# Parts-of-Speech Tagging – Identifying Words

Parts-of-speech (POS) tagging is one of the many tasks in NLP. It is defined as the process of assigning a particular parts-of-speech tag to individual words in a sentence. The parts-of-speech tag identifies whether a word is a noun, verb, adjective, and so on. There are numerous applications of parts-of-speech tagging, such as information retrieval, machine translation, NER, language analysis, and so on.

This chapter will include the following topics:

```
•Creating POS tagged corpora
•Selecting a machine learning algorithm
•Statistical modeling involving the n-gram approach
•Developing a chunker using POS tagged data
```

# Introducing parts-of-speech tagging

Introducing parts-of-speech tagging Parts-of-speech tagging is the process of assigning a category (for example, noun, verb, adjective, and so on) tag to individual tokens in a sentence. In NLTK, taggers are present in the nltk.tag package and it is inherited by the TaggerIbase class.

In [4]:
```
import nltk
text1=nltk.word_tokenize("It is a pleasant day today")
nltk.pos_tag(text1)
```

Out[4]:
```
[('It', 'PRP'),
 ('is', 'VBZ'),
 ('a', 'DT'),
 ('pleasant', 'JJ'),
 ('day', 'NN'),
 ('today', 'NN')]
```

In [5]:
```
nltk.help.upenn_tagset('NNS')
```

```
  NNS: noun, common, plural
      undergraduates scotches bric-a-brac products bodyguards facets coasts
      divestitures storehouses designs clubs fragrances averages
      subjectivists apprehensions muses factory-jobs ...
```

In [6]:
```
nltk.help.upenn_tagset('VB.*')
```

```
VB: verb, base form
    ask assemble assess assign assume atone attention avoid bake balkanize
    bank begin behold believe bend benefit bevel beware bless boil bomb
    boost brace break bring broil brush build ...
VBD: verb, past tense
    dipped pleaded swiped regummed soaked tidied convened halted registered
    cushioned exacted snubbed strode aimed adopted belied figgered
    speculated wore appreciated contemplated ...
VBG: verb, present participle or gerund
    telegraphing stirring focusing angering judging stalling lactating
    hankerin' alleging veering capping approaching traveling besieging
    encrypting interrupting erasing wincing ...
VBN: verb, past participle
    multihulled dilapidated aerosolized chaired languished panelized used
    experimented flourished imitated reunifed factored condensed sheared
    unsettled primed dubbed desired ...
VBP: verb, present tense, not 3rd person singular
    predominate wrap resort sue twist spill cure lengthen brush terminate
    appear tend stray glisten obtain comprise detest tease attract
    emphasize mold postpone sever return wag ...
VBZ: verb, present tense, 3rd person singular
    bases reconstructs marks mixes displeases seals carps weaves snatches
    slumps stretches authorizes smolders pictures emerges stockpiles
    seduces fizzes uses bolsters slaps speaks pleads ...
```

In [7]:
```
text=nltk.word_tokenize("I cannot bear the pain of bear")
nltk.pos_tag(text)
```

Out[7]:
```
[('I', 'PRP'),
 ('can', 'MD'),
 ('not', 'RB'),
 ('bear', 'VB'),
 ('the', 'DT'),
 ('pain', 'NN'),
 ('of', 'IN'),
 ('bear', 'NN')]
```

In [8]:
```
taggedword=nltk.tag.str2tuple('bear/NN')
taggedword
```

Out[8]:
```
('bear', 'NN')
```

In [9]:
```
type(taggedword)
```

Out[9]:
```
tuple
```

In [10]:
```
sentence='''The/DT sacred/VBN Ganga/NNP flows/VBZ in/IN this/DT
region/NN ./. This/DT is/VBZ a/DT pilgrimage/NN ./. People/NNP from/IN
all/DT over/IN the/DT country/NN visit/NN this/DT place/NN ./. '''
```

In [11]:
```
sentence_tags = [nltk.tag.str2tuple(t) for t in sentence.split()]
sentence_tags
```

Out[11]:
```
[('The', 'DT'),
 ('sacred', 'VBN'),
 ('Ganga', 'NNP'),
 ('flows', 'VBZ'),
 ('in', 'IN'),
 ('this', 'DT'),
 ('region', 'NN'),
 ('.', '.'),
 ('This', 'DT'),
 ('is', 'VBZ'),
 ('a', 'DT'),
 ('pilgrimage', 'NN'),
 ('.', '.'),
 ('People', 'NNP'),
 ('from', 'IN'),
 ('all', 'DT'),
 ('over', 'IN'),
 ('the', 'DT'),
 ('country', 'NN'),
 ('visit', 'NN'),
 ('this', 'DT'),
 ('place', 'NN'),
 ('.', '.')]
```

In [12]:
```
taggedtok = ('bear', 'NN')
from nltk.tag.util import tuple2str
tuple2str(taggedtok)
```

Out[12]:
```
'bear/NN'
```

In [17]:
```
from nltk.corpus import treebank
treebank_tagged = treebank.tagged_words(tagset='universal')
tag = nltk.FreqDist(tag for (word, tag) in treebank_tagged)
```

Out[17]:
```
12
```

In [18]:
```
tag
```

Out[18]:
```
FreqDist({'NOUN': 28867, 'VERB': 13564, '.': 11715, 'ADP': 9857, 'DET': 8725,
'X': 6613, 'ADJ': 6397, 'NUM': 3546, 'PRT': 3219, 'ADV': 3171, ...})
```

In [19]:
```
tagpairs = nltk.bigrams(treebank_tagged)
preceders_noun = [x[1] for (x, y) in tagpairs if y[1] == 'NOUN']
freqdist = nltk.FreqDist(preceders_noun)
[tag for (tag, _) in freqdist.most_common()]
```

Out[19]:
```
['NOUN',
 'DET',
 'ADJ',
 'ADP',
 '.',
 'VERB',
 'NUM',
 'PRT',
 'CONJ',
 'PRON',
 'X',
 'ADV']
```

In [20]:
```
tag={}
tag
{}
tag['beautiful']='ADJ'
tag
{'beautiful': 'ADJ'}
tag['boy']='N'
tag['read']='V'
tag['generously']='ADV'
tag
```

Out[20]:
```
{'beautiful': 'ADJ', 'boy': 'N', 'read': 'V', 'generously': 'ADV'}
```

# Default tagging

Default tagging is a kind of tagging that assigns identical parts-of-speech tags to all the tokens. The subclass of SequentialBackoffTagger is DefaultTagger. The choose_tag() method must be implemented by SequentialBackoffTagger.

This method includes the following arguments:

- A collection of tokens
- The index of the token that should be tagged
- The previous tags list

In [21]:
```
from nltk.tag import DefaultTagger
tag = DefaultTagger('NN')
tag.tag(['Beautiful', 'morning'])
```

Out[21]:
```
[('Beautiful', 'NN'), ('morning', 'NN')]
```

In [22]:
```
from nltk.tag import untag
untag([('beautiful', 'NN'), ('morning', 'NN')])
```

Out[22]:
```
['beautiful', 'morning']
```

# Creating POS-tagged corpora

A corpus may be known as a collection of documents. A corpora is the collection of multiple corpus.

In [49]:
```
import nltk
from nltk.corpus import names
```

In [50]:
```
names.fileids()
```

Out[50]:
```
['female.txt', 'male.txt']
```

In [51]:
```
len(names.words('male.txt'))
```

Out[51]:
```
2943
```

In [52]:
```
len(names.words('female.txt'))
```

Out[52]:
```
5001
```

In [53]:
```
from nltk.corpus import words
words.fileids()
```

Out[53]:
```
['en', 'en-basic']
```

In [54]:
```
len(words.words('en'))
```

Out[54]:
```
235886
```

In [55]:
```
len(words.words('en-basic'))
```

Out[55]:
```
850
```

In [72]:
```
from nltk import *
from nltk.tag import *
def pos_tag(tok):
    tagger = load(_POS_TAGGER)
    return tagger.tag(tok)
```

In [73]:
```
def batch_pos_tag(sent):
    tagger = load(_POS_TAGGER)
    return tagger.batch_tag(sent)
```

# Selecting a machine learning algorithm

POS tagging is also referred to as word category disambiguation or grammatical tagging. POS tagging may be of two types: rule-based or stochastic/probabilistic. E. Brill's tagger is based on the rule-based tagging algorithm. A POS classifier takes a document as input and obtains word features. It trains itself with the help of these word features combined with the already available training labels. This type of classifier is referred to as a second order classifier, and it makes use of the bootstrap classifier in order to generate the tags for words. A backoff classifier is one in which backoff procedure is performed. The output is obtained in such a manner that the trigram POS tagger relies on the bigram POS tagger, which in turn relies on the unigram POS tagger.

In NLTK, FastBrillTagger is based on unigram. It makes use of a dictionary of words that are already known and the pos tag information.

# Classification

Classification may be defined as the process of deciding a POS tag for a given input. In supervised classification, a training corpus is used that comprises a word and its correct tag. In unsupervised classification, any pair of words and a correct tag list does not exist:

In supervised classification, during training, a feature extractor accepts the input and labels and generates a set of features. These features set along with the label act as input to machine learning algorithms. During the testing or prediction phase, a feature extractor is used that generates features from unknown inputs, and the output is sent to a classifier model that generates an output in the form of label or pos tag information with the help of machine learning algorithms. The maximum entropy classifier is one in that searches the parameter set in order to maximize the total likelihood of the corpus used for training.

# Statistical modeling involving the n-gram approach

Unigram means a single word. In a unigram tagger, a single token is used to find the particular parts-of-speech tag. Training of UnigramTagger can be performed by providing it with a list of sentences at the time of initialization.

Let's see the following code in NLTK, which performs UnigramTagger training:

In [81]:
```
from nltk.tag import UnigramTagger
from nltk.corpus import treebank
```

In [82]:
```
training= treebank.tagged_sents()[:7000]
```

In [86]:
```
type(training)
```

Out[86]:
```
nltk.collections.LazySubsequence
```

In [87]:
```
len(training)
```

Out[87]:
```
3914
```

In [88]:
```
unitagger=UnigramTagger(training)
```

In [89]:
```
unitagger
```

Out[89]:
```
<UnigramTagger: size=12408>
```

In [90]:
```
type(unitagger)
```

Out[90]:
```
nltk.tag.sequential.UnigramTagger
```

In [94]:
```
treebank.sents()[1]
```

Out[94]:
```
['Mr.',
 'Vinken',
 'is',
 'chairman',
 'of',
 'Elsevier',
 'N.V.',
 ',',
 'the',
 'Dutch',
 'publishing',
 'group',
 '.']
```

In [96]:
```
unitagger.tag(treebank.sents()[1])
```

Out[96]:
```
[('Mr.', 'NNP'),
 ('Vinken', 'NNP'),
 ('is', 'VBZ'),
 ('chairman', 'NN'),
 ('of', 'IN'),
 ('Elsevier', 'NNP'),
 ('N.V.', 'NNP'),
 (',', ','),
 ('the', 'DT'),
 ('Dutch', 'JJ'),
 ('publishing', 'NN'),
 ('group', 'NN'),
 ('.', '.')]
```

To evaluate UnigramTagger, let's see the following code, which calculates the accuracy:

In [97]:
```
testing = treebank.tagged_sents()[2000:]
unitagger.evaluate(testing)
```

Out[97]:
```
0.9619024159944167
```

In [98]:
```
unitag = UnigramTagger(model={'Vinken': 'NN'})
```

In [100]:
```
type(unitag)
```

Out[100]:
```
nltk.tag.sequential.UnigramTagger
```

In [102]:
```
unitag.tag(treebank.sents()[1])
```

Out[102]:
```
[('Mr.', None),
 ('Vinken', 'NN'),
 ('is', None),
 ('chairman', None),
 ('of', None),
 ('Elsevier', None),
 ('N.V.', None),
 (',', None),
 ('the', None),
 ('Dutch', None),
 ('publishing', None),
 ('group', None),
 ('.', None)]
```

Here, in the preceding code, UnigramTagger only tags 'Vinken' with the 'NN' tag and the rest are tagged with the 'None' tag since we have provided the tag for the word 'Vinken' in the context model and no other words are included in the context model.

Backoff tagging may be defined as a feature of SequentialBackoffTagger. All the taggers are chained together so that if one of the taggers is unable to tag a token, then the token may be passed to the next tagger.

Let's see the following code, which uses back-off tagging. Here, DefaultTagger and UnigramTagger are used to tag a token. If any tagger of them is unable to tag a word, then the next tagger may be used to tag it:

In [103]:
```
from nltk.tag import DefaultTagger
testing = treebank.tagged_sents()[2000:]
training= treebank.tagged_sents()[:7000]
tag1=DefaultTagger('NN')
tag2=UnigramTagger(training,backoff=tag1)
tag2.evaluate(testing)
```

Out[103]:
```
0.9619024159944167
```

The subclasses of NgramTagger are UnigramTagger, BigramTagger, and TrigramTagger. BigramTagger makes use of the previous tag as contextual information. TrigramTagger uses the previous two tags as contextual information.

In [105]:
```
training_1= treebank.tagged_sents()[:7000]
bigramtagger=BigramTagger(training_1)
print(treebank.sents()[0])
```

```
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'b
oard', as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']
```

In [106]:
```
print(bigramtagger.tag(treebank.sents()[0]))
```

```
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NN
S'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'),
('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('directo
r', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')]
```

In [107]:
```
testing_1 = treebank.tagged_sents()[2000:]
bigramtagger.evaluate(testing_1)
```

Out[107]: 0.9171131227292321

NgramTagger can be used to generate a tagger for n greater than three as well. Let's see the following code in NLTK, which develops QuadgramTagger:

In [109]:
```
quadgramtag = NgramTagger(4, training)
quadgramtag.evaluate(testing)
```

Out[109]: 0.9304554878173943

In [110]:
```
affixtag = AffixTagger(training)
affixtag.evaluate(testing)          # Uses prefix or suffix as the contextual in
```

Out[110]: 0.2902682841718497

In [111]:
```
prefixtag = AffixTagger(training, affix_length=4)     # learns the use of four ch
prefixtag.evaluate(testing)
```

Out[111]: 0.2094751318841472

The TnT is Trigrams n Tags. TnT is a statistical-based tagger that is based on the second order Markov models

In [ ]:
```
from nltk.tag import tnt
tnt_tagger=tnt.TnT()
tnt_tagger.train(training)
tnt_tagger.evaluate(testing)
```

# Developing a chunker using pos-tagged corpora

Chunking is the process used to perform entity detection. It is used for the segmentation and labeling of multiple sequences of tokens in a sentence. To design a chunker, a chunk grammar should be defined. A chunk grammar holds the rules of how chunking should be done.

Let's consider the example that performs Noun Phrase Chunking by forming the chunk rules:

In [113]:
```
sent=[("A","DT"),("wise", "JJ"), ("small", "JJ"),("girl", "NN"),
("of", "IN"), ("village", "N"), ("became", "VBD"), ("leader", "NN")]
```

In [116]:
```
print(sent)
```

```
[('A', 'DT'), ('wise', 'JJ'), ('small', 'JJ'), ('girl', 'NN'), ('of', 'IN'),
('village', 'N'), ('became', 'VBD'), ('leader', 'NN')]
```

In [117]:
```
grammar = "NP: {<DT>?<JJ>*<NN><IN>?<NN>*}"
find = nltk.RegexpParser(grammar)
res = find.parse(sent)
print(res)
```

```
(S
  (NP A/DT wise/JJ small/JJ girl/NN of/IN)
  village/N
  became/VBD
  (NP leader/NN))
```

In [118]:
```
res.draw()
```

In [125]:
```
noun1=[("financial","NN"),("year","NN"),("account","NN"),("summary","NN")]
gram="NP:{<NN>+}"
find = nltk.RegexpParser(gram)
print(find.parse(noun1))
```

```
(S financial/NN year/NN account/NN summary/NN)
```

In [126]:
```
x=find.parse(noun1)
x.draw()
```

Chunking is the process in which some of the parts of a chunk are eliminated. Either an entire chunk may be used, a part of the chunk may be used from the middle and the remaining parts are eliminated, or a part of chunk may be used either from the beginning of the chunk or from the end of the chunk and the remaining part of the chunk is removed.

In [ ]: