NLP Core using NLTK

Dr. Muhammad Nouman Durrani

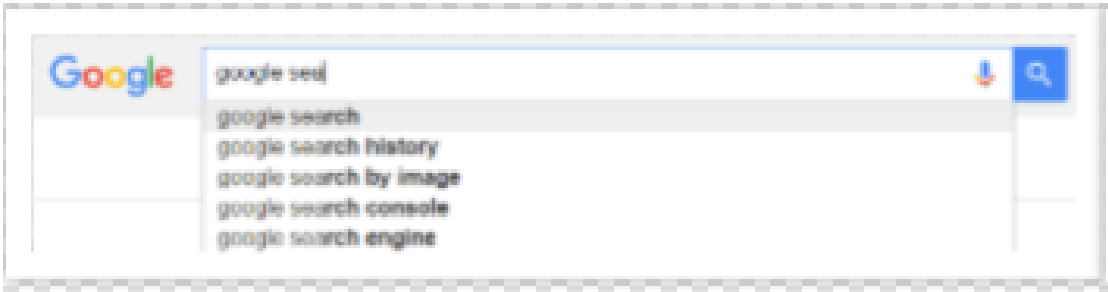
NLTK

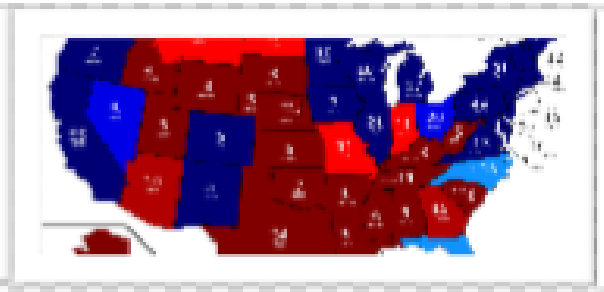
- NLTK is a leading platform for building Python programs to work with human language data.
- It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for:
  - classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries

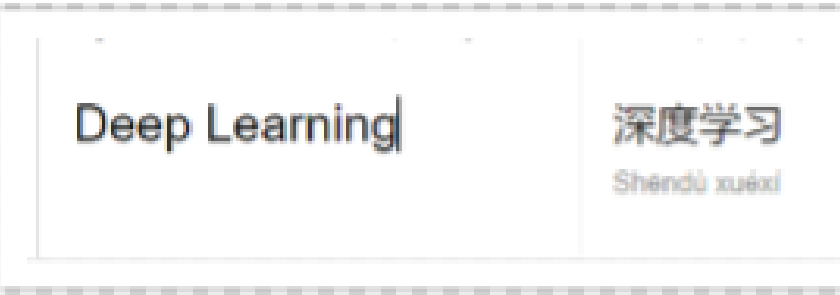
Introduction to Text Processing: Extracting, transforming and selecting features

Consider the following examples:

- Google search
- 2008 U.S Presidential Elections.
- google translate







Introduction to Text Processing: Extracting, transforming and selecting features

- So what do the above examples have in common?
- **TEXT processing**. All the above three scenarios deal with massive amount of text to perform tasks
- Humans deal with text format quite intuitively

Introduction to Text Processing

- A computer can match two strings and tell you whether they are same or not.
- But how do we make computers tell you about football or Ronaldo when you search for Messi?

- Word Embeddings

Tokenization

- The process of breaking down a text paragraph into smaller chunks such as words or sentences is called Tokenization
- Token is a single entity that is building blocks for sentence or paragraph

Sentence Tokenization

- Sentence tokenizer breaks text paragraph into sentences

```
from nltk.tokenize import sent_tokenize
text="""Hello Mr. Smith, how are you doing today? The weather is great, and city is awesome.
The sky is pinkish-blue. You shouldn't eat cardboard"""
tokenized_text=sent_tokenize(text)
print(tokenized_text)
```

```
['Hello Mr. Smith, how are you doing today?', 'The weather is great, and city is awesome.', 'The sky is pinkish-blue.', "You shouldn't eat cardboard"]
```

Tokenization

Word Tokenization

- Word tokenizer breaks text paragraph into words

```
from nltk.tokenize import word_tokenize
tokenized_word=word_tokenize(text)
print(tokenized_word)
```

```
['Hello', 'Mr.', 'Smith', ',', 'how', 'are', 'you', 'doing', 'today', '?', 'The', 'weather', 'is', 'great', ',',
'and', 'city', 'is', 'awesome', '.', 'The', 'sky', 'is', 'pinkish-blue', '.', 'You', 'should', "n't", 'eat',
'cardboard']
```

Tokenization

- **Frequency Distribution**

```
from nltk.probability import FreqDist
fdist = FreqDist(tokenized_word)
print(fdist)
```

```
<FreqDist with 25 samples and 30 outcomes>
```

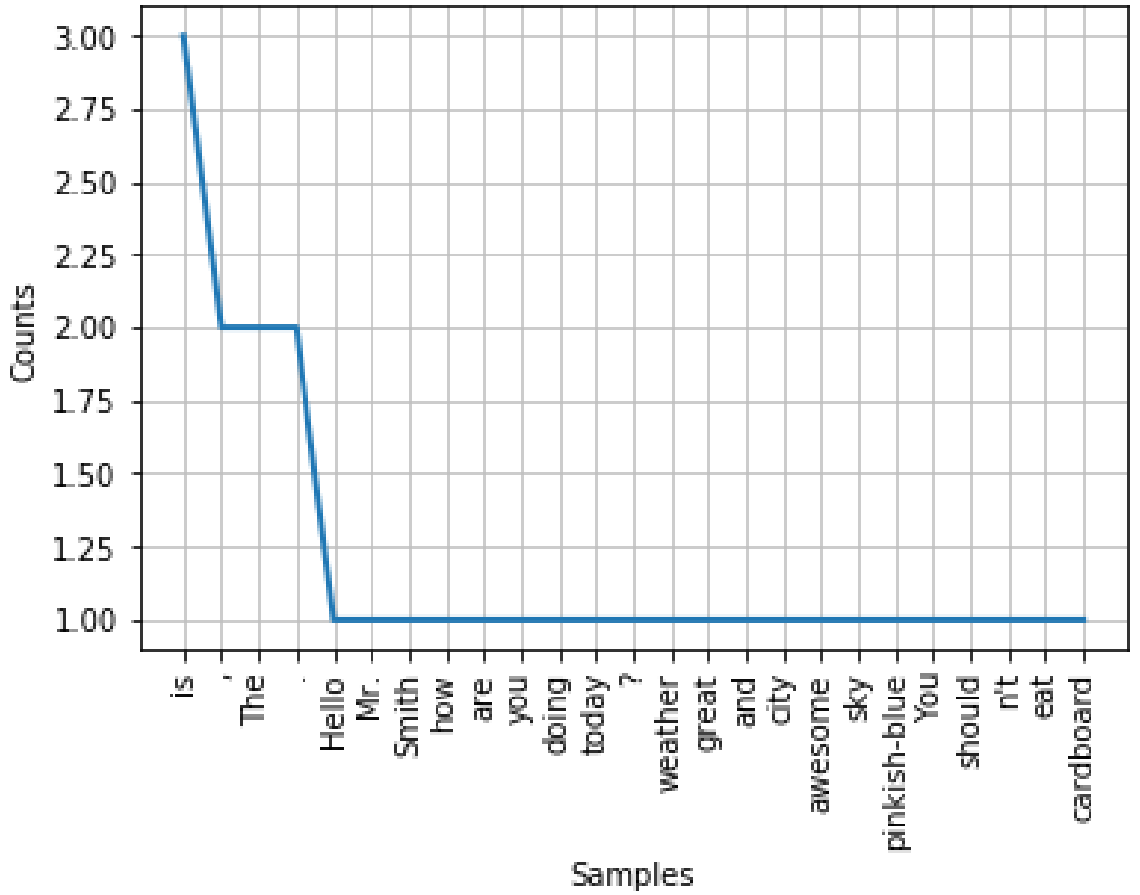
```
fdist.most_common(2)

[('is', 3), (',', 2)]
```

Tokenization

- Frequency Distribution Plot

```
import matplotlib.pyplot as plt
fdist.plot(30,cumulative=False)
plt.show()
```



Tokenize Non-English Languages Text

- To tokenize other languages, you can specify the language like this:

```
from nltk.tokenize import sent_tokenize

mytext = "Bonjour M. Adam, comment allez-vous? J'espère que tout va bien. Aujourd'hui est un bon jour."
print(sent_tokenize(mytext, "french"))
```

The result will be like this:

```
['Bonjour M. Adam, comment allez-vous?', "J'espère que tout va bien.", "Aujourd'hui est un bon jour."]
```

Stopwords

- Stopwords considered as noise in the text. Text may contain stop words such as is, am, are, this, a, an, the, etc.

- In NLTK for removing stopwords, you need to create a list of stopwords and filter out your list of tokens from these words

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

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

Stopwords

```
filtered_sent=[]
for w in tokenized_sent:
    if w not in stop_words:
        filtered_sent.append(w)
print("Tokenized Sentence:",tokenized_sent)
print("Filterd Sentence:",filtered_sent)
```

```
Tokenized Sentence: ['Hello', 'Mr.', 'Smith', ',', 'how', 'are', 'you', 'doing', 'today', '?']
Filterd Sentence: ['Hello', 'Mr.', 'Smith', ',', 'today', '?']
```

Get Synonyms From WordNet

- WordNet is a database built for natural language processing
- It includes groups of synonyms and a brief definition

```
from nltk.corpus import wordnet
syn = wordnet.synsets("pain")
print(syn[0].definition())
print(syn[0].examples())
```

```
a symptom of some physical hurt or disorder

['the patient developed severe pain and distension']
```

Get Synonyms From WordNet

- You can use WordNet to get synonymous words like this:

```
from nltk.corpus import wordnet
synonyms = []
for syn in wordnet.synsets('Computer'):
for lemma in syn.lemmas():
synonyms.append(lemma.name())
print(synonyms)
```

The output is:

['computer', 'computing_machine', 'computing_device', 'data_processor', 'electronic_computer', 'information_processing_system', 'calculator', 'reckoner', 'figurer', 'estimator', 'computer']

Get Antonyms From WordNet

- You can get the antonyms of words the same way
- Use the lemmas before adding them to the array
- it's an antonym or not

```
from nltk.corpus import wordnet
antonyms = []
for syn in wordnet.synsets("small"):
    for l in syn.lemmas():
        if l.antonyms():
            antonyms.append(l.antonyms()[0].name())
print(antonyms)
```

```
['large', 'big', 'big']
```

NLTK Word Stemming

- Word stemming means removing affixes from words and returning the root word. (The stem of the word working is work.)
- Search engines use this technique when indexing pages, so many people write different versions for the same word and all of them are stemmed to the root word
- NLTK has a class called PorterStemmer that uses this algorithm.

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
print(stemmer.stem('working'))
```

The result is: work.

Lemmatizing Words Using WordNet

- Word lemmatizing is similar to stemming, but the difference is the result of lemmatizing is a real word

When we stem some words, it will result as follows:

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
print(stemmer.stem('increases'))
```

The result is: increas.

When we lemmatize the same word using NLTK WordNet, the result is increase:

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize('increases'))
```

The result is increase.

Lemmatizing Words Using WordNet

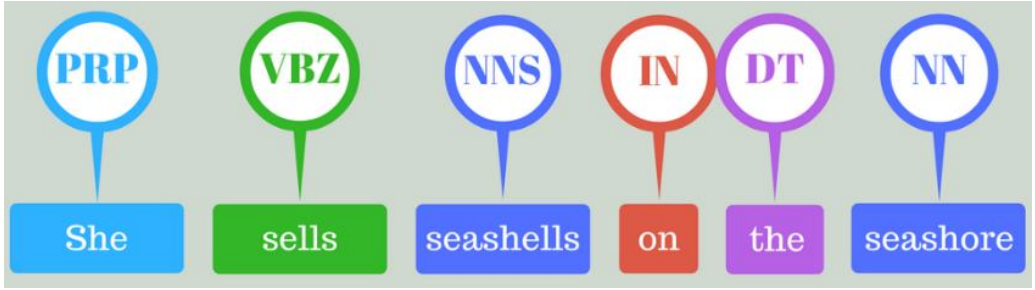
- If we try to lemmatize a word like “playing”, it will end up with the same word
  - This is because the default part of speech is nouns
  - To get verbs, adjective, or adverb, we should specify it (See Example)
  - Actually, this is a very good level of text compression.
  - We end up with about 50% to 60% compression

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize('playing', pos="v"))
print(lemmatizer.lemmatize('playing', pos="n"))
print(lemmatizer.lemmatize('playing', pos="a"))
print(lemmatizer.lemmatize('playing', pos="r"))
```

The result is:

```
play
playing
playing
playing
```

Part of speech tagging (POS)



- Part-of-speech tagging is used to assign parts of speech to each word of a given text (such as nouns, verbs, pronouns, adverb, conjunction, adjectives, interjection) based on its definition and its context.

```
text = “vote to choose a particular man or a group (party) to represent them in parliament”
tex = word_tokenize(text)  #Tokenize the text
for token in tex:
 print(nltk.pos_tag([token]))
```

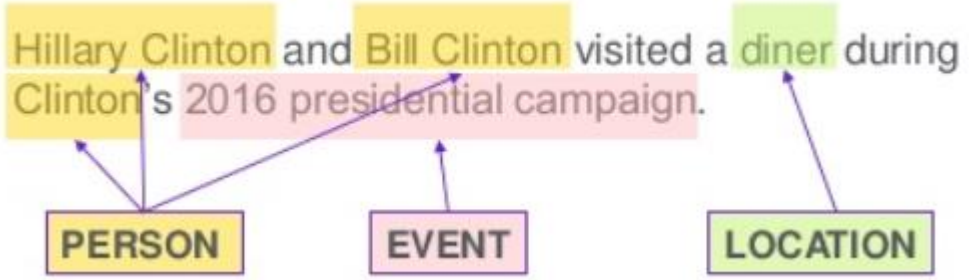
```
[('vote', 'NN')]
[('to', 'TO')]
[('choose', 'NN')]
[('a', 'DT')]
[('particular', 'JJ')]
[('man', 'NN')]
[('or', 'CC')]
[('a', 'DT')]
[('group', 'NN')]
[('(', '(')]
[('party', 'NN')]
[(')', ')')]
[('to', 'TO')]
[('represent', 'NN')]
[('them', 'PRP')]
[('in', 'IN')]
[('parliament', 'NN')]
```

POS : Tags and Descriptions

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Tag	Description
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBR	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Whdeterminer
WP	Whpronoun
WP$	Possessive whpronoun
WRB	Whadverb

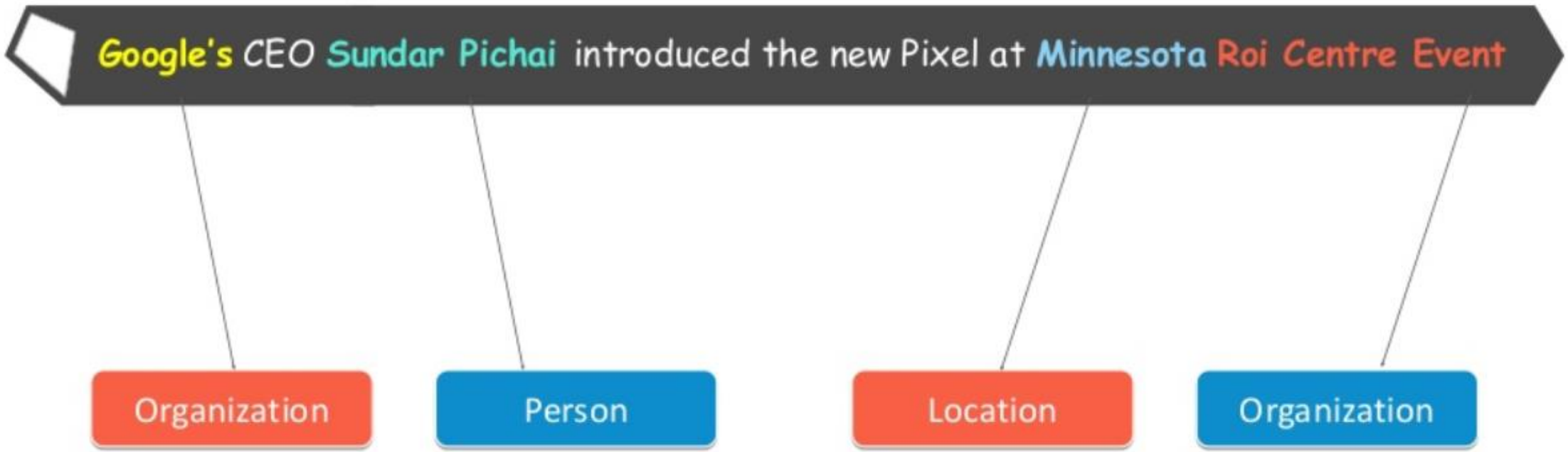
Named entity recognition



- It is the process of detecting the named entities such as the person name, the location name, the company name, the quantities and the monetary value.

```
text = "Google's CEO Sundar Pichai introduced the new Pixel at Minnesota Roi Centre Event"          #importing chunk library from nltk
from nltk import ne_chunk          # tokenize and POS Tagging before doing chunk
token = word_tokenize(text)
tags = nltk.pos_tag(token)
chunk = ne_chunk(tags)
chunk
```

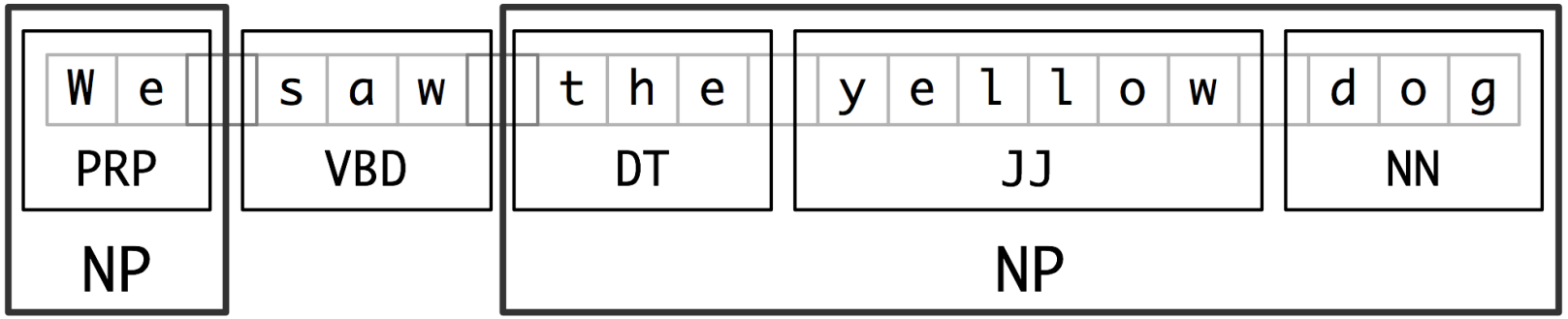
NER : Named Entity Recognition



POS Tagging Output:

Tree('S', [Tree('GPE', [('Google', 'NNP')]), ("'s", 'POS'), Tree('ORGANIZATION', [('CEO', 'NNP'), ('Sundar', 'NNP'), ('Pichai', 'NNP')]), ('introduced', 'VBD'), ('the', 'DT'), ('new', 'JJ'), ('Pixel', 'NNP'), ('at', 'IN'), Tree('ORGANIZATION', [('Minnesota', 'NNP'), ('Roi', 'NNP'), ('Centre', 'NNP')]), ('Event', 'NNP')])

Chunking



- Chunking means picking up individual pieces of information and grouping them into bigger pieces.
- In the context of NLP and text mining, chunking means grouping of words or tokens into chunks.

```
text = “We saw the yellow dog”
token = word_tokenize(text)
tags = nltk.pos_tag(token)
reg = “NP: {<DT>?<JJ>*<NN>}”
a = nltk.RegexpParser(reg)
result = a.parse(tags)
print(result)
```

(S We/PRP saw/VBD (NP the/DT yellow/JJ dog/NN))

What are Word Embeddings?

- **Word Embeddings** are the texts converted into numbers
  - There may be different numerical representations of the same text.

Why do we need **Word Embeddings**?

- Many Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing *strings* or *plain text* in their raw form.

- A Word Embedding format generally tries to map a word using a dictionary to a vector.

What are Word Embeddings?

- **Word Embeddings** are the texts converted into numbers
  - There may be different numerical representations of the same text.

Why do we need **Word Embeddings**?

- Many Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing *strings* or *plain text* in their raw form.
- With huge amount of data that is present in the text format, it is imperative to extract knowledge out of it and build applications.
- They require numbers as inputs to perform any sort of job, be it classification, regression etc. in broad terms.
- Some real world applications of text applications are – sentiment analysis of reviews by Amazon etc., document or news classification or clustering by Google etc.
- A Word Embedding format generally tries to map a word using a dictionary to a vector.

What are Word Embeddings?

Take a look at this example – **sentence=**” **Word Embeddings are Word converted into numbers** ”

- A ***word*** in this **sentence** may be “Embeddings” or “numbers ” etc.
- A ***dictionary*** may be the list of all unique words in the **sentence.**
  - So, a dictionary may look like – [‘Word’,’Embeddings’,’are’,’Converted’,’into’,’numbers’]
- A **vector** representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else.
- The vector representation of “numbers” in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

One-hot encoding (CountVectorizing)

- The most basic and naive method for transforming words into vectors is to count occurrence of each word in each document. Such an approach is called countvectorizing or one-hot encoding.
  - The idea is to collect a set of documents (they can be words, sentences, paragraphs or even articles) and count the occurrence of every word in them.
  - The columns of the resulting matrix are words and the rows are documents.

Example I

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = [
'This is the first document.',
'This document is the second document.',
'And this is the third one.',
'Is this the first document?',
]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
print(X.toarray())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

Example I

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = [
'This is the first document.',
'This document is the second document.',
'And this is the third one.',
'Is this the first document?',
]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
print(X.toarray())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

Example II

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
```

```
sample_text = ["One of the most basic ways we can numerically represent words "
               "is through the one-hot encoding method (also sometimes called "
               "count vectorizing)."]
```

```
# To actually create the vectorizer, we simply need to call fit on the text
# data that we wish to fix
vectorizer.fit(sample_text)
# Now, we can inspect how our vectorizer vectorized the text
# This will print out a list of words used, and their index in the vectors
print('Vocabulary: ')
print(vectorizer.vocabulary_)
```

```
Vocabulary:
{'one': 12, 'of': 11, 'the': 15, 'most': 9, 'basic': 1, 'ways': 18, 'we': 19, 'can': 3, 'numericall
y': 10, 'represent': 13, 'words': 20, 'is': 7, 'through': 16, 'hot': 6, 'encoding': 5, 'method': 8,
'also': 0, 'sometimes': 14, 'called': 2, 'count': 4, 'vectorizing': 17}
```

Example II

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
sample_text = ["One of the most basic ways we can numerically represent words "
               "is through the one-hot encoding method (also sometimes called "
               "count vectorizing)."]
# To actually create the vectorizer, we simply need to call fit on the text
# data that we wish to fix
vectorizer.fit(sample_text)
# Now, we can inspect how our vectorizer vectorized the text
# This will print out a list of words used, and their index in the vectors
print('Vocabulary: ')
print(vectorizer.vocabulary_)
```

The converse mapping from feature name to column index is stored in the vocabulary_ attribute of the vectorizer:

```
Vocabulary:
{'one': 12, 'of': 11, 'the': 15, 'most': 9, 'basic': 1, 'ways': 18, 'we': 19, 'can': 3, 'numericall
y': 10, 'represent': 13, 'words': 20, 'is': 7, 'through': 16, 'hot': 6, 'encoding': 5, 'method': 8,
'also': 0, 'sometimes': 14, 'called': 2, 'count': 4, 'vectorizing': 17}
```

```python
# If we would like to actually create a vector, we can do so by passing the
# text into the vectorizer to get back counts
vector = vectorizer.transform(sample_text)

# Our final vector:
print('Full vector: ')
print(vector.toarray())

# Or if we wanted to get the vector for one word
print('Hot vector: ')
print(vectorizer.transform(['hot']).toarray())

# Or if we wanted to get multiple vectors at once to build matrices
print('Hot, one and Today: ')
print(vectorizer.transform(['hot', 'one', 'of']).toarray())

# We could also do the whole thing at once with the fit_transform method:
print('One swoop:')
new_text = ['Today is the day that I do the thing today, today']
new_vectorizer = CountVectorizer()
print(new_vectorizer.fit_transform(new_text).toarray())
```

```
Full vector:
[[1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1]]
Hot vector:
[[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
Hot, one and Today:
[[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]]
One swoop:
[[1 1 1 1 2 1 3]]
```

Example II

```
# If we would like to actually create a vector, we can do so by passing the
# text into the vectorizer to get back counts
vector = vectorizer.transform(sample_text)

# Our final vector:
print('Full vector: ')
print(vector.toarray())

# Or if we wanted to get the vector for one word:
print('Hot vector: ')
print(vectorizer.transform(['hot']).toarray())

# Or if we wanted to get multiple vectors at once to build matrices
print('Hot, one and Today: ')
print(vectorizer.transform(['hot', 'one', 'of']).toarray())
```

```
Full vector:
[[1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1]]
Hot vector:
[[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
Hot, one and Today:
[[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]]
```

Example II

```
# We could also do the whole thing at once with the fit_transform method:
print('One swoop:')
new_text = ['Today is the day that I do the thing today, today']
new_vectorizer = CountVectorizer()
print(new_vectorizer.fit_transform(new_text).toarray())
```

```
One swoop:
[[1 1 1 1 2 1 3]]
```

Word Frequencies with TfidfVectorizer

- Word counts are a good starting point, but are very basic
  - One issue with simple counts is that some words like “the” or “many other non-stop words” will appear many times and their large counts will not be very meaningful in the encoded vectors
- TF-IDF feature numerical representations where words are represented by their term frequency multiplied by their inverse document frequency
  - **Term Frequency:** This summarizes how often a given word appears within a document

    The number of times a word appears in a document divided by the total number of words in the document. Every document has its own term frequency

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

$$tf_{ij} = \text{number of occurrences of } i \text{ in } j$$

Word Frequencies with TfidfVectorizer

– Inverse Document Frequency: The log of the number of documents divided by the number of documents that contain the word w

Inverse document frequency determines the weight of rare words across all documents in the corpus

This downscales words that appear a lot across documents

$$idf(w) = log(\frac{N}{df_t})$$

df_i = number of documents containing i

N = total number of documents

- Combining these two we come up with the TF-IDF score (w) for a word in a document in the corpus. It is the product of tf and idf:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

- TF-IDF are word frequency scores that try to highlight words that are more interesting, e.g. frequent in a document but not across documents

Word Frequencies with TfidfVectorizer

- Let's take an example to get a clearer understanding.

  Sentence 1 : The car is driven on the road.

  Sentence 2: The truck is driven on the highway.

- In this example, each sentence is a separate document.
- Calculate the TF-IDF for the above two documents, which represent our corpus.

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

```python
from sklearn.feature_extraction.text import TfidfVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog.",
        "The dog.",
        "The fox"]
# create the transform
vectorizer = TfidfVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
# summarize
print(vectorizer.vocabulary_)
print(vectorizer.idf_)
# encode document
vector = vectorizer.transform([text[0]])
# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

With **Tfidftransformer** you will systematically compute word counts using **CountVectorizer** and then compute the **Inverse Document Frequency** (IDF) values and only then compute the Tf-idf scores.

With **Tfidfvectorizer** on the contrary, you will do all three steps at once. Under the hood, it (i) tokenize documents and computes the word counts, (ii) learn the vocabulary and inverse document frequency weightings IDF values, and (iii) Tf-idf scores all using the same dataset.

```
{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}
[1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
 1.69314718 1.        ]
(1, 8)
[[0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646
  0.36388646 0.42983441]]
```

Word Frequencies with TfidfVectorizer

```
from sklearn.feature_extraction.text import TfidfVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog.",
        "The dog.",
        "The fox"]
# create the transform
vectorizer = TfidfVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
```

```
# summarize
print(vectorizer.vocabulary_)
print(vectorizer.idf_)
# encode document
vector = vectorizer.transform([text[0]])
# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

```
{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}
[1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
 1.69314718 1.        ]
(1, 8)
[[0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646
  0.36388646 0.42983441]]
```

How exactly does TF-IDF work?

Document 1

Term	Count
This	1
is	1
about	2
Messi	4

Document 2

Term	Count
This	1
is	2
about	1
Tf-idf	1

- Sample tables give the count of terms(tokens/words) in two documents.
- TF = (Number of times term t appears in a document)/(Number of terms in the document)
  - TF(This,Document1) = 1/8
  - TF(This, Document2)=1/5
- It denotes the contribution of the word to the document i.e words relevant to the document should be frequent.
- For Example: A document about Messi should contain the word ‘Messi’ in large number.

How exactly does TF-IDF work?

- IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

  IDF(This) = log(2/2) = 0.

- **How do we explain the reasoning behind IDF?** Ideally, if a word has appeared in all the document, then probably that word is not relevant to a particular document.
- But if it has appeared in a subset of documents then probably the word is of some relevance to the documents it is present in.

Let us compute IDF for the word 'Messi'.

IDF(Messi) = log(2/1) = 0.301.

Now, let us compare the TF-IDF for a common word 'This' and a word 'Messi'

TF-IDF(This,Document1) = (1/8) * (0) = 0 TF-IDF(This, Document2) = (1/5) * (0) = 0

TF-IDF(Messi, Document1) = (4/8)*0.301 = 0.15

- For Document1, TF-IDF method heavily penalizes the word 'This' but assigns greater weight to 'Messi'.
- So, 'Messi' is an important word for Document1 from the context of the entire corpus.