Natural Language Processing(NLP)

Only 21% of the available data is present in structured form.

Majority of the data exists in the textual form, which is highly unstructured in nature.

Ex: Tweets, posts on social media, user to user chat conversations, news, blogs and articles., patient records in healthcare sector.

A few more recent ones includes chatbots and other voice driven bots.

Natural language processing unearths and uses the power of unstructured text.

NLP is a branch of data science that consists of systematic processes for analyzing, understanding, and deriving information from the text data in a smart and efficient manner.

NLP and its components can organize the massive chunks of text data, perform numerous automated tasks and solve a wide range of problems such as – automatic summarization, machine translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation etc.

1. *Tokenization* – process of converting a text into tokens
2. *Tokens* – words or entities present in the text
3. *Text object* – a sentence or a phrase or a word or an article

Since, text is the most unstructured form of all the available data, various types of noise are present in it and the data is not readily analyzable without any pre-processing.

The entire process of cleaning and standardization of text, making it noise-free and ready for analysis is known as text preprocessing.

Any piece of text which is not relevant to the context of the data and the end-output can be specified as the noise.

For example – *language stopwords* (commonly used words of a language – is, am, the, of, in etc), URLs or links, social media entities (mentions, hashtags), punctuations and industry specific words.

A general approach for noise removal is to prepare a dictionary of noisy entities, and iterate the text object by tokens (or by words), eliminating those tokens which are present in the noise dictionary.

Another approach is to use the regular expressions while dealing with special patterns of noise.

Another type of textual noise is about the multiple representations exhibited by single word.

Ex: "play", "player", "played", "plays" and "playing" are the different variations of the word – "play",

The words mean different but contextually all are similar. The step converts all the disparities of a word into their normalized form (also known as lemma).

Important Libraries for NLP

Scikit-learn: Machine learning in Python

Natural Language Toolkit (NLTK): The complete toolkit for all NLP techniques.

Pattern – A web mining module with tools for NLP and machine learning.

TextBlob – Easy to use nlp tools API, built on top of NLTK and Pattern.

spaCy – Industrial strength N LP with Python and Cython.

Gensim – Topic Modelling for Humans

Stanford Core NLP – NLP services and packages by Stanford NLP Group.

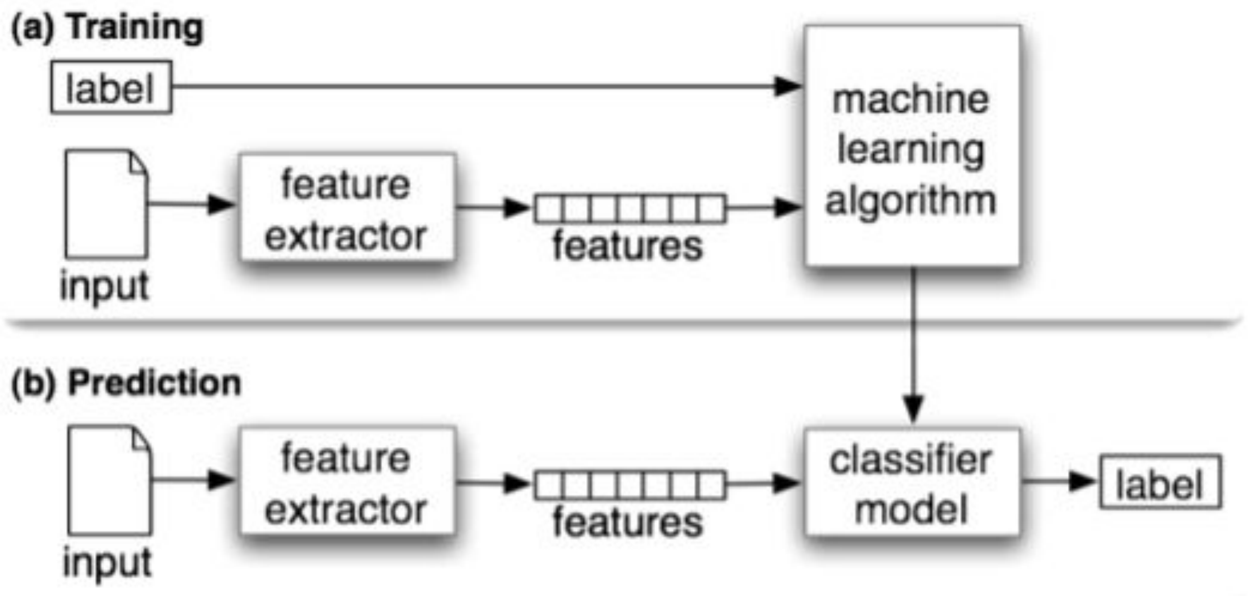
Text Classification

Text classification is one of the classical problem of NLP.

Ex: Email Spam Identification, topic classification of news, sentiment classification and organization of web pages by search engines.

Text classification, in common words is defined as a technique to systematically classify a text object (document or sentence) in one of the fixed category. It is really helpful when the amount of data is too large, especially for organizing, information filtering, and storage purposes.

A typical natural language classifier consists of two parts: (a) Training (b) Prediction as shown in image below. Firstly the text input is processed and features are created. The machine learning models then learn these features and is used for predicting against the new text.



Tokenization

Tokenizing is the process of breaking a large set of texts into smaller meaningful chunks such as sentences, words, phrases.

NLTK library provides *sent_tokenize* for sentence level tokenizing, which uses a pre-trained model PunktSentenceTokenize, to determine punctuation and characters marking the end of sentence for European languages.

Ex:

```
%matplotlib inline

import nltk

from nltk.tokenize import sent_tokenize

text='Statistics skills, and programming skills are equally important for analytics. Statistics

# sent_tokenize uses an instance of PunktSentenceTokenizer from the nltk. tokenize.punkt

sent_tokenize_list = sent_tokenize(text)

print(sent_tokenize_list)
```

Word Tokenize

word_tokenize is a wrapper function that calls tokenize by the TreebankWord- Tokenizer

```
from nltk.tokenize import word_tokenize

print word_tokenize(text)

# Another equivalent call method

from nltk.tokenize import TreebankWordTokenizer

tokenizer = TreebankWordTokenizer()

print tokenizer.tokenize(text)
```

Stemming

Stemming: Stemming is a rudimentary rule-based process of stripping the suffixes (“ing”, “ly”, “es”, “s” etc) from a word.

Lemmatization: Lemmatization, on the other hand, is an organized & step by step procedure of obtaining the root form of the word, it makes use of vocabulary (dictionary importance of words) and morphological analysis (word structure and grammar relations).

Ex:

from nltk.stem.wordnet import WordNetLemmatizer

lem = WordNetLemmatizer()

from nltk.stem.porter import PorterStemmer

stem = PorterStemmer()

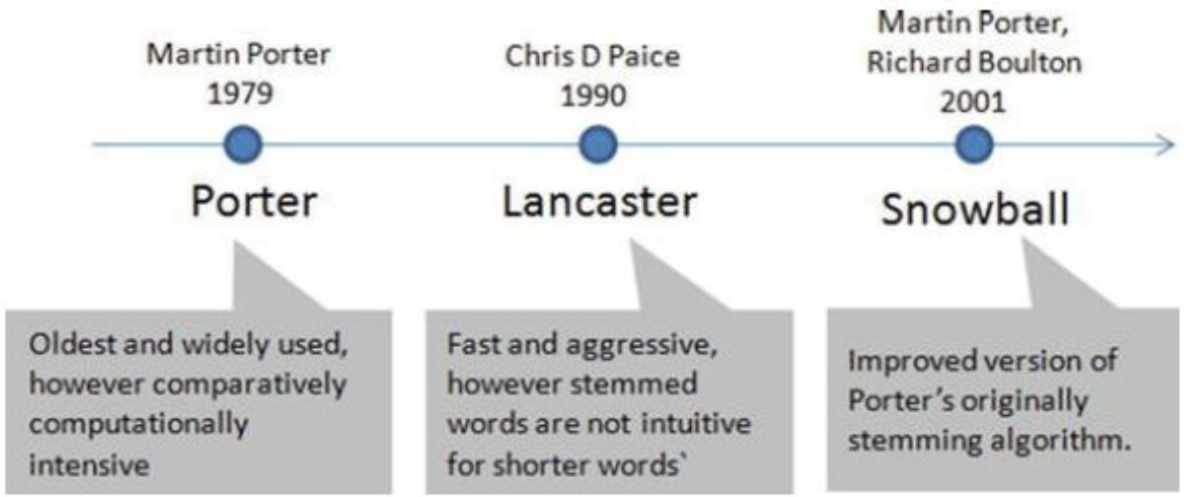
word = "multiplying"

lem.lemmatize(word, "v")

"multiply"

stem.stem(word)

"multipli"



Remove Stopwords and Punctuation

Remove stopwords

from nltk.corpus import stopwords

 # Function to remove stop words

def remove_stopwords(text, lang='english'):

words = nltk.word_tokenize(text)

lang_stopwords = stopwords.words(lang)

stopwords_removed = [w for w in words if w.lower() not in lang_stopwords]

return " ".join(stopwords_removed)

print remove_stopwords('This is a sample English sentence')

Remove punctuations

import string

 # Function to remove punctuations

def remove_punctuations(text):

words = nltk.word_tokenize(text)

punt_removed = [w for w in words if w.lower() not in string.punctuation]

return " ".join(punt_removed)

print remove_punctuations('This is a sample English sentence, with punctuations!')

This is a sample English sentence with punctuations

Remove whitespaces & number , Stemming

```
import re
# Function to remove whitespace
def remove_whitespace(text):
return " ".join(text.split())
# Function to remove numbers
def remove_numbers(text):
return re.sub(r'\d+', '', text)
text = 'This is a sample English sentence, \n with whitespace and numbers print 'Original Text: ', text
print 'Removed whitespace: ', remove_whitespace(text)
print 'Removed numbers: ', remove_numbers(text)
```

Stemming

It is the process of transforming to the root word i.e., it uses an algorithm that removes common word endings for English words, such as “ly”, “es”, “ed” and “s”.

Ex:

Assuming an analysis you may want to consider “carefully”, “cared”, “cares”, “caringly” as “care” instead of

separate words.

```
from IPython.display import Image
Image(filename='../Chapter 5 Figures/Stemmers.png', width=500)
from nltk import PorterStemmer, LancasterStemmer, SnowballStemmer
# Function to apply stemming to a list of words
def words_stemmer(words, type="PorterStemmer", lang="english", encoding="utf8"):
supported_stemmers = ["PorterStemmer","LancasterStemmer","SnowballStemmer"]
if type is False or type not in supported_stemmers:
return words
else:
```

```
stem_words = []
if type == "PorterStemmer":
stemmer = PorterStemmer()
for word in words:
stem_words.append(stemmer.stem(word).encode(encoding))
if type == "LancasterStemmer":
stemmer = LancasterStemmer()
for word in words:
stem_words.append(stemmer.stem(word).encode(encoding))
if type == "SnowballStemmer":
stemmer = SnowballStemmer(lang)
for word in words:
stem_words.append(stemmer.stem(word).encode(encoding))
return " ".join(stem_words)
```

```
words = 'caring cares cared caringly carefully'
print "Original: ", words
print "Porter: ", words_stemmer(nltk.word_tokenize(words), "PorterStemmer")
print "Lancaster: ", words_stemmer(nltk.word_tokenize(words), "LancasterStemmer")
print "Snowball: ", words_stemmer(nltk.word_tokenize(words), "SnowballStemmer")
```

Lemmatizer

It is the process of transforming to the dictionary base form.

```
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()
# Function to apply lemmatization to a list of words
def words_lemmatizer(text, encoding="utf8"):
words = nltk.word_tokenize(text)
lemma_words = []
wl = WordNetLemmatizer()
for word in words:
pos = find_pos(word)
lemma_words.append(wl.lemmatize(word, pos).encode(encoding))
return " ".join(lemma_words)
# Function to find part of speech tag for a word
def find_pos(word):
# Part of Speech constants
# ADJ, ADJ_SAT, ADV, NOUN, VERB = 'a', 's', 'r', 'n', 'v'
# You can learn more about these at http://wordnet.princeton.edu/wordnet/man/wndb.# You can learn more about all the penn tree tags at https://www.ling.upenn.edu/courses/pos = nltk.pos_tag(nltk.word_tokenize(word))[0][1]
# Adjective tags - 'JJ', 'JJR', 'JJS'
if pos.lower()[0] == 'j':
return 'a'
# Adverb tags - 'RB', 'RBR', 'RBS'
elif pos.lower()[0] == 'r':
return 'r'
# Verb tags - 'VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ'
elif pos.lower()[0] == 'v':
return 'v'
# Noun tags - 'NN', 'NNS', 'NNP', 'NNPS'
else:
return 'n'
print "Lemmatized: ", words_lemmatizer(words)
```

N-grams

One of the important concepts in text mining is n-grams, which are fundamentally a set of co-occurring or continuous sequence of n items from a given sequence of large text. The item here could be words, letters, and syllables.

The N-gram technique is relatively simple and simply increasing the value of n will give us more contexts. It is widely used in the probabilistic language model of predicting the next item in a sequence: for example, search engines use this

technique to predict/recommend the possibility of next character/words in the sequence to users as they type.

N-grams

```
from nltk.util import ngrams
from collections import Counter
# Function to extract n-grams from text
def get_ngrams(text, n):
n_grams = ngrams(nltk.word_tokenize(text), n)
return [ ' '.join(grams) for grams in n_grams]
text = 'This is a sample English sentence'
print "1-gram: ", get_ngrams(text, 1)
print "2-gram: ", get_ngrams(text, 2)
print "3-gram: ", get_ngrams(text, 3)
print "4-gram: ", get_ngrams(text, 4)
```

Extract bigram and count their respective frequency

```
text = 'Statistics skills, and programming skills are equally important for analytics. # remove punctuations
text = remove_punctuations(text)
# Extracting bigrams
result = get_ngrams(text,2)
# Counting bigrams
result_count = Counter(result)
print "Words: ", result_count.keys() # Bigrams
print "\nFrequency: ", result_count.values() # Bigram frequency
# Converting to the result to a data frame
import pandas as pd
df = pd.DataFrame.from_dict(result_count, orient='index')
df = df.rename(columns={'index':'words', 0:'frequency'}) # Renaming index and column name
print df
```

Parts of Speech

PoS tagging is the process of assigning language-specific parts of speech such as nouns, verbs, adjectives, and adverbs, etc., for each word in the given text.

NLTK supports multiple PoS tagging models, and the default tagger is maxent_treebank_pos_tagger, which uses Penn (Pennsylvania University) Tree bank corpus.

A sentence (S) is represented by the parser as a tree having three children: a noun phrase (NP), a verbal phrase (VP), and the full stop (.). The root of the tree will be S.

Bag of Words(BoW)

The texts have to be represented as numbers to be able to apply any algorithms.

Bag of words is the method where you count the occurrence of words in a document without giving importance to the grammar and the order of words.

This can be achieved by creating Term Document Matrix (TDM) .

It is simply a matrix with terms as the rows and document names as the columns and a count of the frequency of words as the cells of the matrix.

Ex:

Consider the following three document.

Index (terms)	Doc_1	Doc_2	Doc_3
analytics	1	1	0
and	1	1	1
are	1	1	0
books	0	0	1
domain	0	1	0
equally	1	0	0
for	1	1	0
important	1	1	0
knowledge	0	1	0
like	0	0	1
programming	1	0	0
reading	0	0	1
skills	2	1	0
statistics	1	1	0
travelling	0	0	1

Example

```
import os
import pandas as pd
from sklearn.feature_extraction.text import
CountVectorizer
# Function to create a dictionary with key as file
names and values as text for all files in a given
folder
def CorpusFromDir(dir_path):
result = dict(docs =
[open(os.path.join(dir_path,f)).read() for f in
os.listdir(dir_path)],
ColNames = map(lambda x: x,
os.listdir(dir_path)))
return result
docs = CorpusFromDir('Data/')
# Initialize
vectorizer = CountVectorizer()
doc_vec =
vectorizer.fit_transform(docs.get('docs'))
#create dataFrame
df = pd.DataFrame(doc_vec.toarray().transpose(),
index = vectorizer.get_feature_names())
# Change column headers to be file names
df.columns = docs.get('ColNames')
print df
```

Example

```
import os

import pandas as pd

from sklearn.feature_extraction.text import CountVectorizer

# Create a dictionary with key as file names and values as text for all
files in a given def CorpusFromDir(dir_path):

result = dict(docs = [open(os.path.join(dir_path,f)).read() for f in
os.listdir(dir_ColNames = map(lambda x: x, os.listdir(dir_path)))

return result

docs = CorpusFromDir('Data/text_files/')

# Initialize

vectorizer = CountVectorizer()

doc_vec = vectorizer.fit_transform(docs.get('docs'))

#create dataFrame

df = pd.DataFrame(doc_vec.toarray().transpose(), index =
vectorizer.get_feature_names())

# Change column headers to be file names

df.columns = docs.get('ColNames')

print df
```

Term Frequency-Inverse Document Frequency (TF-IDF)

In the area of information retrieval TF-IDF is a good statistical measure to reflect the relevance of term to the document in a collection of documents or corpus.

Term frequency will tell you how frequently a given term appears.

$$\text{TF (term)} = \frac{Number\ of\ times\ term\ appears\ in\ a\ document}{Total\ number\ of\ terms\ in\ the\ document}$$

For example, consider a document containing 100 words wherein the word, ‘ML’ appears 3 times, then

TF (ML) = 3 / 100 =0.03

Document frequency will tell you how important a term is?

$$\text{DF (term)} = \frac{d\ (number\ of\ documents\ containing\ a\ given\ term)}{D\ (the\ size\ of\ the\ collection\ of\ documents)}$$

Ex:

Assume we have 10 million documents and the word ML appears in one thousand of these, then

DF (ML) = 1000/10,000,000 = 0.0001

To normalize let’s take a log (d/D), that is, log (0.0001) = -4

Quite often D > d and log (d/D) will give a negative value as seen in the above example. So to solve this problem let’s invert the ratio inside the log expression, which is known as Inverse document frequency (IDF). Essentially we are compressing the scale of values so that very large or very small quantities are smoothly compared.

$$\text{IDF (term)} = \log\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ with\ a\ given\ term\ in\ it}\right)$$

IDF(ML) = log(10,000,000 / 1,000) = 4

TF-IDF is the weight product of quantities, that is, for the above example

TF-IDF (ML) = 0.03 * 4 = 0.12