Natural Language Processing

Natural language processing (NLP) is a field that focuses on making natural human language usable by computer programs.

Regex

To work with text data in Python, and in any other language, one has to be familiar with regular expressions (Regex). A regular expression is a sequence of characters that defines a search pattern. They will help us locate,match and deal with text (strings). They provide a very powerfull method to process text. We can do things like:

- Find patterns
- MAke sure the text follows a particular pattern
- Extract and edit substrings from the text

The first thing we need to know is what do have in commmon the text fragments that we are looking after. The second thing is which fragments I don't want. A very interesting thing is that regex is universal for any programming language. In Python there are two important libraries `re` and `trrex`.

Usual Operators

Regex can be considered a language on its own. For this reason, let's see the main operations that we can do to specify regular expressions. Those are known as metacharacters, which are characters that are interpreted in a special way by Regex. Some are `[].^$*+?{}()\|`. Let's see their meaning (for more information visit here):

- **[]** : Square brackets specifies a set of characters you wish to match. For example, [abc] will have a match if the string you are trying to match contains any of the a,b or c.
- **.** : It matches any single character.
- **^** : It is used to check if a string starts with a certain character. For example **^a** will match **abc** but will not match **bca**
- **$** : It is used to check if a string ends with a certain character.
- ***** : It matches zero or more occurrences of the pattern left to it. For example **ma*n** matches with **mn** but not with **main** since **a** is not followed by **n**
- **+** : Matches one or more occurrences of the pattern left to it.
- **?** : Matches zero or one occurrence of the pattern left ot it.
- **{}** : **{n,m}** means that at least **n** and at most **m** repetitions of the pattern left to it.
- **|** : alternation. For example **a|b** matches **ade** or **abe** .
- **()** : It is used to group sub-patterns. For example, **(a|b|c)xz** match any string that matches either **a** or **b** or **c** followed by **xz** .
- **** it is used to escape various characters including all metacharacters.

Special Sequences

Special sequences make commonly used patterns easier to write. Some of those are:

- **\A** : Matches if the specified characters are at the start of a string.
- **\b** : Matches if the specified characters are at the begining or end of a word. Example **\bfoo** matches **football** and **foo \b** matches with **afoo** .
- **\B** : Opposite of **\b** .
- **\d** : Matches any decimal digit.
- **\D** : Matches any non-decimal digit. Equivalent to **[^0-9]**.
- **\s** : Matches where a string contains any whitespace character.
- **\S** : Matches where a string contains any non-whitespace character.
- **\w** : Matches any aplhanumeric character (digits and alphabets).
- **\W** : Matches any non-alphanumeric character.
- **\Z** : Matches if the specified characters are at the end of a string.

Visit here for a complete list of methods.

Regex in python

In Python we will work with the library `re` . There are five main functions that will allow us to work with Regex. Those are:

- **findall**: returns a list with all the matches.
- **sub** : substitues one or more matches with a specified string.
- **split** : returns a list in which the string has been splitted in every match.
- **match** : search fr the pattern of the regex and returns the first occurrence.
- **search**: returns a match object if there is a coincidence in any part of the sequence.

After all this theory, we are now ready to see how all this works!

re.findall()

In [40]:
```
import re
text = "I don't like Econometrics, I prefer Microeconomics 101"
x  = re.findall('cs',text)
print(x)
```

```
['cs', 'cs']
```

In [46]:
```
# To find a word we can use \w
print(re.findall('\w',text))
```

```
['I', 'd', 'o', 'n', 't', 'l', 'i', 'k', 'e', 'E', 'c', 'o', 'n', 'o', 'm', 'e',
't', 'r', 'i', 'c', 's', 'I', 'p', 'r', 'e', 'f', 'e', 'r', 'M', 'i', 'c', 'r',
'o', 'e', 'c', 'o', 'n', 'o', 'm', 'i', 'c', 's', '1', '0', '1']
```

In [47]:
```
print(re.findall('\w+',text))
```

```
['I', 'don', 't', 'like', 'Econometrics', 'I', 'prefer', 'Microeconomics', '101']
```

In [41]:
```
# Suppose we want to find those words that finish with 'cs'.
x = re.findall('\w+cs',text)
print(x)
```

```
['Econometrics', 'Microeconomics']
```

In [42]:
```
# Now find those that start with E and finish with 'cs'
x = re.findall('E\w+cs',text)
print(x)
```

```
['Econometrics']
```

In [43]:
```
# Suppose we want to extract the numbers:
x = re.findall('\d',text)
print(x)
```

```
['1', '0', '1']
```

In [44]:
```
# But what if we want all the number
x = re.findall('\d+',text)
print(x)
```

```
['101']
```

In [56]:
```
# And find words that start with E.
text = "I don't like Econometrics, xxEconskldfh I prefer Microeconomics 101"
print(re.findall(r"\bE\w+", text))   # Important to inlcude  \b !!!
```

```
['Econometrics']
```

In [61]:
```
# Example using ?
text =  'yes yeeees yoo yeeah'
print(re.findall('ye?',text))
```

```
['ye', 'ye', 'y', 'ye']
```

In [77]:
```
# Using *
print(re.findall(r'\bye*',text))
```

```
[]
```

In [80]:
```
# Using {}

print(re.findall(r'\we{3}',text))
```

```
['yeee']
```

In [81]:
```
print(re.findall(r'\we{1,3}',text))
```

```
['ye', 'yeee', 'yee']
```

In [93]:
```
text = 'house House tree Tree'

# using the or

print(re.findall(r'\bh\w+|\bH\w+',text))
```

```
['house', 'House']
```

In [ ]:
```

```

re.sub()

In [99]:
```
text = 'Econometrics is better than macroeconomics'

print(re.sub('better','worse',text))
```

```
Econometrics is worse than macroeconomics
```

In [101…
```
print(re.sub('\we','XX',text))
```

```
EconoXXtrics is XXtXXr than macrXXconomics
```

In [105…]
```
print(re.sub('\we','XX',text,2))   # do it only twice!
```

```
EconoXXtrics is XXtter than macroeconomics
```

In [106…]
```
print(re.sub(r'\bE\w+cs','Sleeping',text))
```

```
Sleeping is better than macroeconomics
```

re.split()

In [107…]
```
# split based on a space
print(re.split('\s',text))
```

```
['Econometrics', 'is', 'better', 'than', 'macroeconomics']
```

In [108…]
```
# split based on a character
print(re.split('a',text))
```

```
['Econometrics is better th', 'n m', 'croeconomics']
```

re.match()

In [119…]
```
text = 'a aa a a a a'
a = re.match('a',text)
print(a)
```

```
<re.Match object; span=(0, 1), match='a'>
```

In [120…]
```
# we can access its elements
a.span()
```

Out[120]:
```
(0, 1)
```

In [121…]
```
a.group()
```

Out[121]:
```
'a'
```

re.search()

In [115…]
```
re.search('cs',text)
```

Out[115]:
```
<re.Match object; span=(10, 12), match='cs'>
```

In [116…]
```
# we cn access itls elements.

print(re.search('cs',text).span())
```

```
(10, 12)
```

In [117…]
```
# or the word

print(re.search('cs',text).group())
```

```
cs
```

For a list of exercices with regular expressions visit here.

Natural Language Toolkit

Now that we know regular expressions, we can move to the real part of Natural Language Processing. There are many packages to do that with Python. We will be using Natural Language Toolkit (NLTK) which is a leading plataform to work with human language data.

Before we can analyize the data we first need to preprocess it. For this we will use NLTK. To install it we just need to type:

In [124...
```
!pip install nltk
```

```
Requirement already satisfied: nltk in c:\users\sergi\anaconda3\lib\site-packages
(3.7)
Requirement already satisfied: regex>=2021.8.3 in c:\users\sergi\anaconda3\lib\sit
e-packages (from nltk) (2022.7.9)
Requirement already satisfied: joblib in c:\users\sergi\anaconda3\lib\site-package
s (from nltk) (1.1.0)
Requirement already satisfied: click in c:\users\sergi\anaconda3\lib\site-packages
(from nltk) (8.0.4)
Requirement already satisfied: tqdm in c:\users\sergi\anaconda3\lib\site-packages
(from nltk) (4.64.1)
Requirement already satisfied: colorama in c:\users\sergi\anaconda3\lib\site-packa
ges (from click->nltk) (0.4.5)
```

Preprocessing Text

The first thing we must do to preprocess text is to clean it using regular expressions. We will assume that we have already done that. The next is to divide it and prepare it for the analysis. For this section we will be following the tutorial here.

Tokenizing

It is the process of spliting the text by words or sentences. This will allow us to work with smaller pieces of text that are stil cohorent and meaningfull even outside of the context of the rest of the text. It is considered the first step in turingn unestructured data into structured data. Let's see how it works:

In [126...
```
from nltk.tokenize import sent_tokenize, word_tokenize

example_string = """
Muad'Dib learned rapidly because his first training was in how to learn.
And the first lesson of all was the basic trust that he could learn.
It's shocking to find how many people do not believe they can learn,
and how many more believe learning to be difficult."""
```

Let's now split the previous text into sentences.

In [131...
```
print(sent_tokenize(example_string))
```

```
["\nMuad'Dib learned rapidly because his first training was in how to learn.", 'An
d the first lesson of all was the basic trust that he could learn.', "It's shockin
g to find how many people do not believe they can learn,\nand how many more believ
e learning to be difficult."]
```

We can also tokenize by word:

In [132...
```
print(word_tokenize(example_string))
```

```
["Muad'Dib", 'learned', 'rapidly', 'because', 'his', 'first', 'training', 'was',
'in', 'how', 'to', 'learn', '.', 'And', 'the', 'first', 'lesson', 'of', 'all', 'wa
s', 'the', 'basic', 'trust', 'that', 'he', 'could', 'learn', '.', 'It', "'s", 'sho
cking', 'to', 'find', 'how', 'many', 'people', 'do', 'not', 'believe', 'they', 'ca
n', 'learn', ',', 'and', 'how', 'many', 'more', 'believe', 'learning', 'to', 'be',
'difficult', '.']
```

Filtering Stop Words

Stop words are words that you want to ignore, so you filter them out of your text. Very common words like *in* ,*is* and *and* are often used as stop words since they don't adda a lot of meaning tot a text for themselves. To do that we use:

In [140…
```
from nltk.tokenize import word_tokenize

text = 'I will study Econometrics at home and then Microeconomics'
tokenized = word_tokenize(text)
print(tokenized)
```

```
['I', 'will', 'study', 'Econometrics', 'at', 'home', 'and', 'then', 'Microeconomic
s']
```

In [ ]:
```

```

In [138…
```
nltk.download("stopwords")
from nltk.corpus import stopwords
print(stopwords.words('english'))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "yo
u've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him',
'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its',
'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who',
'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'we
re', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did',
'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'wh
ile', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'thr
ough', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down',
'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'he
re', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few',
'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'sam
e', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't",
'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren',
"aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "had
n't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "might
n't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\Sergi\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [142…
```
#To filter stop words we can just use a list copmrehension
stop_words = set(stopwords.words('english'))
filtered_sentence = [w for w in tokenized if not w.lower() in stop_words]
print(filtered_sentence)
```

```
['study', 'Econometrics', 'home', 'Microeconomics']
```

Stemming

Stemming is a text processing task in which you reduce words to their root, which is the core part of a word. Fir example, the words *helping* and *helper* share the same root *help*. This

will make text comparisons easier, since we will focus on the basic meaning of a word rather than on the details of how it is being used.

In [144…

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()

string_for_stemming = """The crew of the USS Discovery discovered many discoveries
Discovering is what explorers do."""

words = word_tokenize(string_for_stemming)

stemmed_words = [stemmer.stem(word) for word in words]

print(stemmed_words)
```

```
['the', 'crew', 'of', 'the', 'uss', 'discoveri', 'discov', 'mani', 'discoveri',
'.', 'discov', 'is', 'what', 'explor', 'do', '.']
```

So it looks like steamming went wrong. There are two main reasons why this can happen:

- **Understemming** : When two related words should be reduced to the same stem but aren't.
- **Overstemming** : When two unrelated words are reduced to the same stem

To overcome this issue there are other ways to reduce words to their core meaning. One of those is **lemmatizin** that we will learn on this tutorial. But first, we will cover parts of speech.

Tagging Parts of Speech

Part of speech is a grammatical term that deals with the roles words play when you use them together in sentences. Tagging parts of speech is the task of labeling the words in your text according to their part of speech. Those are:

Part of speech	Role	Examples
Noun	Is a person, place, or thing	mountain, bagel, Poland
Pronoun	Replaces a noun	you, she, we
Adjective	Gives information about what a noun is like	efficient, windy, colorful
Verb	Is an action or a state of being	learn, is, go
Adverb	Gives information about a verb, an adjective, or another adverb	efficiently, always, very
Preposition	Gives information about how a noun or pronoun is connected to another word	from, about, at
Conjunction	Connects two other words or phrases	so, because, and
Interjection	Is an exclamation	yay, ow, wow

To tag parts of the speech we will use `pos_tag()`.

In [148…

```
import nltk
nltk.download('averaged_perceptron_tagger')
from nltk.tokenize import word_tokenize

sagan_quote = """
If you wish to make an apple pie from scratch,
you must first invent the universe."""

words = word_tokenize(sagan_quote)
nltk.pos_tag(words)
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\Sergi\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping taggers\averaged_perceptron_tagger.zip.
```

Out[148]:
```
[('If', 'IN'),
 ('you', 'PRP'),
 ('wish', 'VBP'),
 ('to', 'TO'),
 ('make', 'VB'),
 ('an', 'DT'),
 ('apple', 'NN'),
 ('pie', 'NN'),
 ('from', 'IN'),
 ('scratch', 'NN'),
 (',', ','),
 ('you', 'PRP'),
 ('must', 'MD'),
 ('first', 'VB'),
 ('invent', 'VB'),
 ('the', 'DT'),
 ('universe', 'NN'),
 ('.', '.')]
```

The output is that all the words in the quote are now in a separate tuple, with a tag that represents their part of speech. To get a sense of what each tag means we can do:

In [150…
```
nltk.download('tagsets')
nltk.help.upenn_tagset()
```

```
[nltk_data] Downloading package tagsets to
[nltk_data]     C:\Users\Sergi\AppData\Roaming\nltk_data...
```

$: dollar
    $ -$ --$ A$ C$ HK$ M$ NZ$ S$ U.S.$ US$
'': closing quotation mark
    ' ''
(: opening parenthesis
    ([{
): closing parenthesis
)] }
,: comma
    ,
--: dash
    --
.: sentence terminator
    . ! ?
:: colon or ellipsis
    : ; ...
CC: conjunction, coordinating
    & 'n and both but either et for less minus neither nor or plus so
    therefore times v. versus vs. whether yet
CD: numeral, cardinal
    mid-1890 nine-thirty forty-two one-tenth ten million 0.5 one forty-
    seven 1987 twenty '79 zero two 78-degrees eighty-four IX '60s .025
    fifteen 271,124 dozen quintillion DM2,000 ...
DT: determiner
    all an another any both del each either every half la many much nary
    neither no some such that the them these this those
EX: existential there
    there
FW: foreign word
    gemeinschaft hund ich jeux habeas Haementeria Herr K'ang-si vous
    lutihaw alai je jour objets salutaris fille quibusdam pas trop Monte
    terram fiche oui corporis ...
IN: preposition or conjunction, subordinating
    astride among uppon whether out inside pro despite on by throughout
    below within for towards near behind atop around if like until below
    next into if beside ...
JJ: adjective or numeral, ordinal
    third ill-mannered pre-war regrettable oiled calamitous first separable
    ectoplasmic battery-powered participatory fourth still-to-be-named
    multilingual multi-disciplinary ...
JJR: adjective, comparative
    bleaker braver breezier briefer brighter brisker broader bumper busier
    calmer cheaper choosier cleaner clearer closer colder commoner costlier
    cozier creamier crunchier cuter ...
JJS: adjective, superlative
    calmest cheapest choicest classiest cleanest clearest closest commonest
    corniest costliest crassest creepiest crudest cutest darkest deadliest
    dearest deepest densest dinkiest ...
LS: list item marker
    A A. B B. C C. D E F First G H I J K One SP-44001 SP-44002 SP-44005
    SP-44007 Second Third Three Two * a b c d first five four one six three
    two
MD: modal auxiliary
    can cannot could couldn't dare may might must need ought shall should
    shouldn't will would
NN: noun, common, singular or mass
    common-carrier cabbage knuckle-duster Casino afghan shed thermostat
    investment slide humour falloff slick wind hyena override subhumanity
    machinist ...
NNP: noun, proper, singular
    Motown Venneboerger Czestochwa Ranzer Conchita Trumplane Christos
    Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI Darryl CTCA
    Shannon A.K.C. Meltex Liverpool ...
NNPS: noun, proper, plural

    Americans Americas Amharas Amityvilles Amusements Anarcho-Syndicalists
    Andalusians Andes Andruses Angels Animals Anthony Antilles Antiques
    Apache Apaches Apocrypha ...
NNS: noun, common, plural
    undergraduates scotches bric-a-brac products bodyguards facets coasts
    divestitures storehouses designs clubs fragrances averages
    subjectivists apprehensions muses factory-jobs ...
PDT: pre-determiner
    all both half many quite such sure this
POS: genitive marker
    ' 's
PRP: pronoun, personal
    hers herself him himself hisself it itself me myself one oneself ours
    ourselves ownself self she thee theirs them themselves they thou thy us
PRP$: pronoun, possessive
    her his mine my our ours their thy your
RB: adverb
    occasionally unabatingly maddeningly adventurously professedly
    stirringly prominently technologically magisterially predominately
    swiftly fiscally pitilessly ...
RBR: adverb, comparative
    further gloomier grander graver greater grimmer harder harsher
    healthier heavier higher however larger later leaner lengthier less-
    perfectly lesser lonelier longer louder lower more ...
RBS: adverb, superlative
    best biggest bluntest earliest farthest first furthest hardest
    heartiest highest largest least less most nearest second tightest worst
RP: particle
    aboard about across along apart around aside at away back before behind
    by crop down ever fast for forth from go high i.e. in into just later
    low more off on open out over per pie raising start teeth that through
    under unto up up-pp upon whole with you
SYM: symbol
    % & ' '' ''.)). * + ,. < = > @ A[fj] U.S U.S.S.R * ** ***
TO: "to" as preposition or infinitive marker
    to
UH: interjection
    Goodbye Goody Gosh Wow Jeepers Jee-sus Hubba Hey Kee-reist Oops amen
    huh howdy uh dammit whammo shucks heck anyways whodunnit honey golly
    man baby diddle hush sonuvabitch ...
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 ...

```
WDT: WH-determiner
    that what whatever which whichever
WP: WH-pronoun
    that what whatever whatsoever which who whom whosoever
WP$: WH-pronoun, possessive
    whose
WRB: Wh-adverb
    how however whence whenever where whereby whereever wherein whereof why
``: opening quotation mark
    ` ``
```

```
[nltk_data]   Unzipping help\tagsets.zip.
```

Lemmatizing

Now that we know how to identify the parts of the speech, we can come back to lemmatizing. Like stemming, the goal here is to reduce words to their core meaning, but the output will be a complete English word instead of a fragment like `discoveri`. This is how we use it:

In [156…
```
from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')
nltk.download('omw-1.4')
lemmatizer = WordNetLemmatizer()
lemmatizer.lemmatize("scarves")
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\Sergi\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\Sergi\AppData\Roaming\nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
```

Out[156]:
```
'scarf'
```

But what happens when you lemmatize a word that looks very different from its lemma ?

In [157…
```
lemmatizer.lemmatize("worst")
```

Out[157]:
```
'worst'
```

Lemmatize assumed that *worst* was a noun, so we have to tell him that it is actually an adjective.

In [158…
```
lemmatizer.lemmatize("worst", pos="a")
```

Out[158]:
```
'bad'
```

Concorrdance

A concordance allows us to see each time a word is used, along with its immediate context. To do that we will import some test data.

In [ ]:
```
nltk.download("book")
```

In [161…
```
from nltk.book import *
text8.concordance("man")
```

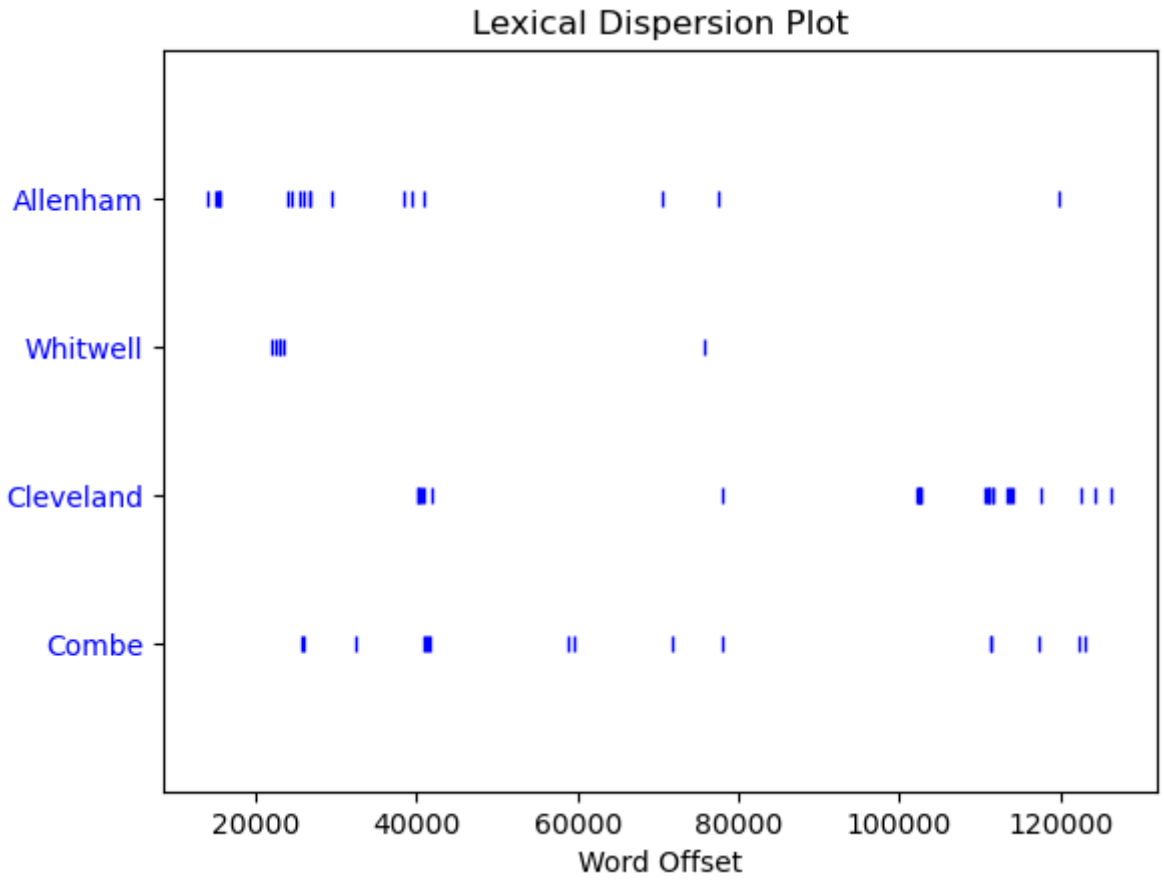
```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
Displaying 14 of 14 matches:
 to hearing from you all . ABLE young man seeks , sexy older women . Phone for
ble relationship . GENUINE ATTRACTIVE MAN 40 y . o ., no ties , secure , 5 ft .
ship , and quality times . VIETNAMESE MAN Single , never married , financially
ip . WELL DRESSED emotionally healthy man 37 like to meet full figured woman fo
 nth subs LIKE TO BE MISTRESS of YOUR MAN like to be treated well . Bold DTE no
eeks lady in similar position MARRIED MAN 50 , attrac . fit , seeks lady 40 - 5
eks nice girl 25 - 30 serious rship . Man 46 attractive fit , assertive , and k
 40 - 50 sought by Aussie mid 40s b / man f / ship r / ship LOVE to meet widowe
discreet times . Sth E Subs . MARRIED MAN 42yo 6ft , fit , seeks Lady for discr
woman , seeks professional , employed man , with interests in theatre , dining
 tall and of large build seeks a good man . I am a nonsmoker , social drinker ,
lead to relationship . SEEKING HONEST MAN I am 41 y . o ., 5 ft . 4 , med . bui
 quiet times . Seeks 35 - 45 , honest man with good SOH & similar interests , f
 genuine , caring , honest and normal man for fship , poss rship . S / S , S /
```

Visualizing words.

It is possible to make dispersion plots or even frequency distributions.

In [162…

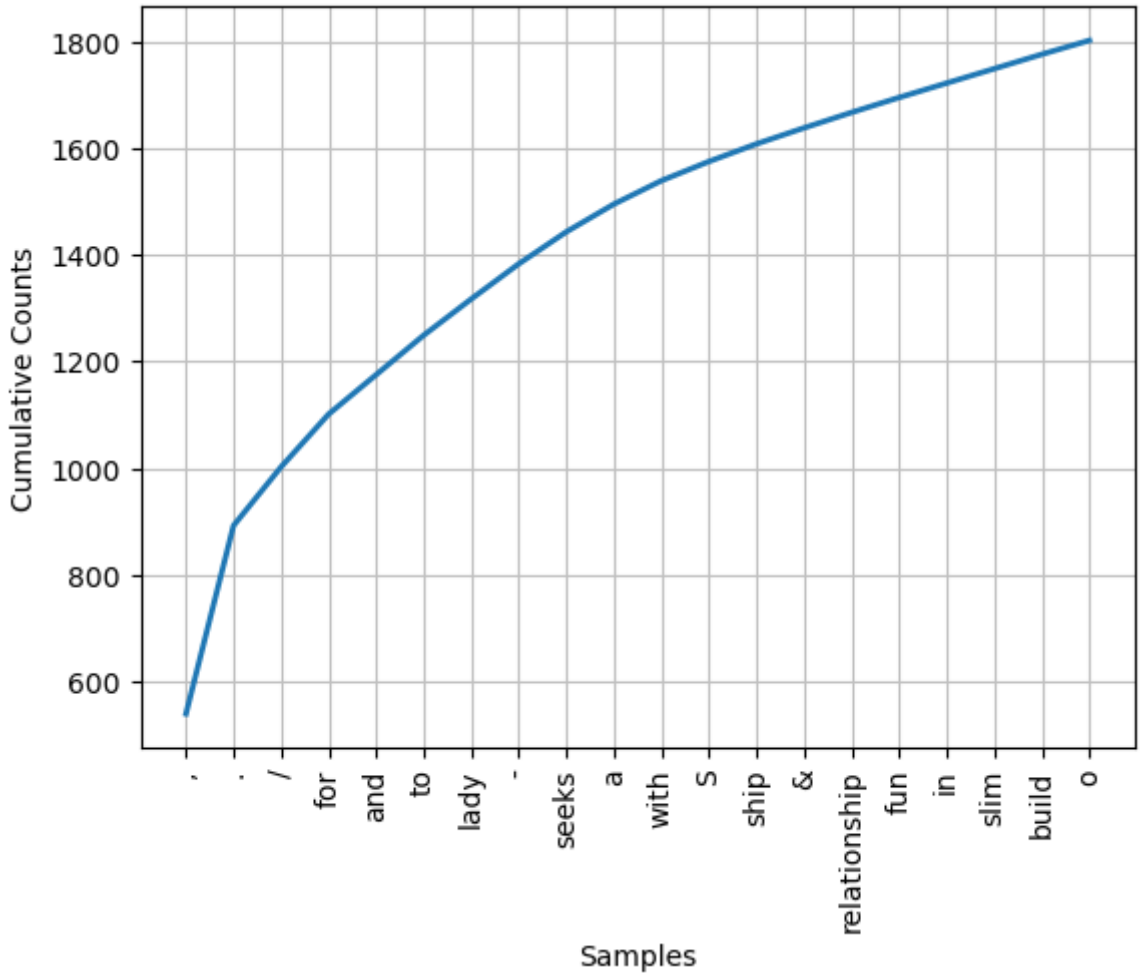
```
text2.dispersion_plot(["Allenham", "Whitwell", "Cleveland", "Combe"])
```



In [163…
```
from nltk import FreqDist
frequency_distribution = FreqDist(text8)
frequency_distribution.most_common(20)
```

Out[163]:
```
[(',', 539),
 ('.', 353),
 ('/', 110),
 ('for', 99),
 ('and', 74),
 ('to', 74),
 ('lady', 68),
 ('-', 66),
 ('seeks', 60),
 ('a', 52),
 ('with', 44),
 ('S', 36),
 ('ship', 33),
 ('&', 30),
 ('relationship', 29),
 ('fun', 28),
 ('in', 27),
 ('slim', 27),
 ('build', 27),
 ('o', 26)]
```

In [164…
```
frequency_distribution.plot(20, cumulative=True)
```



Out[164]: <AxesSubplot:xlabel='Samples', ylabel='Cumulative Counts'>

String to Vector

Most of the time when dealing with human language data we will need to give a mathematical representation. To do that, we can transform the data into vectors according to different criterias. In this section we will learn Bag of Words and the Term frequency inverse document frequency methods.

Bag of Words (BoW)

A bag of words is a representation of text that describes the occurrence of words within a document. We just keep track of word counts and disregard the grammatical details and the word order. It is called a "bag" of words because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document.

One of the biggest problems with text is that it is messy and unstructured, and machine learning algorithms prefer structured, well defined fixed-length inputs and by using the Bag-of-Words technique we can convert variable-length texts into a fixed-length vector.

Suppose we have two sentences:

Sentence 1: "Welcome to Great Learning, Now start learning"

Sentence 2: “Learning is a good practice”

The BoW outcome would be:

Sentence 1 → $[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]$

Sentence 2 → $[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]$

Although BoW provides a very easy methdology to transform words into data, it has some limitations. First of all, it does not consider the semantic meaning at all. It completely ignores the context in which it's used. On the second place, the vector size can be huge resulting in a lot of computation and time.

As an exercice, create a function that given an input, returns the BoW version of the text.

In [165…

```
['welcome', 'to', 'great', 'learning', ',', 'now', 'start', 'learning']
['learning', 'is', 'a', 'good', 'practice']
['welcome', 'to', 'great', 'learning', ',', 'now', 'start', 'is', 'a', 'good', 'pr
actice']
['welcome', 'great', 'learning', 'now', 'start', 'good', 'practice']
[1, 1, 2, 1, 1, 0, 0]
[0, 0, 1, 0, 0, 1, 1]
```

Term frequency-inverse document frequency (TFIDF)

The scoring method being used above takes the count of each word and represents the word in the vector by the number of counts of that particular word. What does a word having high word count signify?

Does this mean that the word is important in retrieving information about documents? The answer is NO. Let me explain, if a word occurs many times in a document but also along with many other documents in our dataset, maybe it is because this word is just a frequent word; not because it is relevant or meaningful.

One approach is to rescale the frequency of words by how often they appear in all documents so that the scores for frequent words like “the” that are also frequent across all documents are penalized. This approach is called term frequency-inverse document frequency or shortly known as Tf-Idf approach of scoring.TF-IDF is intended to reflect how relevant a term is in a given document. So how is Tf-Idf of a document in a dataset calculated?

TF-IDF for a word in a document is calculated by multiplying two different metrics, the term frequency and the inverse document frequency. So let's analyze those elements part by part:

Term Frequency

This measure the frequency of a word in a document. It is individual for each word and calculated as the count of the word divided by the total amount of words in a document.

tf = count of t in d / number of words in d

This will already provide a notion of the frequency of each word. However, the problem is that there are words that are very common and that do not add any value to our text, like for example 'is', 'are' or 'and'. For this reason, we want to penalize words that are very common and prioritize rare words. This is why we compute the document frequency.

Document Frequency

It is a measure of how much information a word provides,i.e., if it is common or rare across all documents. We will count the occurrences of the term `t` in the document set `N`.We consider one occurrence if the term is present in the document at least once. We do not need to know the number of times the term is present.

df(t) = ocurrence of t in N documents

We then take the inverse to prioritize rare words. So the inverse document frequency is :

idf(t) = N/df(t)

The problem here is that when we are dealing with many documents our `N` becomes very large and this explodes. Also, sometimes we might have word that occurs `0` times. For those two reasons, we redifine our measure as:

idf(t) = log(N/df(t)+1))

Finally, we can define the TF-IDF measure as:

tf-idf(t,d) = tf(t,d)* log(N/(df(t)+1))

TF-IDF using scikit-learn

Let's see an example from here.

In [166…
```
# First of all construct the corpus:
corpus = ['data science is one of the most important fields of science',
          'this is one of the best data science courses',
          'data scientists analyze data' ]
```

In [168…
```
# Vectorize the corpus.
from sklearn.feature_extraction.text import TfidfVectorizer
tr_idf_model  = TfidfVectorizer()
tf_idf_vector = tr_idf_model.fit_transform(corpus)
```

We can now analyze the results of our vectorization:

In [169…
```
tf_idf_array = tf_idf_vector.toarray()

print(tf_idf_array)
```

```
[[0.         0.         0.         0.18952581 0.32089509 0.32089509
  0.24404899 0.32089509 0.48809797 0.24404899 0.48809797 0.
  0.24404899 0.        ]
 [0.         0.40029393 0.40029393 0.23642005 0.         0.
  0.30443385 0.         0.30443385 0.30443385 0.30443385 0.
  0.30443385 0.40029393]
 [0.54270061 0.         0.         0.64105545 0.         0.
  0.         0.         0.         0.         0.         0.54270061
  0.         0.        ]]
```

Or even create a data frame!

In [170…]:
```
words_set = tr_idf_model.get_feature_names()

df_tf_idf = pd.DataFrame(tf_idf_array, columns = words_set)

df_tf_idf
```

```
C:\Users\Sergi\anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

Out[170]:

	analyze	best	courses	data	fields	important	is	most	of	
0	0.000000	0.000000	0.000000	0.189526	0.320895	0.320895	0.244049	0.320895	0.488098	0.24
1	0.000000	0.400294	0.400294	0.236420	0.000000	0.000000	0.304434	0.000000	0.304434	0.30
2	0.542701	0.000000	0.000000	0.641055	0.000000	0.000000	0.000000	0.000000	0.000000	0.00

Cosine Similarity

Once we have our data vectorized we might want to have a notion of how similar our documents are. For this we can use cosine similarity. It is defined as the cosie of the angle between two vectors. The good properties is that it always belong to the interval -1,1. For example, two proportional vectors have a cosine similarity of 1, two orthogonal vectors have a similarity of 0, and two opposite vectors have a similarity of -1.

In [171…]:
```
from sklearn.metrics.pairwise import cosine_similarity
cosine_sim = cosine_similarity(tf_idf_array, tf_idf_array)
print(cosine_sim)
```

```
[[1.         0.56488512 0.12149655]
 [0.56488512 1.         0.15155836]
 [0.12149655 0.15155836 1.        ]]
```

tf-idf vectors for TED talks

For the next example we will be following the tutorial here.

We have a data set with 500 TED talks with its transcript.

In [176…]:
```
df = pd.read_csv('Ted.csv')
df.head()
```

Out[176]:

	transcript	url
0	We're going to talk — my — a new lecture, just...	https://www.ted.com/talks/al_seckel_says_our_b...
1	This is a representation of your brain, and yo...	https://www.ted.com/talks/aaron_o_connell_maki...
2	It's a great honor today to share with you The...	https://www.ted.com/talks/carter_emmart_demos_...
3	My passions are music, technology and making t...	https://www.ted.com/talks/jared_ficklin_new_wa...
4	It used to be that if you wanted to get a comp...	https://www.ted.com/talks/jeremy_howard_the_wo...

In [177...

```
# get only the transcripts

ted = df['transcript']
```

The next step is to vectorize it using tf-idf methods.

In [175...

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Create TfidfVectorizer object
vectorizer = TfidfVectorizer()

# Generate matrix of word vectors
tfidf_matrix = vectorizer.fit_transform(ted)

# Print the shape of tfidf_matrix
print(tfidf_matrix.shape)
```

```
(500, 29158)
```

Each row of the previous matrix corresponds to one of the TED talks vectorized.

In [181...

```
cosine_sim = pd.DataFrame(cosine_similarity(tfidf_matrix, tfidf_matrix))
cosine_sim.head()
```

Out[181]:

	0	1	2	3	4	5	6	7	8	
0	1.000000	0.470147	0.431115	0.502149	0.517717	0.352212	0.267109	0.292946	0.473597	0.487
1	0.470147	1.000000	0.442714	0.504594	0.510598	0.385744	0.257842	0.314353	0.480168	0.499
2	0.431115	0.442714	1.000000	0.487968	0.509343	0.419851	0.241377	0.295678	0.474169	0.457
3	0.502149	0.504594	0.487968	1.000000	0.550522	0.438015	0.264298	0.341453	0.520299	0.524
4	0.517717	0.510598	0.509343	0.550522	1.000000	0.437343	0.283068	0.354986	0.555929	0.552

5 rows × 500 columns

Now we could for example find what is the most similar TED talk for each talk. Another thing we could do is compute the sentiment of each talk (are they positive or negative?).

Sentiment Analysis

There are many libraries that allow sentiment analysis in Python. For example, we can use `NLTK` which has a built-in pretrained sentiment analyzed called VADER (Valence Aware

Dictionary and sEntiment Reasoner).

VADER is best suited for language used in social media, like short sentences with some slang and abbreviations. It's less accurate when rating longer, structured sentences, but it's often a good launching point.

Let's see how it works:

In [184…
```
from nltk.sentiment import SentimentIntensityAnalyzer
sia = SentimentIntensityAnalyzer()
sia.polarity_scores("Wow, NLTK is really powerful!")
```

Out[184]:
```
{'neg': 0.0, 'neu': 0.295, 'pos': 0.705, 'compound': 0.8012}
```

You'll get back a dictionary of different scores. The negative, neutral, and positive scores are related: They all add up to 1 and can't be negative. The compound score is calculated differently. It's not just an average, and it can range from -1 to 1.

We can apply it to our TED talks model to see how positive or negative the talks are.

In [192…
```
for i in range(5):
    print(sia.polarity_scores(ted[i]))
```

```
{'neg': 0.036, 'neu': 0.8, 'pos': 0.164, 'compound': 0.9998}
{'neg': 0.037, 'neu': 0.862, 'pos': 0.101, 'compound': 0.9984}
{'neg': 0.011, 'neu': 0.932, 'pos': 0.057, 'compound': 0.99}
{'neg': 0.052, 'neu': 0.819, 'pos': 0.129, 'compound': 0.999}
{'neg': 0.038, 'neu': 0.864, 'pos': 0.098, 'compound': 0.9997}
```

In [ ]:
```

```

In [ ]:
```

```