EXPERIMENT NO.1

AIM: Tokenization of text

RESOURCES REQUIRED:

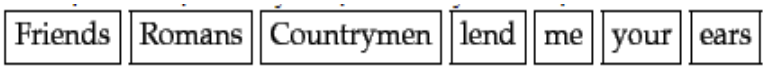
Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

TOKENIZATION:

Given a character sequence and a defined document unit, tokenization is the task of chopping it up into pieces, called tokens , perhaps at the same time throwing away certain characters, such as punctuation. Here is an example of tokenization:

Input: Friends, Romans, Countrymen, lend me your ears;

Output: 

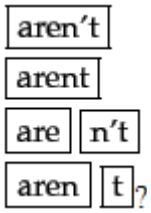
These tokens are often loosely referred to as terms or words, but it is sometimes important to make a type/token distinction. A token is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. A type is the class of all tokens containing the same character sequence. A term is a (perhaps normalized) type that is included in the IR system's dictionary. The set of index terms could be entirely distinct from the tokens, for instance, they could be semantic identifiers in a taxonomy, but in practice in modern IR systems they are strongly related to the tokens in the document. However, rather than being exactly the tokens that appear in the document, they are usually derived from them by various normalization processes.

The major question of the tokenization phase is what are the correct tokens to use? In this example, it looks fairly trivial: you chop on whitespace and throw away punctuation characters. This is a starting point, but even for English there are a number of tricky cases. For example, what do you do about the various uses of the apostrophe for possession and contractions?

Mr. O'Neill thinks that the boys' stories about Chile's capital aren't amusing.

For O'Neill, which of the following is the desired tokenization?

And for aren't, is it:



CHALLENGES IN TOKENIZATION

Challenges in tokenization depends on the type of language. Languages such as English and French are referred to as space-delimited as most of the words are separated from each other by white spaces. Languages such as Chinese and Thai are referred to as unsegmented as words do not have clear boundaries. Tokenising unsegmented language sentences requires additional lexical and morphological information. Tokenization is also affected by writing system and the typographical structure of the words. Structures of languges can be grouped into three categories:

Isolating: Words do not divide into smaller units. Example: Mandarin Chinese

Agglutinative: Words divide into smaller units. Example: Japanese, Tamil

Inflectional: Boundaries between morphemes are not clear and ambiguous in terms of grammatical meaning. Example: Latin.

CONCLUSION:

The process of segmenting running text into words and sentences is called tokenization.Tokenization is a basic pre-processing step in every NLP task. There are two types of tokenization, sentence and word tokenization. Tokenization has been performed on a simple text corpus.

CODE:

```
import nltk

from nltk.corpus import *

from random import choice

samples  = choice(brown.paras(categories="science_fiction"))

sample = " ".join([" ".join(sample) for sample in samples])

tokenized = nltk.word_tokenize(sample)

print(f"Original Corpus:\n{sample}\n\nTokenized Corpus:\n{tokenized}")
```

OUTPUT:

68_Adnan Shaikh

In [1]:
```
import nltk
from nltk.corpus import *
from random import choice
```

In [2]:
```
samples  = choice(brown.paras(categories="science_fiction"))
```

In [3]:
```
sample = " ".join([" ".join(sample) for sample in samples])
tokenized = nltk.word_tokenize(sample)
```

In [4]:
```
print(f"Original Corpus:\n{sample}\n\nTokenized Corpus:\n{tokenized}")
```

```
Original Corpus:
Jubal did not hear of Digby's disappearance when it was announced , and , when he did , while he had a fleeting sus
picion , he dismissed it ; ; if Mike had had a finger in it , he had gotten away with it -- and what happened to su
preme bishops worried Jubal not at all as long as he wasn't bothered .

Tokenized Corpus:
['Jubal', 'did', 'not', 'hear', 'of', 'Digby', "'s", 'disappearance', 'when', 'it', 'was', 'announced', ',', 'and',
',', 'when', 'he', 'did', ',', 'while', 'he', 'had', 'a', 'fleeting', 'suspicion', ',', 'he', 'dismissed', 'it',
';', ';', 'if', 'Mike', 'had', 'had', 'a', 'finger', 'in', 'it', ',', 'he', 'had', 'gotten', 'away', 'with', 'it',
'--', 'and', 'what', 'happened', 'to', 'supreme', 'bishops', 'worried', 'Jubal', 'not', 'at', 'all', 'as', 'long',
'as', 'he', 'was', "n't", 'bothered', '.']
```

In [ ]:

EXPERIMENT NO.2

AIM: Stop word removal.

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

STOP WORD REMOVAL:

Stop words are the most common words in any natural language. For the purpose of analyzing text data and building NLP models, these stop words might not add much value to the meaning of the document.

Consider this text string – "There is a pen on the table". Now, the words "is", "a", "on" and "the" add no meaning to the statement while parsing it. Whereas words like "there", "book", and "table" are the keywords and tell us what the statement is all about.

A basic list of stop words is given below:

```
a about after all also always am an and any are at be been being but by came can ca
nt come
could did didn't do does doesn't doing don't else for from get give goes going had
happen
has have having how i if ill i'm in into is isn't it its i've just keep let like ma
de make
many may me mean more most much no not now of only or our really say see some somet
hing
take tell than that the their them then they thing this to try up us use used uses
very
want was way we what when where which who why will with without wont you your youre
```

Removing stop words is not a hard and fast rule in NLP. It depends upon the task that we are working on. For tasks like text classification, where the text is to be classified into different categories, stop words are removed or excluded from the given text so that more focus can be given to those words which define the meaning of the text.

A few key benefits of removing stop words:

* On removing stop words, dataset size decreases and the time to train the model also decreases
* Removing stop words can potentially help improve the performance as there are fewer and only meaningful tokens left. Thus, it could increase classification accuracy

* Even search engines like Google remove stop words for fast and relevant retrieval of data from the database.

We can remove stop words while performing the following tasks:

Text Classification

Spam Filtering

Language Classification

Genre Classification

Caption Generation

Auto-Tag Generation

CONCLUSION:

Stop word removal is a pre-processing task in natural language processing. Stop word removal is necessary to improve analysis of the corpora in use. Stop word removal helps to understand relationships between the elements of the text and extract features. Stop word removal has been performed on a simple text corpus.

CODE:

```
from random import choice

import nltk

from nltk.corpus import stopwords, brown

samples = choice(brown.paras(categories="fiction"))

corpus = " ".join([" ".join(sample) for sample in samples])

print(f"Original corpus :\n{corpus}\n")

tokens = nltk.word_tokenize(corpus)

print(f"Tokenized words : \n{tokens}\n")

stop_words = set(stopwords.words("english"))

rel_words = [rel for rel in tokens if not rel in stop_words]
```

print(f"Tokens without stop words :\n{rel_words}")

OUTPUT:

68_Adnan Shaikh

In [3]:
```
from random import choice
import nltk
from nltk.corpus import stopwords, brown
```

In [9]:
```
samples = choice(brown.paras(categories="fiction"))
corpus = " ".join([" ".join(sample) for sample in samples])
print(f"Original corpus :\n{corpus}\n")

tokens = nltk.word_tokenize(corpus)
print(f"Tokenized words : \n{tokens}\n")

stop_words = set(stopwords.words("english"))
rel_words = [rel for rel in tokens if not rel in stop_words]
print(f"Tokens without stop words :\n{rel_words}")
```

```
Original corpus :
Day by day , week by week , month by month , the betrayal gnawed at Andrei's heart . He ground his teeth together .
I hate Warsaw , he said to himself . I hate Poland and all the goddamned mothers' sons of them . All of Poland is a
coffin .

Tokenized words :
['Day', 'by', 'day', ',', 'week', 'by', 'week', ',', 'month', 'by', 'month', ',', 'the', 'betrayal', 'gnawed', 'a
t', 'Andrei', "'s", 'heart', '.', 'He', 'ground', 'his', 'teeth', 'together', '.', 'I', 'hate', 'Warsaw', ',', 'h
e', 'said', 'to', 'himself', '.', 'I', 'hate', 'Poland', 'and', 'all', 'the', 'goddamned', 'mothers', "'", 'sons',
'of', 'them', '.', 'All', 'of', 'Poland', 'is', 'a', 'coffin', '.']

Tokens without stop words :
['Day', 'day', ',', 'week', 'week', ',', 'month', 'month', ',', 'betrayal', 'gnawed', 'Andrei', "'s", 'heart', '.',
'He', 'ground', 'teeth', 'together', '.', 'I', 'hate', 'Warsaw', ',', 'said', '.', 'I', 'hate', 'Poland', 'goddamne
d', 'mothers', "'", 'sons', '.', 'All', 'Poland', 'coffin', '.']
```

EXPERIMENT NO.3

AIM: Stemming of text

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

STEMMING:

Stemming is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma. Stemming is important in natural language understanding (NLU) and natural language processing (NLP).

Stemming is a part of linguistic studies in morphology and artificial intelligence (AI) information retrieval and extraction. Stemming and AI knowledge extract meaningful information from vast sources like big data or the Internet since additional forms of a word related to a subject may need to be searched to get the best results. Stemming is also a part of queries and Internet search engines.

Recognizing, searching and retrieving more forms of words returns more results. When a form of a word is recognized it can make it possible to return search results that otherwise might have been missed. That additional information retrieved is why stemming is integral to search queries and information retrieval.

When a new word is found, it can present new research opportunities. Often, the best results can be attained by using the basic morphological form of the word: the lemma. To find the lemma, stemming is performed by an individual or an algorithm, which may be used by an AI system. Stemming uses a number of approaches to reduce a word to its base from whatever inflected form is encountered.

It can be simple to develop a stemming algorithm. Some simple algorithms will simply strip recognized prefixes and suffixes. However, these simple algorithms are prone to error. For example, an error can reduce words like laziness to lazi instead of lazy. Such algorithms may also have difficulty with terms whose inflectional forms don't perfectly mirror the lemma such as with saw and see.

Examples of stemming algorithms include:

Lookups in tables of inflected forms of words. This approach requires all inflected forms be listed.

Suffix strippi . Algorithms recognize known suffixes on inflected words and remove them.

PORTER STEMMER:

A consonant in a word is a letter other than A, E, I, O or U, and other than Y preceded by a consonant. (The fact that the term **consonant** is defined to some extent in terms of itself does not make it ambiguous.) So in TOY the consonants are T and Y, and in SYZYGY they are S, Z and G. If a letter is not a consonant it is a vowel.

A consonant will be denoted by c, a vowel by v. A list ccc... of length greater than 0 will be denoted by C, and a list vvv... of length greater than 0 will be denoted by V. Any word, or part of a word, therefore has one of the four forms:

* CVCV ... C
* CVCV ... V
* VCVC ... C
* VCVC ... V

These may all be represented by the single form

[C]VCVC ... [V]

where the square brackets denote arbitrary presence of their contents. Using (VCmVCm) to denote VC repeated m times, this may again be written as

[C](VCmVCm)[V]

m will be called the measure of any word or word part when represented in this form. The case m = 0 covers the null word. Here are some examples:

* m=0 TR, EE, TREE, Y, BY.
* m=1 TROUBLE, OATS, TREES, IVY.
* m=2 TROUBLES, PRIVATE, OATEN, ORRERY.

The rules for removing a suffix will be given in the form

(condition) S1 -> S2

This means that if a word ends with the suffix S1, and the stem before S1 satisfies the given condition, S1 is replaced by S2. The condition is usually given in terms of m, e.g.

(m > 1) EMENT ->

Here S1 is 'EMENT' and S2 is null. This would map REPLACEMENT to REPLAC, since REPLAC is a word part for which m = 2.

The 'condition' part may also contain the following:

* *S - the stem ends with S (and similarly for the other letters).
* *v* - the stem contains a vowel.
* m=2 TROUBLES, PRIVATE, OATEN, ORRERY.

* *d - the stem ends with a double consonant (e.g. -TT, -SS).
* *o - the stem ends cvc, where the second c is not W, X or Y (e.g. -WIL, -HOP).

And the condition part may also contain expressions with and, or and not, so that:

(m>1 and (*S or *T)) : tests for a stem with m>1 ending in S or T, while
(*d and not (*L or *S or *Z)) : tests for a stem ending witha double consonant other than L, S or Z. Elaborate conditions like this are required only rarely.
In a set of rules written beneath each other, only one is obeyed, and this will be the one with the longest matching S1 for the given word. For example, with

* SSES -> SS
* IES -> I
* SS -> SS
* S ->

(here the conditions are all null) CARESSES maps to CARESS since SSES is the longest match for S1. Equally CARESS maps to CARESS (S1=SS) and CARES to CARE (S1=S).

CONCLUSION:

Stemming is a text pre-processing task used in natural language processing. Stemming is the process of reducing words to their root form or stem. Stemming is useful to simplify text analysis in large corpora. A very common Stemming algorithm is the Porter Stemmer algorithm which has been implemented using the nltk toolkit.

CODE:

```
from random import choice

from nltk import word_tokenize

from nltk.corpus import brown

from nltk.stem.porter import PorterStemmer

samples = choice(brown.paras(categories="humor"))

corpus = " ".join([" ".join(sample) for sample in samples])

print(f"Original corpus :\n{corpus}\n")

tokens = word_tokenize(corpus)

print(f"Tokenized words : \n{tokens}\n")
```

porter = PorterStemmer()

stem_words = [porter.stem(stem) for stem in tokens]

print(f"Stemmed words :\n{stem_words}")

OUTPUT:

68_Adnan Shaikh

In [3]:
```
from random import choice
from nltk import word_tokenize
from nltk.corpus import brown
from nltk.stem.porter import PorterStemmer

samples = choice(brown.paras(categories="humor"))
corpus = " ".join([" ".join(sample) for sample in samples])
print(f"Original corpus :\n{corpus}\n")

tokens = word_tokenize(corpus)
print(f"Tokenized words : \n{tokens}\n")

porter = PorterStemmer()
stem_words = [porter.stem(stem) for stem in tokens]
print(f"Stemmed words :\n{stem_words}")
```

```
Original corpus :
And when they had got to their little lawn , they had had a most twirlingly magnificent time . First , Arlene had p
ut them through some rapid somersaults . They had protested that that wasn't any surprise .

Tokenized words :
['And', 'when', 'they', 'had', 'got', 'to', 'their', 'little', 'lawn', ',', 'they', 'had', 'had', 'a', 'most', 'twi
rlingly', 'magnificent', 'time', '.', 'First', ',', 'Arlene', 'had', 'put', 'them', 'through', 'some', 'rapid', 'so
mersaults', '.', 'They', 'had', 'protested', 'that', 'that', 'was', "n't", 'any', 'surprise', '.']

Stemmed words :
['and', 'when', 'they', 'had', 'got', 'to', 'their', 'littl', 'lawn', ',', 'they', 'had', 'had', 'a', 'most', 'twir
lingli', 'magnific', 'time', '.', 'first', ',', 'arlen', 'had', 'put', 'them', 'through', 'some', 'rapid', 'somersa
ult', '.', 'they', 'had', 'protest', 'that', 'that', 'wa', "n't", 'ani', 'surpris', '.']
```

EXPERIMENT NO.4

AIM:

Lemmatization

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

Lemmatization:

Lemmatization is the process of grouping together the different inflected forms of a word so they can be analysed as a single item. Lemmatization is similar to stemming but it brings context to the words. So it links words with similar meaning to one word.

Text pre-processing includes both Stemming as well as Lemmatization. Many times people find these two terms confusing. Some treat these two as same. Actually, lemmatization is preferred over Stemming because lemmatization does morphological analysis of the words.

Applications of lemmatization are:

* Used in comprehensive retrieval systems like search engines.
* Used in compact indexing

Examples of lemmatization:

rocks : rock

corpora : corpus

better : good

Form	Morphological information	Lemma
studies	Third person, singular number, present tense of the verb study	study
studying	Gerund of the verb study	study
niñas	Feminine gender, plural number of the noun niño	niño
niñez	Singular number of the noun niñez	niñez

CONCLUSION:

Lemmatization is a basic text pre-processing operation in many natural language processing tasks. It is similar to stemming but unlike stemming, it does not truncate any affixes from the morpheme but rather reduces the inflected form to its actual stem. Therefore, lemmatization provides a better result when compared to stemming.

CODE:

```
from random import choice

from nltk import word_tokenize

from nltk.corpus import brown

from nltk.stem import WordNetLemmatizer


samples = choice(brown.paras(categories="fiction"))

corpus = " ".join([" ".join(sample) for sample in samples])

print(f"Original corpus :\n{corpus}\n")


tokens = word_tokenize(corpus)

print(f"Tokenized words : \n{tokens}\n")


lemma = WordNetLemmatizer()

lem_words = [lemma.lemmatize(token) for token in tokens]

print(f"Lemmatized words :\n{lem_words}")
```

OUTPUT:

68_Adnan Shaikh

In [8]:
```
from random import choice
from nltk import word_tokenize
from nltk.corpus import brown
from nltk.stem import WordNetLemmatizer

samples = choice(brown.paras(categories="fiction"))
corpus = " ".join([" ".join(sample) for sample in samples])
print(f"Original corpus :\n{corpus}\n")

tokens = word_tokenize(corpus)
print(f"Tokenized words : \n{tokens}\n")

lemma = WordNetLemmatizer()
lem_words = [lemma.lemmatize(token) for token in tokens]
print(f"Lemmatized words :\n{lem_words}")
```

```
Original corpus :
It was a ridiculous situation and Rector knew it , for Hino , frankly partisan , openly gregarious , would make a p
oor espionage agent . If he wanted to know anything , he would end up asking about it point-blank , but in this gui
leless manner he would probably receive more truthful answers than if he tried to get them by indirection . In all
of his experience in the mission field Rector had never seen a convert quite like Hino . From the moment that Hino
had first walked into the mission to ask for a job , any job -- his qualifications neatly written on a piece of pap
er in a precise hand -- he had been ready to become a Christian . He had already been studying the Bible ; ; he kne
w the fundamentals , and after studying with Fletcher for a time he approached Rector , announced that he wanted to
be baptized and that was that .

Tokenized words :
['It', 'was', 'a', 'ridiculous', 'situation', 'and', 'Rector', 'knew', 'it', ',', 'for', 'Hino', ',', 'frankly', 'p
artisan', ',', 'openly', 'gregarious', ',', 'would', 'make', 'a', 'poor', 'espionage', 'agent', '.', 'If', 'he', 'w
anted', 'to', 'know', 'anything', ',', 'he', 'would', 'end', 'up', 'asking', 'about', 'it', 'point-blank', ',', 'bu
t', 'in', 'this', 'guileless', 'manner', 'he', 'would', 'probably', 'receive', 'more', 'truthful', 'answers', 'tha
n', 'if', 'he', 'tried', 'to', 'get', 'them', 'by', 'indirection', '.', 'In', 'all', 'of', 'his', 'experience', 'i
n', 'the', 'mission', 'field', 'Rector', 'had', 'never', 'seen', 'a', 'convert', 'quite', 'like', 'Hino', '.', 'Fro
m', 'the', 'moment', 'that', 'Hino', 'had', 'first', 'walked', 'into', 'the', 'mission', 'to', 'ask', 'for', 'a',
'job', ',', 'any', 'job', '--', 'his', 'qualifications', 'neatly', 'written', 'on', 'a', 'piece', 'of', 'paper', 'i
n', 'a', 'precise', 'hand', '--', 'he', 'had', 'been', 'ready', 'to', 'become', 'a', 'Christian', '.', 'He', 'had',
'already', 'been', 'studying', 'the', 'Bible', ';', ';', 'he', 'knew', 'the', 'fundamentals', ',', 'and', 'after',
'studying', 'with', 'Fletcher', 'for', 'a', 'time', 'he', 'approached', 'Rector', ',', 'announced', 'that', 'he',
'wanted', 'to', 'be', 'baptized', 'and', 'that', 'was', 'that', '.']

Lemmatized words :
['It', 'wa', 'a', 'ridiculous', 'situation', 'and', 'Rector', 'knew', 'it', ',', 'for', 'Hino', ',', 'frankly', 'pa
rtisan', ',', 'openly', 'gregarious', ',', 'would', 'make', 'a', 'poor', 'espionage', 'agent', '.', 'If', 'he', 'wa
nted', 'to', 'know', 'anything', ',', 'he', 'would', 'end', 'up', 'asking', 'about', 'it', 'point-blank', ',', 'bu
t', 'in', 'this', 'guileless', 'manner', 'he', 'would', 'probably', 'receive', 'more', 'truthful', 'answer', 'tha
n', 'if', 'he', 'tried', 'to', 'get', 'them', 'by', 'indirection', '.', 'In', 'all', 'of', 'his', 'experience', 'i
n', 'the', 'mission', 'field', 'Rector', 'had', 'never', 'seen', 'a', 'convert', 'quite', 'like', 'Hino', '.', 'Fro
m', 'the', 'moment', 'that', 'Hino', 'had', 'first', 'walked', 'into', 'the', 'mission', 'to', 'ask', 'for', 'a',
'job', ',', 'any', 'job', '--', 'his', 'qualification', 'neatly', 'written', 'on', 'a', 'piece', 'of', 'paper', 'i
n', 'a', 'precise', 'hand', '--', 'he', 'had', 'been', 'ready', 'to', 'become', 'a', 'Christian', '.', 'He', 'had',
'already', 'been', 'studying', 'the', 'Bible', ';', ';', 'he', 'knew', 'the', 'fundamental', ',', 'and', 'after',
'studying', 'with', 'Fletcher', 'for', 'a', 'time', 'he', 'approached', 'Rector', ',', 'announced', 'that', 'he',
'wanted', 'to', 'be', 'baptized', 'and', 'that', 'wa', 'that', '.']
```

EXPERIMENT NO.5

AIM:

N-gram model.

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

N-gram Model:

Statistical language models, in its essence, are the type of models that assign probabilities to the sequences of words. In this article, we'll understand the simplest model that assigns probabilities to sentences and sequences of words, the n-gram

You can think of an N-gram as the sequence of N words, by that notion, a 2-gram (or bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (or trigram) is a three-word sequence of words like "please turn your", or "turn your homework".

Let's start with equation P(w|h), the probability of word w, given some history, h. For example,

$$P(the \mid its\ water\ is\ so\ transperant\ that)$$

Here,

w = The

h = its water is so transparent that

And, one way to estimate the above probability function is through the relative frequency count approach, where you would take a substantially large corpus, count the number of times you see its water is so transparent that, and then count the number of times it is followed by the. In other words, you are answering the question:

Out of the times you saw the history h, how many times did the word w follow it

$$P(the \mid its\ water\ is\ so\ transperant\ that) = C(its\ water\ is\ so\ transperant\ that\ the) / C(its\ water\ is\ so\ transperant\ that)$$

Now, you can imagine it is not feasible to perform this over an entire corpus; especially if it is of a significant size.

This shortcoming and ways to decompose the probability function using the chain rule serves as the base intuition of the N-gram model. Here, you, instead of computing probability using the entire corpus, would approximate it by just a few historical words.

The Bigram Model:

As the name suggests, the bigram model approximates the probability of a word given all the previous words by using only the conditional probability of one preceding word. In other words, you approximate it with the probability: P(the | that)

And so, when you use a bigram model to predict the conditional probability of the next word, you are thus making the following approximation:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1})$$

This assumption that the probability of a word depends only on the previous word is also known as the Markov assumption.

Markov models are the class of probabilistic models that assume that we can predict the probability of some future unit without looking too far in the past.

You can further generalize the bigram model to the trigram model which looks two words into the past and can thus be further generalized to the N-gram model.

Now that we understand the underlying base for N-gram models, you'd think, how can we estimate the probability function? One of the most straightforward and intuitive ways to do so is Maximum Likelihood Estimation (MLE)

For example, to compute a particular bigram probability of a word y given a previous word x, you can determine the count of the bigram C(xy) and normalize it by the sum of all the bigrams that share the same first-word x.

There are, of course, challenges, as with every modeling approach, and estimation method. Let's look at the key ones affecting the N-gram model, as well as the use of MLE

Sensitivity to the training corpus

The N-gram model, like many statistical models, is significantly dependent on the training corpus. As a result, the probabilities often encode particular facts about a given training corpus. Besides, the performance of the N-gram model varies with the change in the value of N.

Moreover, you may have a language task in which you know all the words that can occur, and hence we know the vocabulary size V in advance. The closed vocabulary assumption assumes there are no unknown words, which is unlikely in practical scenarios.

Smoothing

A notable problem with the MLE approach is sparse data. Meaning, any N-gram that appeared a sufficient number of times might have a reasonable estimate for its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it.

As a result of it, the N-gram matrix for any training corpus is bound to have a substantial number of cases of putative "zero probability N-grams"

CONCLUSION:

The N-gram model is a statistical language model that has various applications in natural language processing such as spell correction etc. The bi-gram model specifically has been studied in detail and has been implemented.

CODE and OUTPUT:

In [2]:
```
from nltk.util import pad_sequence
from nltk.util import bigrams
from nltk.util import ngrams
from nltk.util import everygrams
from nltk.lm.preprocessing import pad_both_ends
from nltk.lm.preprocessing import flatten
```

In [3]:
```
text = [['a', 'b', 'c'], ['a', 'c', 'd', 'c', 'e', 'f']]
```

In [4]:
```
list(bigrams(text[0]))
```

Out[4]:
```
[('a', 'b'), ('b', 'c')]
```

In [5]:
```
list(ngrams(text[1], n=3))
```

Out[5]:
```
[('a', 'c', 'd'), ('c', 'd', 'c'), ('d', 'c', 'e'), ('c', 'e', 'f')]
```

In [6]:
```
from nltk.util import pad_sequence
list(pad_sequence(text[0],
                  pad_left=True, left_pad_symbol="<s>",
                  pad_right=True, right_pad_symbol="</s>",
                  n=2)) # The n order of n-grams, if it's 2-grams, you pad once, 3-grams pad twice, etc.
```

Out[6]:
```
['<s>', 'a', 'b', 'c', '</s>']
```

In [7]:
```
padded_sent = list(pad_sequence(text[0], pad_left=True, left_pad_symbol="<s>",
                                pad_right=True, right_pad_symbol="</s>", n=2))
list(ngrams(padded_sent, n=2))
```

Out[7]:
```
[('<s>', 'a'), ('a', 'b'), ('b', 'c'), ('c', '</s>')]
```

In [8]:
```
list(pad_sequence(text[0],
                  pad_left=True, left_pad_symbol="<s>",
                  pad_right=True, right_pad_symbol="</s>",
                  n=3)) # The n order of n-grams, if it's 2-grams, you pad once, 3-grams pad twice, etc.
```

Out[8]:
```
['<s>', '<s>', 'a', 'b', 'c', '</s>', '</s>']
```

In [9]:
```
padded_sent = list(pad_sequence(text[0], pad_left=True, left_pad_symbol="<s>",
                                pad_right=True, right_pad_symbol="</s>", n=3))
list(ngrams(padded_sent, n=3))
```

Out[9]:
```
[('<s>', '<s>', 'a'),
 ('<s>', 'a', 'b'),
 ('a', 'b', 'c'),
 ('b', 'c', '</s>'),
 ('c', '</s>', '</s>')]
```

In [10]:
```
from nltk.lm.preprocessing import pad_both_ends
list(pad_both_ends(text[0], n=2))
```

Out[10]:
```
['<s>', 'a', 'b', 'c', '</s>']
```

Combining the two parts discussed so far we get the following preparation steps for one sentence.

In [11]:
```
list(bigrams(pad_both_ends(text[0], n=2)))
```

Out[11]:
```
[('<s>', 'a'), ('a', 'b'), ('b', 'c'), ('c', '</s>')]
```

In [12]:
```
from nltk.util import everygrams
padded_bigrams = list(pad_both_ends(text[0], n=2))
list(everygrams(padded_bigrams, max_len=2))
```

Out[12]:
```
[('<s>',),
 ('a',),
 ('b',),
 ('c',),
 ('</s>',),
 ('<s>', 'a'),
 ('a', 'b'),
 ('b', 'c'),
 ('c', '</s>')]
```

In [13]:
```
from nltk.lm.preprocessing import flatten
list(flatten(pad_both_ends(sent, n=2) for sent in text))
```

Out[13]:
```
['<s>', 'a', 'b', 'c', '</s>', '<s>', 'a', 'c', 'd', 'c', 'e', 'f', '</s>']
```

In [14]:
```
from nltk.lm.preprocessing import padded_everygram_pipeline
train, vocab = padded_everygram_pipeline(2, text)
```

In [15]:
```
training_ngrams, padded_sentences = padded_everygram_pipeline(2, text)
for ngramlize_sent in training_ngrams:
    print(list(ngramlize_sent))
    print()
print('#############')
list(padded_sentences)
```

```
[('<s>',), ('a',), ('b',), ('c',), ('</s>',), ('<s>', 'a'), ('a', 'b'), ('b', 'c'), ('c', '</s>')]

[('<s>',), ('a',), ('c',), ('d',), ('c',), ('e',), ('f',), ('</s>',), ('<s>', 'a'), ('a', 'c'), ('c', 'd'), ('d',
'c'), ('c', 'e'), ('e', 'f'), ('f', '</s>')]

#############
```

Out[15]:
```
['<s>', 'a', 'b', 'c', '</s>', '<s>', 'a', 'c', 'd', 'c', 'e', 'f', '</s>']
```

In [16]:
```
try:
    from nltk import word_tokenize, sent_tokenize
    word_tokenize(sent_tokenize("This is a foobar sentence. Yes it is.")[0])
except:
    import re
    from nltk.tokenize import ToktokTokenizer
    sent_tokenize = lambda x: re.split(r'(?<=[^A-Z].[.?]) +(?=[A-Z])', x)
    toktok = ToktokTokenizer()
    word_tokenize = word_tokenize = toktok.tokenize
```

In [17]:
```
import os
import requests
import io

if os.path.isfile('language-never-random.txt'):
    with io.open('language-never-random.txt', encoding='utf8') as fin:
        text = fin.read()
else:
    url = "https://gist.githubusercontent.com/alvations/53b01e4076573fea47c6057120bb017a/raw/b01ff96a5f76848450e648
f35da6497ca9454e4a/language-never-random.txt"
    text = requests.get(url).content.decode('utf8')
    with io.open('language-never-random.txt', 'w', encoding='utf8') as fout:
        fout.write(text)
```

In [18]:
```
tokenized_text = [list(map(str.lower, word_tokenize(sent)))
                  for sent in sent_tokenize(text)]
```

In [19]:
```
tokenized_text[0]
```

Out[19]:
```
['language',
 'is',
 'never',
 ',',
 'ever',
 ',',
 'ever',
 ',',
 'random',
 'adam',
 'kilgarriff',
 'abstract',
 'language',
 'users',
 'never',
 'choose',
 'words',
 'randomly',
 ',',
 'and',
 'language',
 'is',
 'essentially',
 'non-random',
 '.']
```

In [20]:
```
print(text[:500])
```

```
                    Language is never, ever, ever, random

                                                                ADAM KILGARRIFF



        Abstract
        Language users never choose words randomly, and language is essentially
        non-random. Statistical hypothesis testing uses a null hypothesis, which
        posits randomness. Hence, when we look at linguistic phenomena in cor-
        pora, the null hypothesis will never be true. Moreover, where there is enough
        data, we shall (almost) always be able to establish
```

In [21]:
```
# Preprocess the tokenized text for 3-grams language modelling
n = 3
train_data, padded_sents = padded_everygram_pipeline(n, tokenized_text)
```

In [22]:
```
from nltk.lm import MLE
model = MLE(n) # Lets train a 3-grams model, previously we set n=3
```

In [23]:
```
len(model.vocab)
```

Out[23]:
```
0
```

In [24]:
```
model.fit(train_data, padded_sents)
print(model.vocab)
```

```
<Vocabulary with cutoff=1 unk_label='<UNK>' and 1391 items>
```

In [25]:
```
len(model.vocab)
```

Out[25]:
```
1391
```

In [26]:
```
print(model.vocab.lookup(tokenized_text[0]))
```

```
('language', 'is', 'never', ',', 'ever', ',', 'ever', ',', 'random', 'adam', 'kilgarriff', 'abstract', 'language',
'users', 'never', 'choose', 'words', 'randomly', ',', 'and', 'language', 'is', 'essentially', 'non-random', '.')
```

In [27]:
```
# If we lookup the vocab on unseen sentences not from the training data,
# it automatically replace words not in the vocabulary with `<UNK>`.
print(model.vocab.lookup('language is never random lah .'.split()))
```

```
('language', 'is', 'never', 'random', '<UNK>', '.')
```

Using the N-gram Language Model

In [28]:
```
print(model.counts)
```

```
<NgramCounter with 3 ngram orders and 19611 ngrams>
```

In [29]:
```
model.counts['language'] # i.e. Count('language')
```

Out[29]:
```
25
```

In [30]:
```
model.counts[['language']]['is'] # i.e. Count('is'|'language')
```

Out[30]:
```
11
```

In [31]:
```
model.counts[['language', 'is']]['never'] # i.e. Count('never'|'language is')
```

Out[31]:
```
7
```

In [32]:
```
model.score('language') # P('language')
```

Out[32]:
```
0.003691671588895452
```

In [33]:
```
model.score('is', 'language'.split())  # P('is'|'language')
```

Out[33]:
```
0.44
```

In [34]:
```
model.score('never', 'language is'.split())  # P('never'|'language is')
```

Out[34]:
```
0.6363636363636364
```

In [35]:
```
model.score("<UNK>") == model.score("lah")
```

Out[35]:
```
True
```

In [36]:
```
model.score("<UNK>") == model.score("leh")
```

Out[36]:
```
True
```

In [37]:
```
model.score("<UNK>") == model.score("lor")
```

Out[37]:
```
True
```

In [38]:
```
model.logscore("never", "language is".split())
```

Out[38]:
```
-0.6520766965796932
```

Generation using N-gram Language Model

In [39]:
```
print(model.generate(20, random_seed=7))
```

```
['and', 'hence', ',', 'when', 'we', 'look', 'at', 'linguistic', 'phenom-', 'ena', 'in', 'corpora', ',', '216⩾223',
'.', '</s>', '</s>', '</s>', '</s>', '</s>']
```

In [40]:
```
from nltk.tokenize.treebank import TreebankWordDetokenizer

detokenize = TreebankWordDetokenizer().detokenize

def generate_sent(model, num_words, random_seed=42):
    """
    :param model: An ngram language model from `nltk.lm.model`.
    :param num_words: Max no. of words to generate.
    :param random_seed: Seed value for random.
    """
    content = []
    for token in model.generate(num_words, random_seed=random_seed):
        if token == '<s>':
            continue
        if token == '</s>':
            break
        content.append(token)
    return detokenize(content)
```

In [41]:
```
generate_sent(model, 20, random_seed=7)
```

Out[41]:
```
'and hence, when we look at linguistic phenom- ena in corpora , 216⩾223.'
```

In [42]:
```
print(model.generate(28, random_seed=0))
```

```
['the', 'trouble', 'with', 'quantitative', 'studies', '.', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>',
'</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s
>']
```

In [43]:
```
generate_sent(model, 28, random_seed=0)
```

Out[43]:
```
'the trouble with quantitative studies.'
```

In [44]:
```
generate_sent(model, 20, random_seed=1)
```

Out[44]:
```
'237⩾246.'
```

In [45]:
```
generate_sent(model, 20, random_seed=30)
```

Out[45]:
```
'hypothesis test- ing has been used in the joint conference on empirical methods in nlp and very large corpora altho
ugh'
```

In [46]:
```
generate_sent(model, 20, random_seed=42)
```

Out[46]:
```
'more common or more salient in the last paragraph is ‘ indistinguishable ’.'
```

EXPERIMENT 6

AIM: POS tagging.

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

Part of Speech (hereby referred to as POS) Tags are useful for building parse trees, which are used in building NERs (most named entities are Nouns) and extracting relations between words. POS Tagging is also essential for building lemmatizers which are used to reduce a word to its root form.

POS tagging is the process of marking up a word in a corpus to a corresponding part of a speech tag, based on its context and definition. This task is not straightforward, as a particular word may have a different part of speech based on the context in which the word is used.

For example: In the sentence "Give me your answer", *answer* is a Noun, but in the sentence "Answer the question", *answer* is a verb.

To understand the meaning of any sentence or to extract relationships and build a knowledge graph, POS Tagging is a very important step.

The Different POS Tagging Techniques

There are different techniques for POS Tagging:

- **Lexical Based Methods —** Assigns the POS tag the most frequently occurring with a word in the training corpus.
- **Rule-Based Methods —** Assigns POS tags based on rules. For example, we can have a rule that says, words ending with "ed" or "ing" must be assigned to a verb. Rule-Based Techniques can be used along with Lexical Based approaches to allow POS Tagging of words that are not present in the training corpus but are there in the testing data.
- **Probabilistic Methods —** This method assigns the POS tags based on the probability of a particular tag sequence occurring. Conditional Random Fields (CRFs) and Hidden Markov Models (HMMs) are probabilistic approaches to assign a POS Tag.
- **Deep Learning Methods** — Recurrent Neural Networks can also be used for POS tagging.

Steps Involved:

Tokenize text (word_tokenize)

apply pos_tag to above step that is nltk.pos_tag(tokenize_text)

Some examples are as below:

Abbreviation	Meaning
CC	coordinating conjunction
CD	cardinal digit
DT	determiner
EX	existential there
FW	foreign word
IN	preposition/subordinating conjunction
JJ	adjective (large)
JJR	adjective, comparative (larger)
JJS	adjective, superlative (largest)
LS	list market
MD	modal (could, will)
NN	noun, singular (cat, tree)
NNS	noun plural (desks)
NNP	proper noun, singular (sarah)
NNPS	proper noun, plural (indians or americans)

CONCLUSION:

Part of speech tagging is the process of assigning a word in a corpus word class. Parts of speech tagging have numerous uses such as in Named Entity Recognition. Parts of Speech tagging has been carefully studied and implemented on a text corpus.

CODE:

```
from random import choice

from nltk import pos_tag

from nltk import word_tokenize

from nltk.corpus import brown


samples = choice(brown.paras(categories="adventure"))

corpus = " ".join([" ".join(sample) for sample in samples])

print(f"Original corpus :\n{corpus}\n")


tokens = word_tokenize(corpus)

print(f"Tokenized words : \n{tokens}\n")
```

tagged_words = pos_tag(tokens)

print(f"POS tagged words : \n{tagged_words}")

OUTPUT:

68_Adnan Shaikh

In [3]:
```
from random import choice
from nltk import pos_tag
from nltk import word_tokenize
from nltk.corpus import brown

samples = choice(brown.paras(categories="adventure"))
corpus = " ".join([" ".join(sample) for sample in samples])
print(f"Original corpus :\n{corpus}\n")

tokens = word_tokenize(corpus)
print(f"Tokenized words : \n{tokens}\n")

tagged_words = pos_tag(tokens)
print(f"POS tagged words : \n{tagged_words}")
```

```
Original corpus :
He nodded and , going into the bedroom , brought a needle , thread , and scissors . He said : `` I'm going to bed
'' . He nodded at the door in front of him . `` That's my spare bedroom . The bed isn't made , but you'll find plen
ty of blankets there '' .

Tokenized words :
['He', 'nodded', 'and', ',', 'going', 'into', 'the', 'bedroom', ',', 'brought', 'a', 'needle', ',', 'thread', ',',
'and', 'scissors', '.', 'He', 'said', ':', '``', 'I', "'m", 'going', 'to', 'bed', '``', '.', 'He', 'nodded', 'at',
'the', 'door', 'in', 'front', 'of', 'him', '.', '``', 'That', "'s", 'my', 'spare', 'bedroom', '.', 'The', 'bed', 'i
s', "n't", 'made', ',', 'but', 'you', "'ll", 'find', 'plenty', 'of', 'blankets', 'there', '``', '.']

POS tagged words :
[('He', 'PRP'), ('nodded', 'VBD'), ('and', 'CC'), (',', ','), ('going', 'VBG'), ('into', 'IN'), ('the', 'DT'), ('be
droom', 'NN'), (',', ','), ('brought', 'VBD'), ('a', 'DT'), ('needle', 'NN'), (',', ','), ('thread', 'NN'), (',',
','), ('and', 'CC'), ('scissors', 'NNS'), ('.', '.'), ('He', 'PRP'), ('said', 'VBD'), (':', ':'), ('``', '``'),
('I', 'PRP'), ("'m", 'VBP'), ('going', 'VBG'), ('to', 'TO'), ('bed', 'VB'), ('``', '``'), ('.', '.'), ('He', 'PR
P'), ('nodded', 'VBD'), ('at', 'IN'), ('the', 'DT'), ('door', 'NN'), ('in', 'IN'), ('front', 'NN'), ('of', 'IN'),
('him', 'PRP'), ('.', '.'), ('``', '``'), ('That', 'DT'), ("'s", 'VBZ'), ('my', 'PRP$'), ('spare', 'JJ'), ('bedroo
m', 'NN'), ('.', '.'), ('The', 'DT'), ('bed', 'NN'), ('is', 'VBZ'), ("n't", 'RB'), ('made', 'VBN'), (',', ','), ('b
ut', 'CC'), ('you', 'PRP'), ("'ll", 'MD'), ('find', 'VB'), ('plenty', 'NN'), ('of', 'IN'), ('blankets', 'NNS'), ('t
here', 'EX'), ('``', '``'), ('.', '.')]
```

EXPERIMENT NO 7

AIM: Chunking.

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

Chunking:

Chunking is used to add more structure to the sentence by following parts of speech (POS) tagging. It is also known as shallow parsing. The resulting group of words is called "chunks." In shallow parsing, there is maximum one level between roots and leaves while deep parsing comprises more than one level. Shallow Parsing is also called light parsing or chunking.

The primary usage of chunking is to make a group of "noun phrases." The parts of speech are combined with regular expressions.

Rules for Chunking:

There are no predefined rules, but you can combine them according to need and requirement.

For example, you need to tag Noun, verb (past tense), adjective, and coordinating junction from the sentence. You can use the rule as below

chunk:{<NN.?>*<VBD.?>*<JJ.?>*<CC>?}

Following table shows what the various symbol means:

Name of symbol	Description
.	Any character except new line
*	Match 0 or more repetitions
?	Match 0 or 1 repetitions

Use Case of Chunking

Chunking is used for entity detection. An entity is that part of the sentence by which machine get the value for any intention

Example: Temperature of New York. Here Temperature is the intention and New York is an entity.

In other words, chunking is used as selecting the subsets of tokens. Please follow the below code to understand how chunking is used to select the tokens. In this example, you will see the

graph which will correspond to a chunk of a noun phrase. We will write the code and draw the graph for better understanding.

CONCLUSION:

Chunking is the process of making noun phrases. It is applied after parts of speech tagging. Chunking has been studied and implemented on a text corpus.

CODE:

```
from random import choice

from nltk import pos_tag

from nltk import word_tokenize

from nltk.corpus import brown

from nltk import RegexpParser


samples = choice(brown.paras(categories="adventure"))

corpus = " ".join([" ".join(sample) for sample in samples])

print(f"Original corpus :\n{corpus}\n")


tokens = word_tokenize(corpus)

print(f"Tokenized words : \n{tokens}\n")


tagged_words = pos_tag(tokens)

print(f"POS tagged words : \n{tagged_words}\n")


grammar = """mychunk:{<NN.?>*<VBD.?>*<JJ.?>*<CC>?}"""

chunker = RegexpParser(grammar)

chunk = chunker.parse(tagged_words)


print(f"Chunked Words:\n{chunk}")
```

OUTPUT:

68_Adnan Shaikh

In [2]:
```
from random import choice
from nltk import pos_tag
from nltk import word_tokenize
from nltk.corpus import brown
from nltk import RegexpParser

samples = choice(brown.paras(categories="adventure"))
corpus = " ".join([" ".join(sample) for sample in samples])
print(f"Original corpus :\n{corpus}\n")

tokens = word_tokenize(corpus)
print(f"Tokenized words : \n{tokens}\n")

tagged_words = pos_tag(tokens)
print(f"POS tagged words : \n{tagged_words}\n")

grammar = """mychunk:{<NN.?>*<VBD.?>*<JJ.?>*<CC>?}"""
chunker = RegexpParser(grammar)
chunk = chunker.parse(tagged_words)

print(f"Chunked Words:\n{chunk}")
```

```
Original corpus :
`` I am an honest man '' , the German said with fervor . `` I will give Mr. Roy his due for this dive . I will make
him distributor for all of Florida -- a big market . All tourists come to Florida . This will help him to get out o
f his little tackle shop . Yes ! ! But there is no use causing him to worry at this time '' .

Tokenized words :
['``', 'I', 'am', 'an', 'honest', 'man', "''", ',', 'the', 'German', 'said', 'with', 'fervor', '.', '``', 'I', 'wil
l', 'give', 'Mr.', 'Roy', 'his', 'due', 'for', 'this', 'dive', '.', 'I', 'will', 'make', 'him', 'distributor', 'fo
r', 'all', 'of', 'Florida', '--', 'a', 'big', 'market', '.', 'All', 'tourists', 'come', 'to', 'Florida', '.', 'Thi
s', 'will', 'help', 'him', 'to', 'get', 'out', 'of', 'his', 'little', 'tackle', 'shop', '.', 'Yes', '!', '!', 'Bu
t', 'there', 'is', 'no', 'use', 'causing', 'him', 'to', 'worry', 'at', 'this', 'time', "''", '.']

POS tagged words :
[('``', '``'), ('I', 'PRP'), ('am', 'VBP'), ('an', 'DT'), ('honest', 'NN'), ('man', 'NN'), ("''", "''"), (',',
','), ('the', 'DT'), ('German', 'NNP'), ('said', 'VBD'), ('with', 'IN'), ('fervor', 'NN'), ('.', '.'), ('``', '``
'), ('I', 'PRP'), ('will', 'MD'), ('give', 'VB'), ('Mr.', 'NNP'), ('Roy', 'NNP'), ('his', 'PRP$'), ('due', 'NN'),
('for', 'IN'), ('this', 'DT'), ('dive', 'NN'), ('.', '.'), ('I', 'PRP'), ('will', 'MD'), ('make', 'VB'), ('him', 'P
RP'), ('distributor', 'VB'), ('for', 'IN'), ('all', 'DT'), ('of', 'IN'), ('Florida', 'NNP'), ('--', ':'), ('a', 'D
T'), ('big', 'JJ'), ('market', 'NN'), ('.', '.'), ('All', 'DT'), ('tourists', 'NNS'), ('come', 'VBP'), ('to', 'T
O'), ('Florida', 'NNP'), ('.', '.'), ('This', 'DT'), ('will', 'MD'), ('help', 'VB'), ('him', 'PRP'), ('to', 'TO'),
('get', 'VB'), ('out', 'IN'), ('of', 'IN'), ('his', 'PRP$'), ('little', 'JJ'), ('tackle', 'NN'), ('shop', 'NN'),
('.', '.'), ('Yes', 'UH'), ('!', '.'), ('!', '.'), ('But', 'CC'), ('there', 'EX'), ('is', 'VBZ'), ('no', 'DT'), ('u
se', 'NN'), ('causing', 'VBG'), ('him', 'PRP'), ('to', 'TO'), ('worry', 'VB'), ('at', 'IN'), ('this', 'DT'), ('tim
e', 'NN'), ("''", "''"), ('.', '.')]
```

```
Chunked Words:
(S
  ``/``
  I/PRP
  am/VBP
  an/DT
  (mychunk honest/NN man/NN)
  ``/``
  ,/,
  the/DT
  (mychunk German/NNP said/VBD)
  with/IN
  (mychunk fervor/NN)
  ./.
  ``/``
  I/PRP
  will/MD
  give/VB
  (mychunk Mr./NNP Roy/NNP)
  his/PRP$
  (mychunk due/NN)
  for/IN
  this/DT
  (mychunk dive/NN)
  ./.
  I/PRP
  will/MD
  make/VB
  him/PRP
  distributor/VB
  for/IN
  all/DT
  of/IN
  (mychunk Florida/NNP)
  --/:
  a/DT
  (mychunk big/JJ)
  (mychunk market/NN)
  ./.
  All/DT
  (mychunk tourists/NNS)
  come/VBP
  to/TO
  (mychunk Florida/NNP)
  ./.
  This/DT
  will/MD
  help/VB
  him/PRP
  to/TO
  get/VB
  out/IN
  of/IN
  his/PRP$
  (mychunk little/JJ)
  (mychunk tackle/NN shop/NN)
  ./.
  Yes/UH
  !/.
  !/.
  (mychunk But/CC)
  there/EX
  is/VBZ
  no/DT
  (mychunk use/NN)
  causing/VBG
  him/PRP
  to/TO
  worry/VB
  at/IN
  this/DT
  (mychunk time/NN)
  ``/``
  ./.)
```

EXPERIMENT NO.8

AIM:

Named Entity Recognition

RESOURCES REQUIRED:

Python 3, NLTK toolkit, Text editor, 4 GB RAM and above, i5 processor and above

THEORY:

Named Entity Recognition:

Named-entity recognition (NER) (also known as entity identification, entity chunking and entity extraction) is a sub-task of information extraction that seeks to locate and classify named entities in text into pre-defined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.

NER systems have been created that use linguistic grammar-based techniques as well as statistical models such as machine learning. Hand-crafted grammar-based systems typically obtain better precision, but at the cost of lower recall and months of work by experienced computational linguists . Statistical NER systems typically require a large amount of manually annotated training data. Semi-supervised approaches have been suggested to avoid part of the annotation effort.

Named Entity Recognition has a wide range of applications in the field of Natural Language Processing and Information Retrieval. Few such examples have been listed below :

Automatically Summarizing Resumes

Optimizing Search Engine Algorithms

Powering Recommender Systems

Now that we explained NLP, we can describe how Named Entity Recognition works. NER plays a major role in the semantic part of NLP, which, extracts the meaning of words, sentences and their relationships. Basic NER processes structured and unstructured texts by identifying and locating entities. For example, instead of identifying “Steve” and “Jobs” as different entities, NER understands that “Steve Jobs” is a single entity. More developed NER processes can classify identified entities as well. In this case, NER not only identifies but classifies “Steve Jobs” as a person. In the following, we will describe the two most popular NER methods.

CONCLUSION:

Named Entity Recognition is a technique in natural language processing used to extract real entity names from word corpora. NER can be used to extract names, location, places etc. NER has been carefully studied and implemented.

CODE:

```
from spacy import displacy
from collections import Counter
from nltk.corpus import gutenberg
from nltk import pos_tag
from pprint import pprint
import spacy
import nltk
import essential_generators

enr = spacy.load("en_core_web_sm")
text = essential_generators.DocumentGenerator().paragraph()
text
doc = enr(text)
pprint([(x.text,x.label_) for x in doc.ents])
pprint([(x,x.ent_iob_,x.ent_type_) for x in doc])
displacy.render(doc,jupyter=True,style="ent")
```

OUTPUT:

In [1]:
```
from spacy import displacy
from collections import Counter
from nltk.corpus import gutenberg
from nltk import pos_tag
from pprint import pprint
import spacy
import nltk
import essential_generators
```

In [2]:
```
enr = spacy.load("en_core_web_sm")
text = essential_generators.DocumentGenerator().paragraph()
text
```

Out[2]:
```
"Psychopathology through mobility wing, which. Numerous in nationals are speakers of other indo-european languages made up of 128 senators.. Kurose james germinate, bloom and die in the city's. Health conditions, atlanta garnering most of which. Cabinet ministers, overthrow president hosni mubarak. statistics show that. Inconveniences it month of january 13, 1847, securing american control in california.. Book by service providers, from. Organ, which branched from one. From dehydration the 45th-largest media market in. Spinal cord, 0.56% (40,724) of the continent today. european colonization began when norsemen settled. World, it regime for expatriates. the danish."
```

In [3]:
```
doc = enr(text)
pprint([(x.text,x.label_) for x in doc.ents])
```

```
[('european', 'NORP'),
 ('128', 'CARDINAL'),
 ('Kurose james germinate', 'PERSON'),
 ('atlanta', 'GPE'),
 ('hosni mubarak', 'PERSON'),
 ('american', 'NORP'),
 ('california', 'GPE'),
 ('Organ', 'ORG'),
 ('one', 'CARDINAL'),
 ('45th', 'ORDINAL'),
 ('0.56%', 'PERCENT'),
 ('40,724', 'CARDINAL'),
 ('today', 'DATE'),
 ('european', 'NORP'),
 ('danish', 'NORP')]
```

In [4]:
```
pprint([(x,x.ent_iob_,x.ent_type_) for x in doc])
```

```
[(Psychopathology, 'O', ''),
 (through, 'O', ''),
 (mobility, 'O', ''),
 (wing, 'O', ''),
 (,, 'O', ''),
 (which, 'O', ''),
 (., 'O', ''),
 (Numerous, 'O', ''),
 (in, 'O', ''),
 (nationals, 'O', ''),
 (are, 'O', ''),
 (speakers, 'O', ''),
 (of, 'O', ''),
 (other, 'O', ''),
 (indo, 'O', ''),
 (-, 'O', ''),
 (european, 'B', 'NORP'),
 (languages, 'O', ''),
 (made, 'O', ''),
```

In [5]:
```
displacy.render(doc,jupyter=True,style="ent")
```

Psychopathology through mobility wing, which. Numerous in nationals are speakers of other indo- european NORP languages made up of 128 CARDINAL senators.. Kurose james germinate PERSON , bloom and die in the city's. Health conditions, atlanta GPE garnering most of which. Cabinet ministers, overthrow president hosni mubarak PERSON . statistics show that. Inconveniences it month of january 13, 1847, securing american NORP control in california GPE .. Book by service providers, from. Organ ORG , which branched from one CARDINAL . From dehydration the 45th ORDINAL -largest media market in. Spinal cord, 0.56% PERCENT (40,724 CARDINAL) of the continent today DATE . european NORP colonization began when norsemen settled. World, it regime for expatriates. the danish NORP .