NLTK Book Summary

Natural Language ToolKit – most popular NLP toolkit of Python.

Source: http://www.nltk.org/book/

Summarized by Ido Greenberg, 2017

Contents

Preface

Book goals:

Goals	Background in arts and humanities	Background in science and engineering
Language analysis	Manipulating large corpora, exploring linguistic models, and testing empirical claims.	Using techniques in data modeling, data mining, and knowledge discovery to analyze natural language.
Language technology	Building robust systems to perform linguistic tasks with technological applications.	Using linguistic algorithms and data structures in robust language processing software.

Software requirements:

Python, NLTK, NLTK-Data, NumPy*, Matplotlib*, Stanford NLP Tools*, NetworkX**, Prover9**

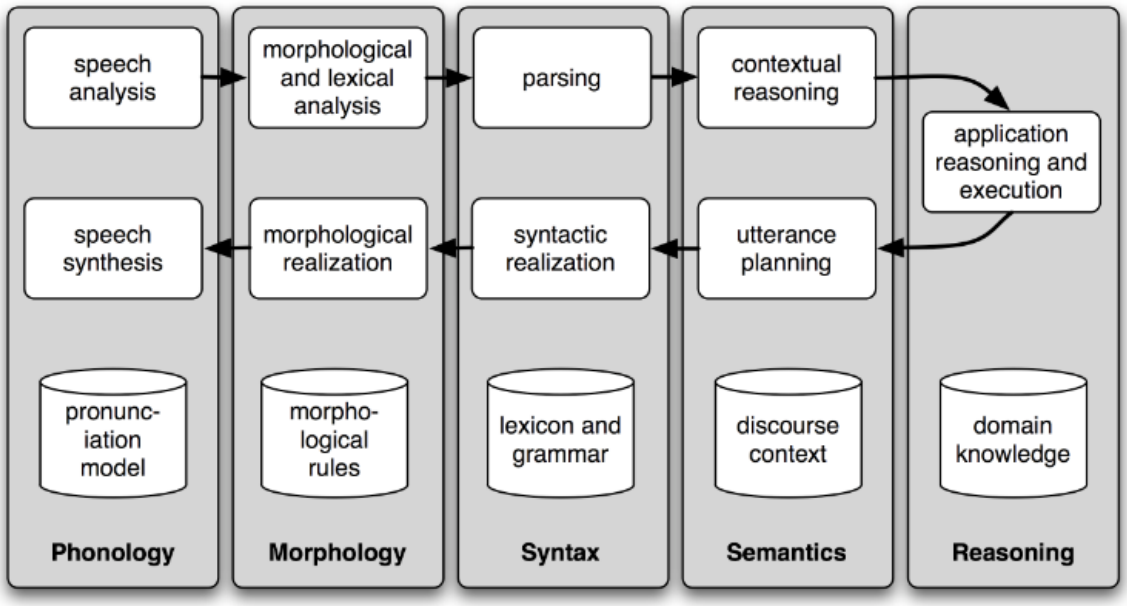
*recommended **optional

NLTK main modules:

Language processing task	NLTK modules	Functionality
Accessing corpora	corpus	standardized interfaces to corpora and lexicons
String processing	tokenize, stem	tokenizers, sentence tokenizers, stemmers
Collocation discovery	collocations	t-test, chi-squared, point-wise mutual information
Part-of-speech tagging	tag	n-gram, backoff, Brill, HMM, TnT
Machine learning	classify, cluster, tbl	decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	chunk	regular expression, n-gram, named-entity
Parsing	parse, ccg	chart, feature-based, unification, probabilistic, dependency
Semantic interpretation	sem, inference	lambda calculus, first-order logic, model checking
Evaluation metrics	metrics	precision, recall, agreement coefficients
Probability and estimation	probability	frequency distributions, smoothed probability distributions
Applications	app, chat	graphical concordancer, parsers, WordNet browser, chatbots
Linguistic fieldwork	toolbox	manipulate data in SIL Toolbox format

Chapter 1: Language Processing and Python

Architecture of spoken dialogue system:



Simple Pipeline Architecture for a Spoken Dialogue System: Spoken input (top left) is analyzed, words are recognized, sentences are parsed and interpreted in context, application-specific actions take place (top right); a response is planned, realized as a syntactic structure, then to suitably inflected words, and finally to spoken output; different types of linguistic knowledge inform each stage of the process.

Chapter 2: Accessing Text Corpora and Lexical Resources

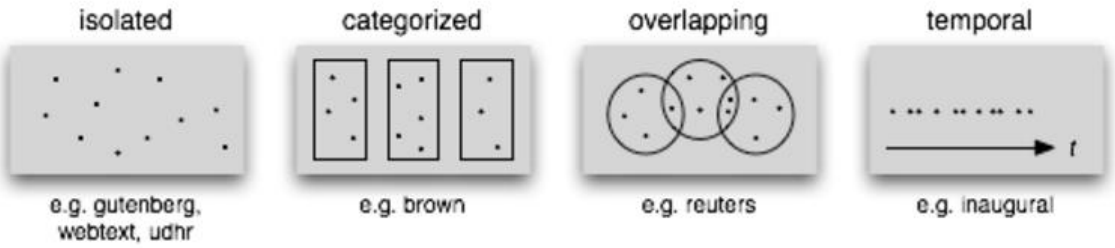
Text corpus = large body of text.

Main corpora in NLTK:

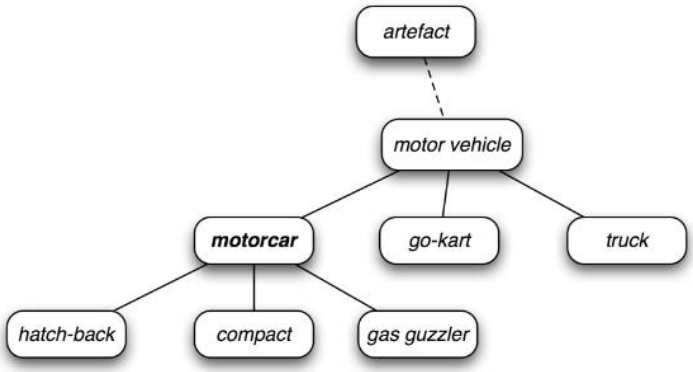
1. Gutenberg: free electronic books
2. Webtext: informal language from 4 sources including Pirates of the Caribbean!
3. **Nps_chat**: anonymous chat collections from public chatrooms (2006)

4. Brown: old collection of Brown University, containing texts of various genres. Useful for Stylistics - studying differences between genres
5. Reuters: 1.3M-words news documents classified into 90 topics and grouped into training & test sets
6. Inaugural: 55 texts of presidential addresses(?)
7. **Words**: the operation system's vocabulary file used by some spell checkers
8. Stopwords: common words, separated by language, which are often omitted before semantic analysis
9. Names: first names separated by gender
10. Cmudict: pronountion dictionary: how to pronounce each word
11. Swadesh: small corpus of the same words in several languages. Contains only tens of words
12. Toolbox: popular multi-language lexical corpus with complex structure
13. **Wordnet**: synonyms corpus

Common corpora structures:



Wordnet corpus hierarchy demonstration:



For more corpora, see http://www.nltk.org/data for instructions and http://www.nltk.org/nltk_data/ for full list; specifically, there seem to be in the list 2 chat dbs – one of public chat and one not well-documented from the 80's, and anyway no explicit private conversations db.

Chapter 3: Processing Raw Text

Unicode decoding & encoding:

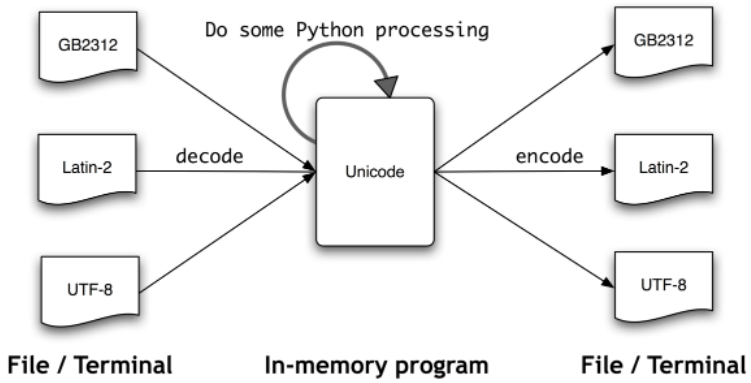


***Figure 3.3**: Unicode Decoding and Encoding*

Basic Regular Expression Meta-Characters:

Operator	Behavior
`.`	Wildcard, matches any character
`^abc`	Matches some pattern *abc* at the start of a string
`abc$`	Matches some pattern *abc* at the end of a string
`[abc]`	Matches one of a set of characters
`[A-Z0-9]`	Matches one of a range of characters
`ed|ing|s`	Matches one of the specified strings (disjunction)
`*`	Zero or more of previous item, e.g. `a*`, `[a-z]*` (also known as *Kleene Closure*)
`+`	One or more of previous item, e.g. `a+`, `[a-z]+`
`?`	Zero or one of the previous item (i.e. optional), e.g. `a?`, `[a-z]?`
`{n}`	Exactly *n* repeats where n is a non-negative integer
`{n,}`	At least *n* repeats
`{,n}`	No more than *n* repeats
`{m,n}`	At least *m* and no more than *n* repeats
`a(b|c)+`	Parentheses that indicate the scope of the operators

Regexes symbols & example (which did not work for me…):

Symbol	Function
`\b`	Word boundary (zero width)
`\d`	Any decimal digit (equivalent to `[0-9]`)
`\D`	Any non-digit character (equivalent to `[^0-9]`)
`\s`	Any whitespace character (equivalent to `[ \t\n\r\f\v]`)
`\S`	Any non-whitespace character (equivalent to `[^ \t\n\r\f\v]`)
`\w`	Any alphanumeric character (equivalent to `[a-zA-Z0-9_]`)
`\W`	Any non-alphanumeric character (equivalent to `[^a-zA-Z0-9_]`)
`\t`	The tab character
`\n`	The newline character

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Word/sentence **segmentation** = separate text into words/sentences, possibly when no structural separation is available (e.g. separate to words based on dictionary rather than spaces – by representation of the separation with a binary sequence of space/non-space, and calculate the probability of various segmentations). That's a generalization of **tokenization** problem (which is just separation to words based on spaces and other separators).

Lemmatization = mapping various forms of a word (e.g. appeared, appears) to the canonical or citation form of the word, AKA the ***lexeme*** or ***lemma*** (e.g. appear).

Chapter 4: Writing Structured Programs

Basic concepts in Python programming.

Important technical note: vocabulary should be represented by a set rather than list, since membership test ("word in vocab") is much faster this way. Times can be measured by Timer.timeit().

Dynamic Programming was introduced – an efficient way to implement recursion where the same input may repeat numerous times: instead of repeated calculation of the function, we add every new call of the function into a *look-up table* (dictionary) and call the function only when the input is not in the table.

There was also a demonstration of matplotlib, networkx, csv & numpy.

Chapter 5: Categorizing and Tagging Words

Part-of-speech (POS) tagging = classify words into ***lexical categories***, e.g. nouns & verbs.

Universal Part-of-Speech Tagset

Tag	Meaning	English Examples
ADJ	adjective	*new, good, high, special, big, local*
ADP	adposition	*on, of, at, with, by, into, under*
ADV	adverb	*really, already, still, early, now*
CONJ	conjunction	*and, or, but, if, while, although*
DET	determiner, article	*the, a, some, most, every, no, which*
NOUN	noun	*year, home, costs, time, Africa*
NUM	numeral	*twenty-four, fourth, 1991, 14:24*
PRT	particle	*at, on, out, over per, that, up, with*
PRON	pronoun	*he, their, her, its, my, I, us*
VERB	verb	*is, say, told, given, playing, would*
.	punctuation marks	. , ; !
X	other	*ersatz, esprit, dunno, gr8, univeristy*

Using tagging and conditional frequency analysis, one can find, for example, the "most something" verbs (something corresponds to the condition of the conditional frequency analysis).

Popular tagged corpora in NLTK: brown, nps_chat, conll2000, treebank.

Smart searches within such corpora – e.g. "the ADJ man" – can be done using the GUI of nltk.app.concordance().

Use defaultdict to generate updating dict:

```
>>> my_dictionary = defaultdict(function to create default value)
>>> for item in sequence:
...       my_dictionary[item_key] is updated with information about item
```

Tagsets are defined differently according to the goal of the tagging. Brown corpus, for example, uses quite high-resolution tagging which expresses some morphosyntactic information, e.g:

Form	Category	Tag
go	base	VB
goes	3rd singular present	VBZ
gone	past participle	VBN
going	gerund	VBG
went	simple past	VBD

Automatic tagging without context (chapter 5.4)

Approaches (performance is tested wrt the same data as):

- 0-order: just assign the most common tag (NN) to all tokens / 13%
- Regex tagging: ***Morphological*** tagger (i.e. exploits the internal structure of the word), using rules such as ".*ed$"→VBD / 20%
- Lookup tagger: remember the most common tags of the 100 most common tokens / 46% (performance rises with the size of the table)

A new tagger can be tested wrt a known test reference tagged corpus named *gold standard* (no link was suggested).

N-gram tagging

- Unigram tagger: as in lookup tagger – assign the most probable tag to each token. Instead of lookup table there's some unexplained training, which seems to be just lookup table of all words in the original text / 93% (81% in generalization test, i.e. separated test set)
- N-Gram tagger: predict tag according to current word and previous N-1 tags. This is a ***syntactic*** tagger, since it's using the context of the word.
    - Requires tokenized sentences in order to avoid sentence-boundary-crossing.
    - Requires lots of data (precision/recall trade-off in information retrieval) (even though using previous tags rather than words makes it better).
- Unknown words tagger (word-invariant tagger): N-Gram tagger which is trained on tagged data where the words are replaced by constant UNK, thus the training is based on previous tags only, hence there's automatic generalization to unknown words. Of course it's useful only as a backoff tagger, since for every word we do need to know the previous tags with high certainty.
- Combined tagger: use precise tagger if there's enough available data (cutoff defines what's enough), otherwise use a backoff tagger.

"We have seen that ambiguity in the training data leads to an upper limit in tagger performance. Sometimes more context will resolve the ambiguity. In other cases however, as noted by (Church, Young, & Bloothooft, 1996), the ambiguity can only be resolved with reference to syntax, or to

world knowledge. Despite these imperfections, part-of-speech tagging has played a central role in the rise of statistical approaches to natural language processing. In the early 1990s, the surprising accuracy of statistical taggers was a striking demonstration that it was possible to solve one small part of the language understanding problem, namely part-of-speech disambiguation, without reference to deeper sources of linguistic knowledge. Can this idea be pushed further? In 7., we shall see that it can."

Morphological (regexes) and syntactic (N-gram) taggers are popular, but semantic (word-meaning based) taggers are not, mainly because semantic are hard to formalize.

Transformation-based tagging

Brill tagger is a kind of transformation-based learner.

Tagger architecture: use unigram tagger, then refine the tagging iteratively using rules of the form "replace all t1 in context C by t2", where the context is usually the previous/following tag, e.g. "replace NN by VB if the previous tag is TO".

- More compact than N-gram tagging, since involves rules rather than huge dictionaries.
- On one hand, the initial guesses are weak, so the contexts are unreliable (compared to the contexts of N-gram models which are the final predictions of the previous words). On the third hand, it permits more general rules, including use of following tags in addition to previous ones.

Training: guess rules (t1,t2,C) and count the benefit of each rule (corrections-wrong modifications).

- Wrt which tagging the training occurs? There can't be corrections wrt the reference tagging. Maybe the training goes on iteratively and greedily from the unigram tagging.
- The training generates **rules** rather than huge tables – and those are **linguistically interpretable**.

Chapter 6: Learning to Classify Text

Standard supervised learning classificatory – we have to encode features of the input, let the trainer learn which ones are useful for the classification, and use validation set in order to check where the classifier fails to generalize and refine features accordingly. Note that beginning with too many features the classifier will just overfit to fit the training data arbitrarily with no generalization ability, hence even the initial features should be chosen wisely.

NLTK supports several classifiers architectures, e.g. ***Naïve Bayes*** and ***Decision Tree*** (which is useful for interpretable learning).

The chapter demonstrates classification of names according to gender; movies reviews according to pos/neg; words according to POS tags (tagging); tokens to end-of-sentence or not (sentence segmentation), **sentences to dialogue acts (based on NPS chat)**, hypotheses and possibly-justifying sentences to T/F (*Recognizing Textual Entailment*)…

Joint classifier is one that applies on groups of input entries together, as in ***sequence/consecutive classifier*** which classifies entries successively, each time using the former prediction as input.

Transformational joint classifiers – as discussed in chapter 5 – allow utilization of information backwards as well. And even sexier than that – ***Hidden Markov Models*** use probabilistic approach to implement consecutive classifier without being greedy and rely on the last hard-prediction. Recommended to read (chapter 6, section 1.7). Also mentioned (but not explained) *Maximum Entropy Markov Models* and *Linear-Chain Conditional Random Field Models*.

Technical note: training via NLTK is programmed in Python, hence slow. "See the NLTK webpage for a list of recommended machine learning packages that are supported by NLTK".

In general, learning models can be either *descriptive* (i.e. just find correlations in data) or *explanatory* (i.e. find abstract concepts that explain patterns in the observed data).

Evaluation tips

- Test set: try to make sure that the least frequent label occurs at least 50 times. Usually 10% is safe enough. If the data consists of clusters (e.g. conversations), in order to test generalization, it's better to take the train & test sets with different clusters.
- Accuracy measure: has to be compared to the frequency of the most common label (e.g. if 95% of the entries are TRUE then it's trivial to have accuracy of 95%...).
- Search task measures: positive label is sparse in the data (e.g. look for docs relevant to specific topic), thus we use the measures: TP, TN, FP (*type I errors*), FN (*Type II errors*); ***Precision***=TP/(TP+FP); ***Recall***=Pd=TP/(TP+FN); ***F-measure***=*F-score*=2*Precision*Recall/(Precision+Recall) (harmonic mean).
- Confusion matrix: matrix indicating how much data of label i were classified as j. Helps to identify common errors.
- Cross validation: divide data into N *folds*, and repeat the train & test process with different fold as test set each time. This allows to have significant evaluation even if every fold size (=test size) is small, which breaks the tradeoff between train & test sizes. It also allows to check the consistency of the evaluation.

Learning methods

Decision tree is just a flowchart that selects labels. It can be seen as hierarchical classification (hence useful, for example, in phylogeny – classification of species to categories). Fundamental terms are *decision nodes*, *leaf nodes*, *root node*.

A decision tree is typically built iteratively using greedy choice of decision nodes (which are called *decision stumps* when they are examined independently). The greedy choice is often based on either of the two:

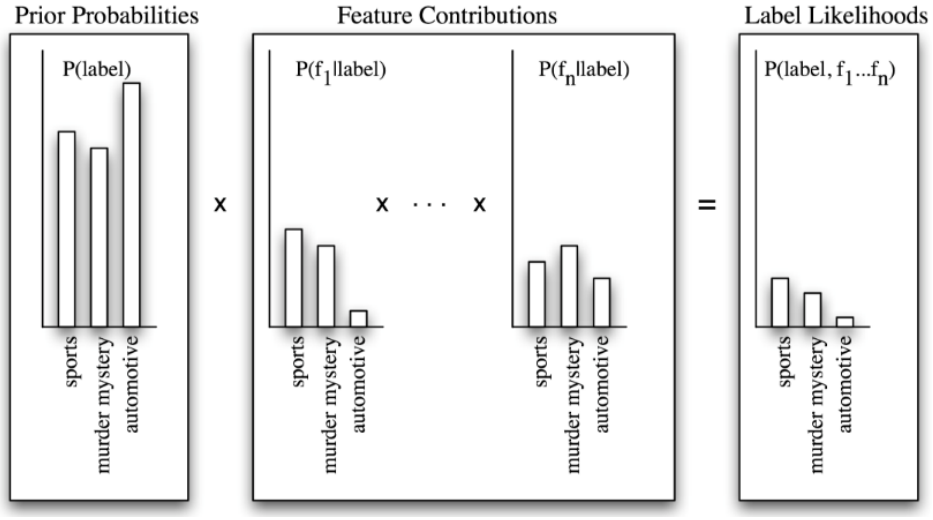
- Which feature gives the best as autonomous classifier for the training data received by the relevant node.
- Maximization of ***information gain***. The information gain is the entropy of the data given to the node as input, minus the weighted average of the entropies of the outputs of the node.

Efficient algorithms for iterative maximization of information gain were not discussed.

The hierarchical architecture of decision trees makes them easily **interpretable**.

Low decision notes have less data, hence more likely to overfit, and should be ***pruned*** (removed) if they don't improve performance on validation set.

When there are many possible predicting features, but they are weak in the sense that each one only yields weak statistical indication, the hierarchical architecture becomes a major limitation. In such case, ***Naïve Bayes Classifier*** can be used. That's a private case of ***Bayesian Network Graph*** that assumes independence between features. It calculates the likelihood of a label given the features ($P(label|features) = P(features, label)/P(features)$), as demonstrated below (up to the constant normalizer P(features)):



To reduce the sensitivity to the data (leading to overfitting), we can ***smooth*** the histograms of the features per labels by averaging them with constant histograms (i.e. adding constant counter to each count of (fi,label)).

Note: Naivete of independence yields overweighting of highly-correlated features ("counted twice").

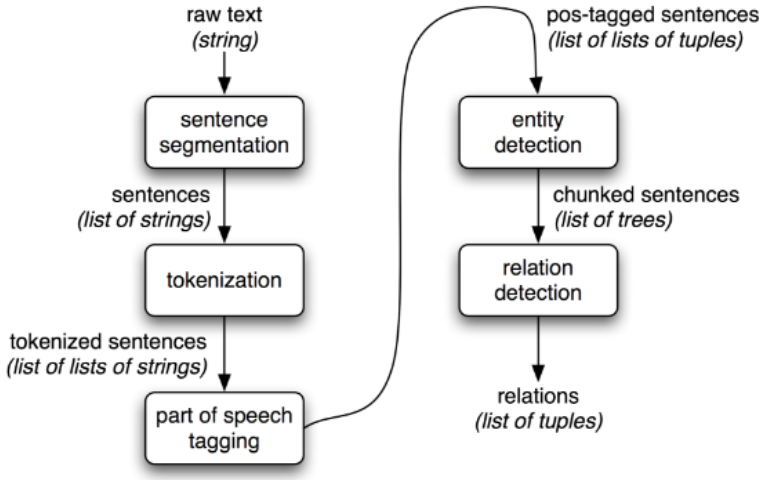
In order to avoid the independence assumption, one can replace the probabilities (which are parameters of a single feature) with general parameters characterizing more than one feature at the same time. In this case the user has to define the *joint-features*, and the corresponding parameters will be denoted by w rather than p. This model is used as part of ***Maximum Entropy Classifiers***. Such classifier maximizes the entropy of the labels under constraints, and it is *conditional* rather than *generative*, i.e. predicts $P(label|input)$ rather than $P(label, input)$. This method is not well demonstrated, and the reason for maximizing the entropy is not very clear, besides the fact that high entropy represents uncertainty between the possible labels. Popular training algorithms are GIS (slow), IIS (slow), CG and BFGS.

Learning method	Generative	Dependence expression	Interpretable
Decision tree	X	~	V
Naïve Bayes	V	X	X
Max. Entropy	X	V	X

Chapter 7: Extracting Information from Text

Practically, all imaginable information is available as electronic text. Unfortunately, NLP is still far from representation of meaning from general text, thus the asked questions are typically restricted, e.g. to relations between entities (such as "who is employed by what company?").

The essence of the process is transformation of natural language into structured representation of the desired relations (e.g. tuples (entity,relation,entity)), as demonstrated below.

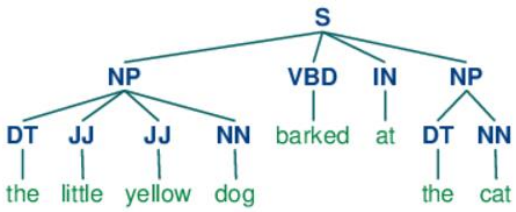


Chunking

Entity detection is done by ***chunking***, which segments multi-token sequences (e.g. "the stupid child" is one chink, though "the stupid child with the flying Frisbee" would be considered as two chunks since it represents two separated entities) and labels them.

POS-tagging is the basis of chunking. For example, simple Noun Phrases can be chunked by ***tag-patterns*** in the format of the regex "<DT>?<JJ>*<NN>", which means "optional determiner, any number of adjs, and a noun".

Chunking is kind of one level higher than tagging, and using the two levels one can already represent a sentence by a syntactic tree.



Note: chunking is not partition of a sentence, since there might be uncovered tokens (though there are not overlapping chunks). In order to notate the chunks, ***IOB-tagging*** (Inside / Outside / Begin) is used on top of the POS-tagging, e.g.:

The/DT/**B**-NP **little**/JJ/**I**-NP **yellow**/JJ/**I**-NP **dog**/NN/**I**-NP **barked**/VBD/**O** **at**/IN/**O** **the**/DT/**B**-NP **cat**/NN/**I**-NP

The last representation is useful when the chunked text is saved within a file. Another representation is by a tree as demonstrated above and used within NLTK.

In addition to ***regex-chunker***, one can use a ***unigram-chunker***, which finds the most frequent chunk type for every POS-tag (just as unigram-tagger finds POS-tag for every word), or generally ***N-gram-chunker***. While regex-chunker learns complete chunks, N-gram-chunker learns the components (I/O/B) separately.

To enrich the information of the learning chunker, one can add input features such as the words themselves (rather than only POS-tags), sequences of POS-tags since last determiner, etc. (see chapter 7, section 3.3).

Chunking can be deeply multi-layered, generating structures of syntactic trees. This can be implemented, for example, by naming the chunks and using these names within the regex-chunker recursively (in the example it wasn't recursive but rather hierarchical, i.e. each chunk consists of components only of lower level, thus the depth of the tree is limited to the number of different hierarchies).

Entity recognition

Named entities are definite noun phrases that refer to specific types of individuals. They are an essential component in extracting information from text. The common types of named entities are:

NE Type	Examples
ORGANIZATION	*Georgia-Pacific Corp., WHO*
PERSON	*Eddy Bonte, President Obama*
LOCATION	*Murray River, Mount Everest*
DATE	*June, 2008-06-29*
TIME	*two fifty a m, 1:30 p.m.*
MONEY	*175 million Canadian Dollars, GBP 10.40*
PERCENT	*twenty pct, 18.75 %*
FACILITY	*Washington Monument, Stonehenge*
GPE	*South East Asia, Midlothian*

Named Entity Recognition (NER) deals with finding the boundaries of a Named Entity, and identifying its type. When looking for information, it allows us to focus in the parts of text with entities which are relevant to what we're looking for.

Using a dedicated vocabulary for each NE type is surprisingly error-prone due to ambiguities and dynamism. For example, the words reading, library, mobile and books would be recognized as geographical places; while many new organizations would not be included in any dictionary.

Instead, a learning classifier can be used, where the output of every word is represented in the IOB format, e.g. B-PER for a word in the beginning of a person-entity (it's not clear in the text what the input is, probably current word and previous POS-tags).

Relation extraction

In order to find relations between certain NE-types, it is possible to look for triplets of the form (entity type I, certain text between entities, entity type II).

Machine learning approach might make more sense, though such one is not described in the section.

Chapter 8: Analyzing Sentence Structure

Since sentences can be arbitrarily long, and since not every sequence of words is a sentence, one needs an systematic way of generating sentences of unlimited length and valid syntax. This is done by defining abstract categories of components (***constituents***) of a sentence, and defining rules for which of these components can fit together and how. The rules are recursive (e.g. one constituent may consist of instances of the same type). The set of rules is called ***Context Free Grammar***.

```
groucho_grammar = nltk.CFG.fromstring("""
S -> NP VP
PP -> P NP
NP -> Det N | Det N PP | 'I'
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
""")
```

The recursive structure of CFG prevents inappropriate merge of parts of a sentence, which may occur in N-gram chunkers, in particular around general conjunctions. For example, “the worst part and clumsy looking” conjoins NP and AP, which is ungrammatical but could occur using N-grams.

Left-recursive productions such as “NP -> NP PP” are usually not allowed since they allow to flexible parsing, and generate infinite recursions in certain parsing algorithms (e.g. Recursive Descent Parsing, see below).

A property of constituents, making them convenient to detect, is that a constituent can be replaced in the sentence by a single word which forms a constituent of the same type:

Det the	Adj little	N bear	V saw	Det the	Adj fine	Adj fat	N trout	P in	Det the	N brook
Det the	Nom bear		V saw	Det the	Nom trout			P in	NP it	
NP He			V saw	NP it				PP there		
NP He			VP ran					PP there		
NP He			VP ran							

Example of syntactic categories:

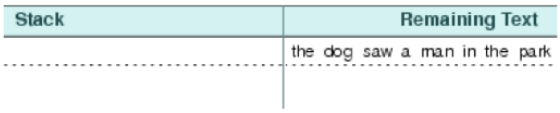
Symbol	Meaning	Example
S	sentence	*the man walked*
NP	noun phrase	*a dog*
VP	verb phrase	*saw a park*
PP	prepositional phrase	*with a telescope*
Det	determiner	*the*
N	noun	*dog*
V	verb	*walked*
P	preposition	*in*

For a CFG and a sentence: no valid tree → *grammatically/syntactically invalid*; more than one valid tree → *structurally ambiguous*.

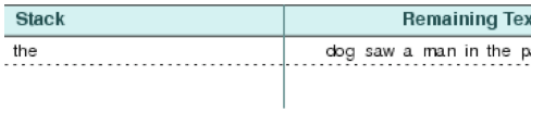
A parser parses (no shit) sentences according to the rules of a CFG. Several parsers:

- ***Recursive Descent Parsing***: define goals in terms of what constituents we're looking for, then define sub-goals iteratively according to the rules (e.g. S, then NP & VP, …), until we reach concrete words that can be matched in the sentence.
    - Quite inefficient top-down parsing: there's a search over all possible trees, where the search is not led by the input, and many calculations repeat many times (e.g. in VP -> V NP | V NP PP, the NP will be fully-parsed twice). In addition, left-recursive productions (e.g. NP -> NP PP) lead to infinite loop.
- ***Shift-Reduce Parsing***: bottom-up parsing that iteratively pushes ("shifts") words from the sentence into a stack, and whenever a sequence of elements in the stack matches a right-hand side of a production (e.g. "Det N" is the right-hand of "NP -> Det N | …"), then it replaced by ("reduced" to) the left-hand constituent.

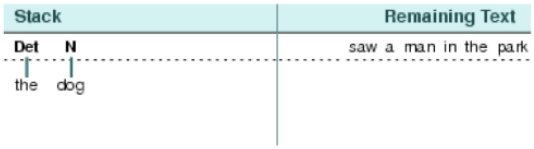
1. Initial state



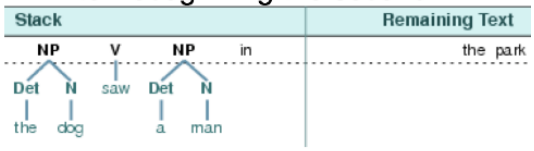
2. After one shift



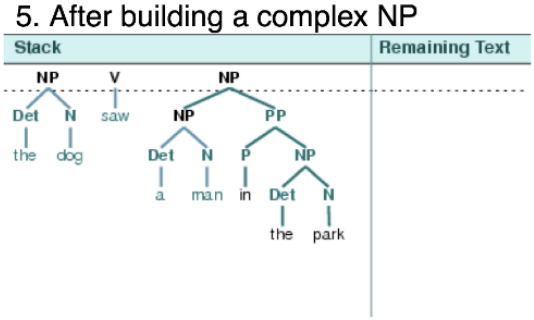
3. After reduce shift reduce



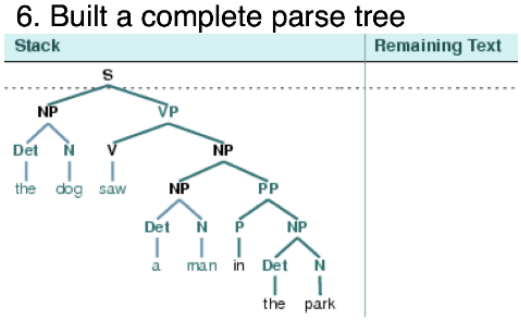
4. After recognizing the second NP



5. After building a complex NP



6. Built a complete parse tree



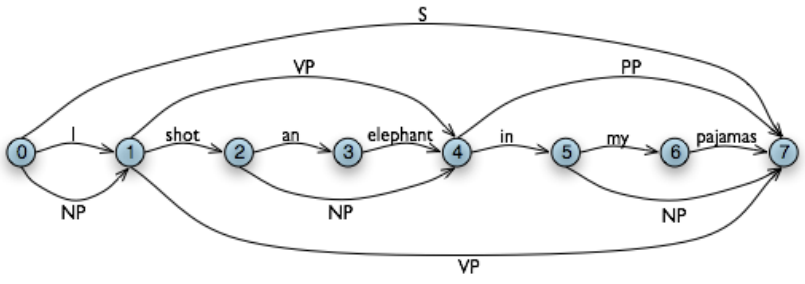
  - Note: the described variant of the algorithm, which is used in NLTK, is very simple and do not include ***backtracking*** (i.e. realization that we did wrong and going back to try different parsing), thus it is not guaranteed to find a valid parsing, and do not find more than one possible parsing anyway. This issue is usually solved in programming-languages compilers, for example, using a generalization called ***lookahead LR parser*** (not explained here).
- ***Left-Corner Parsing***: top-down parser with bottom-up filtering. The sentenced is parsed as in recursive-descent parsing, but subtrees are not expanded unless the beginning of the expanding (its "left corner") may match the beginning of the corresponding text.
  - For example, NP beginning with "John" will not be expanded into "Det N PP", since Det cannot match John.
  - This expanding-filter prevents infinite loops on left-recursive productions.
  - The filter is implemented using a table of the potential left-corners corresponding to every non-terminal constituent. In the following example, a constituent won't be expanded into something beginning with VP, unless the left-corner V matches the beginning of the corresponding text.

Category	Left-Corners (pre-terminals)
S	NP
NP	Det, PropN
VP	V
PP	P

- ***Chart parsing***: more systematic bottom-up algorithm that uses dynamic programming to avoid repeated calculations. The already-computed constituents are stored in a table (***Well-Formed Substring Table***), where the i,j cell contains a valid constituent found for the string from the i'th word to the j'th word:

```
>>> display(wfst1, tokens)
WFST 1    2    3    4    5    6    7
0    NP   .    .    S    .    .    S
1    .    V    .    VP   .    .    VP
2    .    .    Det  NP   .    .    .
3    .    .    .    N    .    .    .
4    .    .    .    .    P    .    PP
5    .    .    .    .    .    Det  NP
6    .    .    .    .    .    .    N
```

  - This can also be represented as a chart:



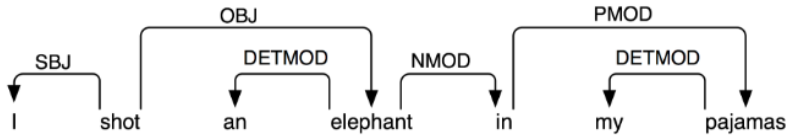
- To build the table, for any n, for any n-gram in the sentence, for any partition to 2 constituents (if it exists in the current WFST), we try to merge the constituents (if the grammar permits), and add the merged constituent to the WFST.
- Generalization of the algorithm is required to detect ambiguities (rather than just overwriting one parsing with another within the loops) and to allow grammars with more than 2 right-hand constituents (by going over multi-partitions).

Summary of the discussed parsers:

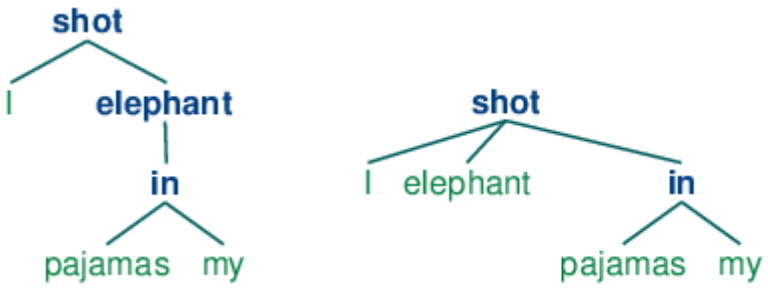
Parser	Completeness		Efficiency	
	Find valid tree when existing	Find all valid trees	Try only relevant sub-trees	Construct each sub-tree only once
Recursive Descent	V	V	X	X
Left-Corner			~	
Shift-Reduce	X	X	V	V
Chart	V	X	V	V

Dependency grammar

Instead of CFG, the structure of a phrase can be represented by direct dependencies between words:



The dependencies can be represented by a tree as well, and to capture ambiguities:



The direction of each dependency (head->dependents) can be defined by various rules, including semantics, which is obligatory in the sentence, etc.

A dependency graph is projective iff for every word, the word along with its dependents (of any order) form a contiguous sequence within the sentence. In the chart representation, it means that the dependencies can be drawn without intersection of edges.

While dependency grammar can be difficult to implement, considering the semantics within the grammar is very important: not every production (e.g. VP -> V S) can correspond to any sequence

(e.g. "I was that I saw her" is invalid). To combine the concept of standard grammar with semantic-considering dependencies, one can define subcategories of constituents, e.g. for verbs:

Symbol	Meaning	Example
IV	intransitive verb	*barked*
TV	transitive verb	*saw a man*
DatV	dative verb	*gave a dog to a man*
SV	sentential verb	*said that a dog barked*

Which permit variant grammatical rules, e.g.:

`VP -> V Adj`	*was*
`VP -> V NP`	*saw*
`VP -> V S`	*thought*
`VP -> V NP PP`	*put*

Anyway, building such database of subcategories corresponding to specific words and their semantic meaning is very hard, in particular since the language keeps expanding dynamically. Problems are also raised by large numbers of ambiguities and exceptions. Grammars can be learnt from tree-banks corpora such as the Penn Treebank, Sinica Treebank (Chinese) and Prepositional Phrase Attachment. Existing grammars are also available in NLTK. See chapter 8, section 6.1 for resources description.

Probabilistic Context Free Grammar (***PCFG***) / ***Weighted Grammar***: due to the enrichment of language, the number of ambiguities grows very rapidly with the length of sentences. To deal with the numerous possible parsing-trees, one can assign probabilities to syntactic productions. This way, long parsing can be interrupted once the probability gets lower than any alternative parsing of the same phrase.

```
grammar = nltk.PCFG.fromstring("""
    S    -> NP VP              [1.0]
    VP   -> TV NP              [0.4]
    VP   -> IV                 [0.3]
    VP   -> DatV NP NP         [0.3]
    TV   -> 'saw'              [1.0]
    IV   -> 'ate'              [1.0]
    DatV -> 'gave'             [1.0]
    NP   -> 'telescopes'       [0.8]
    NP   -> 'Jack'             [0.2]
    """)
```

Chapter 9: Building Feature-Based Grammars

Verb subcategorization: Some features are essential for grammatically-correct sentences generation, e.g. ***Syntactic Agreement*** = covariance between morphological properties of the verb and syntactic properties of the subject:

	singular	plural
1st per	*I run*	*we run*
2nd per	*you run*	*you run*
3rd per	*he/she/it runs*	*they run*

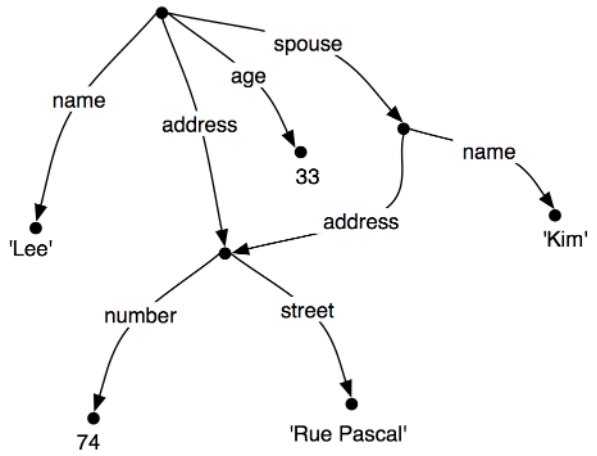
Expressing such features through separate pos-tags, e.g. *NP_SG* for *he* and *NP_PL* for *they*, is inefficient since it increases the size of the grammar quickly. Instead, SG & PL can be defined as possible values of a feature NUM of the NP category, and the extended grammar can be compactly defined in terms of the variables, e.g. to require syntactic agreement:

```
S -> NP[NUM=?n] VP[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
VP[NUM=?n] -> V[NUM=?n]
```

And if the constituent is represented by variables anyway, then its type might as well be a variable, e.g. [POS=N,NUM=SG] instead of N[NUM=SG].

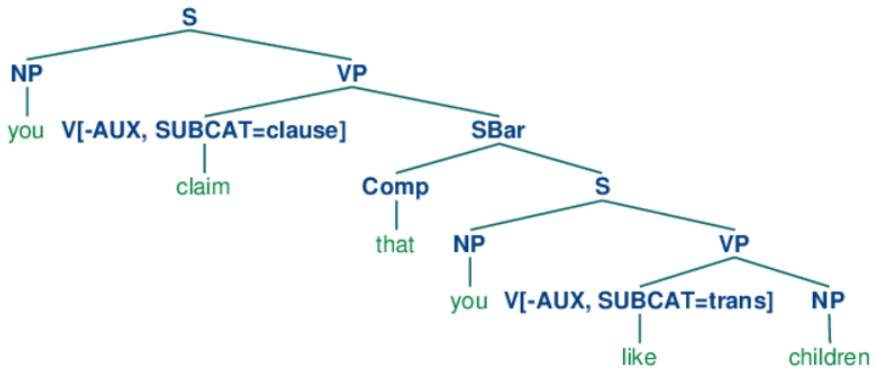
Types of features:

- *Atomic* = can't be decomposed into subparts (e.g. number=sg/pl, tense=present/past…, person=1st/2nd/3rd)
    - *Boolean* = boolean (e.g. auxiliary – can, may… vs. walk, like…)
- *Complex* = feature which represents multiple features grouped together (e.g. agreement=[person,number,gender])
    - Complex feature is typically represented in a format known as *attribute value matrix* (*AVM*)
    - Complex features allow compact representation of grammatic rules, e.g. "S -> NP[AGR=?n] VP[AGR=?n]"
    - Complex feature structures can be represented by directed acyclic graphs. For (non-linguistic) example – a feature set of certain person:



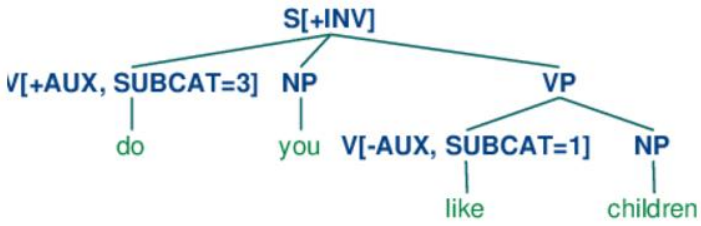
In the context of grammar, features of constituents are called *subcategories*, and the grammar based on them is called ***Generalized Phrase Structure Grammar*** (***GPSG***). Such grammar permits better correspondence between the lexical words and the parsing of the sentence, e.g. by noting

that “claim” is a clause verb whereas “like” is a transitive verb, thus they appear in different syntactic contexts, and the following sentence is parsed accordingly:

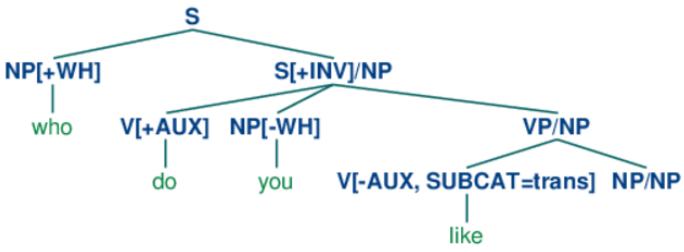


A slightly different representation defines the subcategories directly by the required contexts (e.g. V[SUBCAT=<NP,NP,PP>] for “put” in “I put the book on the table”), rather than defining the subcategory as an abstract entity and expressing the required context through permitted productions (e.g. V[SUBCAT=cat_name] and S -> NP V; V[SUBCAT=cat_name] -> V NP PP). This approach is called ***Head-driven Phrase Structure Grammar***.

Inversion constructions: The order of the noun and the verb in a sentence can be fully or partially ***inverted***, as long as the verb belongs to the set of the ***auxiliaries*** (do, can, have, be, will…). In such cases, inverted sentence is permitted, e.g.:



Unbounded dependency constructions: Some verbs require certain context such as [VP -> V NP PP], and missing context yields ungrammatical sentence. However, sometimes the context is just “somewhere else”, e.g. “Which slot did you put the card into?”. In such case we say that there’s a ***gap*** (into ___) after the verb (put), but it is *licensed* by a ***filler*** (which slot). Generally, the distance between the gap and its filler is unbounded (for demonstration – “Who do you claim that Jody says that you like __?”). parsing of such phrases can be done using ***slash categories***, which are constituents with gaps denoted by a slash:



The slash-category can be represented by just another variable (as any other sub-category of a constituent), which allows the grammatic rules to parse it conveniently. In the example, who is detected as NP[+WH], then the following sentence is classified as S[…,slash=NP], and the "slash=NP" goes down until the empty "NP[slash=NP]" is reached.

Various languages have different grammatical rules, and in some of them (e.g. German and Hebrew) the morphology for agreement is richer and more complex, e.g. by taking into account the gender of the noun.

Chapter 10: Analyzing the Meaning of Sentences

Features-based grammars for meaning analysis: may work fine for restricted domains of Q&A, using corresponding semantic variables.

- For example, for questions of the form "which/what cities are located in ___", one can store the semantical meaning of each word as a variable SEM, and the productions rules can combine the semantics into the semantic meaning of the sentence, encoded for example in SQL ("SELECT City FROM city_table WHERE Country="___ "").
- However, that's quite bit cheating because there're strong assumptions about the input. Part of that is claimed to be due to limitations of SQL, thus usually logic representation or other dedicated languages are used.

Logic model for a set W of sentences: representation of situation in which the sentences in W are true. The representation consists of:

- Domain D of all the entities we care about (e.g. people's names).
- Characteristic expressions (e.g. "boy" or "is running") which are represented as properties of the entities, or belonging of the entities to corresponding groups.

Logic is a structured tool for inference. ***Propositional logic*** is simple (just atomic objects and some Boolean operators), thus it is quite weak (relations between objects have to be represented by dedicated objects, which is very inefficient). ***First-Order Logic*** is stronger, and can handle relations, functions and quantifiers (exists & all).

The generation of FOL representation of a sentence is based on the ***Principle of Compositionality***: The meaning of a whole is a function of the meanings of the parts and of the way they are syntactically combined.

An open expression in FOL is a formula in which at least one variable is not bound to a quantifier but rather free (e.g. exists y.(kill(x,y)). Sometimes we wish to close the expression as a function depending on the free variable. without using the exists/all quantifiers. This is done using the **λ operator** (e.g. λx.(exists y.(kill(x,y)))). In NLTK it is encoded as \x rather than λx. Two uses were demonstrated in the chapter:

- Referring to the action itself, e.g. "running is hard" as hard(\x.run(x)).
- Generation of general objects/formulas which depend on a variable.

Chapter 11: Managing Linguistic Data

The chapter reviews many small issues regarding corpus design, most of them mentioned in the book, most of them known in machine learning, most of them essentially mean "don't be idiot". Intended mainly for scientists who wish to publish a big generic public corpus.

OLAC (*Open Language Archives Community*): database of linguistic resources.

NLTK also has its own repository, accessible through the built-in downloader.

TIMIT: very structured phonetic corpus of read speech.

Heritrix: web crawler useful for obtaining data for a corpus.

Possible layers of annotation: word tokenization, sentence segmentation, paragraph segmentation, POS-tagging, syntactic structure, shallow semantics, dialog & discourse.

XML (*Extensible Markup Language*): format for data storing, which is very dynamic compared to HTML (constant tags) and databases (consistent tags).

- Different ***attributes*** can be assigned to every instance of every ***tag***.

```
<entry>
  <headword>whale</headword>
  <pos>noun</pos>
  <gloss synset="whale.n.02">any of the larger cetacean mammals having
      a streamlined body and breathing through a blowhole on the head</gloss>
  <gloss synset="giant.n.04">a very large person; impressive in size or
      qualities</gloss>
</entry>
```

- Naturally, unstructured use of the XML format makes the data harder to parse. A well-defined format can be forced on an XML file using a ***schema***.
- An XML file can be easily read in python using *xml.etree.ElementTree*.

Chapter 12: The Language Challenge

NLP is very important.

NLP is more extensive and complicated than presented in the book (yeah, actually it hardly talked about ML in NLP).

NLP still has many difficulties with complex or un-defined structures of text.

NLP was developed in grace of research of *formal language theory* (representation of language as a set of strings generated by automata), *symbolic logic* (FOL), and *principle of compositionality* (determining meaning recursively by decomposition into smaller phrases).

Chatbot-Oriented Summary

Steps of processing

- **Phonology** – irrelevant: speech to text
- **Segmentation** – irrelevant: fill in missing spaces (e.g. in URL)
- **Tokenization** (chapter 3): separate text into tokens (e.g. U.S.A. is one token rather than 3 sentences)
    - There is a built-in regex-based tokenizer in NLTK.
- **Lemmatization** (chapter 3): transform each word to its canonical form – useful for dictionary-based analysis where the dictionary does not contain all the forms of the word (e.g. worked→work)
    - There are built-in regex-based stemmers in NLTK. However, the analysis described below was demonstrated in the book for non-lemmatized texts, so this canonization should not be done as default without thinking.
- **Morphology / lexical analysis** (chapter 5): identify the role of each token – its Part-Of-Speech (noun, verb, etc.)
    - There are corresponding learning-based taggers in NLTK, though the tagged dbs for learning may be only partially relevant to informal chat.
- **Syntax** parsing: find the structure of a sentence (represented by tree)
    - Parsing can be done systematically by grammar-based parsing, though most demonstrated grammars (chapters 8,9) were partial, and none were slang-oriented.
    - Heuristics-based classifier was demonstrated in chapter 7 (learning chunks from tags), and this might be more robust to corrupted structures (heuristic rather than systematic).
- **Semantics** reasoning: get meaning from text
    - Entity recognition and relation detection are demonstrated in chapter 7 (e.g. for understanding which organization is located where), and may be generalized to look for certain information within messages. However, the heuristic it used (<something recognized as organization> * in * <something recognized as location>) is very rough and any simple generalization is expected to have limited potential.
    - When a sentence is parsed systematically using grammar, one can describe entities using variables (e.g. recognize entities of type person according to dictionary or classifier), and find the relations between the entities along with the parsing. This approach was nicely demonstrated in chapter 10, but always relied on small, dedicated dictionaries, and on specific semantical rules in the grammar – i.e. one has to know what he's looking for, and that's quite limiting.

Since chatting is too unstructured, trying to parse it to valid syntax may be meaningless. Instead, it is possible to try **less systematic and more straight-forward approach**. For example, one can use Naïve Bayes (=words counting) or Max Entropy Classifier to **classify the topic** of the post (demonstrated for public chat in chapter 6), and then either similar or more systematic approach to **find the meaning** (as demonstrated in chapter 7, it's easier to analyze after we already decided what the topic is and what we're looking for).

One may try to use even more **brute-force approach** and learn to generate replies (output) to messages (input). However, doing so without any abstraction of the text **seems not to fit** the chatting problem:

- Good prediction requires rich input space which requires tones of data for learning.
- The chats are expected to usually talk about one of several specific topics, but to do it in very dynamic and varying forms. Thus, it makes sense to try to figure out the abstract meaning and respond accordingly, rather than learn the words phonetically.

The most promising dictionary observed in the book was the **Wordnet** corpus, which forms tree of concepts ordered by meaning. For example, if one observes that most chats are about music, sports or sex, then the topic of a sentence can be recognized by detecting the entities in the sentence and checking the semantic path of each one of them in the tree, looking for music/sport/sex oriented stuff. This approach actually resembles the Lexicalized Probabilistic Context Free Grammar (**LPCFG**) with back-off models, that was described in the AI course of Udacity as the state-of-the-art type of grammar.

Databases

- There are no private chat corpora in NLTK.
- *Nps_chat* (public chat) may help to practice on informal language.
- *Wordnet* (synonyms and other relations) can help with the semantics.
- *Webtext* (including Pirates of the Caribbean) may be beneficial with certain type of chatters.
- Other corpora (brown, nps_chat) may be helpful for Part-Of-Speech learning (classification to nouns, verbs, etc.).
- *conll2000* allows chunking learning.
- More resources are available in NLTK downloader and in the external OLAC database.
- One can create his own corpus (demonstrated in chapter 2).

Missing subjects

- **Spelling correction** (there're many errors in chats)
- Reasoning meaning out of **unstructured sentences** (chatters don't care about syntax)
- **Scale-up** dictionary-depending processing (though using the dicts corpora along with lemmatization might suffice)

Interesting NLTK features

Chapters 1 and 2 demonstrate a bunch of interesting features that may be useful for variant analysis, such as: print all the contexts in which a certain word appear; find words that appear in similar contexts (see **word2vec** as promising external reference); compute statistics of lengths, frequencies, etc.; n-grams based text generator (generate the next word according to its n-1 formers); find synonyms of a word; find homonyms, nanonyms, dumbnyms, etc. – such as sub-categories of sports or persons.

Additional tips & tricks

Classification of text according to entropy-based similarity: to find the category of a sentence, get text from each category, concatenate the new sentence to each text separately and zip each

pair (text_i, new_sentence) – the best compression will be the one that is most homogeneous, which is a strong indication to similarity.