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))))")

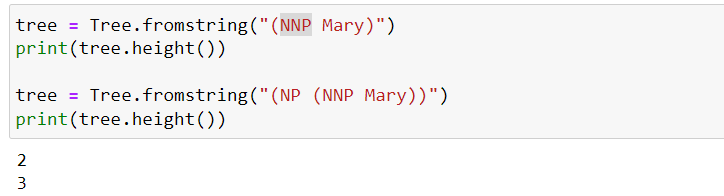
print(tree.height())

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

# Annotated linguistic data

CoNLL-U, Bracketed Text Format, IOB (Inside, Outside, Beginning) Format

# CoNLL-U

CoNLL-U format is specifically designed for annotating linguistic data in natural language processing.

## UD vs CoNLL-U

UD is a project that provides a framework for consistent annotation of grammatical structures in languages across the world.

It provides guidelines for annotating text corpora with morphological, syntactic, and semantic information, and includes guidelines for annotation schemes and a shared task evaluation framework.

CoNLL-U is a file format used to store and share annotated text corpora that follow the UD guidelines.

CoNLL-U files are plain text files with tab-separated values (TSV) format, where each line represents a word in a sentence and provides information about the word's form, lemma, POS tag, morphological features, etc.

## CoNLL-U benefits

* Universality: CoNLL-U is a universal format that can be used to represent a wide variety of linguistic annotations, including morphological, syntactic, and semantic annotations.
* Interoperability: By using a standard file format, NLP tools and libraries can easily exchange data, making it easier to build NLP pipelines that involve multiple tools and libraries.
* Readability: The CoNLL-U format is designed to be human-readable, making it easier for researchers to inspect and debug the annotated data.
* Tool Support: Many NLP tools and libraries support the CoNLL-U format, making it easier for NLP practitioners to use these tools with their data.

It consists of three types of lines:

* word lines (annotations of words/tokens)
* blank lines marking sentence boundaries
* comment lines starting with "#".

Word lines have 10 fields: ID, FORM, LEMMA, UPOS, XPOS, FEATS, HEAD, DEPREL, DEPS, and MISC.

UD

there are over 70 universal dependencies (UD) relationships defined in the UD scheme.

# Parsing

## Shift reduce parsing

* It is used for constructing a dependency parse of a sentence in nlp
* It use stack for buffering
* Resulting structure is like Tree
* Also known as bottom up approach.

## Constituency parsing

* Its task for identifying the parsing in a sentence such as noun phrases, verb phrses, adj phrases
* it make tree like structure
* top down approach
* entity recoginition, semantic role labelling, machine translation.

## Open class words

New words can be added to the class without much difficulty like nouns, verbs, adjectives, and adverbs.

## Closed class words

These are more limited in number and cannot be easily added to the language like articles (the, a, an), prepositions, conjunctions, pronouns, and interjections

# Representing the structure of sentences

## Constituency-based syntax

* structure is represented as tree
* parent child relationship
* parsing algorithm like Earley parsing and CYK parsing

## Dependency syntax

* Structure is represented as directed acyclid graphs(dags)
* Each node in the graph represtn a word and each directed edge represent a syntactic relationship.
* It make graph by finding the head word
* There is no grammer in the same sense as in context free grammer
* It mimic the dependency found in the treebank

Two ways to build automatically dependency tees transition based and graph based :

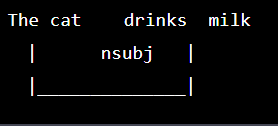
### Transition based parsing

* It construct the dependency tree step at time thn each step perform same task.
* It is faster not much accurate.
* It is also known as shift-reduce parsing.
* it represent the dependency tree for a sentence as a sequence of parser actions.
* It is more suited for large scale of the data.

### Graph based parsing

* It make complete graph of sentence then find the optimal structure to find the relationship between words.
* It more accurate slower in speed

### Dependency syntax graph



Projective dependency trees contain only arcs which do not cross.

Dealing with Projective dependency is by the COLX581 NLP for low-resource languages.

Graph-based parsers can deal with both projective and non-projective trees.

## Graph-based parsers

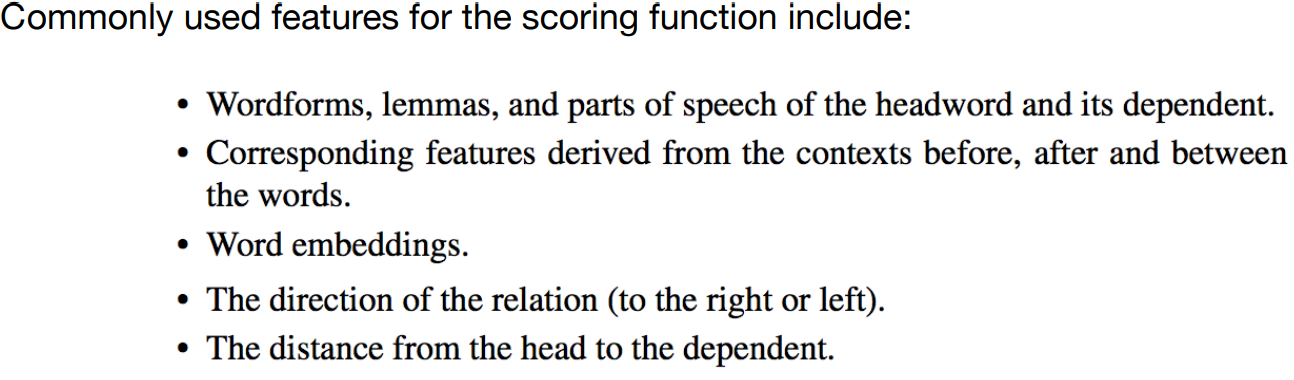
It directly construct dependency graphs

There are two important considerations in graph-based parsing:

1. How do we find the scores for edges?

2. How do we find the dependency graph with maximal score?

binary classifier predicts TRUE for existing dependencies and FALSE for non-existent ones



Dozat and Manning (2016): DEEP BIAFFINE ATTENTION FOR NEURAL DEPENDENCY PARSING

A spanning tree for a directed graph (like a dependency graph) has to fulfill the following:

1. All nodes are connected by a path

2. All nodes have exactly one incoming arc

3. There are no cycles

Chu–Liu-Edmonds algorithm will find us the MST