1. Identify larger syntactic structure.
2. Extract syntactic features from trees
3. Implement key aspects of these algorithms.
4. Use statistics to disambiguate language to find the best parse.
5. Evaluate parsing output relative to gold standard treebanks.

Different nltk syntactic parser :

* The Stanford Parser
* The Chart Parser - disambiguate language : multiple meaning
* The Shift-Reduce Parser
* The Recursive Descent Parser

Nltk regex parser is not syntactic parser. It is normal parser.

# Tree class constructor:

PRP = Tree("PRP", ["I"])

VBD = Tree("VBD", ["met"])

N1 = Tree("NN", ["bill"])

NP1 = Tree("NP", [PRP])

NP = Tree("NP", [N1])

VP = Tree("VP", [VBD, NP])

S= Tree("s", [NP1, VP])

print(S)

#(s (NP (PRP I)) (VP (VBD met) (NP (NN bill))))

# Bracketed string

S = Tree.fromstring("(S (NP (PRP I)) (VP (VBD met) (NP (NN Bill))))")

print(S)

#(S (NP (PRP I)) (VP (VBD met) (NP (NN Bill))))

# Alternative representation

np = Tree("NP", [("I", "PRP")])

vp = Tree("VP", [("met", "VBD"), ("NP", [("Bill", "NN")])])

S= Tree("s", [NP1, VP])

print(S)

#(s (NP I/PRP) (VP (VBD met/VBD) (NP bill/NN)))

The cat sat on the mat

First “The cat” is NP-SBJ

Second ‘the mat’ is only NPs

# nltk.tree.Tree

from nltk.corpus import treebank

from nltk.tree import Tree

# The function parsed\_sents() returns the syntax trees in the treebank.

syntax\_trees = treebank.parsed\_sents()

## tree.label

tree = syntax\_trees[0]

print(tree.label())

for phrase in tree:

print(phrase)

print("---")

## tree.subtrees

for phrase in tree.subtrees():

print(phrase)

print("---")

print(tree.leaves())

## Tree.fromstring

tree = Tree.fromstring("(S (NP (PRP I)) (VP (VBD met) (NP (NNP Bill))))")

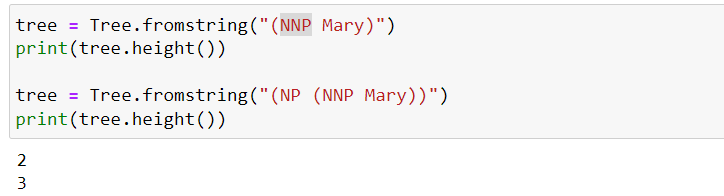
print(tree.height())

print(tree.leaves())

print(tree.height())

print(tree.flatten())

## Height



The only trees with height 1 are preterminals (POS word). All phrases have height >= 2

# Parser :

## nltk.RegexpParser

* can define a set of grammar

from nltk import RegexpParser

simple\_chunker = RegexpParser("NP: {<DT|PRP\$><NN>}")

**parsed\_sent = simple\_chunker.parse(tagged\_sent)**

chunks = []

for phrase in parsed\_sent.subtrees():

if phrase.label() == "NP":

words = [word for word, tag in phrase]

chunks.append(" ".join(words))

"NP: {<DT|PRP\$><NN>}"

The structure of the grammar is:

* "NP:" is label, in this case "NP" for noun phrase.
* "{<DT|PRP$><NN>}" is the actual grammar definition.

The definition consists of multiple tags that are separated by < > :

* <DT> - Defines a determiner such as "the", "a", "an".
* <PRP$> - Defines possessive pronoun such as "my","his","hers"
* <NN> - Defines a noun such as "fox", "dog"
* <RB> - Adverb
* AdvP – adverbail phrses

1. \ character is used to escape the following character.
2. for this case if we want to make $ as regex inside {} then we have to use \ otherwise it will be cosidered as the $ in string.
3. if $ as literal character means as regex

## Chunk dataset, GOLD standard chunk

conll2000 chunking shared task dataset.

from nltk.corpus import conll2000

gold\_chunked\_sents = conll2000.chunked\_sents(chunk\_types=["NP"])

tagged\_sent = conll2000.tagged\_sents()

print(gold\_chunked\_sents[0])

print(tagged\_sent)

print(simple\_chunker.parse(tagged\_sent[0]))

PRP : personal pronoun

I," "you," "he," "she," "it," "we," "they," "me," "him," "her," "us,"

JJ: adjective

## CFG

1. Inside the from stirng we can define the grammer rules.

CFG.fromstring(“”)

1. Now we have to pas the grammer rule inside the parser nd create parser object

parser = EarleyChartParser(grammar)

parses = parser.parse("she gives us bills".split())

for parse in parses:

print(parse

with the help of the non terminal we can extend already existed grammer rule like

new\_grammar = CFG(Nonterminal("S"), grammar.productions() + new\_productions)

new\_productions : newly created grammer rule

## Feature grammer

Feature grammar allows for a more accurate representation of the grammatical properties of the sentence, and also includes more features such as NUM, CASE and VAL, allowing for a more fine-grained control over the generation and parsing of sentences.

NP[NUM=?n, CASE=?m] -> PRP[NUM=?n, CASE=?m]

NUM feature : represent the number of the noun phrase (singular or plural)

CASE feature : represent the grammatical case of the noun phrase (subject, object, etc.)

N – represent number of cases like

M – represent cases like singular ot plural

FeatureEarleyChartParser, FeatureGrammarclass is used

# CNF

Converting a grammar to CNF (Chomsky Normal Form) is a process of transforming a grammar into a specific form that has a specific set of rules.

It is easier to implement.

In CNF, a grammar must have only three types of rules:

* Terminal symbols on the right-hand side of the rules (e.g. "the", "cat", "sat")
* Two non-terminal symbols on the right-hand side of the rules (e.g. S -> AB)
* One non-terminal symbol on the left-hand side and one terminal symbol on the right-hand side of the rule (e.g. S -> "the")

Non-terminal symbols are used to represent the structure of a sentence in a formal grammar, for example, NP, VP, S.

Terminal symbols are the words or phrases that make up a sentence, for example, "cat", "sat" and "on" are terminal symbols.

CYK

* Dynamic programming algorithm for context free grammers.
* Simple probablistic extension
* O(n^3) complexity
* It uses Bottom up : start with smaller then increase the tree then finally a sentence. In top down we start with full sentence.
* We use fence post to point the gaps between words.
* Use table representation

The dog chased the cat

S -> NP VP

NP > DET N

Det > the

N > dog

Vp > chased np

Np > det N

Det > the

N > cat