of N-Grams Model

- A word is just a single token, often known as a *unigram* or *1-gram*. Bag of Words model doesn't consider the order of words. But what if we also wanted to take into account phrases or collection of words that occur in a sequence?

- An N-gram is basically a collection of word tokens from a text document such that these tokens are contiguous and occur in a sequence.

- Bi-gram indicates n-grams of order 2 (two words e.g., [beautiful sky], [sky today]), tri-grams order 3 (three words e.g., [beautiful sky today]), and so on.

CountVectorizer - ngrams

- You can set the n-gram to 1,2 to get unigrams as well as bigrams

>> bv = CountVectorizer(**ngram_range=(2,2)**) #extract bigrams
>> bv_matrix = bv.fit_transform(corpus)

	bacon eggs	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	breakfast sausages	brown fox	dog lazy
0	0	0	0	1	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0
3	1	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	1	1
6	0	0	1	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	1

TF-IDF Model

- Potential problems with BOW model – words (terms) that occur frequently across all documents tend to overshadow other terms.
- **Bag-of-words** feature vectors do not encode **grammar**, **word order**, or **frequencies** of words. The frequency with which a word appears in a document could indicate the extent to which the document relates to the word (**term weights**).
  - Especially words that don't occur as frequently, but might be interesting and effective as features.

Term Frequency

- **Term frequency (TF)** – raw frequency of a term in document $tf(t,d)$
- Words might appear with the same frequency in two documents, but the documents could be dissimilar if one is **many times larger** than the other.
  - "I love **apple**" vs "I love fruits especially **apples**, **oranges**, **pears**, …"
- Mitigate problem by using **normalized** term frequency weights.

$$tf(t,d) = \frac{f(t,d)}{||x||}$$

Term vector for d = tf("I") + tf("love") + tf("apple")

$f(t, d)$ is the frequency of term in document d and x is the L2 norm of the count vector of terms in the document => $\sqrt{{x_1}^2 + {x_2}^2 + {x_3}^2 + \ \ldots}$

Term Frequency-Inverse Document Frequency

- **TF-IDF** is used instead of the raw frequencies of a term to **scale down impact of frequently occurring terms** in a given corpus, which are **less informative** than features that occur in a small fraction of the corpus.
- **Inverse Document Frequency (IDF)** measures how rare or common a word is in a corpus => more rare more significant.

Effect of adding "1" to idf is that terms with zero idf, i.e., terms that occur in all documents (df=n) leading to log(1)=0, will not be entirely ignored.

Standard idf definition

- $(\boldsymbol{i})\ idf(d,t) = \log\left[\frac{(n)}{(df(d,t))}\right]$ $\quad (\boldsymbol{ii})\ idf(d,t) = \log\left[\frac{(1+n)}{(1+df(d,t))}\right] + 1$

Modified idf definition

Effect of adding "1" to df is that terms with zero df, i.e., terms that occur in no documents (df=0) will not lead to division by zero.

- **N** is the total number of documents in the corpus and ***df(d,t)*** is the number of documents in the corpus that contain the term **t**. Log is used when sublinear_tf=True (**Default is LN**)

Term Frequency-Inverse Document Frequency

- A term's **TF-IDF** value is the product of its term frequency and inverse document frequency: ***TF-IDF = TF x IDF.***

$$\textit{TF-IDF} = tf(t,d).\,idf(d,t)$$

$$= \frac{f(t,d)}{||x||}\,.\log\left[\frac{(1+n)}{(1+df(d,t))}\right] + 1$$

- The resulting tf-idf vectors are then normalized by the Euclidean norm based on the tf-idf score of each term:

$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + \dots + {v_n}^2}}$$

CountVectorizer and TfidfTransformer

- CountVectorizer - creation of feature vectors that encode the frequencies of words (term frequencies)

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
corpus = [
    'The dog ate a sandwich and I ate a sandwich',
    'The wizard transfigured a sandwich'
]
vectorizer = CountVectorizer(stop_words='english')
transformer = TfidfTransformer(use_idf=False)
transformerIDF = TfidfTransformer(use_idf=True)
X = vectorizer.fit_transform(corpus)
print('Count vectors:\n', X.todense())
print('Vocabulary:\n', vectorizer.vocabulary_)
print('TF vectors:\n', transformer.fit_transform(X).todense())
print('TF-IDF vectors:\n', transformerIDF.fit_transform(X).todense())
```

```
Count vectors:
 [[2 1 2 0 0]
 [0 0 1 1 1]]
Vocabulary:
 {'dog': 1, 'ate': 0, 'sandwich': 2, 'wizard': 4, 'transfigured': 3}
TF vectors:
 [[ 0.66666667  0.33333333  0.66666667  0.          0.        ]
 [ 0.          0.          0.57735027  0.57735027  0.57735027]]
TF-IDF vectors:
 [[ 0.75458397  0.37729199  0.53689271  0.          0.        ]
 [ 0.          0.          0.44943642  0.6316672   0.6316672 ]]
```

- **TfidfTransformer** returns **TF-IDF weight** when the **use_idf** keyword argument is True (default value) (returns **TF weight** if use_idf set to False)

TF, IDF, and TF-IDF calculations

[ate, doc1] tf = $2/\text{sqrt}(2^2+1^2+2^2) = 2/\text{sqrt}(9) = \mathbf{0.667}$

[wizard, doc2] tf = $1/\text{sqrt}(1^2+1^2+1^2) = 1/\text{sqrt}(3) = \mathbf{0.55735}$

```
Count vectors:
 [[2 1 2 0 0]
 [0 0 1 1 1]]
Vocabulary:
 {'dog': 1, 'ate': 0, 'sandwich': 2, 'wizard': 4, 'transfigured': 3}
TF vectors:
 [[0.66666667 0.33333333 0.66666667 0.         0.        ]
 [0.         0.         0.57735027 0.57735027 0.57735027]]
TF-IDF vectors:
 [[0.75458397 0.37729199 0.53689271 0.         0.        ]
 [0.         0.         0.44943642 0.6316672  0.6316672 ]]
```

[ate, doc1] $tfidf = tf * \boldsymbol{idf} = 0.667 * 1.405 = \mathbf{0.937}$

$$\boldsymbol{idf(d,t)} = \ln\left[\frac{(1+n)}{\left(1+df(d,t)\right)}\right] + 1$$

$$= \ln\left[\frac{(1+2)}{(1+1)}\right] + 1$$

$$= \mathbf{1.405}$$

$$\boldsymbol{normalized}\ tfidf = \frac{0.937}{\sqrt{0.937^2+0.468^2+0.667^2}} = \mathbf{0.754}$$

TfidfVectorizer

- Scikit-learn provides a **TfidfVectorizer** class that wraps **CountVectorizer** and **TfidfTransformer**.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
>>>     'The dog ate a sandwich and I ate a sandwich',
>>>     'The wizard transfigured a sandwich'
>>> ]
>>> vectorizer = TfidfVectorizer(stop_words='english')
>>> print vectorizer.fit_transform(corpus).todense()
[[ 0.75458397  0.37729199  0.53689271  0.          0.        ]
 [ 0.          0.          0.44943642  0.6316672   0.6316672 ]]
```

Computes TF-IDF directly

ate

sandwich

- By comparing the TF-IDF weights to the raw term frequencies, we can see that words that are common to many of the documents in the corpus, such as **sandwich**, have been penalized.

Referenced Materials

- Fundamentals of Predictive Text Mining, Sholom M. Weiss, Nitin Indurkhya, and Tong Zhang, Springer.
  - Chapter 2
- Natural Language Processing, Dan Jurafsky and Christopher Manning, http://www.stanford.edu/~jurafsky/NLPCourseraSlides.html
- Machine Learning, Tom M. Mitchell, McGraw-Hill