# Parsing Structure in Text

This chapter involves a better understanding of deep structure in text and also how to deep parse text and use it in various NLP applications. Now, we are equipped with various NLP preprocessing steps. Let's move to some deeper aspect of the text. The structure of language is so complex that we can describe it by various layers of structural processing. In this chapter we will touch upon all these structures in text, differentiate between them, and provide you with enough details about the usage of one of these. We will talk about **context-free grammar** (**CFG**) and how it can be implemented with NLTK. We will also look at the various parsers and how we can use some of the existing parsing methods in NLTK. We will write a shallow parser in NLTK and will again talk about NER in the context of chunking. We will also provide details about some options that exist in NLTK to do deep structural analysis. We will also try to give you some real-world use cases of information extraction and how it can be achieved by using some of the topics that you will learn in this chapter.

We want you to have an understanding of these topics by the end of this chapter.

In this chapter:

* We will also see what parsing is and what is the relevance of parsing in NLP.
* We will then explore different parsers and see how we can use NLTK for parsing.
* Finally, we will see how parsing can be used for information extraction.

## Shallow versus deep parsing

In deep or full parsing, typically, grammar concepts such as CFG, and **probabilistic context-free grammar** (**PCFG**), and a search strategy is used to give a complete syntactic structure to a sentence. Shallow parsing is the task of parsing a limited part of the syntactic information from the given text. While deep parsing is required for more complex NLP applications, such as dialogue systems and summarization, shallow parsing is more suited for information extraction and text mining varieties of applications. I will talk about these in the next few sections with more details about their pros and cons and how we can use them for our NLP application.

## The two approaches in parsing

There are mainly two views/approaches used to deal with parsing, which are as follows:

|  |  |
| --- | --- |
| **The rule-based approach** | **The probabilistic approach** |
| This approach is based on rules/grammar | In this approach, you learn rules/grammar by using probabilistic models |
| Manual grammatical rules are coded down in CFG, and so on, in this approach | This uses observed probabilities of linguistic features |
| This has a top-down approach | This has a bottom-up approach |
| This approach includes CFG and Regex- based parser | This approach includes PCFG and the Stanford parser |

## Why we need parsing

I again want to take you guys back to school, where we learned grammar. Now tell me why you learnt grammar Do you really need to learn grammar? The answer is definitely yes! When we grow, we learn our native languages. Now, when we typically learn languages, we learn a small set of vocabulary. We learn to combine small chunks of phrases and then small sentences. By learning each example sentence, we learn the structure of the language. Your mom might have corrected you many times when you uttered an incorrect sentence. We apply a similar process when we try to understand the sentence, but the process is so common that we never actually pay attention to it or think about it in detail. Maybe the next time you correct someone's grammar, you will understand.

When it comes to writing a parser, we try to replicate the same process here. If we come up with a set of rules that can be used as a template to write the sentences in a proper order. We also need the words that can fit into these categories. We already talked about this process. Remember POS tagging, where we knew the category of the given word?

Now, if you've understood this, you have learned the rules of the game and what moves are valid and can be taken for a specific step. We essentially follow a very natural phenomenon of the human brain and try to emulate it. One of the simplest grammar concepts to start with is CFG, where we just need a set of rules and a set of terminal tokens.

Let's write our first grammar with very limited vocabulary and very generic rules:

**# toy CFG**

**>>>from nltk import CFG >>>toy\_grammar = nltk.CFG.fromstring( """**

**S -> NP VP # S indicate the entire sentence**

**VP -> V NP # VP is verb phrase the**

**V -> "eats" | "drinks" # V is verb**

**NP -> Det N # NP is noun phrase (chunk that has noun in it)**

**Det -> "a" | "an" | "the" # Det is determiner used in the sentences**

**N -> "president" |"Obama" |"apple"| "coke" # N some example nouns**

**""")**

**>>>toy\_grammar.productions()**

Now, this grammar concept can generate a finite amount of sentences. Think of a situation where you just know how to combine a noun with a verb and the only verbs and nouns you knew were the ones we used in the preceding code. Some of the example sentences we can form from these are:

* President eats apple
* Obama drinks coke

Now, understand what's happening here. Our mind has created a grammar concept to parse based on the preceding rules and substitutes whatever vocabulary we have.

If we are able to parse correctly, we understand the meaning.

So, effectively, the grammar we learnt at school constituted the useful rules of English. We still use those and also keep enhancing them and these are the same rules we use to understand all English sentences. However, today's rules do not apply to William Shakespeare's body of work.

On the other hand, the same grammar can construct meaningless sentences such as:

* Apple eats coke
* President drinks Obama

When it comes to a **syntactic parser**, there is a chance that a syntactically formed sentence could be meaningless. To get to the semantics, we need a deeper understanding of semantics structure of the sentence. I encourage you to look for a semantic parser in case you are interested in these aspects of language.

## Different types of parsers

A parser processes an input string by using a set of grammatical rules and builds one or more rules that construct a grammar concept. Grammar is a declarative specification of a well-formed sentence. A parser is a procedural interpretation of grammar. It searches through the space of a variety of trees and finds an optimal tree for the given sentence. We will go through some of the parsers available and briefly touch upon their workings in detail for awareness, as well as for practical purposes.

### A recursive descent parser

One of the most straightforward forms of parsing is recursive descent parsing. This is a top-down process in which the parser attempts to verify that the syntax of the input stream is correct, as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching them with the terminals from the grammar that describes the syntax of the input. Our recursive descent parser will look ahead one character and advance the input stream reading pointer when it gets a proper match.

### A shift-reduce parser

The shift-reduce parser is a simple kind of bottom-up parser. As is common with all bottom-up parsers, a shift-reduce parser tries to find a sequence of words and phrases that correspond to the right-hand side of a grammar production and replaces them with the left-hand side of the production, until the whole sentence is reduced.

### A chart parser

We will apply the algorithm design technique of dynamic programming to the parsing problem. Dynamic programming stores intermediate results and reuses them when appropriate, achieving significant efficiency gains. This technique can be applied to syntactic parsing. This allows us to store partial solutions to the parsing task and then allows us to look them up when necessary in order to efficiently arrive at a complete solution. This approach to parsing is known as chart parsing.

For a better understanding of the parsers, you can go through an example at <http://www.nltk.org/howto/parse.html>.

### A regex parser

A regex parser uses a regular expression defined in the form of grammar on top of a POS-tagged string. The parser will use these regular expressions to parse the given sentences and generate a parse tree out of this. A working example of the regex parser is given here:

**# Regex parser**

**>>>chunk\_rules=ChunkRule("<.\*>+","chunk everything")**

**>>>import nltk**

**>>>from nltk.chunk.regexp import \***

**>>>reg\_parser = RegexpParser('''**

**NP: {<DT>? <JJ>\* <NN>\*} # NP**

**P: {<IN>} # Preposition**

**V: {<V.\*>} # Verb**

**PP: {<P> <NP>} # PP -> P NP**

**VP: {<V> <NP|PP>\*} # VP -> V (NP|PP)\***

**''')**

**>>>test\_sent="Mr. Obama played a big role in the Health insurance bill"**

**>>>test\_sent\_pos=nltk.pos\_tag(nltk.word\_tokenize(test\_sent))**

**>>>paresed\_out=reg\_parser.parse(test\_sent\_pos)**

**>>> print paresed\_out**

**Tree('S', [('Mr.', 'NNP'), ('Obama', 'NNP'), Tree('VP', [Tree('V',**

**[('played', 'VBD')]), Tree('NP', [('a', 'DT'), ('big', 'JJ'), ('role',**

**'NN')])]), Tree('P', [('in', 'IN')]), ('Health', 'NNP'), Tree('NP',**

**[('insurance', 'NN'), ('bill', 'NN')])])**

The following is a graphical representation of the tree for the preceding code:

Root

Mr

.

Obama

played

DT

VBD

NNP

NP

NP

NP

DT

NN

NN

NN

IN

PP

in

insurance

role

big

JJ

VP

s

a

NNP

NNP

Health

the

bill

In the current example, we define the kind of patterns (a regular expression of the POS) we think will make a phrase, for example, anything that {<DT>? <JJ>\* <NN>\*} has a starting determiner followed by an adjective and then a noun is mostly a noun phrase. Now, this is more of a linguistic rule that we have defined to get the rule-based parse tree.

## Dependency parsing

**Dependency parsing** (**DP**) is a modern parsing mechanism. The main concept of DP is that each linguistic unit (*words*) is connected with each other by a directed link. These links are called **dependencies** in linguistics. There is a lot of work going on in the current parsing community. While **phrase structure parsing** is still widely used for free word order languages (Czech and Turkish), dependency parsing has turned out to be more efficient.

A very clear distinction can be made by looking at the parse tree generated by phrase structure grammar and dependency grammar for a given example, as the sentence "The big dog chased the cat". The parse tree for the preceding sentence is:

**Phrase**

**Structure**

**tree**

**Dependency**

**Tree**

The

big

dog

chased

the

cat

S

NP

VP

Art

Adj

N

V

N

P

Art

N

the

big

dog

chased

the

cat

If we look at both parse trees, the phrase structures try to capture the relationship between words and phrases and then eventually between phrases. While a dependency tree just looks for a dependency between words, for example, *big* is totally dependent on *dog*.

NLTK provides a couple of ways to do dependency parsing. One of them is to use a **probabilistic**, **projective dependency parser**, but it has the restriction of training with a limited set of training data. One of the state of the art dependency parsers is a Stanford parser. Fortunately, NLTK has a wrapper around it and in the following example, I will talk about how to use a Stanford parser with NLTK:

**# Stanford Parser [Very useful]**

**>>>from nltk.parse.stanford import StanfordParser**

**>>>english\_parser = StanfordParser('stanford-parser.jar', 'stanfordparser-3.4-models.jar') >>>english\_parser.raw\_parse\_sents(("this is the english parser test")**

**Parse**

**(ROOT**

**(S**

**(NP (DT this))**

**(VP (VBZ is)**

**(NP (DT the) (JJ english) (NN parser) (NN test))))) Universal dependencies nsubj(test-6, this-1) cop(test-6, is-2) det(test-6, the-3) amod(test-6, english-4) compound(test-6, parser-5) root(ROOT-0, test-6)**

**Universal dependencies, enhanced nsubj(test-6, this-1) cop(test-6, is-2) det(test-6, the-3) amod(test-6, english-4) compound(test-6, parser-5) root(ROOT-0, test-6)**

The output looks quite complex but, in reality, it's not. The output is a list of three major outcomes, where the first is just the POS tags and the parsed tree of the given sentences. The same is plotted in a more elegant way in the following figure. The second is the dependency and positions of the given words. The third is the enhanced version of dependency:

Root

is

NP

s

VBZ

VP

this

DT

NP

DT

NN

NN

parser

JJ

english

the

test

For a better understanding of how to use a Stanford parser, refer to

<http://nlpviz.bpodgursky.com/home> and <http://nlp.stanford.edu:8080/parser/index.jsp>.

## Chunking

Chunking is shallow parsing where instead of reaching out to the deep structure of the sentence, we try to club some chunks of the sentences that constitute some meaning.

A chunk can be defined as the minimal unit that can be processed. So, for example, the sentence "the President speaks about the health care reforms" can be broken into two chunks, one is "the President", which is noun dominated, and hence is called a **noun phrase** (**NP**). The remaining part of the sentence is dominated by a verb, hence it is called a **verb phrase** (**VP**). If you see, there is one more sub-chunk in the part "speaks about the health care reforms". Here, one more NP exists that can be broken down again in "speaks about" and "health care reforms", as shown in the following figure:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VP   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | The | President | speaks | about | The | Health | Care | Reforms |   NP NP |

This is how we broke the sentence into parts and that's what we call chunking. Formally, chunking can also be described as a processing interface to identify non-overlapping groups in unrestricted text.

Now, we understand the difference between shallow and deep parsing. When we reach the syntactic structure of the sentences with the help of CFG and understand the syntactic structure of the sentence. Some cases we need to go for semantic parsing to understand the meaning of the sentence. On the other hand, there are cases where, we don't need analysis this deep. Let's say, from a large portion of unstructured text, we just want to extract the key phrases, named entities, or specific patterns of the entities. For this, we will go for shallow parsing instead of deep parsing because deep parsing involves processing the sentence against all the grammar rules and also the generation of a variety of syntactic tree till the parser generates the best tree by using the process of backtracking and reiterating. This entire process is time consuming and cumbersome and, even after all the processing, you might not get the right parse tree. Shallow parsing guarantees the shallow parse structure in terms of chunks which is relatively faster. So, let's write some code snippets to do some basic chunking:

**# Chunking**

**>>>from nltk.chunk.regexp import \***

**>>>test\_sent="The prime minister announced he had asked the chief government whip, Philip Ruddock, to call a special party room meeting for 9am on Monday to consider the spill motion."**

**>>>test\_sent\_pos=nltk.pos\_tag(nltk.word\_tokenize(test\_sent))**

**>>>rule\_vp = ChunkRule(r'(<VB.\*>)?(<VB.\*>)+(<PRP>)?', 'Chunk VPs') >>>parser\_vp = RegexpChunkParser([rule\_vp],chunk\_label='VP')**

**>>>print parser\_vp.parse(test\_sent\_pos)**

**>>>rule\_np = ChunkRule(r'(<DT>?<RB>?)?<JJ|CD>\*(<JJ|CD><,>)\*(<NN.\*>)+',**

**'Chunk NPs')**

**>>>parser\_np = RegexpChunkParser([rule\_np],chunk\_label="NP")**

**>>>print parser\_np.parse(test\_sent\_pos)**

**(S The/DT prime/JJ minister/NN (VP announced/VBD he/PRP) (VP had/VBD asked/VBN) the/DT chief/NN government/NN whip/NN ….**

**….**

**…. (VP consider/VB) the/DT spill/NN motion/NN ./.)**

**(S (NP The/DT prime/JJ minister/NN) # 1st noun phrase announced/VBD he/PRP had/VBD asked/VBN**

**(NP the/DT chief/NN government/NN whip/NN) # 2nd noun phrase ,/,**

**(NP Philip/NNP Ruddock/NNP)**

**,/,**

**to/TO call/VB**

**(NP a/DT special/JJ party/NN room/NN meeting/NN) # 3rd noun phrase for/IN 9am/CD on/IN (NP Monday/NNP) # 4th noun phrase to/TO consider/VB (NP the/DT spill/NN motion/NN) # 5th noun phrase**

**./.)**

The preceding code is good enough to do some basic chunking of verb and noun phrases. A conventional pipeline in chunking is to tokenize the POS tag and the input string before they are ed to any chunker. Here, we use a regular chunker, as rule NP / VP defines different POS patterns that can be called a verb/noun phrase. For example, the NP rule defines anything that starts with the determiner and then there is a combination of an adverb, adjective, or cardinals that can be chunked in to a noun phrase. Regular expression-based chunkers rely on chunk rules defined manually to chunk the string. So, if we are able to write a universal rule that can incorporate most of the noun phrase patterns, we can use regex chunkers. Unfortunately, it's hard to come up with those kind of generic rules; the other approach is to use a machine learning way of doing chunking. We briefly touched upon ne\_chunk() and the Stanford NER tagger that both use a pre-trained model to tag noun phrases.

## Information extraction

We learnt about taggers and parsers that we can use to build a basic information extraction engine. Let's jump directly to a very basic IE engine and how a typical IE engine can be developed using NLTK.

Any sort of meaningful information can be drawn only if the given input stream goes to each of the following NLP steps. We already have enough understanding of sentence tokenization, word tokenization, and POS tagging. Let's discuss NER and relation extraction as well.

A typical information extraction pipeline looks very similar to that shown in the following figure:

Sentence

T

okenization

W

ord

T

okenization

P

art-of

Speech

-

T

agging

Entity

Detection

Relation

Extraction

raw

text

relations

list

(

of

strings)

list

(

of

list

of

strings)

list

(

of

list

of

tuples)

list

(

of

trees)

String

Some of the other preprocessing steps, such as stop word removal and

stemming, are generally ignored and do not add any value to an IE

engine.

### Named-entity recognition (NER)

We already briefly discussed NER generally in the last chapter. Essentially, NER is a way of extracting some of the most common entities, such as names, organizations, and locations. However, some of the modified NER can be used to extract entities such as product names, biomedical entities, author names, brand names, and so on. Let's start with a very generic example where we are given a text file of the content and we need to extract some of the most insightful named entities from it:

**# NP chunking (NER)**

**>>>f=open(# absolute path for the file of text for which we want NER)**

**>>>text=f.read()**

**>>>sentences = nltk.sent\_tokenize(text)**

**>>>tokenized\_sentences = [nltk.word\_tokenize(sentence) for sentence in sentences]**

**>>>tagged\_sentences = [nltk.pos\_tag(sentence) for sentence in tokenized\_ sentences] >>>for sent in tagged\_sentences:**

**>>>print nltk.ne\_chunk(sent)**

In the preceding code, we just followed the same pipeline provided in the preceding figure. We took all the preprocessing steps, such as sentence tokenization, tokenization, POS tagging, and NLTK. NER (pre-trained models) can be used to extract all NERs.

### Relation extraction

Relation extraction is another commonly used information extraction operation. Relation extraction as it sound is the process of extracting the different relationships between different entities. There are variety of the relationship that exist between the entities. We have seen relationship like inheritance/synonymous/analogous. The definition of the relation can be dependent on the Information need. For example in the case where we want to look from unstructured text data who is the writer of which book then authorship could be a relation between the author name and book name. With NLTK the idea is to use the same IE pipeline that we used till NER and extend it with a relation pattern based on the NER tags.

So, in the following code, we used an inbuilt corpus of ieer, where the sentences are tagged till NER and the only thing we need to specify is the relation pattern we want and the kind of NER we want the relation to define. In the following code, a relationship between an organization and a location has been defined and we want to extract all the combinations of these patterns. This can be applied in various ways, for example, in a large corpus of unstructured text, we will be able to identify some of the organizations of our interest with their corresponding location:

**>>>import re**

**>>>IN = re.compile(r'.\*\bin\b(?!\b.+ing)') >>>for doc in nltk.corpus.ieer.parsed\_docs('NYT\_19980315'):**

**>>> for rel in nltk.sem.extract\_rels('ORG', 'LOC', doc, corpus='ieer', pattern = IN): >>>print(nltk.sem.rtuple(rel))**

**[ORG: u'WHYY'] u'in' [LOC: u'Philadelphia']**

**[ORG: u'McGlashan &AMP; Sarrail'] u'firm in' [LOC: u'San Mateo']**

**[ORG: u'Freedom Forum'] u'in' [LOC: u'Arlington']**

**[ORG: u'Brookings Institution'] u', the research group in' [LOC: u'Washington']**

**[ORG: u'Idealab'] u', a self-described business incubator based in' [LOC: u'Los Angeles'] ..**

## Summary

We moved beyond the basic preprocessing steps in this chapter. We looked deeper at NLP techniques, such as parsing and information extraction. We discussed parsing in detail, which parsers are available, and how to use NLTK to do any NLP parsing. You understood the concept of CFG and PCFG and how to learn from a tree bank and build a parser. We talked about shallow and deep parsing and what the difference is between them.

We also talked about some of the information extraction essentials, such as entity extraction and relation extraction. We talked about a typical information extraction engine pipeline. We saw a very small and simple IE engine that can be built in less than 100 lines of code. Think about this kind of system running on an entire Wikipedia dump or an entire web content related to an organization. Cool, isn't it?

We will use some of the topics we've learnt in this chapter in further chapters to build some useful NLP applications.