

The logo for Natural Language Processing (NLP) is displayed on a black square background. It consists of the letters 'N' and 'L' in a bold, yellow, sans-serif font, and the letters 'P' and 'T' in a bold, white, sans-serif font. The letters are arranged in a 2x2 grid: 'N' and 'L' on the left, and 'P' and 'T' on the right.

N
L P T

Natural Language Processing

Index

- Introduction
- Data Sources and Problem Types
- Stage of NLP
- Application of NLP
- Text Data Cleaning and Regular Expressions
- Phonetic Hashing and Spell Corrections
- Stopwords and Tokenization, NLTK
- Stemming and Lemmatization
- N-Grams
- Part of Speech Tagging
- Word Embedding - TD IDF, Word2Vec, etc
- Bag of Words
- Cosine Similarity
- Topic Modelling
- SpaCy and Stanford NLP

Introduction

NLP is an automated way to understand & analyze natural human language & extract useful information from such data by applying various machine learning algorithms

Applications

- Social Media analytics
- Bank loan processing
- Insurance claim processing
- Customer relationship processing
- Security and counter terrorism
- Medical transcriptions

Components:

1. NLU - Natural Language Understanding

Mapping an input in natural language into useful representations.

2. NLG - Natural Language Generation

- a. Text Planning
- b. Sentence Planning
- c. Text Realization

Data Sources and Problem Types

Well, below are few sources of data:

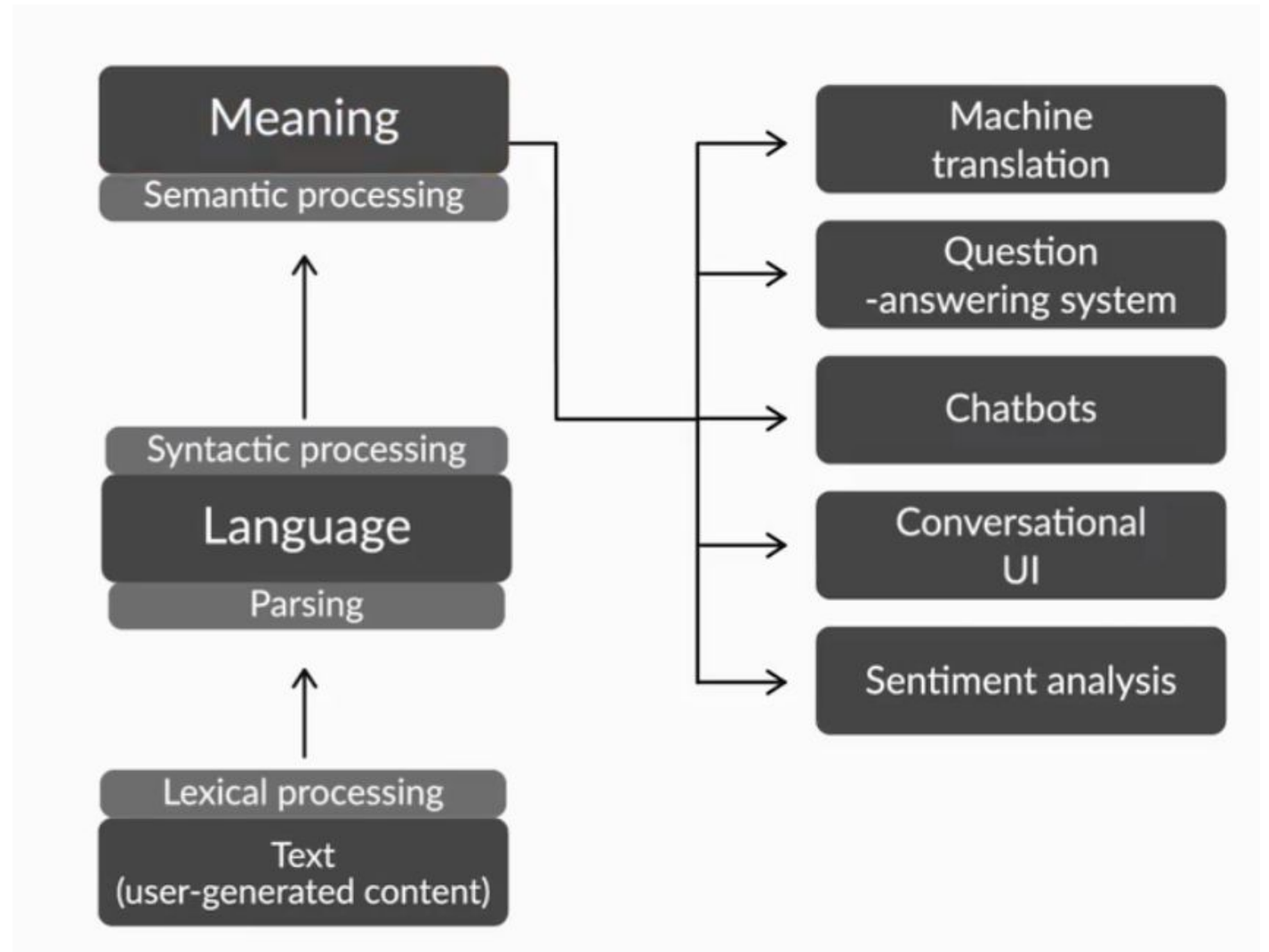
1. Linguistic Data Consortium (<https://www ldc.upenn.edu/>)
2. Web Crawling/Scraping
3. WordNet
4. <https://lionbridge.ai/datasets/the-best-25-datasets-for-natural-language-processing/>
5. API's: Twitter, Wordnik etc.
6. University sites & academic communities.

Below are few Classic NLP Problems:

1. Linguistically-motivated: Segmentation, Tagging, Parsing etc.
2. Analytical: Classification, Sentiment Analysis
3. Transformation: Translation, Correction, Generation
4. Conversation: Question Answering, Dialog etc.

Stages

- Text-Lexical processing(Tokenizations)
- Parsing(PoS tagging)
- Language
- Syntactic processing
- Semantic processing
- Meaning



Application

1. **Lexical processing:** In this stage, you will be required to do text preprocessing and text cleaning steps such as tokenization, stemming, lemmatization, correcting spellings, etc.
2. **Syntactic processing:** In this step, you will be required to extract more meaning from the sentence, by using its syntax this time. Instead of just blindly looking at what the words are, you'll look at the syntactic structures, i.e., the grammar of the language to understand what the meaning is.
3. **Semantic processing:** Lexical and syntactic processing do not suffice when it comes to building advanced NLP applications such as language translation, chatbots etc. The machine, after the two steps given above, will still be incapable of understanding the meaning of each word. Here, you will try and extract the hidden meaning behind the words which is also the most difficult part for computers. Semantic processing helps you to build advanced applications such as chatbots and language translators.

Cleaning

Cleaning data is the first task, which means splitting it into words and normalizing issues such as :

1. Upper & lower case characters
2. Spelling mistakes & regional variations
3. Unicode characters
4. Punctuation
5. Numbers such as amounts and dates.

Regular expressions

Regular expressions help you immensely in the text preprocessing stage. You looked at various characters that have special meaning in the regular expressions' engine. Here is a list of all the special characters and their meaning. Powerful programming tools that are used for a variety of purposes such as feature extraction from text, string replacement and other string manipulations.

E.g. extract all the hashtags from a tweet.

Quantifiers allow you to mention and have control over how many times you want the character(s) in your pattern to occur.

Sr. no	Character	Meaning
1	*	Matches the preceding character or character set zero or more times.
2	+	Matches the preceding character or character set one or more times.
3	?	Matches the preceding character or character set zero or one time.
4	{m, n}	Matches the preceding character or character set if it is present from 'm' times to 'n' times.

5	^	Marks the start of a string. If used inside a character set, acts as negation set.
6	\$	Marks the end of a string.
7	.	The wildcard character (used to match any character)
8	[a-z0-9]	Character set. Matches the character that are present inside the set.
9	\w	Meta-sequence used to match all the alphanumeric characters
10	\W	Meta-sequence used to match all the characters except the alphanumeric characters
11	\d	Meta-sequence used to match all the numeric digits.
12	\D	Meta-sequence used to match all the characters except the numeric digits.
13	\s	Meta-sequence used to match all the whitespace characters.
14	\S	Meta-sequence used to match all the characters except the whitespace characters.
15	*?	Non-greedy search. Search stops as soon as the pattern is found. '?' can be used after any of the quantifiers to make the search non-greedy.
16		Used as an OR operator inside character sets or groups
17	()	Used to group parts of regex to extract separately.

Python Functions

The most useful functions are listed below:

Sr. no.	Function	Meaning
1	<code>re.match(pattern ,string)</code>	Tries to look for the pattern in the string from the very beginning. Returns None if match not found at the start of the string.
2	<code>re.search(pattern, string)</code>	Tries to look for the pattern in the given string. Returns None if match not found in the entire string.
3	<code>re.findall(pattern, string)</code>	Tries to look for each occurrence of the pattern in the string and returns all of them in a list. Returns None if match not found.
4	<code>re.finditer(pattern, string)</code>	Similar to <code>re.findall()</code> but iterated over the matches one by one.
5	<code>re.sub(pattern, replacement, string)</code>	Tries to look for each occurrence of the pattern in the string and replaces all of them by the replacement string. Returns the same original if match not found.

Phonetic Hashing

There are certain words which have different pronunciations in different languages. As a result, they end up being spelt differently. Examples of such words include names of people, city names, names of dishes, etc. Take, for example, the capital of India - New Delhi. Delhi is also pronounced as Dilli in Hindi. Performing stemming or lemmatization to these words will not help us as much because the problem of redundant tokens will still be present. Hence, we need to reduce all the variations of a particular word to a common word. For this, you learn an algorithm called Soundex.

Using Soundex, you can reduce all the words to a four-digit code. All the words that have the same Soundex can then be mapped to a common word. For example, the Soundex of Bengaluru and Bangalore is B524.

Soundex

Algorithm

For a given word, apply the following steps:

1. Retain the first letter of the word as is
2. Replace consonants with digits according to the following:
 - b, f, p, v \Rightarrow 1
 - c, g, j, k, q, s, x, z \Rightarrow 2
 - d, t \Rightarrow 3
 - l \Rightarrow 4
 - m, n \Rightarrow 5
 - r \Rightarrow 6
 - h, w, y \Rightarrow unencoded
3. Remove all the vowels and the unencoded letters. And then merge any consecutive same numbers into a single number
4. Continue till the code has one letter and three numbers. If the length of Code is greater than four then truncate it to four letters. In case the code is shorter than four letters, pad it with zeroes

Example

Soundex of the word 'Mississippi'. To calculate the hash code, you'll make changes to the same word, in-place, as follows:

1. Phonetic hashing is a four-letter code. The first letter of the code is the first letter of the input word. Hence it is retained as is. The first character of the phonetic hash is 'M'. Now, we need to make changes to the rest of the letters of the word.
2. Now, we need to map all the consonant letters (except the first letter). All the vowels are written as is and 'H's, 'Y's and 'W's remain unencoded (unencoded means they are removed from the word). After mapping the consonants, the code becomes MI22I22I11I.
3. The third step is to remove all the vowels. 'I' is the only vowel. After removing all the 'I's, we get the code M222211. Now, you would need to merge all the consecutive duplicate numbers into a single unique number. All the '2's are merged into a single '2'. Similarly, all the '1's are merged into a single '1'. The code that we get is M21.
4. The fourth step is to force the code to make it a four-letter code. You either need to pad it with zeroes in case it is less than four characters in length. Or you need to truncate it from the right side

in case it is more than four characters in length. Since the code is less than four characters in length, you'll pad it with one '0' at the end. The final code is M210.

Edit Distance

An edit distance is the number of edits that are needed to convert a source string to a target string. You learnt that there are three edit operations allowed in the Levenshtein edit distance:

1. Insertion of a letter
2. Deletion of a letter
3. Substitution of a letter with another letter

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
q	2	1	1	1	2	3	4	5
u	3	2	2	2	1	2	3	4
i	4	3	3	3	2	1	2	3
r	5	4	4	4	3	2	1	2
e	6	5	5	5	4	3	2	1

Edit Distance

An edit distance is the number of edits that are needed to convert a source string to a target string. You learnt that there are three edit operations allowed in the Levenshtein edit distance:

1. Insertion of a letter
2. Deletion of a letter
3. Substitution of a letter with another letter

EDIT DISTANCE

	.	a	c	q	u	i	r	e
.	0	1	2	3	4	5	6	7
a	1	0	1	2	3	4	5	6
q	2	1	1	1	2	3	4	5
u	3	2	2	2	1	2	3	4
i	4	3	3	3	2	1	2	3
r	5	4	4	4	3	2	1	2
e	6	5	5	5	4	3	2	1

Spell Corrector

At last, the final technique to canonicalize words was correcting the spellings of the incorrect words. You learnt about the edit distance calculation between two strings. You'll use the concept of edit distance while building the spell corrector. You learnt how to build the famous Norvig's spell corrector. You need a seed document which can be any large corpus of text with correct spellings. The seed document will also act as a dictionary lookup for our spell corrector. There were four main functions that you used to find the correct spelling of the word:

1. `edits_one()`: It created all the possible words that are one edit distance away from the input word.
2. `edits_two()`: Similar to `edits_one()`, this function is used to return all the possible spellings that are two edit distance away from the input word.
3. `known()`: This function takes a list of words and output only the valid English words.
4. `possible_corrections()`: This function takes the input word and uses all of the above three functions to return possible correct spellings for the given input word. First, it looks as if the spelling is correct or not. If it is correct, it returns the input word. If the spelling is not correct, it looks for all the dictionary words that are one edit away from it and returns them. If there are no dictionary words that are one edit distance away from the given word, then it looks for all the dictionary words that are two edits away and returns them. If there are no such words that are two edits away, the original word is returned which means no correct spelling was found.
5. `prob()`: The probability function takes multiple words and returns the word that is most frequent among the input words in the seed document. The word returned by this is the correct spelling.

Pointwise Mutual Information

Suppose there is an article titled “Higher Technical Education in India” which talks about the state of Indian education system in engineering space. Let’s say, it contains names of various Indian colleges such as ‘International Institute of Information Technology, Bangalore’, ‘Indian Institute of Technology, Mumbai’, ‘National Institute of Technology, Kurukshetra’ and many other colleges. Now, when you tokenise this document, all these college names will be broken into individual words such as ‘Indian’, ‘Institute’, ‘International’, ‘National’, ‘Technology’ and so on. But we don’t want this. We want an entire college name to be represented by one token.

$$\text{pmi}(x; y) = \log \frac{p(x, y)}{p(x) p(y)}$$

Text Encoding

The alphabets and special characters were to be converted to a numeric value first before they could be stored. Hence, the concept of **encoding** came into existence. All the non-numeric characters were encoded to a number using a code. Also, the encoding techniques had to be standardised so that different computer manufacturers won't use different encoding techniques.

Symbol	ASCII code		UTF-8 code		UTF-16 (BE) code	
	Binary	Hex	Binary	Hex	Binary	Hex
\$ (Dollar sign)	00100100	24	00100100	24	00000000 00100100	0024
₹ (Indian Rupee sign)	Doesn't exist	Doesn't exist	11100010 10000010 10111001	E282B9	00100000 10111001	20B9

Stop Words

Broadly, there are three kinds of words present in any text corpus:

- Highly frequent words called as stop words, such as 'is', 'an', 'the', etc.
- Significant words, which are very helpful for any analysis.
- Rarely occurring words.

Stop words are removed from the text because of two reasons:

1. They provide no useful information in most of the applications.
2. They use a lot of memory because of such high frequency in which they are present.

Tokenization

Tokenization is a kind of pre-processing in a sense; an identification of basic units to be processed. It is conventional to concentrate on pure analysis or generation while taking basic units for granted. Yet without these basic units clearly segregated it is impossible to carry out any analysis or generation.

Before you move ahead with any kind of lexical processing, you need to break the text corpus into different words or sentences or paragraphs according to the end application in mind.

Manual Tokenization

```
words = text.split()
```

NLTK library has various tokenizers:

1. **Word** tokeniser: Use this tokeniser to break your text corpus into a list of words.
2. **Sentence** tokeniser: Use this tokeniser to break your text into different sentences.
3. **Tweet** tokeniser: Use this tokeniser to tokenize text into different words. This also tokenises social media elements such as emojis and hashtags correctly unlike the word tokeniser.
4. **Regexp** Tokeniser: Use this to tokenise the text using a regular expression.

Stemming

Producing morphological
variants of a root

Affections/Affection/Affected/
Affects/Affect → Affect

likes/like/likely → like

Over Stemming: 2 words stemmed to a same root that are of
different stems (FP)

A: No, P: Yes

Example: universal/universe/university → universe/univers

Under Stemming: 2 words stemmed to same root that are not of
different stems (FN)

A:Y, P:N

Example: Alumnus/Alumni → Alumnu, Alumni

Stemming

Stemmers:

1. **PorterStemmer** → Most common & sophisticated/gentle stemmer, fast but not very precise
2. **Snowball Stemmer** → Also called as English stemmer, or Porter2 stemmer, better in comparison to PorterStemmer, works faster on larger datasets.
3. **Lancaster Stemmer**
Aggressive algorithm, hugely trims down the datasets, so it has its pros & cons → Sometimes it is good, sometimes it is not good

Lemmatization

- This is a more sophisticated technique that is more intelligent in the sense that it doesn't just chop off the suffix of a word. Instead, it takes an input word and searches for its base word by going recursively through all the variations of **dictionary** words.
- The base word, in this case, is called the 'lemma'.
- Words such as 'feet', 'drove', 'arose', 'bought', etc. can't be reduced to their correct base form using a stemmer. But a lemmatizer can reduce them to their correct base form.
- The most popular lemmatizer is the WordNet lemmatizer created by a team of professors at Princeton University.

Applications of Stemming/Lemmatization

Widely used in text mining.

Text Mining tasks:

1. Text categorizing/classifying
2. Text clustering
3. Sentiment Analysis
4. Document Summarization
5. Entity relation modelling etc.

nGrams

Easiest concept to understand.

Snake is the poisonous creature.

If $n=2$.

Output:

Snake is

is the

the poisonous

poisonous creature

Why ngrams?

In NLP, n-grams has varieties of usage. Some examples include auto completion of sentences (such as the one we see in Gmail these days), auto spell check (yes, we can do that as well), and to a certain extent, we can check for grammar in a given sentence. We'll see some examples of this later in the post when we talk about assigning probabilities to n-grams.

nGrams

nGram Probability

1. Thank you so much for your help.
2. I really appreciate your help.
3. Excuse me, do you know what time it is?
4. I'm really sorry for not inviting you.
5. I really like your watch.
6. I really like your wall.

New Sentence: I really like

really → appreciate → 25%

really → sorry → 25%

really → like → 50%

So you have 4 n-grams in this case. When **N=1**, this is referred to as **unigrams** and this is essentially the individual words in a sentence. When **N=2**, this is called **bigrams** and when **N=3** this is called **trigrams**. When **N>3** this is usually referred to as four grams or five grams and so on.

Entity Detection

Entities are the words or groups of words that represent information about common things such as persons, locations, organizations, etc. These entities have proper names.

For example, consider the following sentence:

"Donald Trump will meet the chairman of Google in New York City"

In this sentence, the entities are "Donald Trump", "Google", and "New York City".

Let's now see how spaCy recognizes named entities in a sentence.

Output:

Indians NORP, over \$71 billion MONEY, 2018 DATE

```
spacy.explain("NORP")
```

Output: 'Nationalities or religious or political groups'

NLTK Alternative

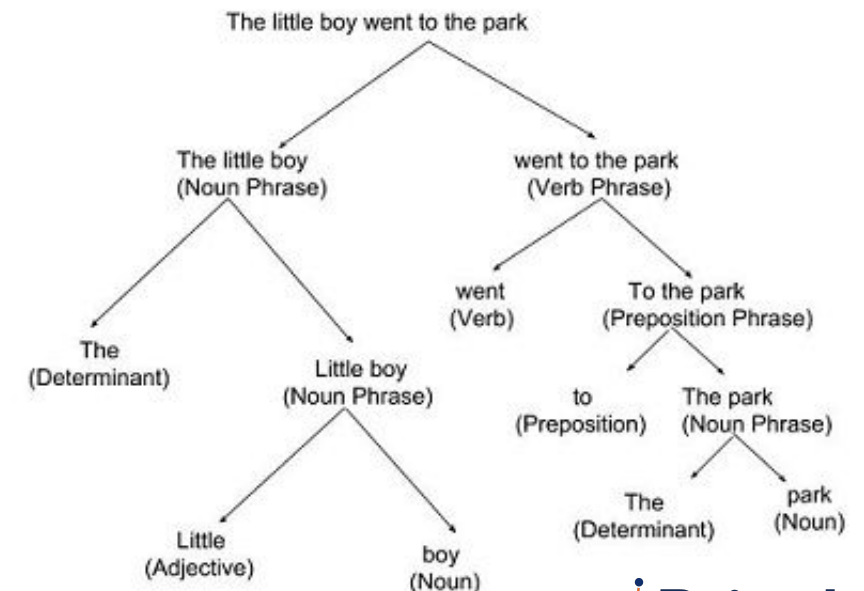
```
namedEnt = nltk.ne_chunk(tagged)
namedEnt.draw()
```

Syntactic processing - Parsing

POS tagging is the task of assigning a part of speech tag (POS tag) to each word. The POS tags identify the linguistic role of the word in the sentence. The POS tags of the sentence are:

The	little	boy	went	to	the	park
Determinant	Adjective	Noun	Verb	Preposition	Determinant	Noun

Constituency parsers divide the sentence into constituent phrases such as noun phrase, verb phrase, prepositional phrase etc. Each constituent phrase can itself be divided into further phrases. The constituency parse tree given below divides the sentence into two main phrases - a noun phrase and a verb phrase. The verb phrase is further divided into a verb and a prepositional phrase, and so on.



PoS Tagging - Tags

A frequently asked question is “*What do the Part of Speech tags (VB, JJ, etc) mean?*” The bottom line is that these tags mean whatever they meant in your original training data. You are free to invent your own tags in your training data, as long as you are consistent in their usage.

Training data generally takes a lot of work to create, so a pre-existing corpus is typically used. These usually use the **Penn Treebank** or **Brown Corpus** tags.

The most common part of speech (POS) tag schemes are those developed for the Penn Treebank and Brown Corpus. Penn Treebank is probably the most common, but both corpora are available with NLTK.

<https://www.winwaed.com/blog/2011/11/08/part-of-speech-tags/>

Here are the POS tags used in the Penn Treebank:

POS Tag	Description	Example
CC	coordinating conjunction	and
CD	cardinal number	1, third
DT	determiner	the
EX	existential there	there is
FW	foreign word	d'hoevre
IN	preposition/subordinating conjunction	in, of, like
JJ	adjective	big
JJR	adjective, comparative	bigger
JJS	adjective, superlative	biggest
LS	list marker	1)
MD	modal	could, will
NN	noun, singular or mass	door
NNS	noun plural	doors
NNP	proper noun, singular	John
NNPS	proper noun, plural	Vikings
PDT	predeterminer	both the boys
POS	possessive ending	friend's
PRP	personal pronoun	I, he, it
PRP\$	possessive pronoun	my, his
RB	adverb	however, usually, naturally, here, good
RBR	adverb, comparative	better
RBS	adverb, superlative	best
RP	particle	give up
TO	to	to go, to him
UH	interjection	uhhuhhuhh
VB	verb, base form	take
VBD	verb, past tense	took
VBG	verb, gerund/present participle	taking
VBN	verb, past participle	taken
VBP	verb, sing. present, non-3d	take
VBZ	verb, 3rd person sing. present	takes
WDT	wh-determiner	which
WP	wh-pronoun	who, what
WP\$	possessive wh-pronoun	whose
WRB	wh-abverb	where, when

PoS Tagging Approaches

Now that you are familiar with the commonly used POS tags, we will discuss techniques and algorithms for POS tagging. We will look at the following 3 main techniques used for POS tagging:

1. **Lexicon-based**-The lexicon-based approach uses the following simple statistical algorithm: for each word, it assigns the POS tag that most frequently occurs for that word in some training corpus.
2. **Rule-based**-first assign the tag using the lexicon method and then apply predefined rules. e.g. Change the tag to VBG for words ending with '-ing'
3. **Probabilistic (or stochastic) techniques**- Probabilistic taggers don't naively assign the highest frequency tag to each word, instead, they look at slightly longer parts of the sequence and often use the tag(s) and the word(s) appearing before the target word to be tagged.

PoS Tagging -Probabilistic- Hidden Markov

Why Part-of-Speech tagging?

Part-of-Speech tagging in itself may not be the solution to any particular NLP problem. It is however something that is done as a pre-requisite to simplify a lot of different problems. Let us consider a few applications of POS tagging in various NLP tasks.

Text to Speech Conversion

They refuse to permit us to obtain the refuse permit.

The word `refuse` is being used twice in this sentence and has two different meanings here. *refUSE* (/rə'fyʊz/) is a verb meaning “deny,” while *REFuse* (/ref.yʊs/) is a noun meaning “trash” (that is, they are not homophones). Thus, we need to know which word is being used in order to pronounce the text correctly. (For this reason, text-to-speech systems usually perform POS-tagging.)

PoS Tagging - Hidden Markov - Contd.

Word Sense Disambiguation

Let's talk about this kid called Peter. Since his mother is a neurological scientist, she didn't send him to school. His life was devoid of science and math.

One day she conducted an experiment, and made him sit for a math class. Even though he didn't have any prior subject knowledge, Peter thought he aced his first test. His mother then took an example from the test and published it as below. (Kudos to her!)

~ PETER

MATHS EXAM

Q.1) Expand $(a+b)^n$

Ans) $(a+b)^n$

$= (a + b)^n$


$= (a + b)^n$

$= (a + b)^n$

Very funny, +

Peter

?



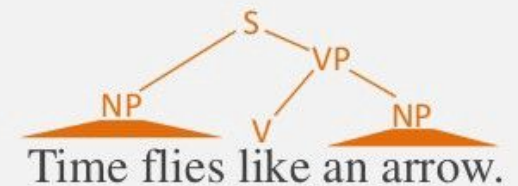
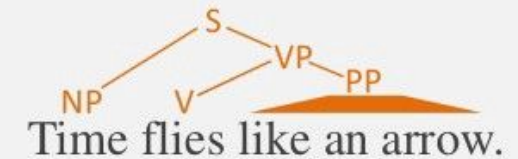
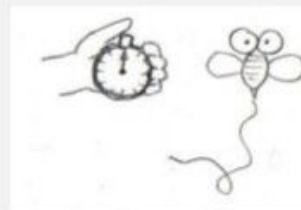
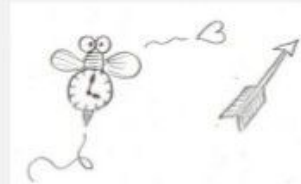
PoS Tagging - Hidden Markov - Contd.

Multiple interpretations possible for the given sentence.

Different interpretations yield different kinds of part of speech tags for the words.

This example shows us that a single sentence can have three different POS tag sequences assigned to it that are equally likely. That means that it is very important to know what specific meaning is being conveyed by the given sentence whenever it's appearing. **This is word sense disambiguation, as we are trying to find out THE sequence.**

Syntactic Ambiguity

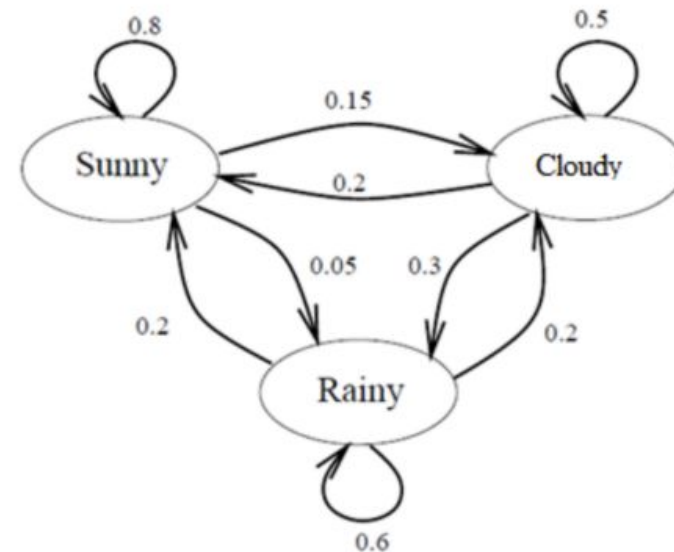


PoS Tagging - Markov Model- Contd.

Markov's Model

Say you have a sequence. Something like this:

Sunny, Rainy, Cloudy, Cloudy, Sunny, Sunny,
Sunny, Rainy etc etc.....



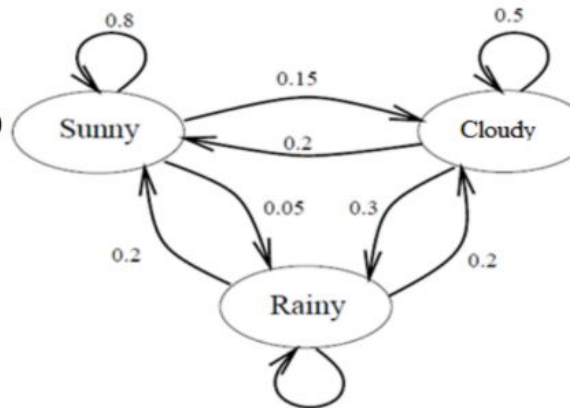
$$\begin{aligned} & \left. \begin{aligned} P(\text{Sunny}|\text{Sunny}) &= 0.8 \\ P(\text{Rainy}|\text{Sunny}) &= 0.05 \\ P(\text{Cloudy}|\text{Sunny}) &= 0.15 \end{aligned} \right\} 1 \\ & \left. \begin{aligned} P(\text{Sunny}|\text{Rainy}) &= 0.2 \\ P(\text{Rainy}|\text{Rainy}) &= 0.6 \\ P(\text{Cloudy}|\text{Rainy}) &= 0.2 \end{aligned} \right\} 1 \\ & \left. \begin{aligned} P(\text{Sunny}|\text{Cloudy}) &= 0.2 \\ P(\text{Rainy}|\text{Cloudy}) &= 0.3 \\ P(\text{Cloudy}|\text{Cloudy}) &= 0.5 \end{aligned} \right\} 1 \end{aligned}$$

$$P(q_1, \dots, q_n) = \prod_{i=1}^n P(q_i | q_{i-1})$$

PoS Tagging - Markov Model- Contd.

- Exercise 1: Given that today is Sunny, what's the probability that tomorrow is Sunny and the next day Rainy?

$$\begin{aligned} P(q_2, q_3|q_1) &= P(q_2|q_1)P(q_3|q_1, q_2) \\ &= P(q_2|q_1) P(q_3|q_2) \\ &= P(\text{Sunny}|\text{Sunny}) P(\text{Rainy}|\text{Sunny}) \\ &= (0.8)(0.05) \\ &= 0.04 \end{aligned}$$

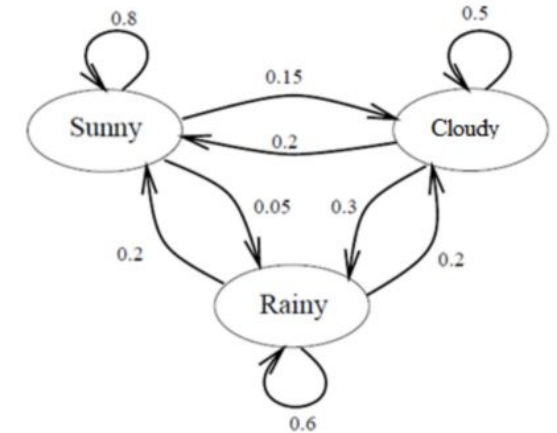


- Exercise 2: Assume that yesterday's weather was Rainy, and today is Cloudy, what is the probability that tomorrow will be Sunny?

$$P(q_3|q_1, q_2) = P(q_3|q_2)$$

$$= P(\text{Sunny}|\text{Cloudy})$$

$$= 0.2$$



$$P(q_1, \dots, q_n) = \prod_{i=1}^n P(q_i|q_{i-1})$$

Hidden Markov for PoS

HMMs are probabilistic graphical models used to predict a sequence of hidden (unknown) states from a set of observable states.

This class of models follows the Markov processes assumption:

“The future is independent of the past, given that we know the present”

Therefore, when working with Hidden Markov Models, we just need to know our present state in order to make a prediction about the next one (we don't need any information about the previous states).

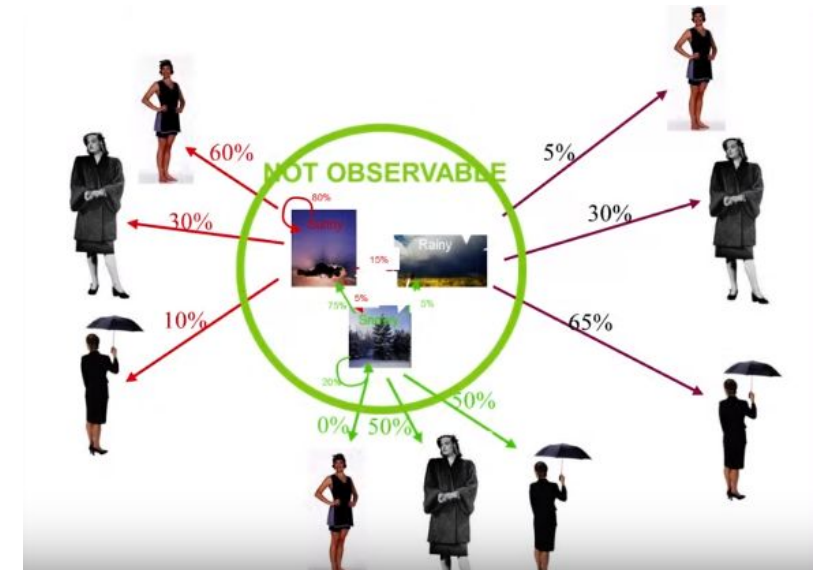
To make our predictions using HMMs we just need to calculate the joint probability of our hidden states and then select the sequence which yields the highest probability (the most likely to happen).

In order to calculate the joint probability we need three main types of information:

Initial condition: the initial probability we have to start our sequence in any of the hidden states.

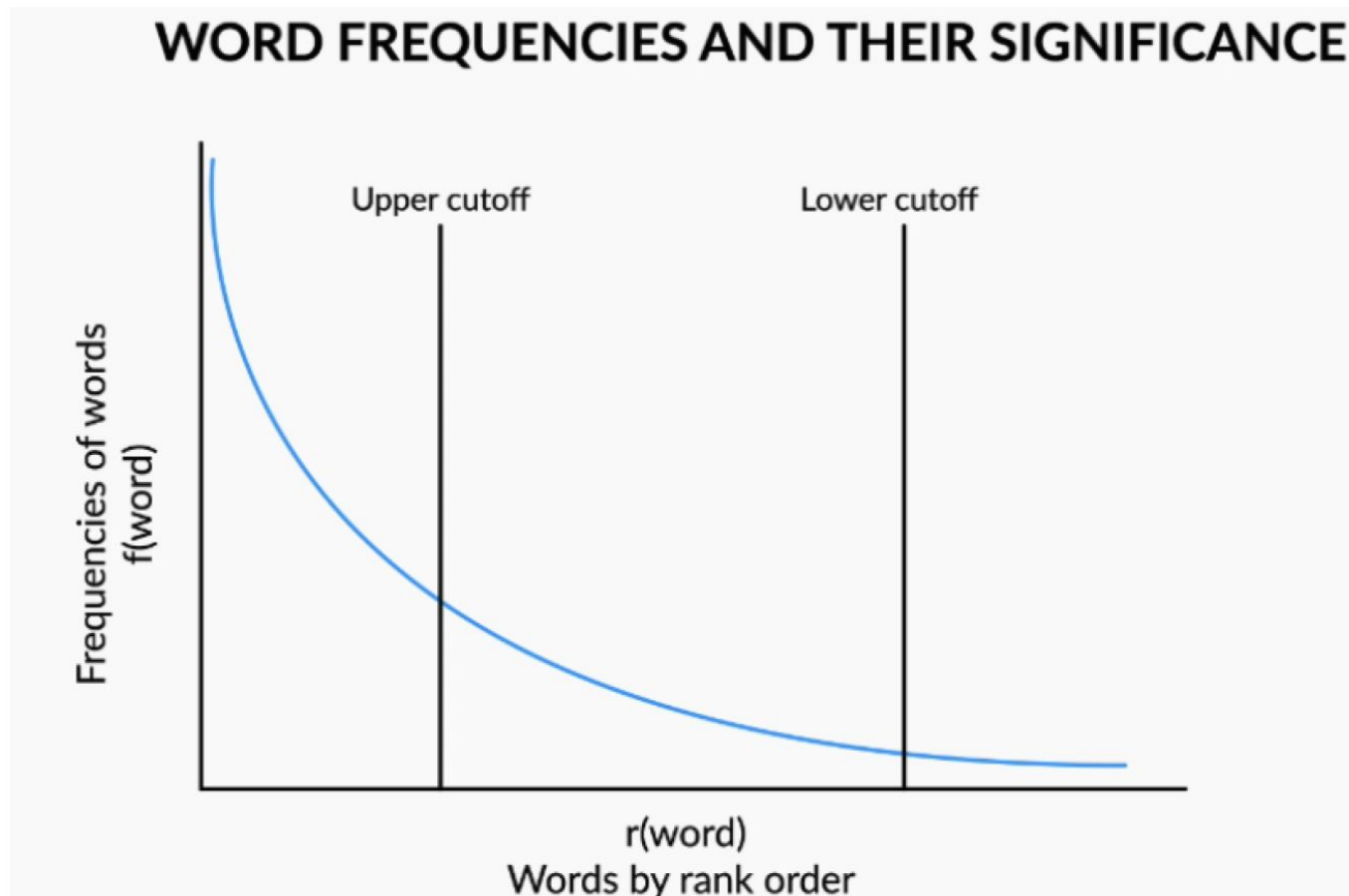
Transition probabilities: the probabilities of moving from one hidden state to another.

Emission probabilities: the probabilities of moving from a hidden state to an observable state.



Word Frequencies

When you plot any text document that is large enough (say, a hundred thousand words), you will notice that the word frequencies follow Zipf distribution as shown in the following image



Word Embeddings

Word embedding is a dense representation of words in the form of numeric vectors. It can be learned using a variety of language models. The word embedding representation is able to reveal many hidden relationships between words. For example, $\text{vector}(\text{"cat"}) - \text{vector}(\text{"kitten"})$ is similar to $\text{vector}(\text{"dog"}) - \text{vector}(\text{"puppy"})$.

Why do we use word embedding?

Words aren't things that computers naturally understand. By encoding them in a numeric form, we can apply mathematical rules and do matrix operations to them

One-Hot-Encoding (Count Vectorizing) - Unique words counts in a corpus represented as a vector e.g. {'machine': 7, 'learning': 6, 'introduction': 5, 'to': 13, 'and': 0, 'hands': 4, 'on': 11, 'experience': 3, 'the': 12, 'various': 14, 'applications': 1, 'of': 10, 'ml': 8, 'deep': 2, 'nlp': 9}

Co-Occurrence Matrix - Shows which words occur together in a one-hot encoding format. e.g. see below

	a	and	earned	foolish	is	penny	pound	saved	wise
a	0	0	0	0	0	2	0	0	0
penny	0	0	1	0	0	0	0	1	1

TF-IDF Representation

To calculate the tf-idf representation or tf-idf model, you need to calculate a tf-idf score for each word in each document. The tf-idf score comprises of two terms – the 'tf' (term frequency), and the 'idf' (inverse document frequency). The formula of tf-idf is shown in the following image:

$$tf_{t,d} = \frac{\text{frequency of term 't' in document 'd'}}{\text{total terms in document 'd'}}$$

$$idf_t = \log \frac{\text{total number of documents}}{\text{total documents that have the term 't'}}$$

$$tf - idf = tf_{t,d} * idf_t$$

Bag-of-Words

The central idea is that any given piece of text, i.e., tweets, articles, messages, emails etc., can be “represented” by a list of all the words that occur in it (after removing the stopwords), where the sequence of occurrence does not matter. You can visualise it as the “bag” of all “words” that occur in it.

Document 1 : “Gangs of Wasseypur is a great movie.”

Document 2 : “The success of a movie depends on the performance of the actors.”

Document 3 : “There are no new movies releasing this week.”

	actors	great	depends	gangs	movie	movies	new	performance	releasing	success	wasseygur	week
1	0	1	0	1	1	0	0	0	0	0	1	0
2	1	0	1	0	1	0	0	1	0	1	0	0
3	0	0	0	0	0	1	1	0	1	0	0	1

Vector Theory

A vector has length and direction. The vector's absolute position is not relevant. Two vectors may be same even though their starting positions are different. For example, the vectors in the first slide have different starting positions. They can all be moved to have the same starting position at the (0,0) origin by simply focusing on their movements in X and Y space.

For example, Vector **b** moves 1 unit in the x direction and 1 unit in the y direction, so this vector can be translated to (1,1) with respect to the (0,0) origin.

Vector Length

The length of a vector in two dimensional space is the length of its hypotenuse. The length of a vector is called its "*norm*" and is defined below:

Unit Vectors

To scale different vectors to a common length, each vector is divided by its length. This ensures that the resultant vector has a length of one. This is called the unit vector as shown below:

Dot Product

The dot product of two vectors is simply multiplying each component of the vectors and then adding the results. This is shown below:

Assuming that z is (2,0), then the dot product of b (1,1) and z (2,0) would be $(1 \times 2) + (1 \times 0) = 2$.

$$\| \mathbf{v} \| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

$$\mathbf{b} \cdot \mathbf{z} = b_1 z_1 + b_2 z_2 + \dots + b_n z_n$$

Cosine Similarity

What is Cosine Similarity

Identifies Similarities between two or more vectors.

It is the similarity measure.

Now that the dot product and norm has been defined, then the cosine similarity of two vectors is simply the dot product of two unit vectors. This is shown below:

Given that vector b moves up and to the right by equal amounts, it would be expected that this vector is 45 degrees to the x axis. If the x axis is represented by z (2,0). The result of the cosine similarity between b and z is equal to: 0.7071. The inverse cosine of this value is .7855 radians or 45 degrees.

$$\cos(\theta) = \frac{\mathbf{b} \cdot \mathbf{z}}{\|\mathbf{b}\| \cdot \|\mathbf{z}\|}$$

<https://beingdatum.com/text-matching-using-cosine-similarity-flask-api/>

Word2Vect

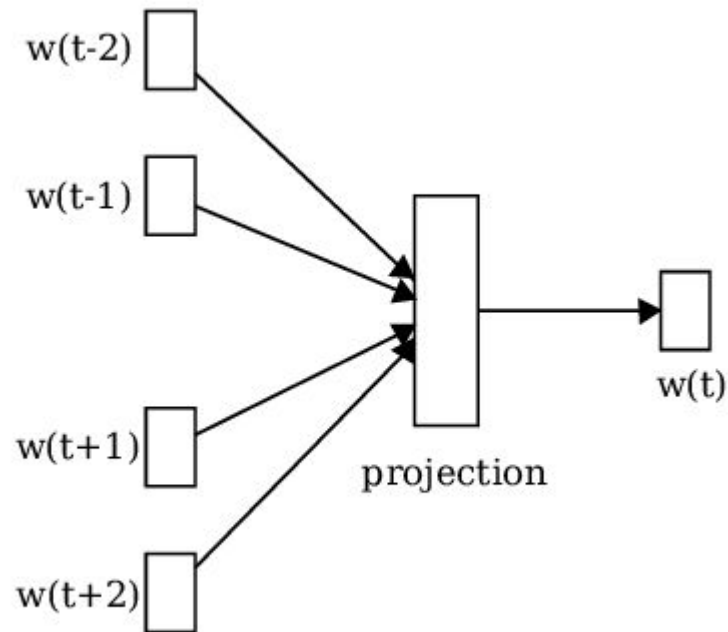
Each word is represented by a vector (Array of numbers based on Embedding Size).

In general the Word2Vec is based on Window Method, where we have to assign a Window size.

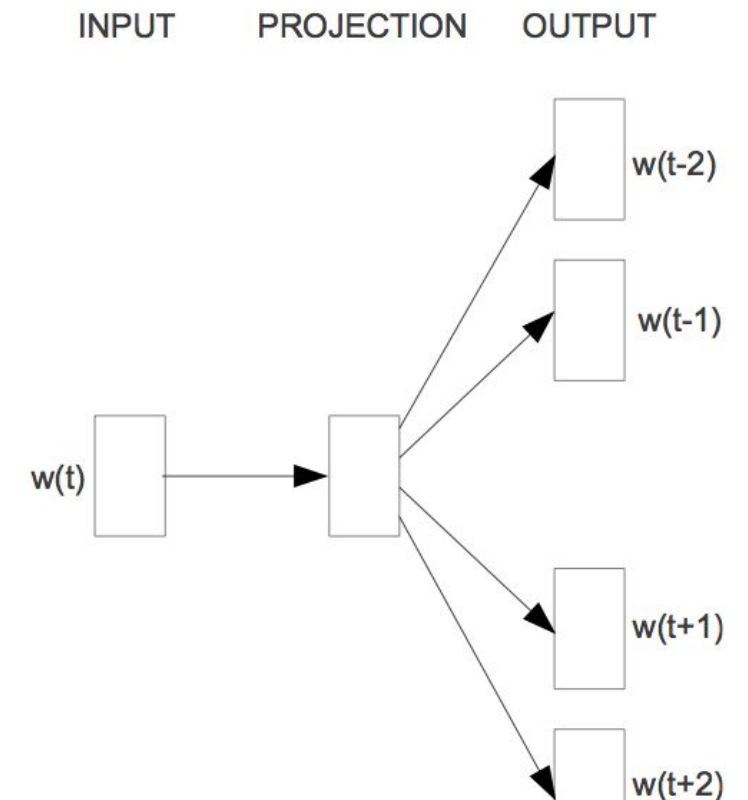
There are 2 types of Algorithms : CBOW and Skip-gram.



Continuous Bag of Words (CBOW) :



Skip-gram :

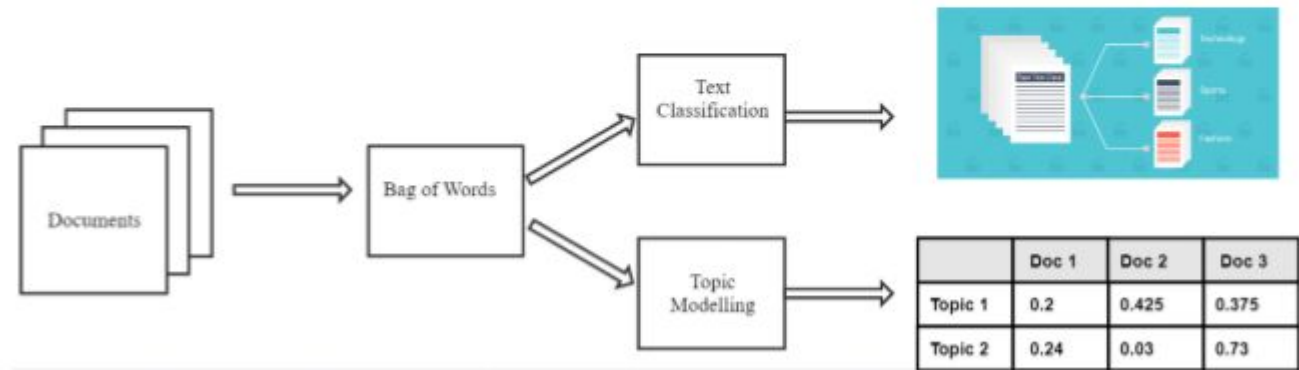


Topic Modelling

While Analyzing text messages, paragraphs → We need to analyse by it's type, whether it's related to politics, sports etc.

When volume of data increases we need to ensure that we bucket different articles based on their semantic similarity, so that it's easier to analyze.

How Topic Modelling is different from Text Classification



Topic Modelling- LSA

LSA (Latent Semantic Analysis) also known as LSI (Latent Semantic Index) LSA uses bag of words(BoW) model, which results in a term-document matrix(occurrence of terms in a document). Rows represent terms and columns represent documents. LSA learns latent topics by performing a matrix decomposition on the document-term matrix using Singular value decomposition. LSA is typically used as a dimension reduction or noise reducing technique.

Pros and Cons of LSA:

1. LSA is the simplest method which is easy to understand and implement.
2. It offers better results compared to vector space model.
3. Faster compared to other available algorithms because it involves document term matrix decomposition only.

Use Cases of Topic Modelling

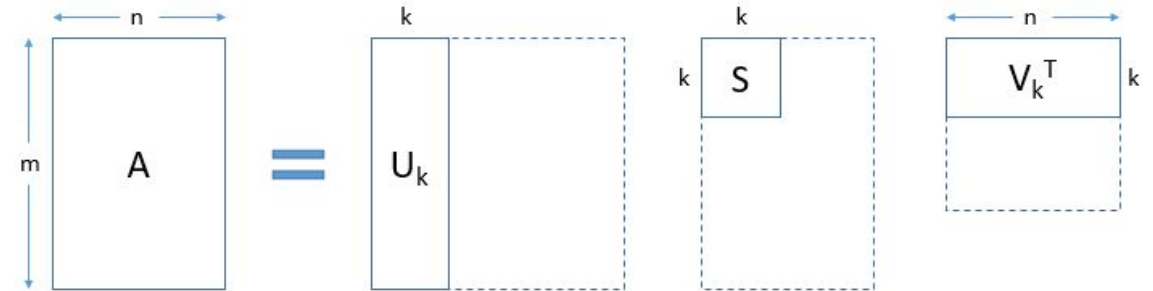
1. Resume Summarization
2. Search Engine Optimization
3. Recommender System Optimization
4. Improving customer support

Topic Modelling- LSA

Generate a document-term matrix of shape $m \times n$ having TF-IDF scores

	Terms				
	T1	T2	T3	...	Tn
D1	0.2	0.1	0.5	...	0.1
D2	0.1	0.3	0.4	...	0.3
D3	0.3	0.1	0.1	...	0.5
...
Dm	0.2	0.1	0.2	...	0.1

$$A = USV^T$$



- Then, we will reduce the dimensions of the above matrix to k (no. of desired topics) dimensions, using singular-value decomposition (SVD).
- SVD decomposes a matrix into three other matrices. Suppose we want to decompose a matrix A using SVD. It will be decomposed into matrix U, matrix S, and VT (transpose of matrix V)

- Each row of the matrix **U_k (document-term matrix)** is the vector representation of the corresponding document. The length of these vectors is k, which is the number of desired topics. Vector representation for the terms in our data can be found in the matrix **V_k (term-topic matrix)**.
- So, SVD gives us vectors for every document and term in our data. The length of each vector would be **k**. We can then use these vectors to find similar words and similar documents using the cosine similarity method.

SpaCy

Dep tree	Token	Dep type
	The	det
	programmer	nsubjpass
	was	auxpass
	pleased	ROOT
	by	agent
	the	det
	nicely	advmod
	formatted	amod
	parse	compound
	tree	pobj
	.	punct

- One of the most powerful feature of spacy is the extremely fast and accurate syntactic **dependency** parser which can be accessed via lightweight API. The parser can also be used for sentence boundary detection and phrase chunking. The relations can be accessed by the properties “.children” , “.root” , “.ancestor” etc. Every sentence has a grammatical structure to it and with the help of dependency parsing, we can extract this structure. It can also be thought of as a directed graph, where nodes correspond to the words in the sentence and the edges between the nodes are the corresponding dependencies between the word. The dependency tag ROOT denotes the main verb or action in the sentence. The other words are directly or indirectly connected to the ROOT word of the sentence.
- Spacy also provides **inbuilt** integration of dense, real valued **vectors** representing distributional similarity information. It uses [GloVe](#) vectors to generate vectors. GloVe is an unsupervised learning algorithm for obtaining vector representations for words.
- SpaCy is **faster** than NLTK.
- Downside of SpaCY is **multi language** support is **not present**.

Stanford NLP

Advantages:

1. Its out-of-the-box support for multiple languages
2. The fact that it is going to be an official Python interface for CoreNLP. This means it will only improve in functionality and ease of use going forward
3. It is fairly fast (barring the huge memory footprint)
4. Straightforward set up in Python

Disadvantages:

1. The size of the language models is too large (English is 1.9 GB, Chinese ~ 1.8 GB)
2. The library requires a lot of code to churn out features. Compare that to NLTK where you can quickly script a prototype – this might not be possible for StanfordNLP
3. Currently missing visualization features. It is useful to have for functions like dependency parsing. StanfordNLP falls short here when compared with libraries like SpaCy



NLP

Thank You!

BeingDatum.com
contact@beingdatum.com