Text Wrangling & Cleansing

Bal Krishna Nyaupane
Assistant Professor
Department of Electronics and Computer Engineering
Paschimanchal Campus, IOE
bkn@wrc.edu.np

What is Text Wrangling?

- Text wrangling converts and ***transforms information at different levels of granularity.*** For example, information in a list could be converted into a table, or vice versa.
- Text wrangling ***can restructure and renarrate*** information. It can also clean up content from different sources, such as standardizing spelling or wording.
- Instead of editing a single document, text wrangling involves ***gathering text from many sources, and rewriting and consolidating that text into a unified document*** or content repository.
- Text wrangling applies large scale changes to text, by automating some low level transformations.
- The process involves ***data munging, text cleansing, specific preprocessing, tokenization, stemming or lemmatization and stop word*** removal.

What is Text Cleansing?

- Text cleansing is loosely used for most ***of the cleaning to be done on text***, depending on the data source, parsing performance, external noise and so on.
- In that sense, what ***we do for cleaning the html using*** **html_clean**, can be labeled as text cleansing. In another case, where we are ***parsing a PDF***, there could be ***unwanted noisy characters, non ASCII characters to be removed***, and so on. Before going on to next steps we want to remove these to get a clean text to process further.
- With a data source like xml, we might only be interested in some specific elements of the tree, with databases we may have to manipulate splitters, and sometimes we are only interested in specific columns.
- In summary, ***any process that is done with the aim to make the text cleaner and to remove all the noise surrounding the text can be termed as text cleansing***.

Extracting the Data

1. ***Text data collection using APIs***
   - *Twitter APIs, Social media*
2. ***Reading PDF file in Python***
3. ***Reading word document***
4. ***Reading JSON object***
5. ***Reading HTML page and HTML parsing***
6. ***Web scraping***

Text Preprocessing steps

- *Converting Text Data to Lowercase*
- *Removing Punctuation*
- *Standardizing Text*
- *Tokenization*
- *Stemming*
- *Lemmatization*
- *Stop words removal*

Converting Text Data to Lowercase

- The ***lower() method*** converts all uppercase characters in a string into lowercase characters and returns them.

```
text1 = 'Testing To Lower Case for NLP PROGRAMMING.'
text2 = text1.lower()
print(text2)
```

```
testing to lower case for nlp programming.
```

Removing Punctuation using Regular Expression

```
s = "I. like. This book!"
s1 = re.sub(r'[^\w\s]',"",s)
s1
```

```
'I like This book'
```

```
txt = "The mountains in beautiful Nepal"
x = re.search("^The.*Nepal$", txt)
x
```

```
<re.Match object; span=(0, 32), match='The mountains in beautiful Nepal'>
```

```
NameAge=''' Ram is 23 and Hari is 34
            Shyam is 40 and Gita is 50'''
ages=re.findall(r'\d{1,2}',NameAge) # d{1,2} for two digit.{1,3} for three digits
Names=re.findall(r'[A-Z][a-z]*',NameAge) # First letter must be Capital.
print(ages)
print(Names)
```

```
['23', '34', '40', '50']
['Ram', 'Hari', 'Shyam', 'Gita']
```

Regular Expression RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Regular Expression Metacharacters

Character	Description
[]	A set of characters
\	Signals a special sequence (can also be used to escape special characters)
.	Any character (except newline character)
^	Starts with
$	Ends with
*	Zero or more occurrences
+	One or more occurrences
?	Zero or one occurrences
{}	Exactly the specified number of occurrences
|	Either or
()	Capture and group

Regular Expression Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Regular Expression Sets

A set is a set of characters inside a pair of square brackets `[]` with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (`a` , `r` , or `n`) are present
[a-n]	Returns a match for any lower case character, alphabetically between `a` and `n`
[^arn]	Returns a match for any character EXCEPT `a` , `r` , and `n`
[0123]	Returns a match where any of the specified digits (`0` , `1` , `2` , or `3`) are present
[0-9]	Returns a match for any digit between `0` and `9`
[0-5][0-9]	Returns a match for any two-digit numbers from `00` and `59`
[a-zA-Z]	Returns a match for any character alphabetically between `a` and `z` , lower case OR upper case
[+]	In sets, `+` , `*` , `.` , `|` , `()` , `$` , `{}` has no special meaning, so `[+]` means: return a match for any `+` character in the string

Standardizing Text

- Most of the text data is in the form of either customer reviews, blogs, or tweets, where there is a high chance of people using short words and abbreviations to represent the same meaning.
- We can write our own custom dictionary to look for short words and abbreviations.

```
import re
lookup_dict = {'nlp':'natural language processing',
               'ur':'your',
               'wbu' : 'what about you'}

def text_std(input_text):
    words = input_text.split()
    new_words = []
    for word in words:
        word = re.sub(r'[^\w\s]'," ",word)
        if word.lower() in lookup_dict:
            word1 = lookup_dict[word.lower()]
            new_words.append(word1)
            new_text = " ".join(new_words)
            input_text=re.sub(word, word1, input_text)
    return [new_text,input_text]

txt=text_std("I like nlp it's ur choice.") # function call
print(txt[0])
print(txt[1])
```

```
natural language processing your
I like natyoural language processing it's your choice.
```

Tokenization

- A word (Token) is the ***minimal unit that a machine can understand*** and process. So any text string cannot be further processed without going through tokenization.
- Tokenization is the ***process of splitting a string into a list of pieces or tokens.*** A token is a piece of a whole, so a word is a token in a sentence, and a sentence is a token in a paragraph.
- The complexity of tokenization varies according to the need of the NLP application, and the complexity of the language itself.
- For example, in English it can be as simple as choosing only words and numbers through a regular expression. But for Chinese and Japanese, it will be a very complex task.

Tokenizing text into sentences

```
from nltk.tokenize import sent_tokenize
text="""Tokenization is the first step in text analytics.
        The process of breaking down a text paragraph into smaller chunks such as words or sentence is called Tokenization.
        Token is a single entity that is building blocks for sentence or paragraph.
        Sentence tokenizer breaks text paragraph into sentences."""
sentences=sent_tokenize(text)
print(sentences)
```

```
['Tokenization is the first step in text analytics.', 'The process of breaking down a text paragraph into smaller chunks suc
h as words or sentence is called Tokenization.', 'Token is a single entity that is building blocks for sentence or paragrap
h.', 'Sentence tokenizer breaks text paragraph into sentences.']
```

- The ***sent_tokenize*** function uses an instance of ***PunktSentenceTokenizer*** from the ***nltk.tokenize.punkt*** module. This instance has already been trained and works well for many European languages. So it knows what punctuation and characters mark the end of a sentence and the beginning of a new sentence.

Tokenizing sentences into words

- We must consider things like: do we want to remove punctuation from tokens, and if so,
  - *Should we make punctuation marks tokens themselves?*
  - *Should we preserve hyphenated words as compound elements or break them apart?*
  - *Should we approach contractions as one token or two, and if they are two tokens, where should they be split?*
- We can select different tokenizers depending on our responses to these questions.

- Many word tokenizers available in NLTK *(e.g., TreebankWordTokenizer, WordPunctTokenize, PunktWordTokenizer, etc.)*, a common choice for tokenization is ***word_tokenize***, which invokes ***the Treebank tokenizer*** and ***uses regular expressions*** to tokenize text as in Penn Treebank.

Tokenizing sentences into words

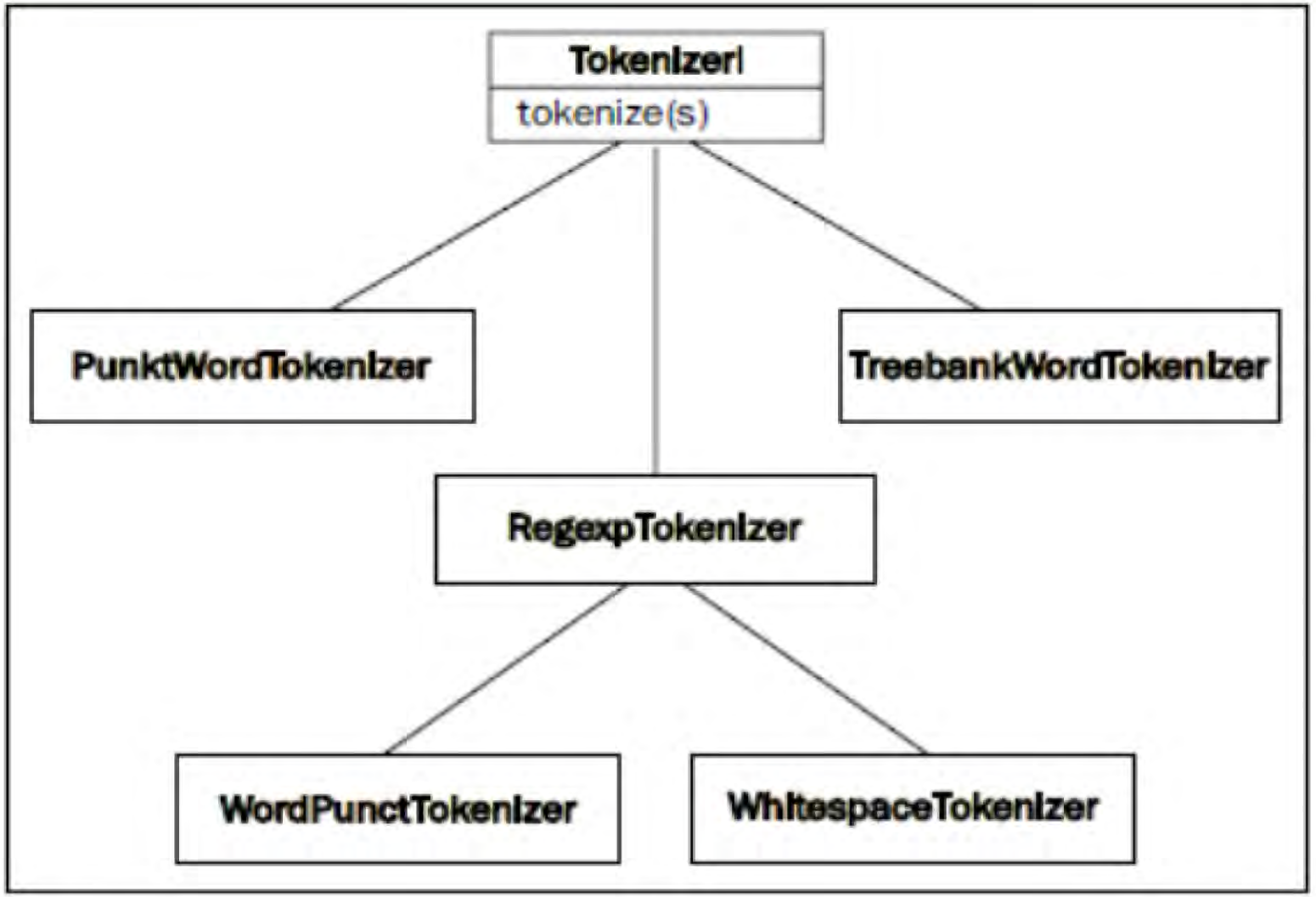
- This includes splitting standard contractions (e.g., ***"wouldn't"*** becomes ***"would"*** and ***"n't")*** and treating punctuation marks (like commas, single quotes, and periods followed by whitespace) as separate tokens.

- By contrast, ***WordPunctTokenizer*** is based on the ***RegexpTokenizer*** class, which splits strings using the regular expression ***\w+|[^\w\s]+*** , matching either tokens or separators between tokens and resulting in a sequence of alphabetic and nonalphabetic characters.

- You can also use the ***RegexpTokenizer*** class to create your own custom tokenizer.

Tokenizing sentences into words

- These differ from ***TreebankWordTokenizer*** by how they handle punctuation and contractions, but they all inherit from ***TokenizerI*** interface.
- The inheritance tree looks like what's shown in the following diagram:



Tokenizing sentences into words

```
txt="""The process of breaking down a text paragraph into smaller chunks such as
       words or sentence is called Tokenization.Token is a single entity that
       is building blocks for sentence or paragraph."""
from nltk.tokenize import word_tokenize
from nltk.tokenize import TreebankWordTokenizer
print(word_tokenize(txt))
```

```
['The', 'process', 'of', 'breaking', 'down', 'a', 'text', 'paragraph', 'into', 'smaller', 'chunks', 'such', 'as', 'words',
'or', 'sentence', 'is', 'called', 'Tokenization.Token', 'is', 'a', 'single', 'entity', 'that', 'is', 'building', 'blocks',
'for', 'sentence', 'or', 'paragraph', '.']
```

Tokenizing sentences into words

```
from nltk.tokenize import TreebankWordTokenizer
from nltk.tokenize import wordpunct_tokenize

txt1 = "Can't is a contraction."
tokenizer = TreebankWordTokenizer()
token1 = word_tokenize(sentence)
token2 = tokenizer.tokenize(sentence)
token3=wordpunct_tokenize(txt1)

print("Using word_tokenize : ",token1)
print("=======================================")
print("Using TreebankWordTokenizer : ",token2)
print("=======================================")
print("Using wordpunct_tokenize : ",token3)
```

```
Using word_tokenize :  ['Ca', "n't", 'is', 'a', 'contraction', '.']
=======================================
Using TreebankWordTokenizer :  ['Ca', "n't", 'is', 'a', 'contraction', '.']
=======================================
Using wordpunct_tokenize :  ['Can', "'", 't', 'is', 'a', 'contraction', '.']
```

Tokenizing sentences into words using regular expressions

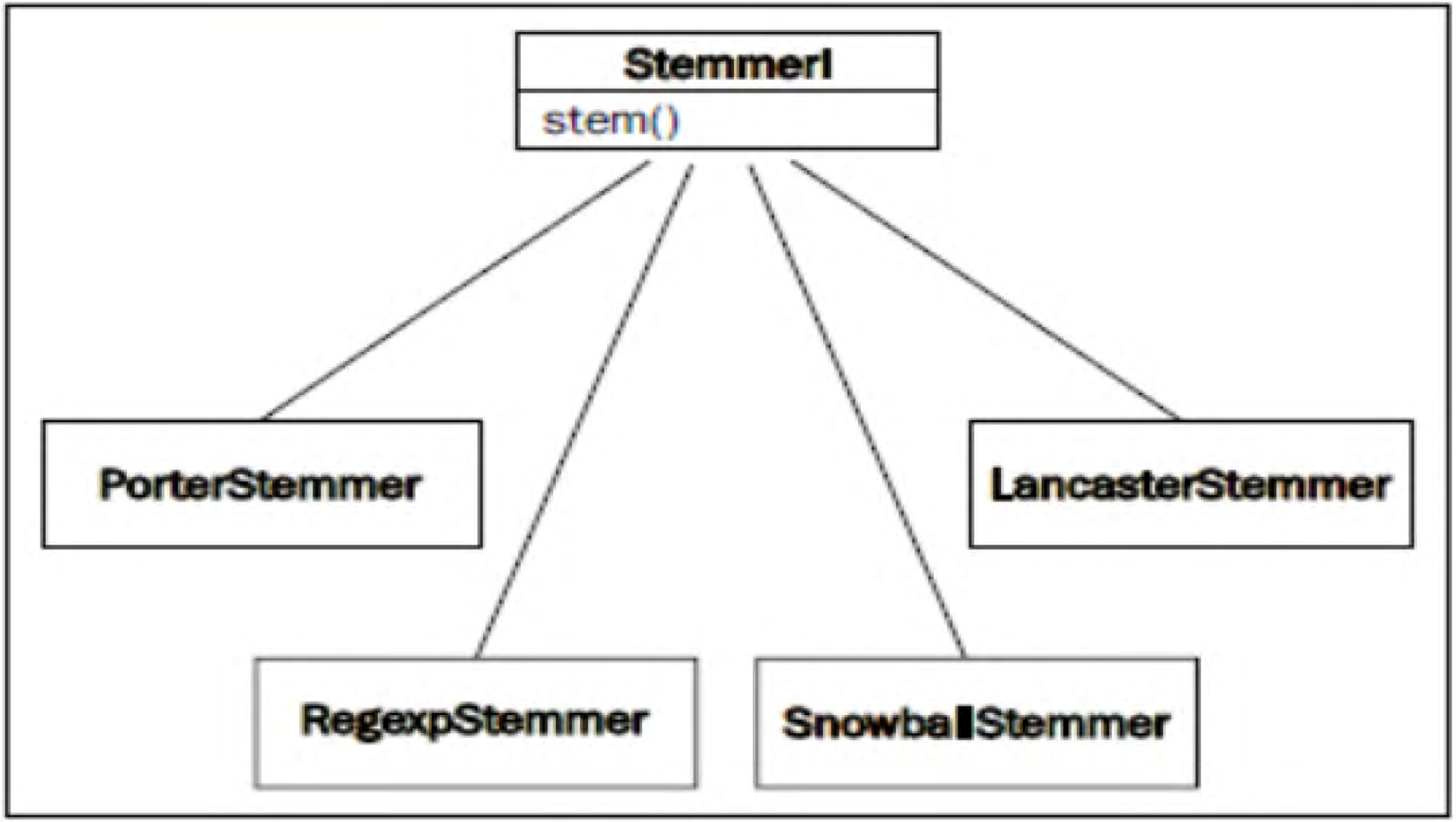
```
import re
from nltk.tokenize import regexp_tokenize

text="Hi Everyone ! Its gr8 time."
text1=regexp_tokenize(text, pattern='\w+')
print(text1)
text2=regexp_tokenize(text, pattern='\d+')
print(text2)

text3 = re.findall("[\w]+", text)
print(text3)
txt1 = "Can't is a contraction."
text3 = re.findall("[\w]+", txt1)
print(text3)
text3 = re.findall("[\w']+", txt1)
print(text3)
```

```
['Hi', 'Everyone', 'Its', 'gr8', 'time']
['8']
['Hi', 'Everyone', 'Its', 'gr8', 'time']
['Can', 't', 'is', 'a', 'contraction']
["Can't", 'is', 'a', 'contraction']
```

Stemming

- Stemming, in literal terms, is ***the process of cutting down the branches of a tree*** to its stem. Stemming is more of a ***crude rule-based process*** by which we want to club together different variations of the token. For example, the word ***eat will have variations like eating, eaten, eats, and*** so on.
- Stemming is a ***technique to remove affixes from a word***, ending up with the stem. For example, the stem of ***cooking is cook***, and a good stemming algorithm knows that the ***ing*** suffix can be removed.
- Stemming is most commonly used by search engines for indexing words. Instead of storing all forms of a word, a search engine can store only the stems, greatly reducing the size of index while increasing retrieval accuracy.
- One of the most common stemming algorithms is the ***Porter stemming algorithm*** by Martin Porter. It is designed to remove and replace well-known suffixes of English words.

Stemming


StemmerI
stem()
PorterStemmer
LancasterStemmer
RegexpStemmer
SnowballStemmer

Stemming

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
print(stemmer.stem('cooking'))
print(stemmer.stem('cookery'))
print(stemmer.stem('eats'))
print(stemmer.stem('eaten'))
print(stemmer.stem('shopping'))
```

```
cook
cookeri
eat
eaten
shop
```

```
from nltk.stem import LancasterStemmer
stemmer = LancasterStemmer()
print(stemmer.stem('cooking'))
print(stemmer.stem('cookery'))
print(stemmer.stem('eats'))
print(stemmer.stem('eaten'))
print(stemmer.stem('shopping'))
```

```
cook
cookery
eat
eat
shop
```

Stemming: PorterStemmer

```
txt="""The process of breaking down a text paragraph into smaller chunks such as
       words or sentence is called Tokenization. Token is a single entity that
       is building blocks for sentence or paragraph.
       I like fishing. I eat fish. There are many fishes in pound."""

stemmed_words=[]
for w in txt.split():
    stemmed_words.append(stemmer.stem(w))

print("Stemmed Sentence:",stemmed_words)
```

```
Stemmed Sentence: ['the', 'process', 'of', 'break', 'down', 'a', 'text', 'paragraph', 'into', 'sma
ller', 'chunk', 'such', 'as', 'word', 'or', 'sentenc', 'is', 'call', 'tokenization.', 'token', 'i
s', 'a', 'singl', 'entiti', 'that', 'is', 'build', 'block', 'for', 'sentenc', 'or', 'paragraph.',
'i', 'like', 'fishing.', 'i', 'eat', 'fish.', 'there', 'are', 'mani', 'fish', 'in', 'pound.']
```

Stemming: LancasterStemmer

```
txt="""The process of breaking down a text paragraph into smaller chunks such as
        words or sentence is called Tokenization. Token is a single entity that
        is building blocks for sentence or paragraph.
        I like fishing. I eat fish. There are many fishes in pound."""

stemmed_words=[]
for w in txt.split():
    stemmed_words.append(stemmer.stem(w))

print("Stemmed Sentence:",stemmed_words)
```

```
Stemmed Sentence: ['the', 'process', 'of', 'break', 'down', 'a', 'text', 'paragraph', 'into', 'sma
l', 'chunk', 'such', 'as', 'word', 'or', 'sent', 'is', 'cal', 'tokenization.', 'tok', 'is', 'a',
'singl', 'ent', 'that', 'is', 'build', 'block', 'for', 'sent', 'or', 'paragraph.', 'i', 'lik', 'fi
shing.', 'i', 'eat', 'fish.', 'ther', 'ar', 'many', 'fish', 'in', 'pound.']
```

Lemmatization

- Lemmatization uses ***context and part of speech*** to determine the inflected form of the word and applies different normalization rules for each part of speech to get the root word (lemma).
- Lemmatization is a process of extracting a root word by considering the vocabulary. For example, “good,” “better,” or “best” is lemmatized into good.
- Lemmatization reduces words to their base word, which is linguistically correct lemmas. It transforms root word with the use of vocabulary and morphological analysis.
- Lemmatization is usually more ***sophisticated than stemming***.

- Stemmer works on an individual word without knowledge of the context.
- For example, The word ***"better" has "good" as its lemma***.

Lemmatization

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

text = "studies studying cries cry"
tokenization = word_tokenize(text)
for w in tokenization:
    print("Stemming for {} ========> {}".format(w,stemmer.stem(w)))
```

```
Stemming for studies ========> studi
Stemming for studying ========> studi
Stemming for cries ========> cri
Stemming for cry ========> cri
```

```
from nltk.stem.wordnet import WordNetLemmatizer
lem = WordNetLemmatizer()

text = "studies studying cries cry"
tokenization = word_tokenize(text)
for w in tokenization:
    print("Lemma for {} ========> {}".format(w, lem.lemmatize(w)))
```

```
Lemma for studies ========> study
Lemma for studying ========> studying
Lemma for cries ========> cry
Lemma for cry ========> cry
```

Lemmatization

```
from nltk.stem.wordnet import WordNetLemmatizer
lem = WordNetLemmatizer()

print("rocks :", lem.lemmatize("rocks"))
print("corpora :", lem.lemmatize("corpora"))
# a denotes adjective in "pos"
print("better :", lem.lemmatize("better", pos ="a"))
print("cooking :",lem.lemmatize('cooking'))
print("cooking :",lem.lemmatize('cooking', pos='v'))
print("cooking :",lem.lemmatize('cooking', pos='v'))
print("flying :",lem.lemmatize('flying', pos='v'))
```

```
rocks : rock
corpora : corpus
better : good
cooking : cooking
cooking : cook
cooking : cook
flying : fly
```

Stop Words Removal

- Stop word removal is one of the most commonly used preprocessing steps across different NLP applications.
- Stop words are ***very common words that carry no meaning or less meaning*** compared to other keywords. If we remove the words that are less commonly used, we can focus on the important keywords instead.

- A very simple way to build a stop word list ***is based on word's document frequency*** (Number of documents the word presents), where the words present across the corpus can be treated as stop words.
- Enough research has been done to get the optimum list of stop words for some specific corpus.
- NLTK comes with a pre-built list of stop words for around 22 languages.

Stop Words Removal

- For example, if your search query is ***"How to develop chatbot using python,"*** if the search engine tries to find web pages that contained the terms "how," "to," "develop," "chatbot," "using," "python," the search engine is going to find ***a lot more pages that contain the terms "how" and "to"*** than pages that contain information about ***developing chatbot*** because the terms "how" and "to" are so commonly used in the English language.

- So, if we remove such terms, the search engine can actually focus on retrieving pages that contain the keywords: ***"develop," "chatbot," "python***" – which would more closely bring up pages that are of real interest.

- Similarly we can remove more common words and rare words as well.

Stop Words Removal

```
from nltk.corpus import stopwords
stop_words=set(stopwords.words("english"))
print(stop_words)
```

```
{"you're", 'our', 'no', 'being', "didn't", "weren't", "needn't", 'with', "it's", 'both', 'becaus
e', "should've", 'yours', 'having', 'himself', 'too', 'can', 'am', 'down', 'won', 'from', "don't",
'itself', 'up', 'when', 'a', 'there', 'were', 'hasn', 'against', 'mustn', 'here', 'about', 'would
n', 'and', "mightn't", "hasn't", 'his', 'until', 'between', 'herself', 'hers', 'further', 'or', 'w
hy', "shouldn't", 'ourselves', 'after', 'my', 're', 'an', 'we', 'to', 'this', 'but', 'does', 'myse
lf', 'needn', 'as', "that'll", 'll', 'those', 'if', 'him', "shan't", 'such', 'her', 'before', 'ai
n', "you'd", 't', "you've", 'very', 'not', 'same', 'at', 'its', 'all', 'their', 'wasn', 'was', 'on
ly', 'm', 'be', "wasn't", 'them', 'during', 'isn', 'ours', 'your', 've', 'do', 'theirs', 'few', 'h
e', 'shouldn', 'for', 'off', "couldn't", 'ma', 'have', 'who', 'some', 'below', 'most', 'should',
"haven't", "mustn't", 'any', 'each', 'than', "won't", 'shan', 'out', 'over', 'couldn', 'doesn', 't
hemselves', 'been', 'under', 'has', 'where', 'you', 'hadn', 'into', 'in', 'of', 'how', 'whom', 'no
r', 'while', "isn't", 'will', 'just', 'these', 'are', 'weren', 'now', "aren't", 's', 'o', 'again',
'didn', 'through', "hadn't", 'haven', 'by', 'yourselves', 'own', 'd', 'once', "doesn't", 'more',
"she's", "you'll", 'it', 'other', 'she', 'y', 'that', 'then', 'i', 'me', 'don', 'the', "wouldn't",
'is', 'did', 'so', 'had', 'above', 'which', 'on', 'mightn', 'yourself', 'what', 'they', 'aren', 'd
oing'}
```

Stop Words Removal

```
text="""The process of breaking down a text paragraph into smaller chunks such as
        words or sentence is called Tokenization. Token is a single entity that
        is building blocks for sentence or paragraph.
        I like fishing. I eat fish. There are many fishes in pound."""

# Removing Stopwords
tokenized_sent=word_tokenize(text)

filtered_sent=[]
for w in tokenized_sent:
    if w not in stop_words:
        filtered_sent.append(w)
print("After removing Stop Words:",filtered_sent)
print("\nTokenized Sentence:",tokenized_sent)
```

```
After removing Stop Words: ['The', 'process', 'breaking', 'text', 'paragraph', 'smaller', 'chunk
s', 'words', 'sentence', 'called', 'Tokenization', '.', 'Token', 'single', 'entity', 'building',
'blocks', 'sentence', 'paragraph', '.', 'I', 'like', 'fishing', '.', 'I', 'eat', 'fish', '.', 'The
re', 'many', 'fishes', 'pound', '.']
```

Exploring Text Data

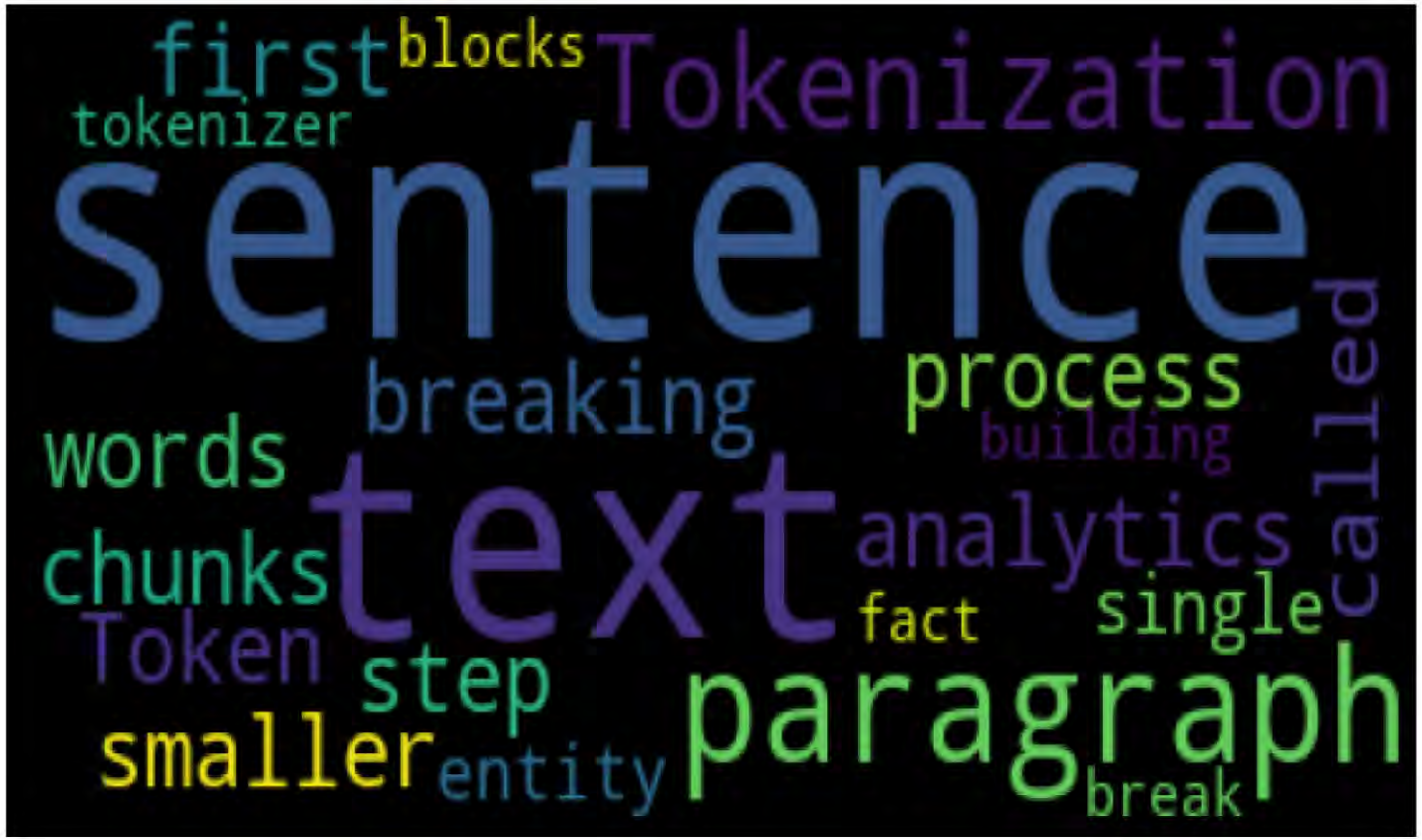
- Read the text data
- Check number of words in the data
- Compute the frequency of all words
- Consider words with different length and plot them
- Word Count - Total number of words
- Character Count - Total number of characters excluding spaces
- Word Density - Average length of the words
- Punctuation Count - Total number of punctuations used
- Word cloud: which represent the frequency or the importance of each word

Word cloud

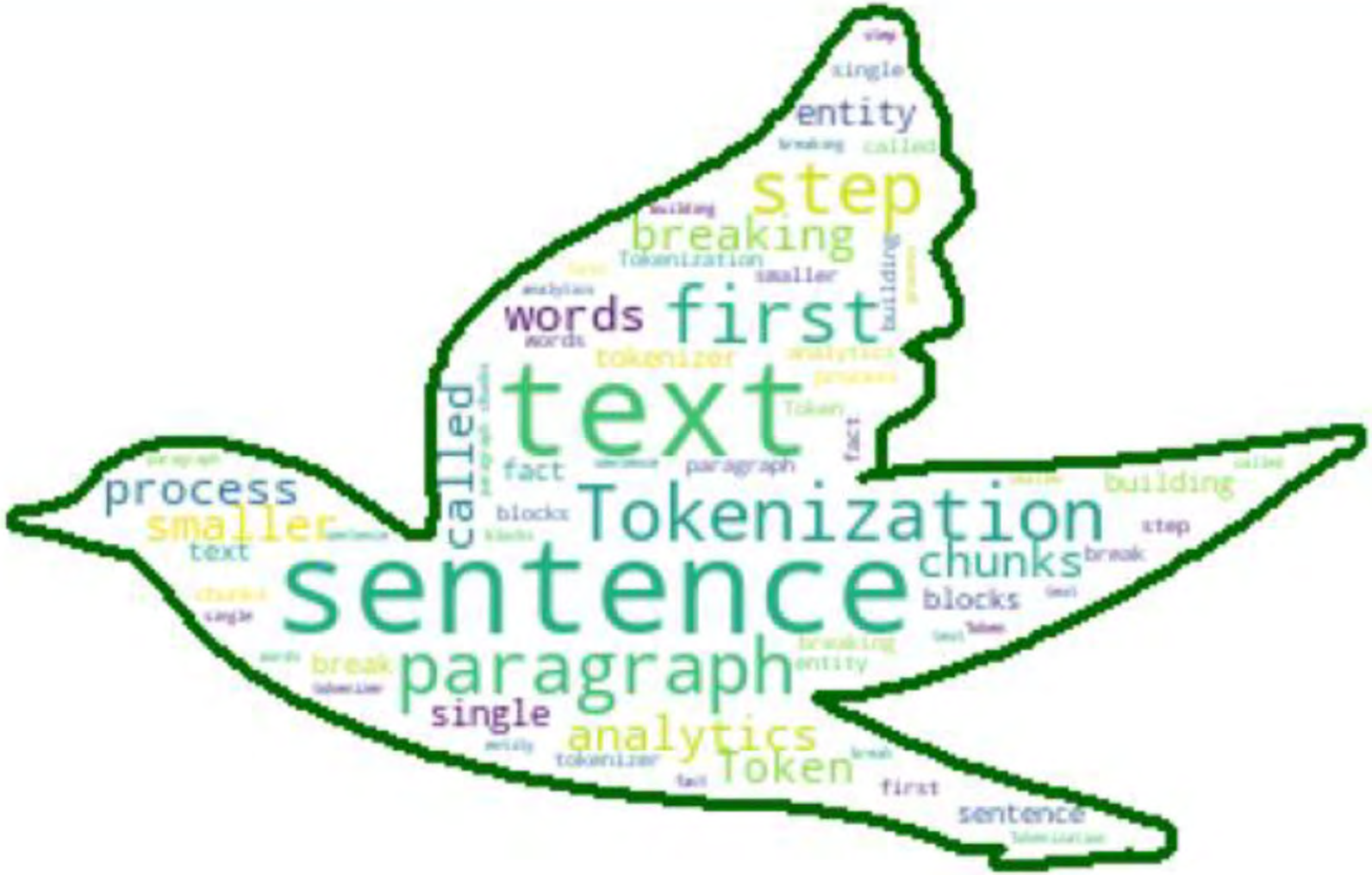
- Word Clouds (Word Clouds) are quite often called Tag clouds.
- It is a visual representation of text data. Size and colors are used to show the relative importance of words or terms in a text.
- The bigger a term is the greater is its weight. So the size reflects the frequency of a words, which may correspond to its importance.
- A word cloud is a visually prominent presentation of "keywords" that appear frequently in text data. The rendering of keywords forms a cloud-like color picture, so that you can appreciate the main text data at a glance.
- Significant textual data points can be highlighted using a word cloud. Word clouds are widely used for analyzing data from social network websites.
- For generating word cloud in Python, modules needed are – matplotlib, pandas and word cloud.

Word cloud

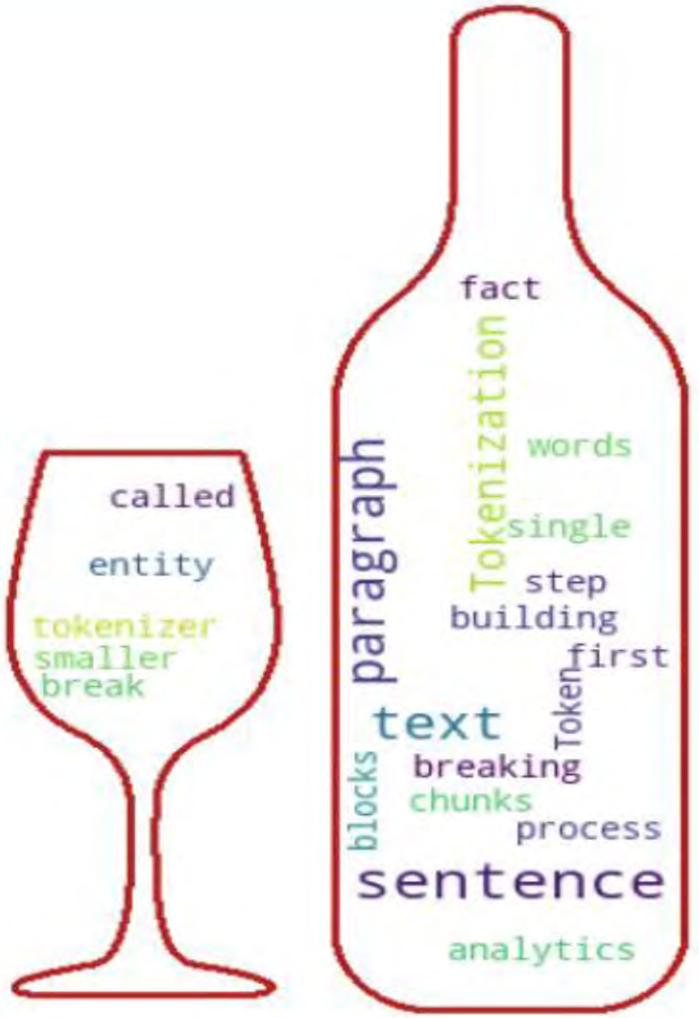
text = """Tokenization is the first step in text analytics. The process of breaking down a text paragraph into smaller chunks such as words or sentence is called Tokenization. Token is a single entity that is building blocks for sentence or paragraph. Does sentence tokenizer break text paragraph into sentences? What is fact?"""



Word cloud


entity
step
breaking
words
first
text
called
process
fact
Tokenization
smaller
chunks
sentence
blocks
paragraph
single
analytics
Token
building

Word cloud


called
entity
tokenizer
smaller
break
fact
Tokenization
words
paragraph
single
step
building
first
Token
text
breaking
blocks
chunks
process
sentence
analytics

Thank You
???

- ***References:***

1) Foundations of Statistical Natural Language Processing - Christopher D. Manning.
2) Text Data Management and Analysis - A Practical Introduction to Information Retrieval and Text Mining
3) Natural Language Processing using Python- Apress (2019)
4) Applied Text Analysis with Python O'Reilly Media (2018)
5) Natural Language Processing Python and NLTK-Packt Publishing (2016)
6) Building Machine Learning Systems with Python-Packt Publishing (2015)