Introduction

Welcome to the module on Syntactic Processing. The last module on Lexical Processing focussed on text preprocessing and feature extraction, in which you had learnt the following techniques:

*.Regular expressions

*.Tokenization, Stemming, Lemmatization

*.TF-IDF model

*.Phonetic hashing

*.The minimum edit distance algorithm

You also learnt to build a spam detector and spell corrector in the module.

In this module, you will learn algorithms and techniques used to analyse the syntax or the grammatical structure of sentences. In the first session, you will learn the basics of grammar (part-of-speech tags etc.) and write your own algorithms such as HMMs to build POS taggers. In the second session, you will study algorithms to parse the grammatical structure of sentences such as CFGs, PCFGs and dependency parsing. Finally, in the third session, you will learn to build an Information Extraction (IE) system to parse flight booking queries for users using techniques such as Named Entity Recognition (NER). You will also study a class of models called Conditional Random Fields (CRFs) which are widely used for building NER systems.

All these techniques fall under what is called syntactic processing.

Syntactic processing is widely used in applications such as question answering systems, information extraction, sentiment analysis, grammar checking etc.

In this session This session will introduce you to the following topics:

*The What and Why of Syntactic Processing

*Basics of Grammar and Parsing

*Algorithms for Part of Speech Tagging and Hidden Markov Models

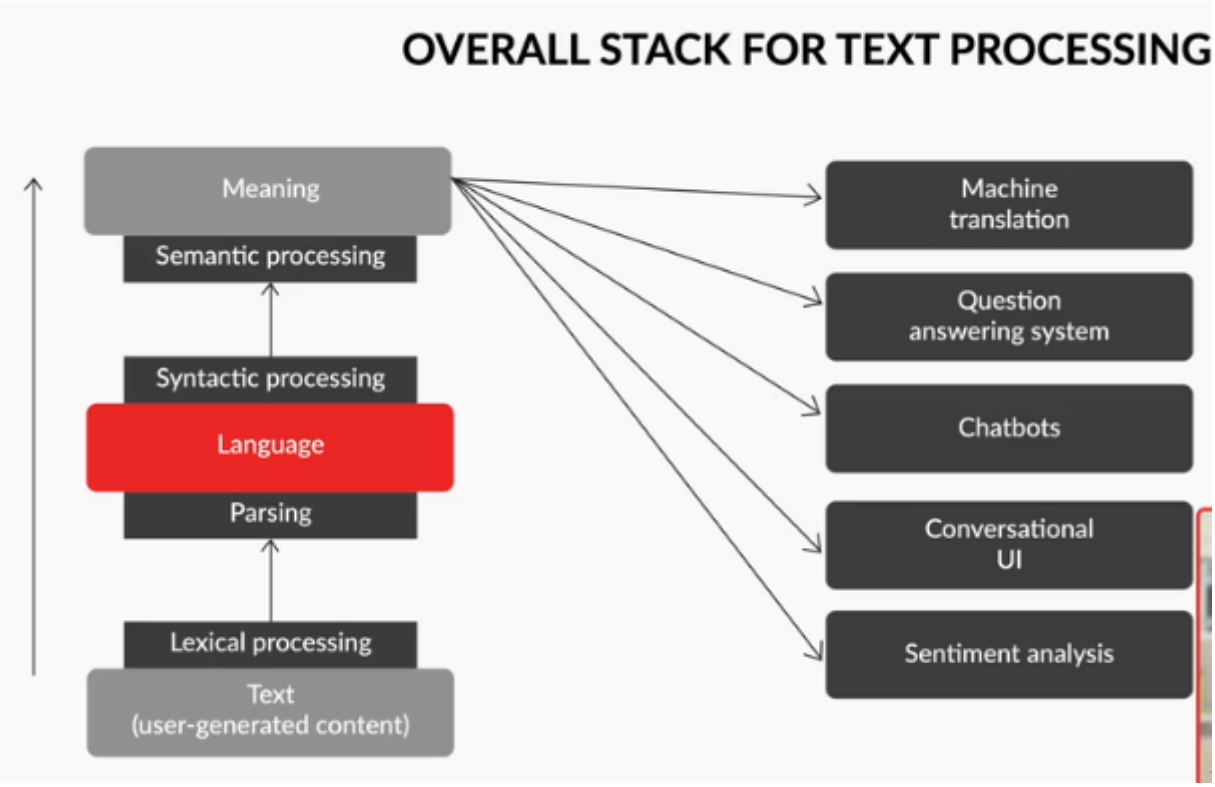
The What and Why of Syntactic Processing

Let’s start with an example to understand Syntactic Processing:

Canberra is the capital of Australia. Is Canberra the of Australia capital. Both sentences have the same set of words, but only the first one is syntactically correct and comprehensible. Basic lexical processing techniques wouldn't be able to tell this difference. Therefore, more sophisticated syntactic processing techniques are required to understand the relationship between individual words in the sentence.

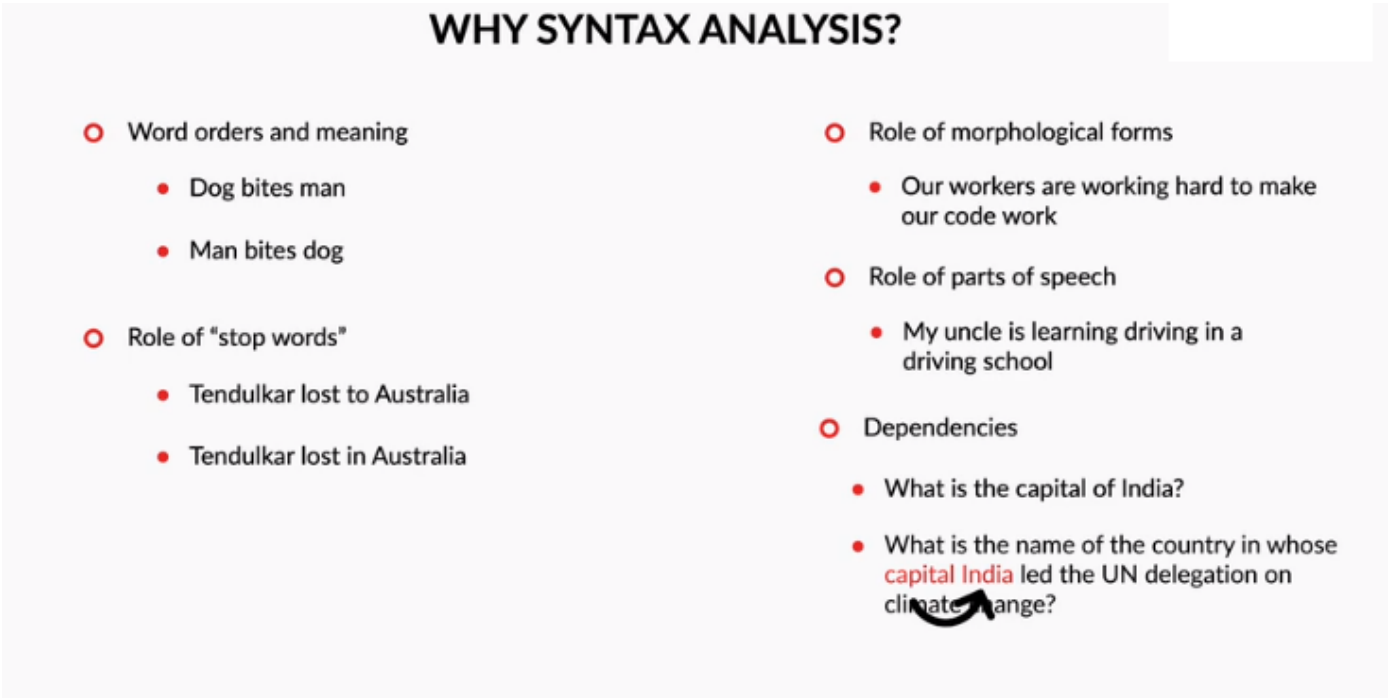
Prof. Me will explain the concept and different industrial applications of syntactic processing.

Recap



You've learnt about lexical processing in the last module. Lexical analysis aims at data cleaning and feature extraction, which it does by using techniques such as lemmatization, removing stopwords, rectifying misspelt words, etc. But, in syntactic analysis, our aim will be to understand the roles played by the words in the sentence, the relationship between words and to parse the grammatical structure of sentences.

Prof. Me will now discuss which lexical processing techniques are irrelevant for syntactic processing.



Syntactical Analysis

Syntactical analysis looks at the following aspects in the sentence which lexical doesn't.

```
1.Words order and meaning: Syntactical analysis aims to find how words are depende
nt on each other. Changing word order will make it difficult to comprehend the sen
tence

2.Retaining stopwords:Removing stopwords can altogether change the meaning of a se
ntence.

3.Morphology of words: Stemming, lemmatisation will bring the words to its base fo
rm, thus modifying the grammar of the sentence.

4.Parts-of-speech of words in a sentence:Identifying correct part-of-speech of a w
ord is important.
Example:

'cuts and bruises on his face' (Here 'cuts' is a noun)

'he cuts an apple' (Here, 'cuts' is a verb)
```

Now that you understand the basic idea of syntactic processing, let's study the different levels of syntactic analysis.

Note that we'll be using the NLTK toolkit of Python for syntactical processing. It's recommended that you download and install the NLTK toolkit (https://www.nltk.org/data.html (https://www.nltk.org/data.html)) on your system, if not already done

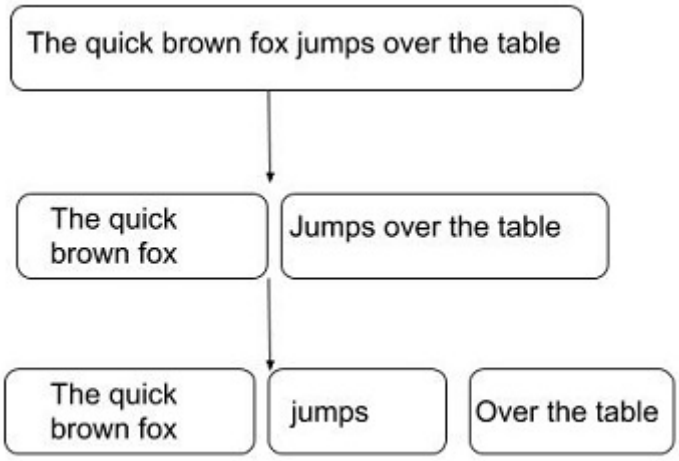
Parsing

A key task in syntactical processing is parsing. It means to break down a given sentence into its 'grammatical constituents'. Parsing is an important step in many applications which helps us better understand the linguistic structure of sentences.

Let's understand parsing through an example. Let's say you ask a question answering (QA) system, such as Amazon's Alexa or Apple's Siri, the following question: "Who won the cricket world cup in 2015?"

The QA system can respond meaningfully only if it can understand that the phrase 'cricket world cup' is related to the phrase 'in 2015'. The phrase 'in 2015' refers to a specific time frame, and thus modifies the question significantly. Finding such dependencies or relations between the phrases of a sentence can be achieved using parsing techniques.

Let's take another example sentence to understand how a parsed sentence looks like: "The quick brown fox jumps over the table". The figure given below shows the three main constituents of this sentence. Note that actual parse trees are different from the simplified representation below.



This structure divides the sentence into three main constituents:

* 'The quick brown fox' is a noun phrase

* 'jumps' is a verb phrase

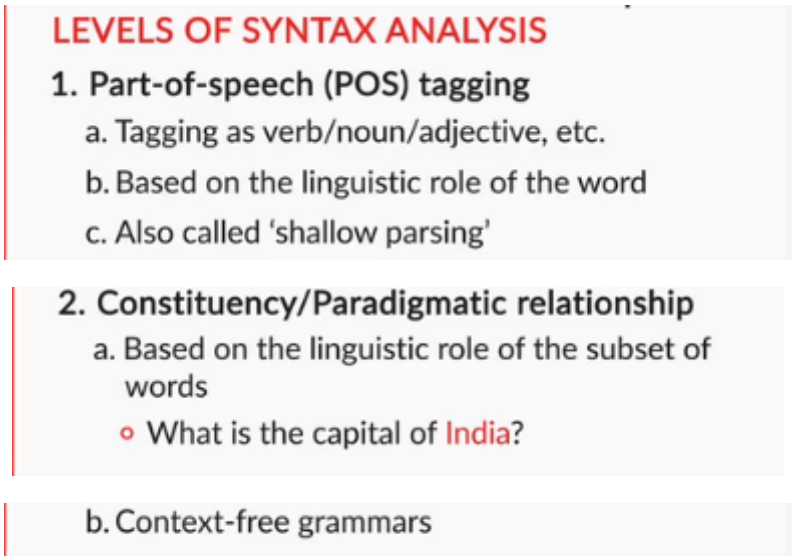
* 'over the table' is a prepositional phrase.

You will study elements of grammar and parsing techniques in the segments that follow. Prof. Me will introduce you to the following different levels of syntactical analysis:

* Part-of-speech tagging

* Constituency parsing

* Dependency parsing



3. Dependencies
   a. Subject, object, modifier, etc
      - The animals that were trapped in the rapidly spreading forest fire, and that spent three days without food or shelter on an isolated barn, were finally rescued yesterday

animals is rescued so we must be able to identify this

To summarise, you will study the following levels of syntactic analysis in this module:

* Part-of-speech (POS) tagging

* Constituency parsing

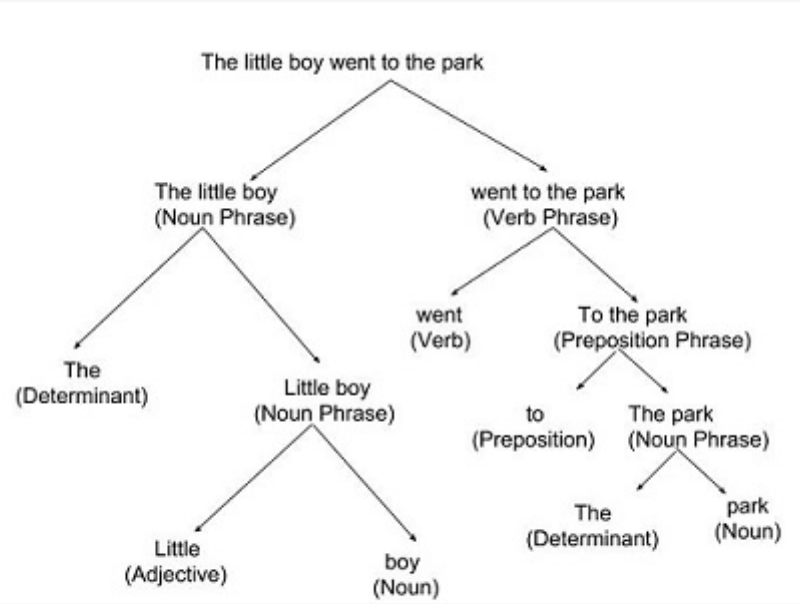
* Dependency parsing

Let's understand the levels of syntax analysis using an example sentence: "The little boy went to the park."

POS tagging is the task of assigning a part of speech tag (POS tag) to each word. The POS tags identify the linguistic role of the word in the sentence. The POS tags of the sentence are:

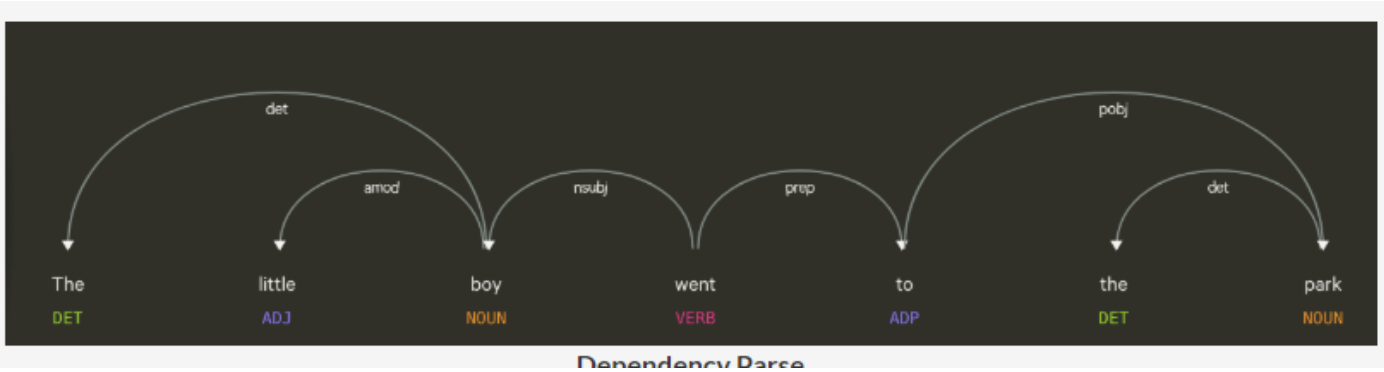
The	little	boy	went	to	the	park
Determinant	**Adjective**	**Noun**	**Verb**	**Preposition**	**Determinant**	**Noun**

Constituency parsers divide the sentence into constituent phrases such as noun phrase, verb phrase, prepositional phrase etc. Each constituent phrase can itself be divided into further phrases. The constituency parse tree given below divides the sentence into two main phrases - a noun phrase and a verb phrase. The verb phrase is further divided into a verb and a prepositional phrase, and so on.



Constituency Parsing

Dependency Parsers do not divide a sentence into constituent phrases, but rather establish relationships directly between the words themselves. The figure below is an example of a dependency parse tree of the sentence given above (generated using the spaCy dependency visualiser https://explosion.ai/demos/displacy?text=The%20little%20boy%20went%20to%20the%20park&model=en_core_web_sm&cpu=0&cph=0) (https://explosion.ai/demos/displacy?text=The%20little%20boy%20went%20to%20the%20park&model=en_core_web_sm&cpu=0&cph=0)). In this module, you'll understand when dependency parsing is more useful than constituency parsing and study the elements of dependency grammar.



Dependency Parse

Parts-of-Speech

Let's start with the first level of syntactic analysis- POS (parts-of-speech) tagging. A word can be tagged as a noun, verb, adjective, adverb, preposition etc. depending upon its role in the sentence. Assigning the correct tag such as noun, verb, adjective etc. is one of the most fundamental tasks in syntactic analysis.

Let's say you ask your smart home device a question - "Ok Google, where can I get the permit to work in Australia?". Now, the word 'permit' can potentially have two POS tags - noun and a verb. In the phrase 'I need a work permit', the correct tag of 'permit' is 'noun'. On the other hand, in the phrase "Please permit me to take the exam.", the word 'permit' is a 'verb'.

Assigning the correct POS tags helps us better understand the intended meaning of a phrase or a sentence and is thus a crucial part of syntactic processing. In fact, all the subsequent parsing techniques (constituency parsing, dependency parsing etc.) use the part-of-speech tags to parse the sentence.

Prof. Me will explain the commonly occurring parts of speech tags.

PART-OF-SPEECH TAGGING

1. Also called 'shallow parsing'
2. What is a 'part of speech'?
3. Terms of the same parts of speech can be substituted with one another to make a syntactically correct sentence
   a. Example: My brother goes to school

Even if I replace brother by cag i.e the noun the sentence is syntactically correct

Example: My cat goes to school

PART-OF-SPEECH TAGS

UpGrad

Part of Speech	Tag	Definition	Example
Noun	NN	Represent entities and concepts	People (Ronnie) Animals (cat) Things (table) Etc.
Proper noun	NNP	Names of entities	City Names (Chennai, Mumbai, Delhi)
Adverbial noun	NR	Nouns that modify other parts of speech	North Delhi, yesterday
Pronoun	PP	Nouns that are salient in a discourse context	He, she He went to school. His name is Ram.
Subject Case Pronoun	PPS	Pronouns that form the subject	They went to school
Object Case Pronoun	PPO	Pronouns that form the object	The bus went to them
Possessive Pronoun	PP$	Object describes a possessor	My car
Reflexive Pronoun	PPL	Refers to a possession relationship	Myself, himself, herself

Note: You do not need to remember all the POS tags except for a few which are listed later on this page. You'll pick up most of these tags as you work on the problems in the coming segments, but it's important to be aware of all the types of tags. Now, let's look at some other tags.

PART-OF-SPEECH TAGS

Part of Speech	Tag	Definition	Example
Determiner	DT	Describes a particular reference to a noun	This, That
Article	AT	Special case of determiners	A, an, the
Adjective	JJ	Describes properties of a noun	Red rose Fast car Lazy cat
Interrogative pronoun	WDT	Wh- determiner Word that starts a question	Which, what, whom Which city did you come from? What did you do?
Verb	VB	Word that descibes an action Can also represent assertion	Ran, threw, walked The capital of India is Delhi
Adverb	RB	Word that modifies a verb	I rarely travel abroad
Preposition	IN	Represents a spatial relationship	Under the bus Over the city In the car
Conjunction	CC	Connects two parts of a sentence	And, because You cannot start a sentence with because because because is a conjunction

LIST OF ALL POS TAGS

Penn Treebank Project

Number	Tag	Description
1.	CC	Coordinating conjuction
2.	CD	Cardinal number
3.	DT	Determiner
4.	EX	Existential *there*
5.	FW	Foreign word
6.	IN	Preposition or subordinating conjuction
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

Number	Tag	Description
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

There are 36 POS tags in the Penn treebank in NLTK. The file given below enlists the most commonly used POS tags. It is recommended to remember these common tags. It'll help you avoid the trouble of looking up meanings of tags in the upcoming segments, so you can focus on the core concepts.

Data/POS_tags.pdf

Note that the set of POS tags is not standard - some books/applications may use only the base forms such as NN, VB, JJ etc without using granular forms, though NLTK uses this set of tags (https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html (https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html)). Since we'll use NLTK heavily, we recommend you to read through this list of tags at least once.

In [17]:

```
import nltk

nltk.help.upenn_tagset()
```

```
$: dollar
    $ -$ --$ A$ C$ HK$ M$ NZ$ S$ U.S.$ US$
'': closing quotation mark
    ' ''
(: opening parenthesis
    ( [ {
): closing parenthesis
    ) ] }
,: comma
    ,
--: dash
    --
.: sentence terminator
    . ! ?
:: colon or ellipsis
    : ; ...
CC: conjunction, coordinating
    & 'n and both but either et for less minus neither nor or plus so
    therefore times v. versus vs. whether vet
```

Different Approaches to POS Tagging

Now that you are familiar with the commonly used POS tags, we will discuss techniques and algorithms for POS tagging. We will look at the following four main techniques used for POS tagging:

* Lexicon-based

* Rule-based

* Probabilistic (or stochastic) techniques

* Deep learning techniques

This session will cover the first three tagging approaches in detail and the basics of deep-learning based taggers. Deep-learning based models will be covered in detail in the Neural Networks course.

Lexicon based Tagger :

Given a actual corpus, for every word in the actual corpus just replace it with the tag which is most likely to occur like below.

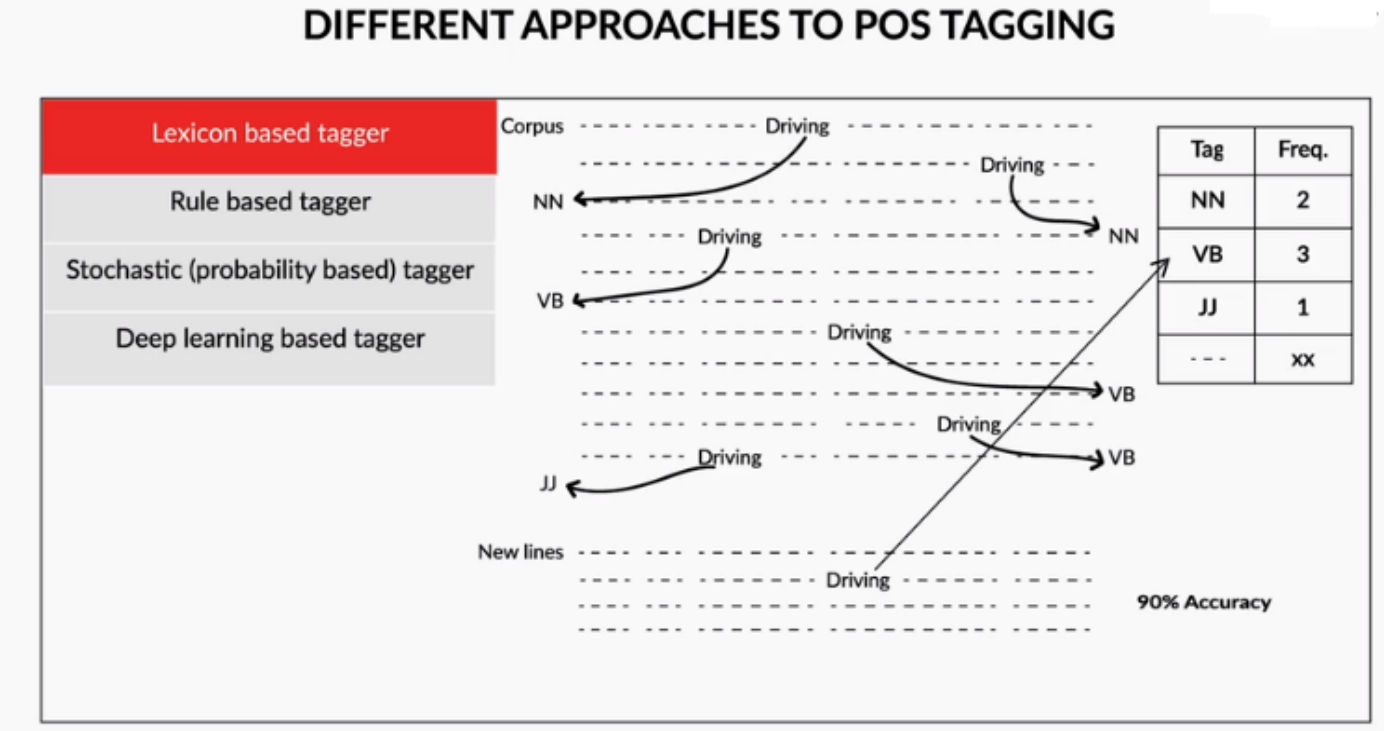
The lexicon-based approach uses the following simple statistical algorithm: for each word, it assigns the POS tag that most frequently occurs for that word in some training corpus. Such a tagging approach cannot handle unknown/ambiguous words. For example:

I went for a run/NN

I run/VB in the morning

Lexicon tagger will tag 'run' basis the highest frequency tag. In most contexts, 'run' is likely to appear as a verb, implying that 'run' will be wrongly tagged in the first sentence.

But if there's a rule that is applied to the entire text, such as, 'replace VB with NN if the previous tag is DT', or 'tag all words ending with ing as VBG', the tag can be corrected. Rule-based tagging methods use such an approach.



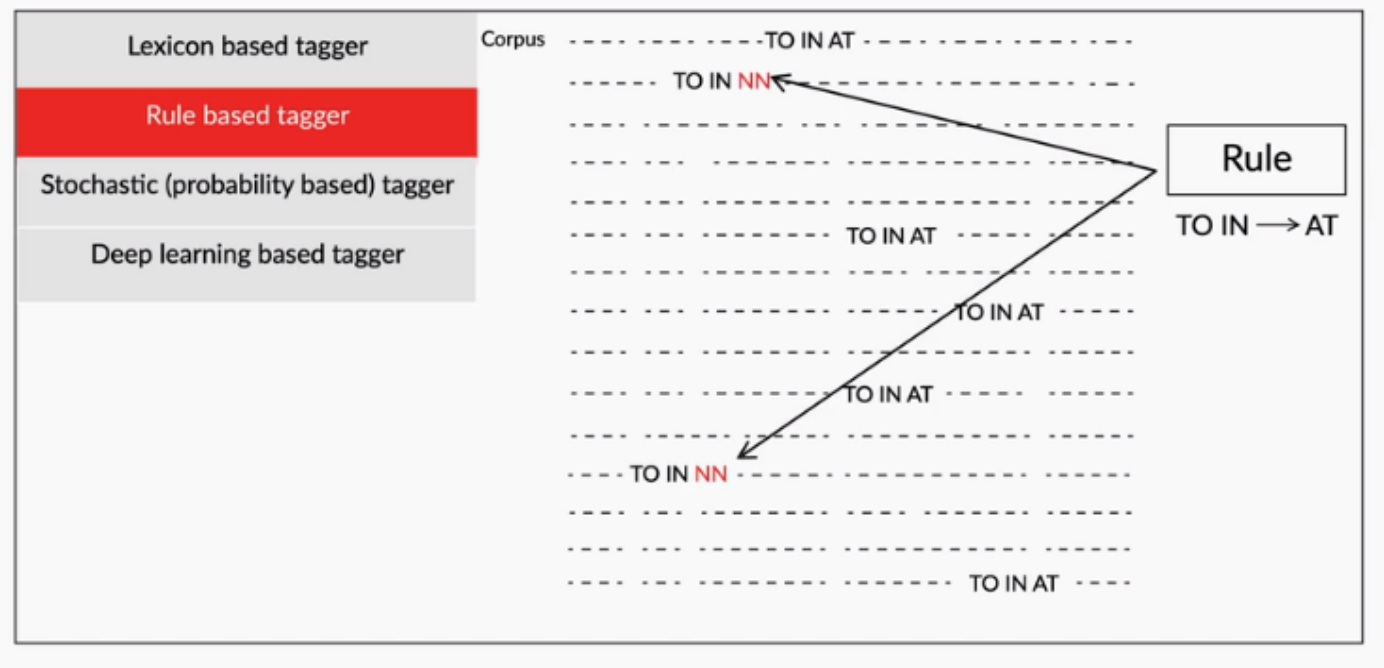
Rule based Tagger

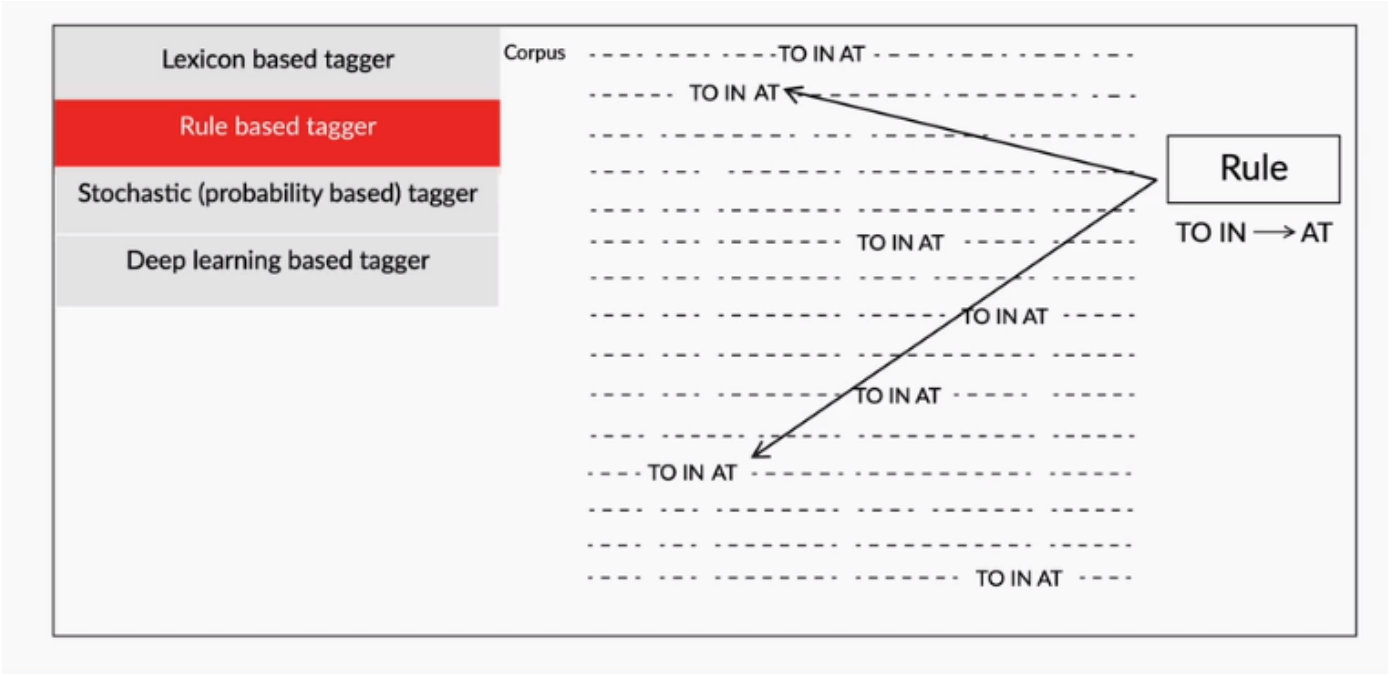
Rule-based taggers first assign the tag using the lexicon method and then apply predefined rules. Some examples of rules are:

Change the tag to VBG for words ending with '-ing'

Changes the tag to VBD for words ending with '-ed'

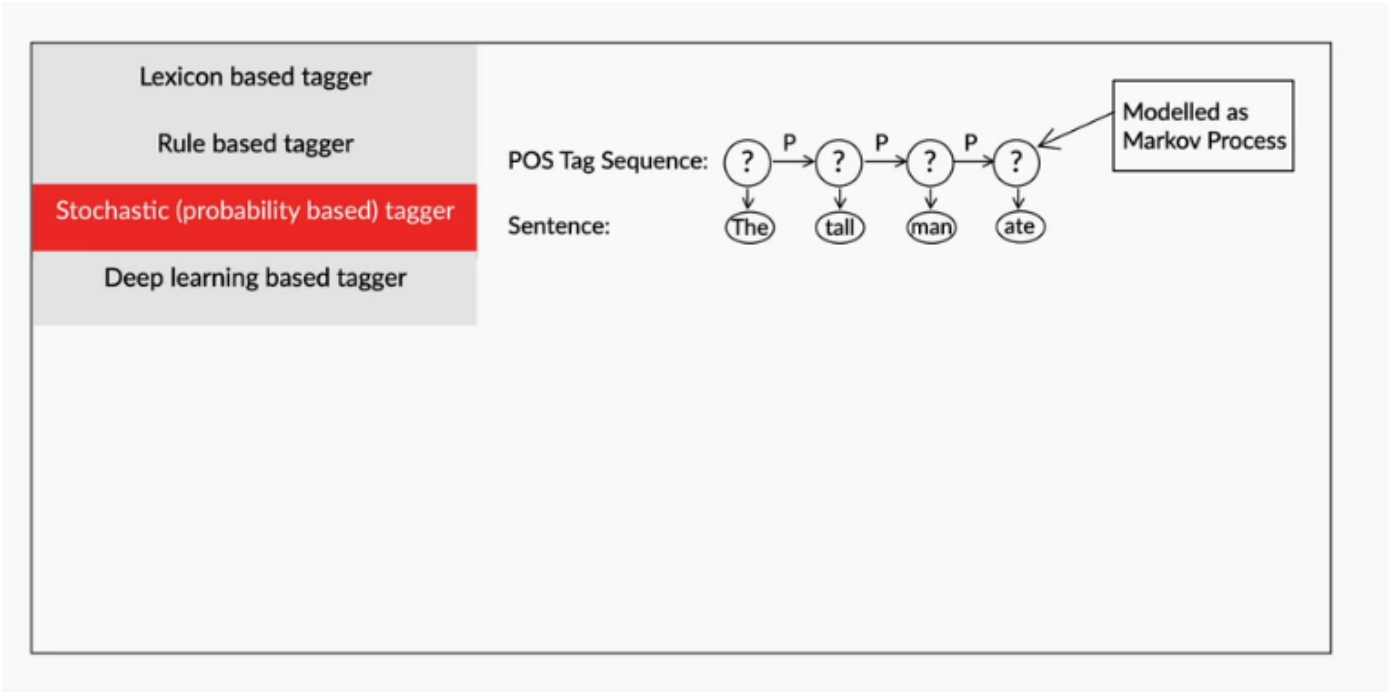
Replace VBD with VBN if the previous word is 'has/have/had'





Probabilistic based Taggers

Probabilistic taggers don't naively assign the highest frequency tag to each word, instead, they look at slightly longer parts of the sequence and often use the tag(s) and the word(s) appearing before the target word to be tagged.



Lexicon and Rule-based POS Tagging

You saw that the lexicon tagger uses a simple statistical tagging algorithm: for each token, it assigns the most frequently assigned POS tag. For example, it will assign the tag "verb" to any occurrence of the word "run" if "run" is used as a verb more often than any other tag.

Rule-based taggers first assign the tag using the lexicon method and then apply predefined rules. Some examples of rules are:

1.Change the tag to VBG for words ending with ‘-ing’

2.Changes the tag to VBD for words ending with ‘-ed’

3.Replace VBD with VBN if the previous word is 'has/have/had'

Defining such rules require some exploratory data analysis and intuition.

In this segment, you’ll learn to implement the lexicon and rule-based tagger on the Treebank corpus of NLTK (http://www.nltk.org/howto/corpus.html#corpus-reader-objects) (http://www.nltk.org/howto/corpus.html#corpus-reader-objects)). Let’s first explore the corpus.

Programming Exercise - Exploratory Analysis for POS Tagging

In the following practice exercise, you will use the Jupyter notebook attached below to answer the following questions. The notebook contains some starter code to read the data and explain its structure. We recommend you to try answering the questions by writing code to conduct some exploratory analysis. The solution to these questions is provided in the TA videos at the bottom of this page.

POS Tagging - Lexicon and Rule Based Taggers

Let's look at the two most basic tagging techniques - lexicon based (or unigram) and rule-based.

In this guided exercise, you will explore the WSJ (wall street journal) POS-tagged corpus that comes with NLTK and build a lexicon and rule-based tagger using this corpus as the tarining data.

This exercise is divided into the following sections:

1. Reading and understanding the tagged dataset
2. Exploratory analysis

1. Reading and understanding the tagged dataset

In [18]:

```
# Importing libraries
import nltk
import numpy as np
import pandas as pd
import pprint, time
import random
from sklearn.model_selection import train_test_split
from nltk.tokenize import word_tokenize
import math
```

In [19]:

```
# reading the Treebank tagged sentences
wsj = list(nltk.corpus.treebank.tagged_sents())
```

In [20]:

```
# samples: Each sentence is a list of (word, pos) tuples
wsj[:3]
```

Out[20]:

```
[[('Pierre', 'NNP'),
  ('Vinken', 'NNP'),
  (',', ','),
  ('61', 'CD'),
  ('years', 'NNS'),
  ('old', 'JJ'),
  (',', ','),
  ('will', 'MD'),
  ('join', 'VB'),
  ('the', 'DT'),
  ('board', 'NN'),
  ('as', 'IN'),
  ('a', 'DT'),
  ('nonexecutive', 'JJ'),
  ('director', 'NN'),
  ('Nov.', 'NNP'),
  ('29', 'CD'),
  ('.', '.')],
 [('Mr.', 'NNP'),
  ('Vinken', 'NNP'),
  ('is', 'VBZ'),
  ('chairman', 'NN'),
  ('of', 'IN'),
  ('Elsevier', 'NNP'),
  ('N.V.', 'NNP'),
  (',', ','),
  ('the', 'DT'),
  ('Dutch', 'NNP'),
  ('publishing', 'VBG'),
  ('group', 'NN'),
  ('.', '.')],
 [('Rudolph', 'NNP'),
  ('Agnew', 'NNP'),
  (',', ','),
  ('55', 'CD'),
  ('years', 'NNS'),
  ('old', 'JJ'),
  ('and', 'CC'),
  ('former', 'JJ'),
  ('chairman', 'NN'),
  ('of', 'IN'),
  ('Consolidated', 'NNP'),
  ('Gold', 'NNP'),
  ('Fields', 'NNP'),
  ('PLC', 'NNP'),
  (',', ','),
  ('was', 'VBD'),
  ('named', 'VBN'),
  ('*-1', '-NONE-'),
  ('a', 'DT'),
  ('nonexecutive', 'JJ'),
```

```
  ('director', 'NN'),
  ('of', 'IN'),
  ('this', 'DT'),
  ('British', 'JJ'),
  ('industrial', 'JJ'),
  ('conglomerate', 'NN'),
  ('.', '.')]]
```

In the list mentioned above, each element of the list is a sentence. Also, note that each sentence ends with a full stop '.' whose POS tag is also a '.'. Thus, the POS tag '.' demarcates the end of a sentence.

Also, we do not need the corpus to be segmented into sentences, but can rather use a list of (word, tag) tuples. Let's convert the list into a (word, tag) tuple.

In [22]:

```
# converting the list of sents to a list of (word, pos tag) tuples
tagged_words = [tup for sent in wsj for tup in sent]
print(len(tagged_words))
tagged_words[:10]
```

```
100676
```

Out[22]:

```
[('Pierre', 'NNP'),
 ('Vinken', 'NNP'),
 (',', ','),
 ('61', 'CD'),
 ('years', 'NNS'),
 ('old', 'JJ'),
 (',', ','),
 ('will', 'MD'),
 ('join', 'VB'),
 ('the', 'DT')]
```

We now have a list of about 100676 (word, tag) tuples. Let's now do some exploratory analyses.

2. Exploratory Analysis

Let's now conduct some basic exploratory analysis to understand the tagged corpus. To start with, let's ask some simple questions:

How many unique tags are there in the corpus? Which is the most frequent tag in the corpus? Which tag is most commonly assigned to the following words: "bank" "executive"

In [23]:

```
# question 1: Find the number of unique POS tags in the corpus
# you can use the set() function on the list of tags to get a unique set of tags,
# and compute its length
tags =  [pair[1] for pair in tagged_words]
unique_tags = set(tags)
len(unique_tags)
```

Out[23]:

```
46
```

In [26]:

```
# question 2: Which is the most frequent tag in the corpus
# to count the frequency of elements in a list, the Counter() class from collections
# module is very useful, as shown below

from collections import Counter
tag_counts = Counter(tags)
tag_counts
#type(tags)
```

Out[26]:

```
Counter({'NNP': 9410,
         ',': 4886,
         'CD': 3546,
         'NNS': 6047,
         'JJ': 5834,
         'MD': 927,
         'VB': 2554,
         'DT': 8165,
         'NN': 13166,
         'IN': 9857,
         '.': 3874,
         'VBZ': 2125,
         'VBG': 1460,
         'CC': 2265,
         'VBD': 3043,
         'VBN': 2134,
         '-NONE-': 6592,
         'RB': 2822,
         'TO': 2179,
         'PRP': 1716,
         'RBR': 136,
         'WDT': 445,
         'VBP': 1321,
         'RP': 216,
         'PRP$': 766,
         'JJS': 182,
         'POS': 824,
         '``': 712,
         'EX': 88,
         "''": 694,
         'WP': 241,
         ':': 563,
         'JJR': 381,
         'WRB': 178,
         '$': 724,
         'NNPS': 244,
         'WP$': 14,
         '-LRB-': 120,
         '-RRB-': 126,
         'PDT': 27,
         'RBS': 35,
         'FW': 4,
         'UH': 3,
         'SYM': 1,
```

```
          'LS': 13,
          '#': 16})
```

In [29]:

```
# the most common tags can be seen using the most_common() method of Counter
tag_counts.most_common(5)
```

Out[29]:

```
[('NN', 13166), ('IN', 9857), ('NNP', 9410), ('DT', 8165), ('-NONE-', 6592)]
```

Thus, NN is the most common tag followed by IN, NNP, DT, -NONE- etc. You can read the exhaustive list of tags using the NLTK documentation as shown below.

In [30]:

```
# list of POS tags in NLTK
nltk.help.upenn_tagset()
```

```
$: dollar
    $ -$ --$ A$ C$ HK$ M$ NZ$ S$ U.S.$ US$
'': closing quotation mark
    ' ''
(: opening parenthesis
    ( [ {
): closing parenthesis
    ) ] }
,: comma
    ,
--: dash
    --
.: sentence terminator
    . ! ?
:: colon or ellipsis
    : ; ...
CC: conjunction, coordinating
    & 'n and both but either et for less minus neither nor or plus so
    therefore times v. versus vs. whether yet
```

In [34]:

```
# question 3: Which tag is most commonly assigned to the word w. Get the tags list that app
# Try w ='bank'
bank = [pair for pair in tagged_words if pair[0].lower() == 'bank']
bank
```

Out[34]:

```
[('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
```

```
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('bank', 'NN'),
 ('Bank', 'NNP'),
 ('bank', 'NN')]
```

In [35]:

```
Counter(bank)
```

Out[35]:

```
Counter({('bank', 'NN'): 38, ('Bank', 'NNP'): 32})
```

In [36]:

```
# question 3: Which tag is most commonly assigned to the word w. Try 'executive'
executive = Counter([pair for pair in tagged_words if pair[0].lower() == 'executive'])
Counter(executive)
```

Out[36]:

```
Counter({('executive', 'NN'): 40, ('executive', 'JJ'): 28})
```

2. Exploratory Analysis Contd.

Until now, we were looking at the frequency of tags assigned to particular words, which is the basic idea used by lexicon or unigram taggers. Let's now try observing some rules which can potentially be used for POS tagging.

To start with, let's see if the following questions reveal something useful:

4. What fraction of words with the tag 'VBD' (verb, past tense) end with the letters 'ed'
5. What fraction of words with the tag 'VBG' (verb, present participle/gerund) end with the letters 'ing'

In [42]:

```
# 4. What fraction of words with the tag 'VBD' (verb, past tense) end with the letters 'ed'

past_tense_verbs = [pair for pair in tagged_words if pair[1] == 'VBD']
ed_verbs = [pair for pair in past_tense_verbs if pair[0].endswith('ed')]
print(len(ed_verbs)/len(past_tense_verbs))
ed_verbs[:20]
```

```
0.3881038448899113
```

Out[42]:

```
[('reported', 'VBD'),
 ('stopped', 'VBD'),
 ('studied', 'VBD'),
 ('led', 'VBD'),
 ('worked', 'VBD'),
 ('explained', 'VBD'),
 ('imposed', 'VBD'),
 ('dumped', 'VBD'),
 ('poured', 'VBD'),
 ('mixed', 'VBD'),
 ('described', 'VBD'),
 ('ventilated', 'VBD'),
 ('contracted', 'VBD'),
 ('continued', 'VBD'),
 ('eased', 'VBD'),
 ('ended', 'VBD'),
 ('lengthened', 'VBD'),
 ('reached', 'VBD'),
 ('resigned', 'VBD'),
 ('approved', 'VBD')]
```

In [44]:

```
# 5 .What fraction of words with the tag 'VBG' (verb, present participle/gerund) end with t
participle_verbs = [pair for pair in tagged_words if pair[1] == 'VBG']
ing_verbs = [pair for pair in participle_verbs if pair[0].endswith('ing')]
print(len(ed_verbs)/len(participle_verbs))
ing_verbs[:20]
```

```
0.9972602739726028
```

Out[44]:

```
[('publishing', 'VBG'),
 ('causing', 'VBG'),
 ('using', 'VBG'),
 ('talking', 'VBG'),
 ('having', 'VBG'),
 ('making', 'VBG'),
 ('surviving', 'VBG'),
 ('including', 'VBG'),
 ('including', 'VBG'),
 ('according', 'VBG'),
 ('remaining', 'VBG'),
 ('according', 'VBG'),
 ('declining', 'VBG'),
 ('rising', 'VBG'),
 ('yielding', 'VBG'),
 ('waiving', 'VBG'),
 ('holding', 'VBG'),
 ('holding', 'VBG'),
 ('cutting', 'VBG'),
 ('manufacturing', 'VBG')]
```

In [ ]:

In [ ]:

1. what fraction of adjectives JJ are followed by a noun NN
2. what fraction of determiners DT are followed by a noun NN
3. what fraction of modals MD are followed by a verb VB?

In [48]:

```
# question: what fraction of adjectives JJ are followed by a noun NN

# create a list of all tags (without the words)
tags = [pair[1] for pair in tagged_words]

# create a list of JJ tags
jj_tags = [t for t in tags if t == 'JJ']

# create a list of (JJ, NN) tags
jj_nn_tags = [(t,tags[index + 1]) for index, t in enumerate(tags)
              if t == 'JJ' and tags[index + 1] == 'NN']

print(len(jj_tags))
print(len(jj_nn_tags))
print(len(jj_nn_tags) / len(jj_tags))
jj_nn_tags[:10]
```

```
5834
2611
0.4475488515598217
```

Out[48]:

```
[('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN'),
 ('JJ', 'NN')]
```

In [50]:

```
# question: what fraction of determiners DT are followed by a noun NN
dt_tags = [t for t in tags if t == 'DT']
dt_nn_tags = [(t,tags[index + 1]) for index, t in enumerate(tags)
              if t == 'DT' and tags[index + 1] == 'NN']

print(len(dt_tags))
print(len(dt_nn_tags))
print(len(dt_nn_tags) / len(dt_tags))
```

```
8165
3844
0.470789957134109
```

In [51]:

```
# question: what fraction of modals MD are followed by a verb VB?
md_tags = [t for t in tags if t == 'MD']
md_vb_tags = [(t,tags[index + 1]) for index, t in enumerate(tags)
              if t == 'MD' and tags[index + 1] == 'VB']

print(len(md_tags))
print(len(md_vb_tags))
print(len(md_vb_tags) / len(md_tags))
```

```
927
756
0.8155339805825242
```

Thus, we see that the probability of certain tags appearing after certain other tags is quite high, and this fact can be used to build quite efficient POS tagging algorithms.

Now that you have an intuition of how lexicon and rule-based taggers work, let's build these taggers in NLTK. Since NLTK comes with built-in functions for lexicon and rule-based taggers, called Unigram and Regular Expression taggers respectively in NLTK, we'll use them to train taggers using the Penn Treebank corpus

3. Lexicon and Rule-Based Models for POS Tagging

Let's now see lexicon and rule-based models for POS tagging. We'll first split the corpus into training and test sets and then use built-in NLTK taggers.

3.1 Splitting into Train and Test Sets

In [52]:

```
# splitting into train and test
random.seed(1234)
train_set, test_set = train_test_split(wsj, test_size=0.3)

print(len(train_set))
print(len(test_set))
print(train_set[:2])
```

```
2739
1175
[[('Mr.', 'NNP'), ('Chase', 'NNP'), ('did', 'VBD'), ("n't", 'RB'), ('retur
n', 'VB'), ('a', 'DT'), ('telephone', 'NN'), ('call', 'NN'), ('to', 'TO'),
('his', 'PRP$'), ('office', 'NN'), ('.', '.')], [('If', 'IN'), ('he', 'PR
P'), ('does', 'VBZ'), ('not', 'RB'), (',', ','), ('he', 'PRP'), ('will', 'M
D'), ('help', 'VB'), ('*-1', '-NONE-'), ('realize', 'VB'), ('Madison', 'NN
P'), ("'s", 'POS'), ('fear', 'NN'), ('in', 'IN'), ('The', 'DT'), ('Federalis
t', 'NNP'), ('No.', 'NN'), ('48', 'CD'), ('of', 'IN'), ('a', 'DT'), ('legisl
ature', 'NN'), ('``', '``'), ('everywhere', 'RB'), ('extending', 'VBG'), ('t
he', 'DT'), ('sphere', 'NN'), ('of', 'IN'), ('its', 'PRP$'), ('activity', 'N
N'), ('and', 'CC'), ('drawing', 'VBG'), ('all', 'DT'), ('powers', 'NNS'),
('into', 'IN'), ('its', 'PRP$'), ('impetuous', 'JJ'), ('vortex', 'NN'),
('.', '.'), ("''", "''")]]
```

3.2 Lexicon (Unigram) Tagger

Let's now try training a lexicon (or a unigram) tagger which assigns the most commonly assigned tag to a word.

In NLTK, the `UnigramTagger()` can be used to train such a model.

In [53]:

```
# Lexicon (or unigram tagger)
unigram_tagger = nltk.UnigramTagger(train_set)
unigram_tagger.evaluate(test_set)
```

```
...
```

Even a simple unigram tagger seems to perform fairly well.

3.3. Rule-Based (Regular Expression) Tagger

Now let's build a rule-based, or regular expression based tagger. In NLTK, the RegexpTagger() can be provided with handwritten regular expression patterns, as shown below.

In the example below, we specify regexes for gerunds and past tense verbs (as seen above), 3rd singular present verb (creates, moves, makes etc.), modal verbs MD (should, would, could), possesive nouns (partner's, bank's etc.), plural nouns (banks, institutions), cardinal numbers CD and finally, if none of the above rules are applicable to a word, we tag the most frequent tag NN.

In [54]:

```
# specify patterns for tagging
# example from the NLTK book
patterns = [
    (r'.*ing$', 'VBG'),               # gerund
    (r'.*ed$', 'VBD'),                # past tense
    (r'.*es$', 'VBZ'),                # 3rd singular present
    (r'.*ould$', 'MD'),               # modals
    (r'.*\'s$', 'NN$'),               # possessive nouns
    (r'.*s$', 'NNS'),                 # plural nouns
    (r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # cardinal numbers
    (r'.*', 'NN')                     # nouns
]
```

In [55]:

```
regexp_tagger = nltk.RegexpTagger(patterns)
# help(regexp_tagger)
```

In [56]:

```
regexp_tagger.evaluate(test_set)
```

Out[56]:

```
0.22165121590756415
```

3.4 Combining Taggers

Let's now try combining the taggers created above. We saw that the rule-based tagger by itself is quite ineffective since we've only written a handful of rules. However, if we could combine the lexicon and the rule-based tagger, we can potentially create a tagger much better than any of the individual ones.

NLTK provides a convenient way to combine taggers using the 'backup' argument. In the following code, we create a regex tagger which is used as a backup tagger to the lexicon tagger, i.e. when the tagger is not able to tag using the lexicon (in case of a new word not in the vocabulary), it uses the rule-based tagger.

Also, note that the rule-based tagger itself is backed up by the tag 'NN'.

In [57]:

```
# rule based tagger
rule_based_tagger = nltk.RegexpTagger(patterns)

# lexicon backed up by the rule-based tagger
lexicon_tagger = nltk.UnigramTagger(train_set, backoff=rule_based_tagger)

lexicon_tagger.evaluate(test_set)
```

Out[57]:

```
0.9082695149805186
```

In [58]:

```
bigram_tagger = nltk.BigramTagger(train_set,backoff=lexicon_tagger)
bigram_tagger.evaluate(test_set)
```

Out[58]:

```
0.9171033185543463
```

You can refer to this Stack Overflow answer to learn more about the backoff technique (https://stackoverflow.com/questions/17259970/tagging-pos-in-nltk-using-backoff-ngrams (https://stackoverflow.com/questions/17259970/tagging-pos-in-nltk-using-backoff-ngrams)). Next, you will study a widely used probabilistic POS tagging model - the Hidden Markov Model (HMM).

Stochastic Parsing

In the following segments, you'll study a commonly used probabilistic algorithm for POS tagging - the Hidden Markov Model (HMM).

Before moving ahead, it will be useful to recall the Bayes' theorem and the chain rule of probability which you had learnt in the Naive Bayes module. Say you have two features X= (x1, x2) and a binary target variable y (class = 0/1). According to the Bayes' rule, the probability of a point (x1, x2) belonging to the class c1 is given by:

$$P(class = c1|x1, x2) = \frac{P(x1,x2|c1).P(c1)}{P(x1,x2)}$$

Now, according to the chain rule of probability, the term P(x1, x2 | c1) can be rewritten as P(x1|c1). P(x2|c1). You can now compute all the probabilities on the right-hand side to compute P(class = c1 | x1, x2).

A similar idea is used by probabilistic parsers to assign POS tags to sequences of words. Say you want to tag the word sequence 'The high cost' and you want to compute the probability that the tag sequence is (DT, JJ, NN). For simplicity, let's work with only these three POS tags, and also assume that only three tag sequences are possible - (DT, JJ, NN), (DT, NN, JJ) and (JJ, DT, NN).

The probability that the tag sequence is (DT, JJ, NN) can be computed as:

$$P(DT, JJ, NN|The, high, cost) = \frac{P(The,high,cost|DT,JJ,NN).P(DT,JJ,NN)}{P(The,high,cost)}$$

Similarly, we can compute the probabilities of the other two sequences and assign the sequence that has the maximum probability. We will need to use some simplifying assumptions to compute the right-hand side of the equation. Let's see how.

Now, according to the chain rule of probability, the term P(x1, x2 | c1) can be rewritten as P(x1|c1). P(x2|c1). You can now compute all the probabilities on the right-hand side to compute P(class = c1 | x1, x2).

A similar idea is used by probabilistic parsers to assign POS tags to sequences of words. Say you want to tag the word sequence 'The high cost' and you want to compute the probability that the tag sequence is (DT, JJ, NN). For simplicity, let's work with only these three POS tags, and also assume that only three tag sequences are possible - (DT, JJ, NN), (DT, NN, JJ) and (JJ, DT, NN).

The probability that the tag sequence is (DT, JJ, NN) can be computed as:

$$P(DT, JJ, NN|The, high, cost) = \frac{P(The,high,cost|DT,JJ,NN).P(DT,JJ,NN)}{P(The,high,cost)}$$

In [ ]: