Text Analytics

As we have seen cases of different types of data and different ways of processing them to obtain important information out of them, in this section we will talk about one of the most sought upon data in today’s time. Customer Information is the backbone for most of the renowned companies in 21st century. Let’s take example of Google. How do you suppose Google is providing so many services for free? What profit does it make by giving you free services? Well it’s your information that google sells to different companies by analyzing your searches and makes profit through it. The moment you search for a product on google, you will start seeing the product recommendation on every website. This is the most basic usage of text analytics, product recommendation to customers by analyzing their searches. Similarly, many companies analyze the positive or negative reviews given by customers and try to predict the customer behavior. There is a wealth of such unstructured data present such as emails, google searches, online surveys, twitter, online reviews etc. which can be processed using text analysis. Many key information about people, customers can be derived by processing the unstructured text and analyzing.

Natural Language Processing (NLP)

NLP is a part of Artificial Intelligence, developed for the machine to understand human language. The ultimate goal of NLP is to read, understand and make valuable conclusion of human language. It is a very tough job to do as human language has a lot of variation in terms of language, pronunciation etc. Although, in recent times there has been a major breakthrough in the field of NLP.

Siri and Alexa are one such example of uses of NLP.

We will use NLP for text analytics.

There many libraries available for NLP in python. we will focus on the two most important one's :

- Natural Languange Tool Kit (NLTK)
- Spacy

Before we dive into text analytics using NLTK or Spacy. Let's understand about some important terminologies:

Tokenization

Tokenization is a process of breaking down a given paragraph of text into a list of sentence or words. When paragraph is broken down into list of sentences, it is called sentence tokenization. Similarly, if the sentences are further broken down into list of words, it is known as Word tokenization.

Let's understand this with an example. Below is a given paragraph, let's see how tokenization works on it:

"India (Hindi: Bhārat), officially the Republic of India, is a country in South Asia. It is the seventh-largest country by area, the second-most populous country, and the most populous democracy in the world. Bounded by the Indian Ocean on the south, the Arabian Sea on the southwest, and the Bay of Bengal on the southeast, it shares land borders with Pakistan to the west; China, Nepal, and Bhutan to the north; and Bangladesh and Myanmar to the east. In the Indian Ocean, India is in the vicinity of Sri Lanka and the Maldives; its Andaman and Nicobar Islands share a maritime border with Thailand and Indonesia."

- Sentence Tokenize:

['India (Hindi: Bhārat), officially the Republic of India, is a country in South Asia.',

'It is the seventh-largest country by area, the second-most populous country, and the most populous democracy in the world.',

'Bounded by the Indian Ocean on the south, the Arabian Sea on the southwest, and the Bay of Bengal on the southeast, it shares land borders with Pakistan to the west; China, Nepal, and Bhutan to the north; and Bangladesh and Myanmar to the east.',

'In the Indian Ocean, India is in the vicinity of Sri Lanka and the Maldives; its Andaman and Nicobar Islands share a maritime border with Thailand and Indonesia.']

- Word tokenize:

['India', '(', 'Hindi', ':', 'Bhārat', ')', ',', 'officially', 'the', 'Republic', 'of', 'India', ',', 'is', 'a', 'country', 'in', 'South', 'Asia', '.', 'It', 'is', 'the', 'seventh-largest', 'country', 'by', 'area', ',', 'the', 'second-most', 'populous', 'country', ',', 'and', 'the', 'most', 'populous', 'democracy', 'in', 'the', 'world', '.', 'Bounded', 'by', 'the', 'Indian', 'Ocean', 'on', 'the', 'south', ',', 'the', 'Arabian', 'Sea', 'on', 'the', 'southwest', ',', 'and', 'the', 'Bay', 'of', 'Bengal', 'on', 'the', 'southeast', ',', 'it', 'shares', 'land', 'borders', 'with', 'Pakistan', 'to', 'the', 'west', ';', 'China', ',', 'Nepal', ',', 'and', 'Bhutan', 'to', 'the', 'north', ';', 'and', 'Bangladesh', 'and', 'Myanmar', 'to', 'the', 'east', '.', 'In', 'the', 'Indian', 'Ocean', ',', 'India', 'is', 'in', 'the', 'vicinity', 'of', 'Sri', 'Lanka', 'and', 'the', 'Maldives', ';', 'its', 'Andaman', 'and', 'Nicobar', 'Islands', 'share', 'a', 'maritime', 'border', 'with', 'Thailand', 'and', 'Indonesia', '.']

Hope this example clears up the concept of tokenization. We will understand why it is done when we will dive into text analysis.

In [1]:

```
# Tokenizing using NLTK
import nltk

data = "India (Hindi: Bhārat), officially the Republic of India, is a country i

nltk.sent_tokenize(data)
```

Out[1]:

```
['India (Hindi: Bhārat), officially the Republic of India, is a countr
y in South Asia.',
 'It is the seventh-largest country by area, the second-most populous
country, and the most populous democracy in the world.',
 'Bounded by the Indian Ocean on the south, the Arabian Sea on the sou
thwest, and the Bay of Bengal on the southeast, it shares land borders
with Pakistan to the west; China, Nepal, and Bhutan to the north; and
Bangladesh and Myanmar to the east.',
 'In the Indian Ocean, India is in the vicinity of Sri Lanka and the M
aldives; its Andaman and Nicobar Islands share a maritime border with
Thailand and Indonesia.']
```

In [2]:

```
nltk.word_tokenize(data)
```

Out[2]:

```
['India',
 '(',
 'Hindi',
 ':',
 'Bhārat',
 ')',
 ',',
 'officially',
 'the',
 'Republic',
 'of',
 'India',
 ',',
 'is',
 'a',
 'country',
 'in',
 'South',
```

POS Tags and Chunking

- Parts Of Speech Tagging(POS tags)

As the name suggests, it is a method of tagging individual words on the basis of it's parts of speech.

Wikipedia definition : Part-of-speech tagging (POS tagging or PoS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech,based on both its definition and its context i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is commonly taught to school-age children, in the identification of words as nouns, verbs, adjectives, adverbs, etc.

There are 9 parts of speech in grammars, but in NLP there are more than 9 POS tags based on different set of rules, such as:

- NN noun, singular 'table'
- NNS noun plural 'tables'
- NNP proper noun, singular
- NNPS proper noun, plural

There are 4 types of division for noun only. Similarly, there are multiple divisions for other part of speeches.

In [3]:

```
data =' We will see an example of POS tagging.'

pos = nltk.pos_tag(nltk.word_tokenize(data))

pos
```

Out[3]:

```
[('We', 'PRP'),
 ('will', 'MD'),
 ('see', 'VB'),
 ('an', 'DT'),
 ('example', 'NN'),
 ('of', 'IN'),
 ('POS', 'NNP'),
 ('tagging', 'NN'),
 ('.', '.')]
```

- Chunking

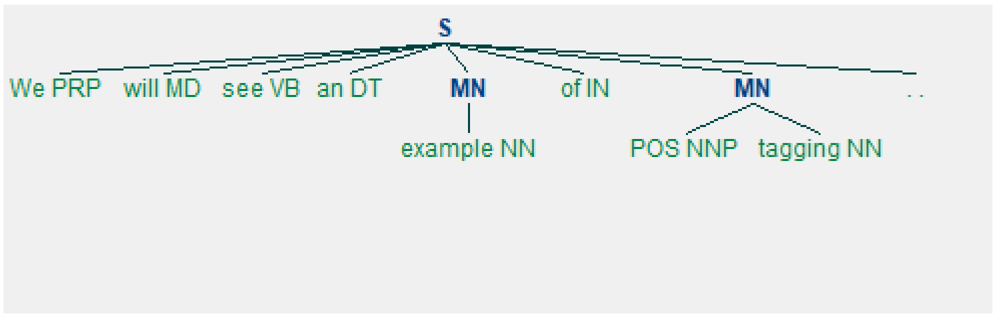
After using parts of speech, Chunking can be used to make data more structured by giving a specific set of rules. Chunking is also known as shallow parser. Let's understand more about chunking by following example :

In [4]:

```
data =' We will see an example of POS tagging.'

pos = nltk.pos_tag(nltk.word_tokenize(data))

# now once the POS tag has been done. Let's say we want to further structure da
# categorized under one specific node defined by us :

my_node = "MN: {<NNP>*<NN>}"

chunk  =nltk.RegexpParser(my_node)
result = chunk.parse(pos)
print(result)
result.draw()    # It will draw the pattern graphically which can be seen in No
```

```
(S
  We/PRP
  will/MD
  see/VB
  an/DT
  (MN example/NN)
  of/IN
  (MN POS/NNP tagging/NN)
  ./.)
```

- Graphical representation



We can see that both NN and NNP are now categorised into "MN" (as the given tag_name).

So, whenever we need to categorise different tags into one tag, we can use chunking for this purpose.

Stop Words

Stop words are such words which are very common in occurrence such as ‘a’,’an’,’the’, ‘at’ etc. We ignore such words during the preprocessing part since they do not give any important information and would just take additional space. We can make our custom list of stop words as well if we want. Different libraries have different stop words list. Let’s see the stop words list for NLTK:

In [5]:

```python
from nltk.corpus import stopwords

stop_words = stopwords.words('english')
stop_words
```

Out[5]:

```
['i',
 'me',
 'my',
 'myself',
 'we',
 'our',
 'ours',
 'ourselves',
 'you',
 "you're",
 "you've",
 "you'll",
 "you'd",
 'your',
 'yours',
 'yourself',
 'yourselves',
 'he',
```

In [6]:

```
# We also have punctuations which we can ignore from our set of words just like

import string

punct =string.punctuation
punct
```

Out[6]:

```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

In [7]:

```
# Let's word tokenize the given sample after we remove the stopwords and punctu

import nltk
import string
from nltk.corpus import stopwords

stop_words = stopwords.words('english')
punct =string.punctuation

data = "India (Hindi: Bhārat), officially the Republic of India, is a country i
clean_data =[]
for word in nltk.word_tokenize(data):
    if word not in punct:
        if word not in stop_words:
            clean_data.append(word)

clean_data
```

Out[7]:

```
['India',
 'Hindi',
 'Bhārat',
 'officially',
 'Republic',
 'India',
 'country',
 'South',
 'Asia',
 'It',
 'seventh-largest',
 'country',
 'area',
 'second-most',
 'populous',
 'country',
 'populous',
 'democracy',
 'world',
 'Bounded',
 'Indian',
 'Ocean',
 'south',
 'Arabian',
 'Sea',
 'southwest',
 'Bay',
 'Bengal',
 'southeast',
 'shares',
 'land',
 'borders',
 'Pakistan',
 'west',
 'China',
 'Nepal',
 'Bhutan',
 'north',
 'Bangladesh',
 'Myanmar',
 'east',
 'In',
 'Indian',
 'Ocean',
 'India',
 'vicinity',
 'Sri',
 'Lanka',
 'Maldives',
 'Andaman',
 'Nicobar',
 'Islands',
 'share',
 'maritime',
 'border',
 'Thailand',
 'Indonesia']
```

Great!! Our data looks so much cleaner now after removing stop words and punctuation.

Hope, this clears up why we should remove stop words and punctuation before processing our data.

Let's see pos tagging for our cleaned data:

In [8]:

```
nltk.pos_tag(clean_data)
```

Out[8]:

```
[('India', 'NNP'),
 ('Hindi', 'NNP'),
 ('Bhārat', 'NNP'),
 ('officially', 'RB'),
 ('Republic', 'NNP'),
 ('India', 'NNP'),
 ('country', 'NN'),
 ('South', 'NNP'),
 ('Asia', 'IN'),
 ('It', 'PRP'),
 ('seventh-largest', 'JJ'),
 ('country', 'NN'),
 ('area', 'NN'),
 ('second-most', 'RB'),
 ('populous', 'JJ'),
 ('country', 'NN'),
 ('populous', 'JJ'),
 ('democracy', 'NN'),
 ('world', 'NN'),
 ('Bounded', 'NNP'),
 ('Indian', 'JJ'),
 ('Ocean', 'NNP'),
 ('south', 'NN'),
 ('Arabian', 'NNP'),
 ('Sea', 'NNP'),
 ('southwest', 'JJS'),
 ('Bay', 'NNP'),
 ('Bengal', 'NNP'),
 ('southeast', 'NN'),
 ('shares', 'NNS'),
 ('land', 'VBP'),
 ('borders', 'NNS'),
 ('Pakistan', 'NNP'),
 ('west', 'JJS'),
 ('China', 'NNP'),
 ('Nepal', 'NNP'),
 ('Bhutan', 'NNP'),
 ('north', 'JJ'),
 ('Bangladesh', 'NNP'),
 ('Myanmar', 'NNP'),
 ('east', 'NN'),
 ('In', 'IN'),
 ('Indian', 'JJ'),
 ('Ocean', 'NNP'),
 ('India', 'NNP'),
 ('vicinity', 'NN'),
 ('Sri', 'NNP'),
 ('Lanka', 'NNP'),
 ('Maldives', 'NNP'),
 ('Andaman', 'NNP'),
 ('Nicobar', 'NNP'),
 ('Islands', 'NNP'),
 ('share', 'NN'),
 ('maritime', 'JJ'),
 ('border', 'NN'),
 ('Thailand', 'NNP'),
 ('Indonesia', 'NNP')]
```

Stemming and Lemmatization

Many words that are used in a sentence are not always used in their basic form but are used as per the rules of grammar e.g.

running ---> run (base word)

runs ---> run (base word)

ran ---> run (base word)

Although, the underlying meaning will be same but form of the base word changes to preserve the correct grammatical meaning.

Stemming and Lemmatization are basically used to bring such words to their basic forms, so that the words with same base are treated as same words rather than treated differently.

The only difference in Stemming and Lemmatization is the way in which they change the word to its base form.

- Stemming

Stemming means mapping a group of words to the same stem by removing prefixes or suffixes without giving any value to the “grammatical meaning” of the stem formed after the process.

e.g.

computation --> comput

computer --> comput

hobbies --> hobbi

We can see that stemming tries to bring the word back to their base word but the base word may or may not have correct grammatical meanings.

There are typically two types of stemmers available in NLTK package. 1) Porter Stemmer 2) Lancaster Stemmer

Let’s see how to use both of them:

In [9]:

```python
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer, SnowballStemmer

lancaster = LancasterStemmer()
porter = PorterStemmer()
Snowball = SnowballStemmer("english")
print('Porter stemmer')
print(porter.stem("hobby"))
print(porter.stem("hobbies"))
print(porter.stem("computer"))
print(porter.stem("computation"))
print("*************************")
print('lancaster stemmer')
print(lancaster.stem("hobby"))
print(lancaster.stem("hobbies"))
print(lancaster.stem("computer"))
print(porter.stem("computation"))
print("*************************")
print('Snowball stemmer')
print(Snowball.stem("hobby"))
print(Snowball.stem("hobbies"))
print(Snowball.stem("computer"))
print(Snowball.stem("computation"))

```

```
Porter stemmer
hobbi
hobbi
comput
comput
*************************
lancaster stemmer
hobby
hobby
comput
comput
*************************
Snowball stemmer
hobbi
hobbi
comput
comput
```

In [10]:

```
sent = "I was going to the office on my bike when i saw a car passing by hit th
token = list(nltk.word_tokenize(sent))
for stemmer in (Snowball, lancaster, porter):
    stemm = [stemmer.stem(t) for t in token]
    print(" ".join(stemm))
```

```
i was go to the offic on my bike when i saw a car pass by hit the tree
.
i was going to the off on my bik when i saw a car pass by hit the tre
.
I wa go to the offic on my bike when i saw a car pass by hit the tree
.
```

lancaster algorithm is faster than porter but it is more complex. Porter stemmer is the oldest algorithm present and was the most popular to use.

Snowball stemmer, also known as porter2, is the updated version of the Porter stemmer and is currently the most popular stemming algorithm.

Snowball stemmer is available for multiple languages as well.

In [11]:

```
print(porter.stem("running"))
print(porter.stem("runs"))
print(porter.stem("ran"))
```

```
run
run
ran
```

- Lemmatization

Lemmatization also does the same thing as stemming and try to bring a word to its base form, but unlike stemming it do keep in account the actual meaning of the base word i.e. the base word belongs to any specific language. The ‘base word’ is known as ‘Lemma’.

We use WordNet Lemmatizer for Lemmatization in nltk.

In [12]:

```
from nltk.stem import WordNetLemmatizer

lemma = WordNetLemmatizer()

print(lemma.lemmatize('running'))
print(lemma.lemmatize('runs'))
print(lemma.lemmatize('ran'))
```

```
running
run
ran
```

Here, we can see the lemma has changed for the words with same base.

This is because, we haven't given any context to the Lemmatizer.

Generally, it is given by passing the POS tags for the words in a sentence. e.g.

In [13]:

```
print(lemma.lemmatize('running',pos='v'))
print(lemma.lemmatize('runs',pos='v'))
print(lemma.lemmatize('ran',pos='v'))
```

```
run
run
run
```

Lemmatizer is very complex and takes a lot of time to calculate.

So, it should only when the real meaning of words or the context is necessary for processing, else stemming should be preferred.

It completely depends on the type of problem you are trying to solve.

Named Entity Recognition(NER)

In chunking, we read that we can set rules to keep different POS tags under one sinlge user defined tag. One such form of chunking in NLP is known as Named Entity Recognition.

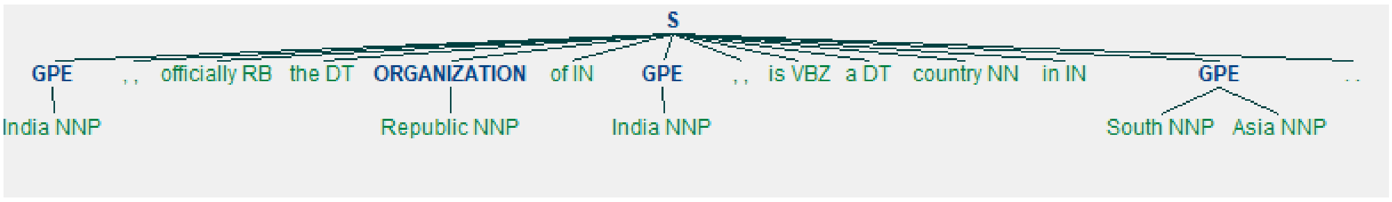
In NER, we try to group entities like people, places, countries, things etc. together.

In [136]:

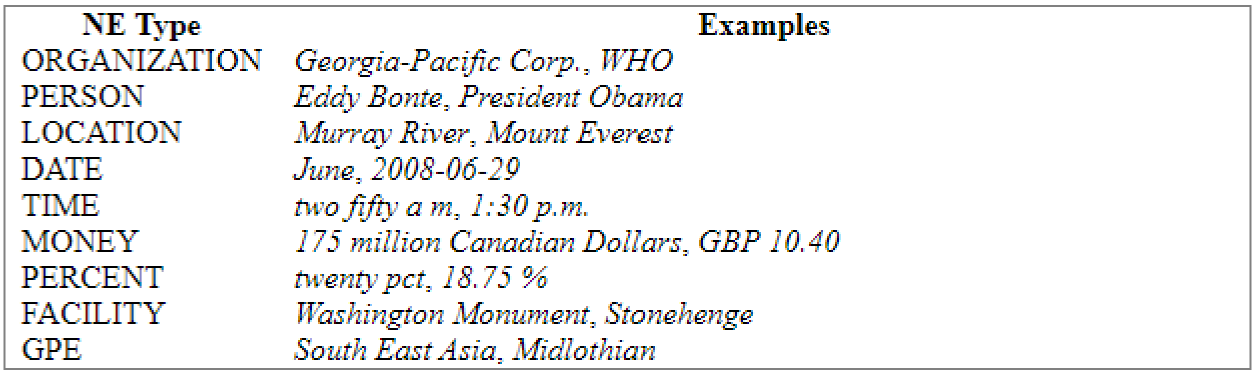
```
sent = "India, officially the Republic of India, is a country in South Asia."

words = nltk.word_tokenize(sent)
pos_tag = nltk.pos_tag(words)
namedEntity = nltk.ne_chunk(pos_tag)
print(namedEntity)
namedEntity.draw()
```

```
(S
  (GPE India/NNP)
  ,/,
  officially/RB
  the/DT
  (ORGANIZATION Republic/NNP)
  of/IN
  (GPE India/NNP)
  ,/,
  is/VBZ
  a/DT
  country/NN
  in/IN
  (GPE South/NNP Asia/NNP)
  ./.)
```

Graph



Commonly Used Types of Named Entity



NE Type	Examples
ORGANIZATION	*Georgia-Pacific Corp., WHO*
PERSON	*Eddy Bonte, President Obama*
LOCATION	*Murray River, Mount Everest*
DATE	*June, 2008-06-29*
TIME	*two fifty a m, 1:30 p.m.*
MONEY	*175 million Canadian Dollars, GBP 10.40*
PERCENT	*twenty pct, 18.75 %*
FACILITY	*Washington Monument, Stonehenge*
GPE	*South East Asia, Midlothian*

image source= nltk.org

With the help of NER, we can select any particular category from a given word document or sentence. Suppose we need all the names mentioned in a document, we can use NER and select the words with tag "Person".

Parsing

Parsing is the process of determining the structure of a given a text on the basis of a given grammatical rule.

e.g.

- Divide a sentence as Noun_phrase and Verb_phrase.
- Break Noun_phrase further in Proper noun, determiner, noun
- Break Verb_phrase in verb, noun_phrase

These are set of grammatical rules we will use to parse a given text.

text = " Ram ate a mango."

- Proper_noun = Ram
- Noun = mango
- determiner = a
- verb = ate

Once you pass the above mentioned set of grammatical rules in a parser. It will break the given "text" on the basis of it and output like this:

Noun_Phrase(Proper_Noun Ram) , Verb_Phrase(verb (ate), noun_phrase(determiner(a), noun(mango))

Such type of parser is called a **Recursive Parser**.

There are many different types of Parsers available in Nltk library. Each one has a different usecase and can be used on the basis of requirement.

In [15]:

```python
dir(nltk.parse)
```

Out[15]:

```
['BllipParser',
 'BottomUpChartParser',
 'BottomUpLeftCornerChartParser',
 'BottomUpProbabilisticChartParser',
 'ChartParser',
 'CoreNLPDependencyParser',
 'CoreNLPParser',
 'DependencyEvaluator',
 'DependencyGraph',
 'EarleyChartParser',
 'FeatureBottomUpChartParser',
 'FeatureBottomUpLeftCornerChartParser',
 'FeatureChartParser',
 'FeatureEarleyChartParser',
 'FeatureIncrementalBottomUpChartParser',
 'FeatureIncrementalBottomUpLeftCornerChartParser',
 'FeatureIncrementalChartParser',
 'FeatureIncrementalTopDownChartParser',
 'FeatureTopDownChartParser',
 'IncrementalBottomUpChartParser',
 'IncrementalBottomUpLeftCornerChartParser',
 'IncrementalChartParser',
 'IncrementalLeftCornerChartParser',
 'IncrementalTopDownChartParser',
 'InsideChartParser',
 'LeftCornerChartParser',
 'LongestChartParser',
 'MaltParser',
 'NaiveBayesDependencyScorer',
 'NonprojectiveDependencyParser',
 'ParserI',
 'ProbabilisticNonprojectiveParser',
 'ProbabilisticProjectiveDependencyParser',
 'ProjectiveDependencyParser',
 'RandomChartParser',
 'RecursiveDescentParser',
 'ShiftReduceParser',
 'SteppingChartParser',
 'SteppingRecursiveDescentParser',
 'SteppingShiftReduceParser',
 'TestGrammar',
 'TopDownChartParser',
 'TransitionParser',
 'UnsortedChartParser',
 'ViterbiParser',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 'api',
 'bllip',
 'chart',
 'corenlp',
 'dependencygraph',
 'earleychart',
 'evaluate',
 'extract_test_sentences',
 'featurechart',
 'load_parser',
 'malt',
 'nonprojectivedependencyparser',
 'pchart',
 'projectivedependencyparser',
 'recursivedescent',
 'shiftreduce',
 'transitionparser',
 'util',
 'viterbi']
```

In [62]:

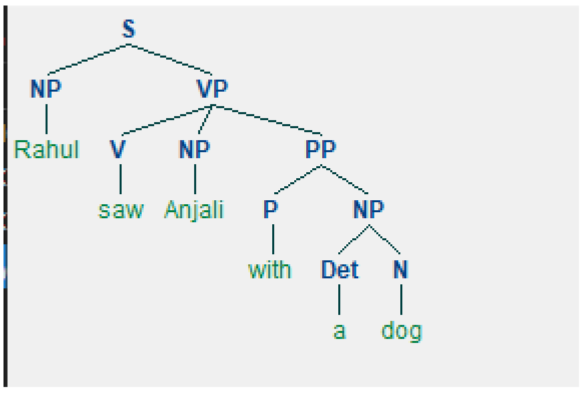
```
grammar = nltk.CFG.fromstring("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "slept" | "walked"
  NP -> "Rahul" | "Anjali" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park"
  P -> "in" | "on" | "by" | "with"
  """)
```

In [67]:

```
sent = "Rahul saw Anjali with a dog".split()
parser = nltk.RecursiveDescentParser(grammar)
for tree in parser.parse(sent):
    print(tree)
    tree.draw()
```

```
(S
  (NP Rahul)
  (VP (V saw) (NP Anjali) (PP (P with) (NP (Det a) (N dog)))))
```

Let's visualise the graph after parsing ::



Word Vectorization (Word Embedding)

Word vectorization is the process of mapping words to a set of real numbers or vectors. This is done to process the given words using machine learning techniques and extract relevant information from them such that it can be used in further predicting words. Vectorization is done by comparing a given word to the corpus(collection) of the available words. There are many different methods used for vectorizing a given set of words. let's see some of the mosed popular ones:

Count Vectorizer

Count vectorizer uses two of the following models as the base to vectorize the given words on the basis of frequency of words.

Bag of Words Model

BOW model is used in NLP to represent the given text/sentence/document as a collection (bag) of words without giving any importance to grammar or the occurrence order of the words. It keeps the account of frequency of the words in the text document, which can be used as features in many models.

Let’s understand this with an example:

Text1 = “I went to have a cup of coffee but I ended up having lunch with her.”

Text2 = “I don’t understand, what is the problem here?”

BOW1 = {I :2, went : 1, to : 1,have : 1, a : 1, cup: 1, of :1, coffee : 1, but :1, ended : 1, up :1,having : 1, with :1, her :1}

BOW2 = {I : 1, don’t : 1, understand:1, what : 1 , is :1, the : 1, problem : 1, here : 1}

BOW is mainly used for feature selection. The above dictionary is converted as a list with only the frequency terms there and on that basis, weights are given to the most occurring terms. But the “stop words” are the most frequent words that appears in raw document. Thus, having a word with high frequency count doesn’t mean that the word is as important. To resolve this problem, “Tf-idf” was introduced. We will discuss about it later.

n-gram model

As discussed in bag of words model, BOW model doesn’t keep the sequence of words in a given text, only the frequency of words matters. It doesn’t take into account the context of the given sentence, or care for grammatical rules such as verb is following a proper noun in the given text.n-gram model is used in such cases to keep the context of the given text intact. N-gram is the sequence of n words from a given text/document.

When, n= 1, we call it a “unigram”.

```
n=2, it is called a “bigram”.

n=3, it is called a “trigram”.
```

And so on.

Let’s understand this with an example:

Text1 = “I went to have a cup of coffee but I ended up having lunch with her.”

- Unigram

[I, went, to, have, a, cup, of, coffee, but, I, ended, up, having, lunch, with, her]

- Bi-gram

[I went], [went to],[to have],[have a],[a cup],[cup f],[of coffee],[coffee but],[but I],[I ended],[ended up], [up having], [having lunch],[lunch with],[with her]

- Tri-gram

[I went to], [went to have], [to have a], [have a cup],[a cup of], [cup of coffee],[of coffee but],[coffee but I],[but I ended],[I ended up],[ended up having],[up having lunch],[having lunch with],[lunch with her].

Note: We can clearly see that BOW model is nothing but n-gram model when n=1.

Skip-grams

Skip grams are type of n-grams where the words are not necessarily in the same order as are in the given text i.e. some words can be skipped. Example:

Text2 = “I don’t understand, what is the problem here?”

1-skip 2-grams (we have to make 2-gram while skipping 1 word)

[I understand, don’t what, understand is, what the, is problem, the here].

Let's see the implementation of Count vectorizer in python:

In [104]:

```python
# n-grams
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize import word_tokenize

string = ["This is an example of n-gram!"]
vect1 = CountVectorizer(ngram_range=(1,1))
vect1.fit_transform(string)
vect2 = CountVectorizer(ngram_range=(2,2))
vect2.fit_transform(string)
vect3 = CountVectorizer(ngram_range=(3,3))
vect3.fit_transform(string)
vect4 = CountVectorizer(ngram_range=(4,4))
vect4.fit_transform(string)
print("1-gram  :",vect1.get_feature_names())
print("2-gram  :",vect2.get_feature_names())
print("3-gram  :",vect3.get_feature_names())
print("4-gram  :",vect4.get_feature_names())
```

```
1-gram  : ['an', 'example', 'gram', 'is', 'of', 'this']
2-gram  : ['an example', 'example of', 'is an', 'of gram', 'this is']
3-gram  : ['an example of', 'example of gram', 'is an example', 'this
is an']
4-gram  : ['an example of gram', 'is an example of', 'this is an examp
le']
```

In [105]:

```
## Bag Of Words
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize import word_tokenize

string = ["This is an example of bag of words!"]
vect1 = CountVectorizer()
vect1.fit_transform(string)
print("bag of words :",vect1.get_feature_names())
```

```
bag of words : ['an', 'bag', 'example', 'is', 'of', 'this', 'words']
```

Tf-Idf (Term frequency–Inverse document frequency)

Wikipedia definition: ” Tf-Idf, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. The Tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general. Tf–idf is one of the most popular term-weighting schemes today.”

Term Frequency

It is simply the frequency in which a word appears in a document in comparison to the total number words in the document. Mathematically given as:

Term frequency = (Number of times a word appears in the document) / (Total number of words in the document)

Inverse Document Frequency

Term frequency has a disadvantage that it tends to give higher weights to words with higher frequency. In such cases words like ‘a’, ‘the’, ‘in’, ’of’ etc. appears more in the documents than other regular words. Thus, more important words are wrongly given lower weights as their frequency is less. To tackle this problem IDF was introduced. IDF decreases the weights of such high frequency terms and increases the weight of terms with rare occurrence. Mathematically it is given as:

Inverse Document Frequency = log [(Number of documents)/(Number of documents the word appears in)]

note: [log has base 2]

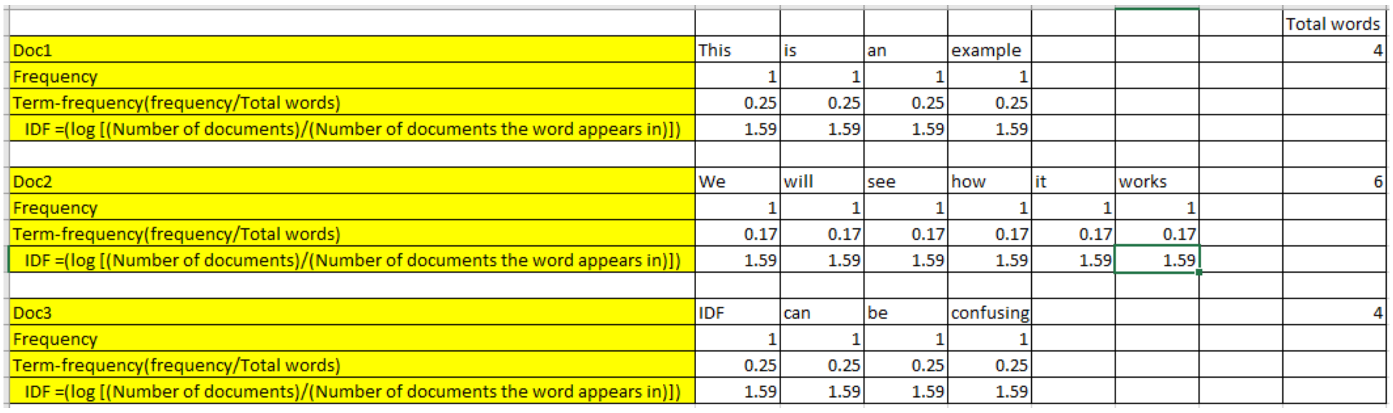
*Tf-Idf Score = Term frequency * Inverse Document Frequency*

Let's understand more with an example:

Doc 1: This is an example.

Doc 2: We will see how it works.

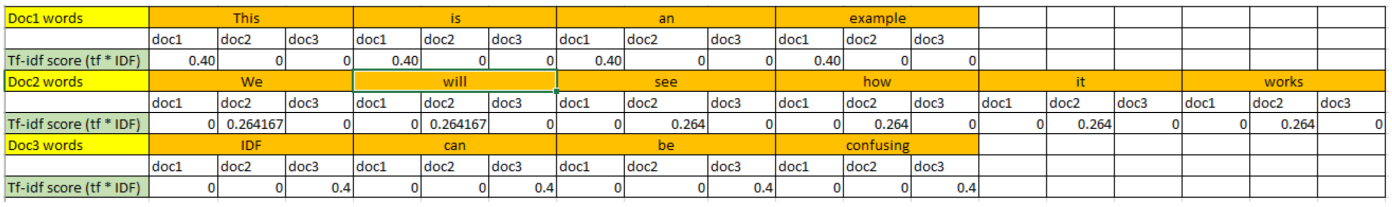
Doc 3: IDF can be confusing.



								Total words
Doc1	This	is	an	example				4
Frequency	1	1	1	1				
Term-frequency(frequency/Total words)	0.25	0.25	0.25	0.25				
IDF =(log [(Number of documents)/(Number of documents the word appears in)])	1.59	1.59	1.59	1.59				
Doc2	We	will	see	how	it	works		6
Frequency	1	1	1	1	1	1		
Term-frequency(frequency/Total words)	0.17	0.17	0.17	0.17	0.17	0.17		
IDF =(log [(Number of documents)/(Number of documents the word appears in)])	1.59	1.59	1.59	1.59	1.59	1.59		
Doc3	IDF	can	be	confusing				4
Frequency	1	1	1	1				
Term-frequency(frequency/Total words)	0.25	0.25	0.25	0.25				
IDF =(log [(Number of documents)/(Number of documents the word appears in)])	1.59	1.59	1.59	1.59				

In the above table, we have calculated the term frequency as well as inverse document frequency of each of the words present in the 3 documents given.

Now, let's calculate the tf-idf score for each term. Since, words of one document is not present in another document, we will have tf-idf value 0 for them e.g. words of doc1 will have 0 tf-idf for doc2 and doc3.



Doc1 words	This			is			an			example								
	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3						
Tf-idf score (tf * IDF)	0.40	0	0	0.40	0	0	0.40	0	0	0.40	0	0						
Doc2 words	We			will			see			how			it			works		
	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3
Tf-idf score (tf * IDF)	0	0.264167	0	0	0.264167	0	0	0.264	0	0	0.264	0	0	0.264	0	0	0.264	0
Doc3 words	IDF			can			be			confusing								
	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3	doc1	doc2	doc3						
Tf-idf score (tf * IDF)	0	0	0.4	0	0	0.4	0	0	0.4	0	0	0.4						

Great, hope this example must have cleared how Tf-Idf works.

let's see the python implementation for it:

In [7]:

```python
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

tfid = TfidfVectorizer(smooth_idf=False)

doc= ["This is an example.","We will see how it works.","IDF can be confusing"]

doc_vector = tfid.fit_transform(doc)
#print(tfid.get_feature_names())
df= pd.DataFrame(doc_vector.todense(),columns=tfid.get_feature_names())
df
#print(doc_vector)





```

Out[7]:

```
    an   be  can  confusing  example       how  idf   is        it       see  this        we
0  0.5  0.0  0.0        0.0      0.5  0.000000  0.0  0.5  0.000000  0.000000   0.5  0.000000  (
1  0.0  0.0  0.0        0.0      0.0  0.408248  0.0  0.0  0.408248  0.408248   0.0  0.408248  (
2  0.0  0.5  0.5        0.5      0.0  0.000000  0.5  0.0  0.000000  0.000000   0.0  0.000000  (
```

Although we are using the same data set as we used while doing manual calculation, the results are different than what we got.

This is because sklearn package have some modifications done to the formula to avoid complete avoidance of terms as well as to counter dividing by zero.

You can know more by going through the official doumentation of sklearn as below:

" *The formula that is used to compute the tf-idf for a term t of a document d in a document set is tf-idf(t, d) = tf(t, d) * idf(t), and the idf is computed as idf(t) = log [n / df(t)] + 1 (if* `smooth_idf=False` *), where n is the total number of documents in the document set and df(t) is the document frequency of t; the document frequency is the number of documents in the document set that contain the term t. The effect of adding "1" to the idf in the equation above is that terms with zero idf, i.e., terms that occur in all documents in a training set, will not be entirely ignored. (Note that the idf formula above differs from the standard textbook notation that defines the idf as idf(t) = log [n / (df(t) + 1)]). If* `smooth_idf=True` *(the default), the constant "1" is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions: idf(d, t) = log [(1 + n) / (1 + df(d, t))] + 1."*

In [8]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

tfid = TfidfVectorizer()

doc= ["Let's use python!", "Sklearn has package for Tf-idf.","Vectorization is

doc_vector = tfid.fit_transform(doc)
#print(tfid.get_feature_names())
df= pd.DataFrame(doc_vector.todense(),columns=tfid.get_feature_names())

#print(doc_vector)
```

In [4]:

```
df
```

Out[4]:

	for	fun	has	idf	is	let	package	python	sklearn	
0	0.000000	0.00000	0.000000	0.000000	0.00000	0.57735	0.000000	0.57735	0.000000	0.000
1	0.408248	0.00000	0.408248	0.408248	0.00000	0.00000	0.408248	0.00000	0.408248	0.408
2	0.000000	0.57735	0.000000	0.000000	0.57735	0.00000	0.000000	0.00000	0.000000	0.000

Let's understand the above result!!!!

The left most column has values like (a,b) where 'a' denotes the number of document and 'b' is the word vector index assigned to the words in that document.

The right column denotes the tf-idf score for each word.

In [ ]:

```

```