**NLTK (Natural Language Toolkit) Tutorial in Python**

**What is Natural Language Processing?**

Natural Language Processing is manipulation or understanding text or speech by any software or machine. An analogy is that humans interact, understand each other views, and respond with the appropriate answer. In NLP, this interaction, understanding, the response is made by a computer instead of a human.

**What is NLTK?**

NLTK stands for Natural Language Toolkit. This toolkit is one of the most powerful NLP libraries which contains packages to make machines understand human language and reply to it with an appropriate response. Tokenization, Stemming, Lemmatization, Punctuation, Character count, word count are some of these packages which will be discussed in this tutorial.

**Various NLP Libraries**

|  |  |
| --- | --- |
| **NLP Library** | **Description** |
| NLTK | This is one of the most usable and mother of all NLP libraries. |
| spaCy | This is completely optimized and highly accurate library widely used in deep learning |
| Stanford CoreNLP Python | For client-server based architecture this is a good library in NLTK. This is written in JAVA, but it provides modularity to use it in Python. |
| TextBlob | This is an NLP library which works in Pyhton2 and python3. This is used for processing textual data and provide mainly all type of operation in the form of API. |
| Gensim | Genism is a robust open source NLP library support in python. This library is highly efficient and scalable. |
| Pattern | It is a light-weighted NLP module. This is generally used in Web-mining, crawling or such type of spidering task. p |
| Polyglot | For massive multilingual applications, Polyglot is best suitable NLP library. Feature extraction in the way on Identity and Entity. |
| PyNLPl | PyNLPI also was known as 'Pineapple' and supports Python. It provides a parser for many data format like FoLiA/Giza/Moses/ARPA/Timbl/CQL. |
| Vocabulary | This library is best to get Semantic type information from the given text. |

In this tutorial, we will only discuss one of the most popular NLP library NLTK.

**Here is what we cover in the Course**

# NLP (Natural Language Processing) Tutorial: What is, History, Example

## What is Natural Language Processing?

Natural Language Processing (NLP) is a branch of AI that helps computers to understand, interpret and manipulate human language.

NLP helps developers to organize and structure knowledge to perform tasks like translation, summarization, named entity recognition, relationship extraction, speech recognition, topic segmentation, etc.

NLP is a way of computers to analyze, understand and derive meaning from a human languages such as English, Spanish, Hindi, etc.

In this nlp tutorial, you will learn:

* [What is Natural Language Processing?](https://www.guru99.com/nlp-tutorial.html#1)
* [History of NLP](https://www.guru99.com/nlp-tutorial.html#2)
* [How does NLP work?](https://www.guru99.com/nlp-tutorial.html#3)
* [Components of NLP](https://www.guru99.com/nlp-tutorial.html#4)
* [NLP and writing systems](https://www.guru99.com/nlp-tutorial.html#5)
* [How to implement NLP](https://www.guru99.com/nlp-tutorial.html#6)
* [NLP Examples](https://www.guru99.com/nlp-tutorial.html#7)
* [Future of NLP](https://www.guru99.com/nlp-tutorial.html#8)
* [Natural language vs. Computer Language](https://www.guru99.com/nlp-tutorial.html#9)
* [Advantages of NLP](https://www.guru99.com/nlp-tutorial.html#10)
* [Disadvantages of NLP](https://www.guru99.com/nlp-tutorial.html#11)

**History of NLP**

Here, is are important events in the history of Natural Language Processing:

**1950-**NLP started when Alan Turing published an article called "Machine and Intelligence."

**1950-**Attempts to automate translation between Russian and English

**1960-**The work of Chomsky and others on formal language theory and generative syntax

**1990-**Probabilistic and data-driven models had become quite standard

**2000-**A Large amount of spoken and textual data become available

**How does NLP work?**

Before we learn how NLP works, let's understand how humans use language-

Every day, we say thousand of a word that other people interpret to do countless things. We, consider it as a simple communication, but we all know that words run much deeper than that. There is always some context that we derive from what we say and how we say it., NLP never focuses on voice modulation; it does draw on contextual patterns.

Example:

Man is to woman as king is to \_\_\_\_\_\_\_\_\_\_?

Meaning (king) – meaning (man) + meaning ( woman)=?

The answer is- queen

Here, we can easily co-relate because man is male gender and woman is female gender. In the same way, the king is masculine gender, and its female gender is queen.

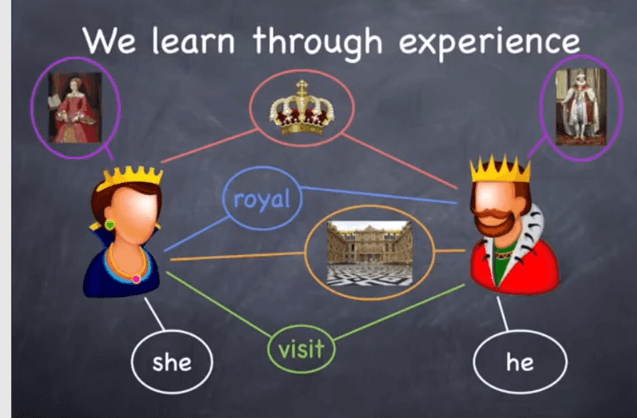
Example:

Is King to kings as the queen is to\_\_\_\_\_\_\_?

The answer is--- queens

Here, we can see two words kings and kings where one is singular and other is plural. Therefore, when the world queen comes, it automatically co-relates with queens again singular plural.

Here, the biggest question is that how do we know what words mean? Let's, say who will call it queen?



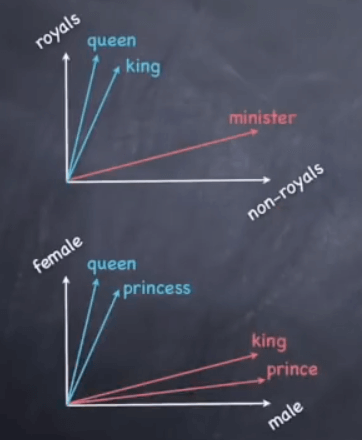
The answer is we learn this thinks through experience. However, here the main question is that how computer know about the same?

We need to provide enough data for Machines to learn through experience. We can feed details like

* Her Majesty the Queen.
* The Queen's speech during the State visit
* The crown of Queen Elizabeth
* The Queens's Mother
* The queen is generous.

With above examples the machine understands the entity Queen.

The machine creates word vectors as below. A word vector is built using surrounding words.



The machine creates these vectors

* As it learns from multiple datasets
* Use Machine learning (e.g., Deep Learning algorithms)
* A word vector is built using surrounding words.

Here is the formula:

Meaning (king) – meaning (man) + meaning (woman)=?

This amounts to performing simple algebraic operations on word vectors:

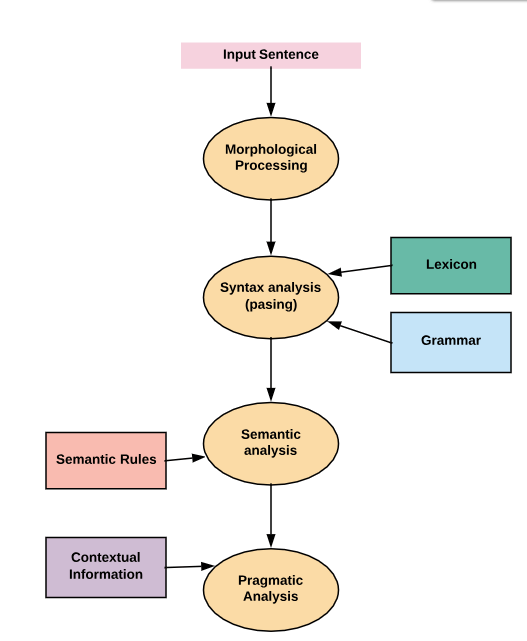
Vector ( king) – vector (man) + vector (woman)= vector(?)

To which the machine answers queen.

**Components of NLP**

Five main Component of Natural Language processing are:

* Morphological and Lexical Analysis
* Syntactic Analysis
* Semantic Analysis
* Discourse Integration
* Pragmatic Analysis



**Morphological and Lexical Analysis**

Lexical analysis is a vocabulary that includes its words and expressions. It depicts analyzing, identifying and description of the structure of words. It includes dividing a text into paragraphs, words and the sentences

Individual words are analyzed into their components, and nonword tokens such as punctuations are separated from the words.

**Semantic Analysis**

Semantic Analysis is a structure created by the syntactic analyzer which assigns meanings. This component transfers linear sequences of words into structures. It shows how the words are associated with each other.

Semantics focuses only on the literal meaning of words, phrases, and sentences. This only abstracts the dictionary meaning or the real meaning from the given context. The structures assigned by the syntactic analyzer always have assigned meaning

E.g.. "colorless green idea." This would be rejected by the Symantec analysis as colorless Here; green doesn't make any sense.

**Pragmatic Analysis**

Pragmatic Analysis deals with the overall communicative and social content and its effect on interpretation. It means abstracting or deriving the meaningful use of language in situations. In this analysis, the main focus always on what was said in reinterpreted on what is meant.

Pragmatic analysis helps users to discover this intended effect by applying a set of rules that characterize cooperative dialogues.

E.g., "close the window?" should be interpreted as a request instead of an order.

**Syntax analysis**

The words are commonly accepted as being the smallest units of syntax. The syntax refers to the principles and rules that govern the sentence structure of any individual languages.

Syntax focus about the proper ordering of words which can affect its meaning. This involves analysis of the words in a sentence by following the grammatical structure of the sentence. The words are transformed into the structure to show hows the word are related to each other.

**Discourse Integration**

It means a sense of the context. The meaning of any single sentence which depends upon that sentences. It also considers the meaning of the following sentence.

For example, the word "that" in the sentence "He wanted that" depends upon the prior discourse context.

**NLP and writing systems**

The kind of writing system used for a language is one of the deciding factors in determining the best approach for text pre-processing. Writing systems can be

1. Logographic: a Large number of individual symbols represent words. Example Japanese, Mandarin
2. Syllabic: Individual symbols represent syllables
3. Alphabetic: Individual symbols represent sound

Majority of the writing systems use the Syllabic or Alphabetic system. Even English, with its relatively simple writing system based on the Roman alphabet, utilizes logographic symbols which include Arabic numerals, Currency symbols (S, £), and other special symbols.

This pose following challenges

* Extracting meaning(semantics) from a text is a challenge
* NLP is dependent on the quality of the corpus. If the domain is vast, it's difficult to understand context.
* There is a dependence on the character set and language

**How to implement NLP**

Below, given are popular methods used for Natural Learning Process:

**Machine learning:**The learning nlp procedures used during machine learning. It automatically focuses on the most common cases. So when we write rules by hand, it is often not correct at all concerned about human errors.

**Statistical inference:**NLP can make use of statistical inference algorithms. It helps you to produce models that are robust. e.g., containing words or structures which are known to everyone.

**NLP Examples**

Today, Natual process learning technology is widely used technology.

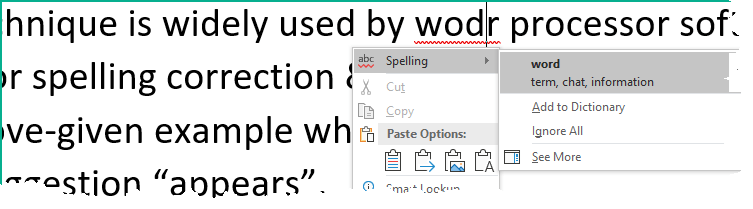
Here, are common Application' of NLP:

**Information retrieval & Web Search**

Google, Yahoo, Bing, and other search engines base their machine translation technology on NLP deep learning models. It allows algorithms to read text on a webpage, interpret its meaning and translate it to another language.

**Grammar Correction:**

NLP technique is widely used by word processor software like MS-word for spelling correction & grammar check.



**Question Answering**

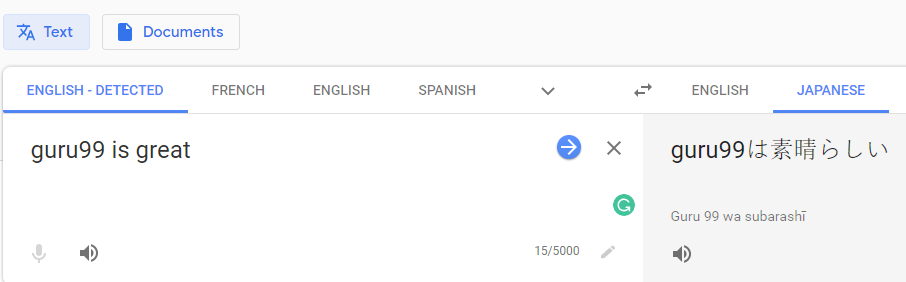
Type in keywords to ask Questions in Natural Language.

**Text Summarization**

The process of summarising important information from a source to produce a shortened version

**Machine Translation**

Use of computer applications to translate text or speech from one natural language to another.



**Sentiment analysis**

NLP helps companies to analyze a large number of reviews on a product. It also allows their customers to give a review of the particular product.

**Future of NLP**

* Human readable natural language processing is the biggest Al- problem. It is all most same as solving the central artificial intelligence problem and making computers as intelligent as people.
* Future computers or machines with the help of NLP will able to learn from the information online and apply that in the real world, however, lots of work need to on this regard.
* Naturla language toolkit or nltk become more effective
* Combined with natural language generation, computers will become more capable of receiving and giving useful and resourceful information or data.

**Natural language vs. Computer Language**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Natural Language** | **Computer Languages** |
| Ambiguous | They are ambiguous in nature. | They are designed to unambiguous. |
| Redundancy | Natural languages employ lots of redundancy. | Formal languages are less redundant. |
| Literalness | Natural languages are made of idiom & metaphor | Formal languages mean exactly what they want to say |

**Advantages of NLP**

* Users can ask questions about any subject and get a direct response within seconds.
* NLP system provides answers to the questions in natural language
* NLP system offers exact answers to the questions, no unnecessary or unwanted information
* The accuracy of the answers increases with the amount of relevant information provided in the question.
* NLP process helps computers communicate with humans in their language and scales other language-related tasks
* Allows you to perform more language-based data compares to a human being without fatigue and in an unbiased and consistent way.
* Structuring a highly unstructured data source

**Disadvantages of NLP**

* Complex Query Language- the system may not be able to provide the correct answer it the question that is poorly worded or ambiguous.
* The system is built for a single and specific task only; it is unable to adapt to new domains and problems because of limited functions.
* NLP system doesn't have a user interface which lacks features that allow users to further interact with the system

**Summary**

* Natural Language Processing is a branch of AI which helps computers to understand, interpret and manipulate human language
* NLP started when Alan Turing published an article called "Machine and Intelligence".
* NLP never focuses on voice modulation; it does draw on contextual patterns
* Five essential components of Natural Language processing are 1) Morphological and Lexical Analysis 2)Syntactic Analysis 3) Semantic Analysis 4) Discourse Integration 5) Pragmatic Analysis
* Three types of the Natural process writing system are 1)Logographic 2) Syllabic 3) Alphabetic
* Machine learning and Statistical inference are two methods to implementation of Natural Process Learning
* Essential Applications of NLP are Information retrieval & Web Search, Grammar Correction Question Answering, , Text Summarization, Machine Translation, etc.
* Future computers or machines with the help of NLP and Data Science will able to learn from the information online and apply that in the real world, however, lots of work need to on this regard
* NLP is are ambiguous while open source computer language is designed to unambiguous
* The biggest advantage of the NLP system is that it offers exact answers to the questions, no unnecessary or unwanted information
* The biggest draw back of the NLP system is built for a single and specific task only so it is unable to adapt to new domains and problems because of limited functions

**How to Download & Install NLTK on Windows/Mac**

In this tutorial, you will learn –

* [Installing NLTK in Windows](https://www.guru99.com/download-install-nltk.html#1)
* [Installing Python in Windows](https://www.guru99.com/download-install-nltk.html#2)
* [Installing NLTK in Mac/Linux](https://www.guru99.com/download-install-nltk.html#3)
* [Installing NLTK through Anaconda](https://www.guru99.com/download-install-nltk.html#4)
* [NLTK Dataset](https://www.guru99.com/download-install-nltk.html#5)
* [How to Download all packages of NLTK](https://www.guru99.com/download-install-nltk.html#6)
* [Running the NLP Script](https://www.guru99.com/download-install-nltk.html#7)
* [How to Run NLTK Script](https://www.guru99.com/download-install-nltk.html#8)

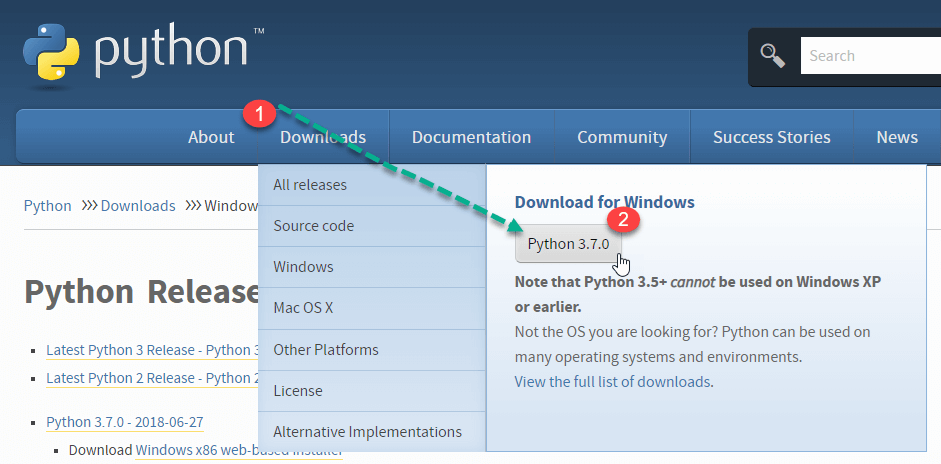
**Installing NLTK in Windows**

In this part, we will learn that how to make setup NLTK via terminal (Command prompt in windows).

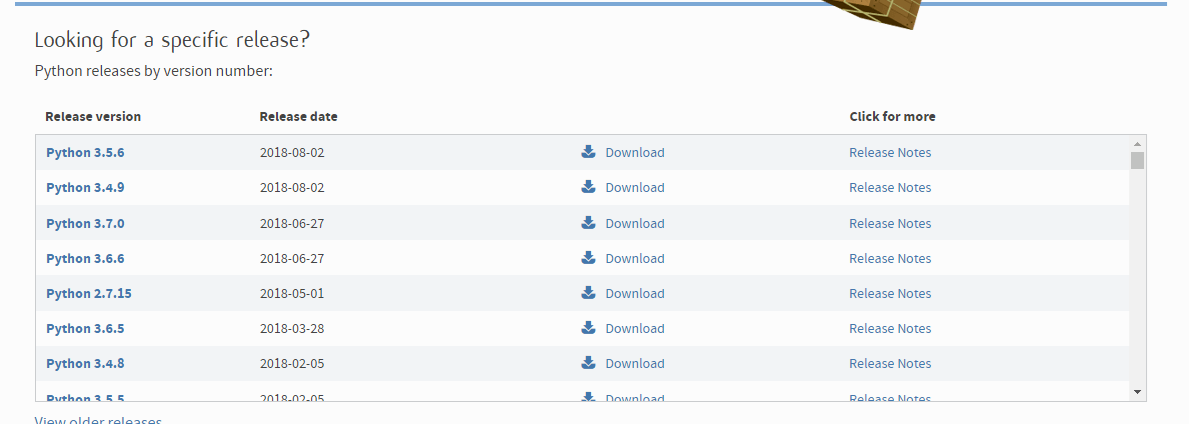
The instruction given below are based on the assumption that you don't have python installed. So, first step is to install python.

### Installing Python in Windows:

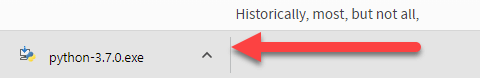
**Step 1)**Go to link [**https://www.python.org/downloads/**](https://www.python.org/downloads/)**,**and select the latest version for windows.



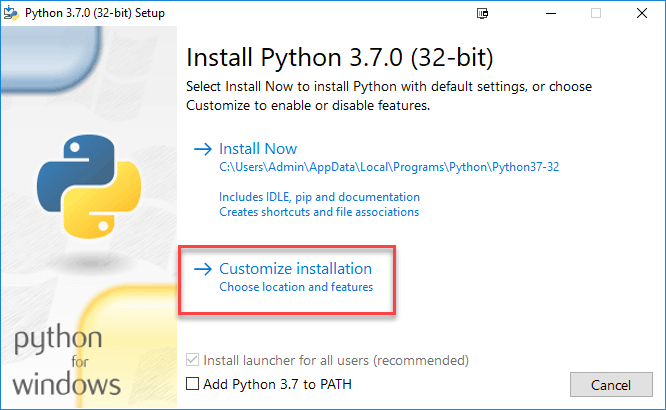
**Note**: If you don't want to download the latest version, you can visit the download tab and see all releases.



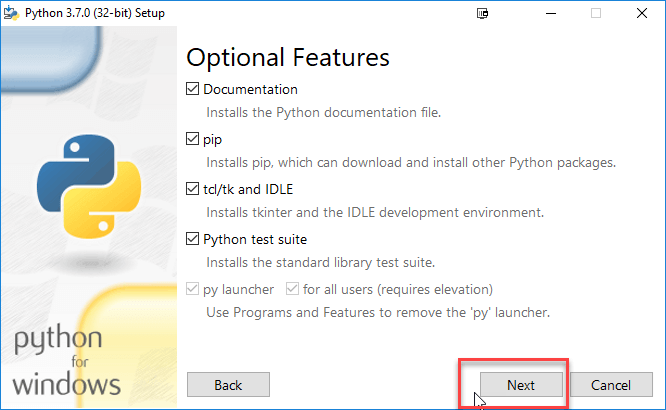
**Step 2)**Click on the Downloaded File



**Step 3)**Select Customize Installation

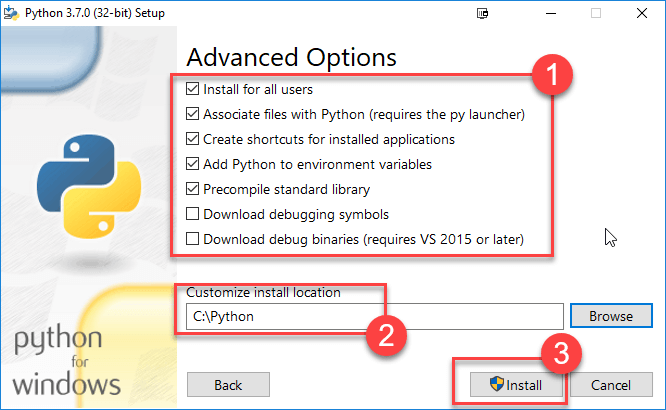


**Step 4)**Click NEXT

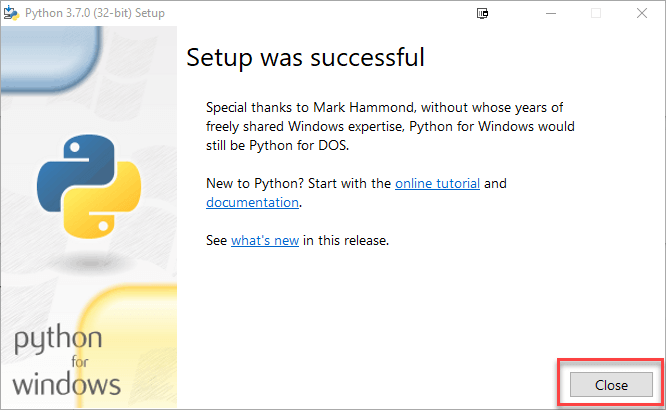


**Step 5)**In next screen

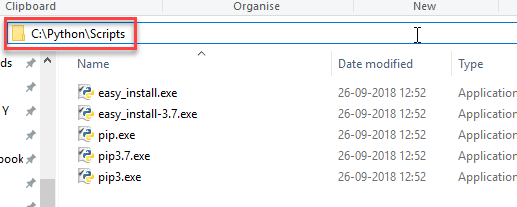
1. Select the advanced options
2. Give a Custom install location. In my case, a folder on C drive is chosen for ease in operation
3. Click Install



**Step 6)**Click Close button once install is done.



**Step 7)**Copy the path of your Scripts folder.

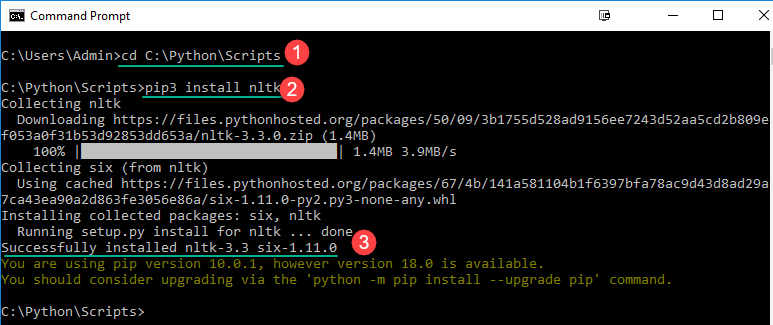


**Step 8)**In windows command prompt

* Navigate to the location of the pip folder
* Enter command to install NLTK

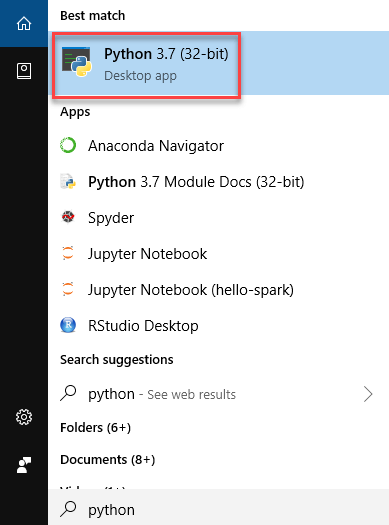
pip3 install nltk

* Installation should be done successfully



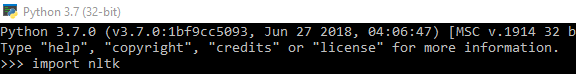
**NOTE**: For Python2 use the commandpip2 install nltk

**Step 9)**In Windows Start Menu, search and open PythonShell



**Step 10)**You can verify whether the installation is accurate supplying the below command

import nltk



If you see no error, Installation is complete.

## Installing NLTK in Mac/Linux

Installing NLTK in Mac/Unix requires python package manager pip to install nltk. If pip is not installed, please follow the below instructions to complete the process

**Step1)** Update the package index by typing the below command

sudo apt update

**Step2)** Installing pip for Python 3:

sudo apt install python3-pip

You can also install pip using easy\_install.

sudo apt-get install python-setuptools python-dev build-essential

Now easy\_install is installed. Run the below command to install pip

sudo easy\_install pip

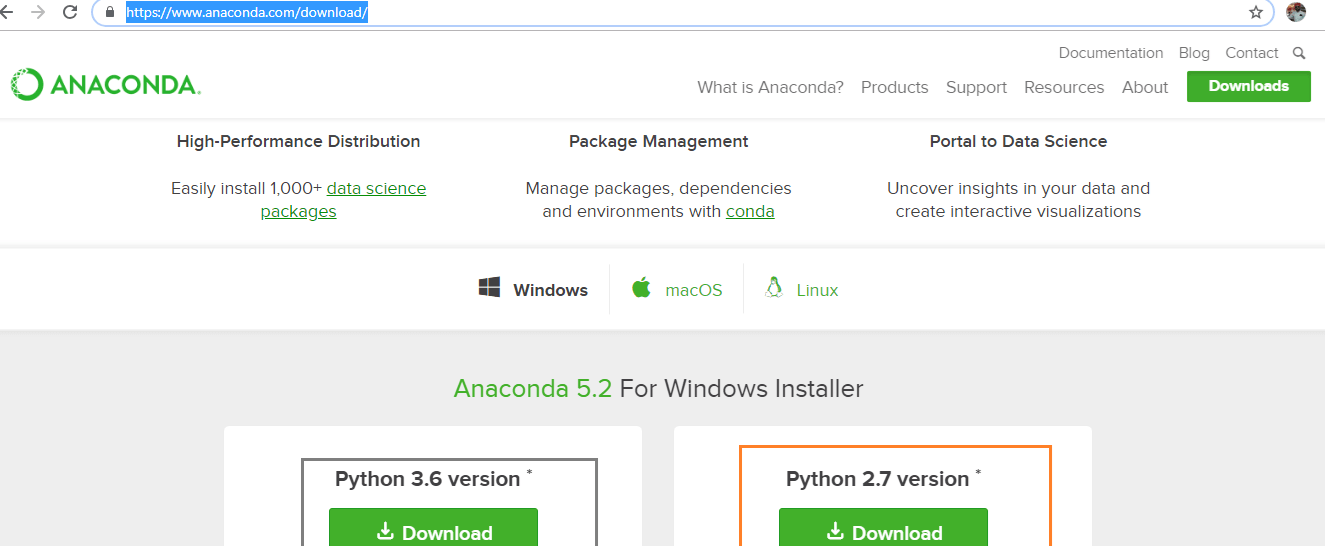
**Step3)**Use following command to install NLTK

sudo pip install -U nltk

sudo pip3 install -U nltk

## Installing NLTK through Anaconda

**Step1)** Please install anaconda (which can also be used to install different packages) by visiting <https://www.anaconda.com/download/> and select which version of python you need to install for anaconda.



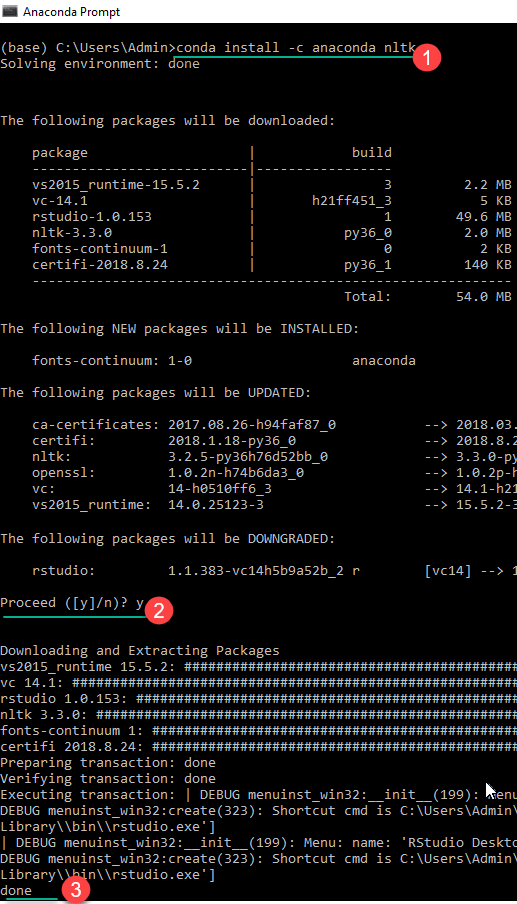
Note: Refer to this tutorial for detailed steps to [install anaconda](https://www.guru99.com/download-install-r-rstudio.html)

**Step 2)**In the Anaconda prompt,

1. Enter command

conda install -c anaconda nltk

1. Review the package upgrade, downgrade, install information and enter yes
2. NLTK is downloaded and installed



**NLTK Dataset**

NLTK module has many datasets available that you need to download to use. More technically it is called **corpus**. Some of the examples are **stopwords**, **gutenberg**, **framenet\_v15**, **large\_grammars**and so on.

**How to Download all packages of NLTK**

**Step 1)**Run the Python interpreter in Windows or Linux

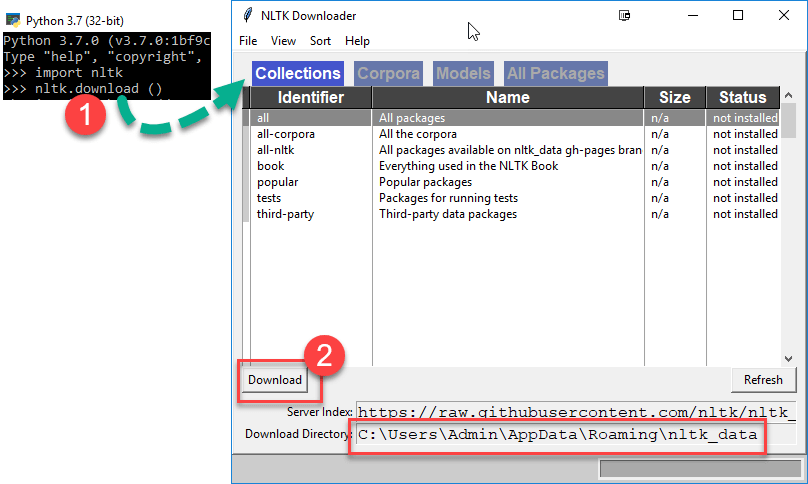
**Step 2)**

1. Enter the commands

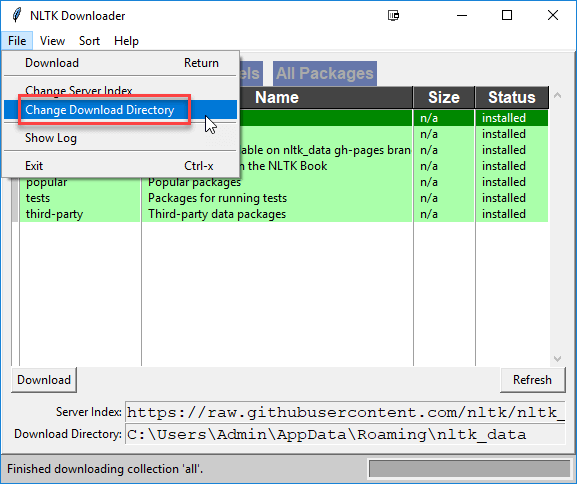
import nltk

nltk.download ()

1. NLTK Downloaded Window Opens. Click the Download Button to download the dataset. This process will take time, based on your internet connection



**NOTE:**You can change the download location by Clicking File> Change Download Directory

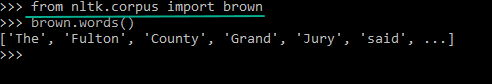


**Step 3)**To test the installed data use the following code

>>> from nltk.corpus import brown

>>>brown.words()

['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]



## Running the NLP Script

We are going to discuss how NLP script will be executed on our local PC. There are many libraries for Natural Language Processing present in the market. So choosing a library depends on fitting your requirements. Here is the list of [NLP libraries](http://www.aisangam.com/blog/natural-language-toolkit-guide/).

## How to Run NLTK Script

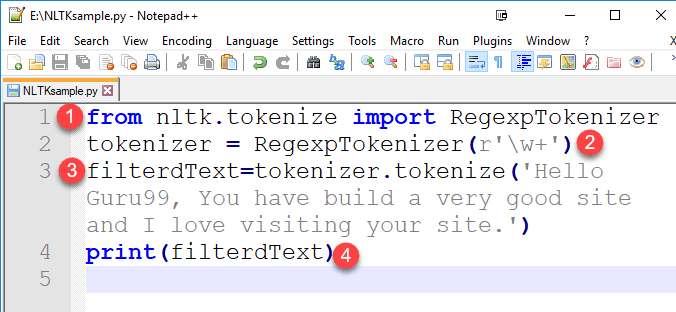
**Step1)**In your favorite code editor, copy the code and save the file as**"**NLTKsample.py**"**

from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'\w+')

filterdText=tokenizer.tokenize('Hello Guru99, You have build a very good site and I love visiting your site.')

print(filterdText)

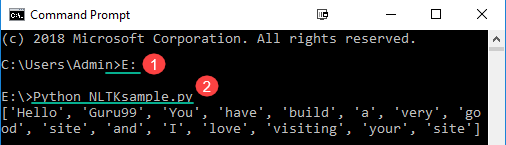


**Code Explanation:**

1. In this program, the objective was to remove all type of punctuations from given text. We imported "RegexpTokenizer" which is a module of NLTK. It removes all the expression, symbol, character, numeric or any things whatever you want.
2. You just have passed the regular Expression to the "RegexpTokenizer" module.
3. Further, we tokenized the word using "tokenize" module. The output is stored in the "filterdText" variable.
4. And printed them using "print()."

**Step2)**In the command prompt

* Navigate to the location where you have saved the file
* Run the command Python NLTKsample.py



This will show output as :

['Hello', 'Guru99', 'You', 'have', 'build', 'a', 'very', 'good', 'site', 'and', 'I', 'love', 'visiting', 'your', 'site']

# Tokenize Words and Sentences with NLTK

**What is Tokenization?**

Tokenization is the process by which big quantity of text is divided into smaller parts called **tokens**.

Natural language processing is used for building applications such as Text classification, intelligent chatbot, sentimental analysis, language translation, etc. It becomes vital to understand the pattern in the text to achieve the above-stated purpose. **These tokens are very useful for finding such patterns as well as is considered as a base step for stemming and lemmatization.**

For the time being, don't worry about stemming and lemmatization but treat them as steps for textual data cleaning using NLP (Natural language processing). We will discuss stemming and lemmatization later in the tutorial. Tasks such as **Text classification or spam filtering** makes use of NLP along with deep learning libraries such as Keras and Tensorflow.

Natural Language toolkit has very important module **tokenize** which further comprises of sub-modules

1. word tokenize
2. sentence tokenize

**Tokenization of words**

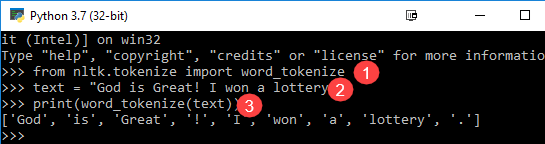
We use the method **word\_tokenize()** to split a sentence into words. The output of word tokenization can be converted to Data Frame for better text understanding in machine learning applications. It can also be provided as input for further text cleaning steps such as punctuation removal, numeric character removal or stemming. Machine learning models need numeric data to be trained and make a prediction. Word tokenization becomes a crucial part of the text (string) to numeric data conversion. Please read about [Bag of Words or CountVectorizer](https://en.wikipedia.org/wiki/Bag-of-words_model). Please refer to below example to understand the theory better.

from nltk.tokenize import word\_tokenize

text = "God is Great! I won a lottery."

print(word\_tokenize(text))

Output: ['God', 'is', 'Great', '!', 'I', 'won', 'a', 'lottery', '.']



**Code Explanation**

1. word\_tokenize module is imported from the NLTK library.
2. A variable "text" is initialized with two sentences.
3. Text variable is passed in word\_tokenize module and printed the result. This module breaks each word with punctuation which you can see in the output.

**Tokenization of Sentences**

Sub-module available for the above is sent\_tokenize. An obvious question in your mind would be **why sentence tokenization is needed when we have the option of word tokenization**. Imagine you need to count average words per sentence, how you will calculate? For accomplishing such a task, you need both sentence tokenization as well as words to calculate the ratio. Such output serves as an important feature for machine training as the answer would be numeric.

Check the below example to learn how sentence tokenization is different from words tokenization.

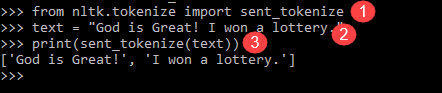
from nltk.tokenize import sent\_tokenize

text = "God is Great! I won a lottery."

print(sent\_tokenize(text))

Output: ['God is Great!', 'I won a lottery ']

We have **12**words and **two sentences** for the same input.



**Explanation of the program:**

1. In a line like the previous program, imported the sent\_tokenize module.
2. We have taken the same sentence. Further sent module parsed that sentences and show output. It is clear that this function breaks each sentence.

Above examples are good settings stones to understand the mechanics of the word and sentence tokenization.

# POS (Part-Of-Speech) Tagging & Chunking with NLTK

## POS Tagging

Parts of speech Tagging is responsible for reading the text in a language and assigning some specific token (Parts of Speech) to each word.

e.g.

Input: Everything to permit us.

Output: [('Everything', NN),('to', TO), ('permit', VB), ('us', PRP)]

**Steps Involved:**

* Tokenize text (word\_tokenize)
* apply pos\_tag to above step that is nltk.pos\_tag(tokenize\_text)

**Some examples are as below:**

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| CC | coordinating conjunction |
| CD | cardinal digit |
| DT | determiner |
| EX | existential there |
| FW | foreign word |
| IN | preposition/subordinating conjunction |
| JJ | adjective (large) |
| JJR | adjective, comparative (larger) |
| JJS | adjective, superlative (largest) |
| LS | list market |
| MD | modal (could, will) |
| NN | noun, singular (cat, tree) |
| NNS | noun plural (desks) |
| NNP | proper noun, singular (sarah) |
| NNPS | proper noun, plural (indians or americans) |
| PDT | predeterminer (all, both, half) |
| POS | possessive ending (parent\ 's) |
| PRP | personal pronoun (hers, herself, him,himself) |
| PRP$ | possessive pronoun (her, his, mine, my, our ) |
| RB | adverb (occasionally, swiftly) |
| RBR | adverb, comparative (greater) |
| RBS | adverb, superlative (biggest) |
| RP | particle (about) |
| TO | infinite marker (to) |
| UH | interjection (goodbye) |
| VB | verb (ask) |
| VBG | verb gerund (judging) |
| VBD | verb past tense (pleaded) |
| VBN | verb past participle (reunified) |
| VBP | verb, present tense not 3rd person singular(wrap) |
| VBZ | verb, present tense with 3rd person singular (bases) |
| WDT | wh-determiner (that, what) |
| WP | wh- pronoun (who) |
| WRB | wh- adverb (how) |

POS tagger is used to assign grammatical information of each word of the sentence. Installing, Importing and downloading all the packages of NLTK is complete.

## Chunking

Chunking is used to add more structure to the sentence by following parts of speech (POS) tagging. It is also known as shallow parsing. The resulted group of words is called "**chunks**." In shallow parsing, there is maximum one level between roots and leaves while deep parsing comprises of more than one level. Shallow Parsing is also called light parsing or chunking.

The primary usage of chunking is to make a group of "noun phrases." The parts of speech are combined with regular expressions.

**Rules for Chunking:**

There are no pre-defined rules, but you can combine them according to need and requirement.

For example, you need to tag Noun, verb (past tense), adjective, and coordinating junction from the sentence. You can use the rule as below

chunk:{<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?}

Following table shows what the various symbol means:

|  |  |
| --- | --- |
| **Name of symbol** | **Description** |
| . | Any character except new line |
| \* | Match 0 or more repetitions |
| ? | Match 0 or 1 repetitions |

Now Let us write the code to understand rule better

from nltk import pos\_tag

from nltk import RegexpParser

text ="learn php from guru99 and make study easy".split()

print("After Split:",text)

tokens\_tag = pos\_tag(text)

print("After Token:",tokens\_tag)

patterns= """mychunk:{<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?}"""

chunker = RegexpParser(patterns)

print("After Regex:",chunker)

output = chunker.parse(tokens\_tag)

print("After Chunking",output)

**Output**

After Split: ['learn', 'php', 'from', 'guru99', 'and', 'make', 'study', 'easy']

After Token: [('learn', 'JJ'), ('php', 'NN'), ('from', 'IN'), ('guru99', 'NN'), ('and', 'CC'), ('make', 'VB'), ('study', 'NN'), ('easy', 'JJ')]

After Regex: chunk.RegexpParser with 1 stages:

RegexpChunkParser with 1 rules:

<ChunkRule: '<NN.?>\*<VBD.?>\*<JJ.?>\*<CC>?'>

After Chunking (S

(mychunk learn/JJ)

(mychunk php/NN)

from/IN

(mychunk guru99/NN and/CC)

make/VB

(mychunk study/NN easy/JJ))

The conclusion from the above example: "make" is a verb which is not included in the rule, so it is not tagged as mychunk

#### Use Case of Chunking

Chunking is used for entity detection. An entity is that part of the sentence by which machine get the value for any intention

Example:

Temperature of New York.

Here Temperature is the intention and New York is an entity.

In other words, chunking is used as selecting the subsets of tokens. Please follow the below code to understand how chunking is used to select the tokens. In this example, you will see the graph which will correspond to a chunk of a noun phrase. We will write the code and draw the graph for better understanding.

### Code to Demonstrate Use Case

import nltk

text = "learn php from guru99"

tokens = nltk.word\_tokenize(text)

print(tokens)

tag = nltk.pos\_tag(tokens)

print(tag)

grammar = "NP: {<DT>?<JJ>\*<NN>}"

cp =nltk.RegexpParser(grammar)

result = cp.parse(tag)

print(result)

result.draw() # It will draw the pattern graphically which can be seen in Noun Phrase chunking

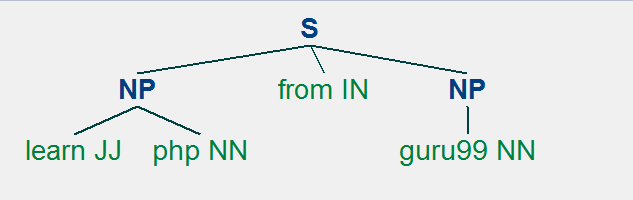
Output:

['learn', 'php', 'from', 'guru99'] -- These are the tokens

[('learn', 'JJ'), ('php', 'NN'), ('from', 'IN'), ('guru99', 'NN')] -- These are the pos\_tag

(S (NP learn/JJ php/NN) from/IN (NP guru99/NN)) -- Noun Phrase Chunking

**Graph**



**Noun Phrase chunking Graph**

From the graph, we can conclude that "learn" and "guru99" are two different tokens but are categorized as Noun Phrase whereas token "from" does not belong to Noun Phrase.

Chunking is used to categorize different tokens into the same chunk. The result will depend on grammar which has been selected. Further chunking is used to tag patterns and to explore text corpora.

# Stemming and Lemmatization with Python NLTK

## What is Stemming?

Stemming is a kind of normalization for words. Normalization is a technique where a set of words in a sentence are converted into a sequence to shorten its lookup. The words which have the same meaning but have some variation according to the context or sentence are normalized.

In another word, there is one root word, but there are many variations of the same words. For example, the root word is "eat" and it's variations are "eats, eating, eaten and like so". In the same way, with the help of Stemming, we can find the root word of any variations.

**For example**

He was riding.

He was taking the ride.

In the above two sentences, the meaning is the same, i.e., riding activity in the past. A human can easily understand that both meanings are the same. But for machines, both sentences are different. Thus it became hard to convert it into the same data row. In case we do not provide the same data-set, then machine fails to predict. So it is necessary to differentiate the meaning of each word to prepare the dataset for machine learning. And here stemming is used to categorize the same type of data by getting its root word.

Let's implement this with a Python program.NLTK has an algorithm named as "PorterStemmer". This algorithm accepts the list of tokenized word and stems it into root word.

**Program for understanding Stemming**

from nltk.stem import PorterStemmer

e\_words= ["wait", "waiting", "waited", "waits"]

ps =PorterStemmer()

for w in e\_words:

rootWord=ps.stem(w)

print(rootWord)

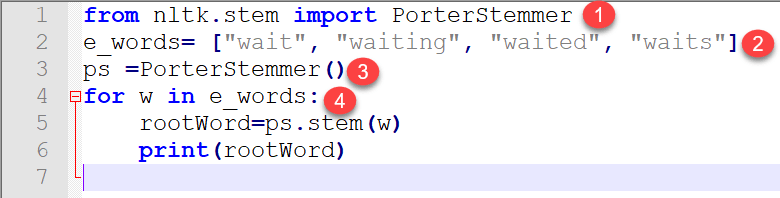
**Output**:

wait

wait

wait

wait



**Code Explanation:**

* There is a stem module in NLTk which is imported. If ifyou import the complete module, then the program becomes heavy as it contains thousands of lines of codes. So from the entire stem module, we only imported "PorterStemmer."
* We prepared a dummy list of variation data of the same word.
* An object is created which belongs to class nltk.stem.porter.PorterStemmer.
* Further, we passed it to PorterStemmer one by one using "for" loop. Finally, we got output root word of each word mentioned in the list.

From the above explanation, it can also be concluded that stemming is considered as an important preprocessing step because it removed redundancy in the data and variations in the same word. As a result, data is filtered which will help in better machine training.

Now we pass a complete sentence and check for its behavior as an output.

**Program:**

from nltk.stem import PorterStemmer

from nltk.tokenize import sent\_tokenize, word\_tokenize

sentence="Hello Guru99, You have to build a very good site and I love visiting your site."

words = word\_tokenize(sentence)

ps = PorterStemmer()

for w in words:

rootWord=ps.stem(w)

print(rootWord)

**Output:**

hello

guru99

,

you

have

build

a

veri

good

site

and

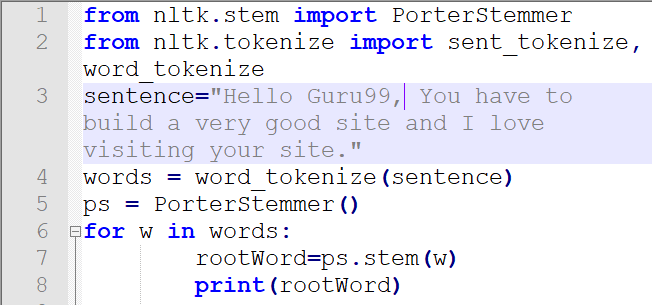
I

love

visit

your

site



**Code Explanation**

* Package PorterStemer is imported from module stem
* Packages for tokenization of sentence as well as words are imported
* A sentence is written which is to be tokenized in the next step.
* Word tokenization is implemented in this step.
* An object for PorterStemmer is created here.
* Loop is run and stemming of each word is done using the object created in the code line 5

**Conclusion:**

Stemming is a data-preprocessing module. The English language has many variations of a single word. These variations create ambiguity in machine learning training and prediction. To create a successful model, it's vital to filter such words and convert to the same type of sequenced data using stemming. Also, this is an important technique to get row data from a set of sentence and removal of redundant data also known as normalization.

**What is Lemmatization?**

Lemmatization is the algorithmic process of finding the lemma of a word depending on their meaning. Lemmatization usually refers to the morphological analysis of words, which aims to remove inflectional endings. It helps in returning the base or dictionary form of a word, which is known as the lemma. The NLTK Lemmatization method is based on WorldNet's built-in morph function. Text preprocessing includes both stemming as well as lemmatization. Many people find the two terms confusing. Some treat these as same, but there is a difference between these both. Lemmatization is preferred over the former because of the below reason.

**Why is Lemmatization better than Stemming?**

Stemming algorithm works by cutting the suffix from the word. In a broader sense cuts either the beginning or end of the word.

On the contrary, Lemmatization is a more powerful operation, and it takes into consideration morphological analysis of the words. It returns the lemma which is the base form of all its inflectional forms. In-depth linguistic knowledge is required to create dictionaries and look for the proper form of the word. Stemming is a general operation while lemmatization is an intelligent operation where the proper form will be looked in the dictionary. Hence, lemmatization helps in forming better machine learning features.

**Code to distinguish between Lemmatization and Stemming**

Stemming code

import nltk

from nltk.stem.porter import PorterStemmer

porter\_stemmer = PorterStemmer()

text = "studies studying cries cry"

tokenization = nltk.word\_tokenize(text)

for w in tokenization:

print("Stemming for {} is {}".format(w,porter\_stemmer.stem(w)))

Output:

Stemming for studies is studi

Stemming for studying is studi

Stemming for cries is cri

Stemming for cry is cri

**Lemmatization code**

import nltk

from nltk.stem import WordNetLemmatizer

wordnet\_lemmatizer = WordNetLemmatizer()

text = "studies studying cries cry"

tokenization = nltk.word\_tokenize(text)

for w in tokenization:

print("Lemma for {} is {}".format(w, wordnet\_lemmatizer.lemmatize(w)))

Output:

Lemma for studies is study

Lemma for studying is studying

Lemma for cries is cry

Lemma for cry is cry

**Discussion of output:**

If you look stemming for studies and studying, output is same (studi) but lemmatizer provides different lemma for both tokens study for studies and studying for studying. So when we need to make feature set to train machine, it would be great if lemmatization is preferred.

**Use Case of Lemmatizer:**

Lemmatizer minimizes text ambiguity. Example words like bicycle or bicycles are converted to base word bicycle. Basically, it will convert all words having the same meaning but different representation to their base form. It reduces the word density in the given text and helps in preparing the accurate features for training machine. Cleaner the data, the more intelligent and accurate your machine learning model, will be. Lemmatizerwill also saves memory as well as computational cost.

**Real Time example showing use of Wordnet Lemmatization and POS Tagging in Python**

from nltk.corpus import wordnet as wn

from nltk.stem.wordnet import WordNetLemmatizer

from nltk import word\_tokenize, pos\_tag

from collections import defaultdict

tag\_map = defaultdict(lambda : wn.NOUN)

tag\_map['J'] = wn.ADJ

tag\_map['V'] = wn.VERB

tag\_map['R'] = wn.ADV

text = "guru99 is a totally new kind of learning experience."

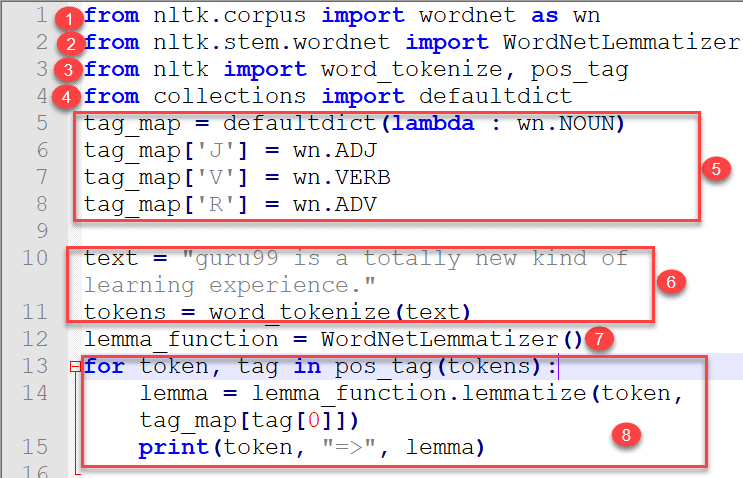
tokens = word\_tokenize(text)

lemma\_function = WordNetLemmatizer()

for token, tag in pos\_tag(tokens):

lemma = lemma\_function.lemmatize(token, tag\_map[tag[0]])

print(token, "=>", lemma)



**Code Explanation**

* Firstly, the corpus reader wordnet is imported.
* WordNetLemmatizer is imported from wordnet
* Word tokenize as well as parts of speech tag are imported from nltk
* Default Dictionary is imported from collections
* Dictionary is created where pos\_tag (first letter) are the key values whose values are mapped with the value from wordnet dictionary. We have taken the only first letter as we will use it later in the loop.
* Text is written and is tokenized.
* Object lemma\_function is created which will be used inside the loop
* Loop is run and lemmatize will take two arguments one is token and other is a mapping of pos\_tag with wordnet value.

Output:

guru99 => guru99

is => be

totally => totally

new => new

kind => kind

of => of

learning => learn

experience => experience

. => .

Lemmatization has a close relation with wordnet dictionary, so it is essential to study this topic, so we keep this as the next topic

# WordNet with NLTK: Finding Synonyms for words in Python

## What is Wordnet?

Wordnet is an NLTK corpus reader, a lexical database for English. It can be used to find the meaning of words, synonym or antonym. One can define it as a semantically oriented dictionary of English. It is imported with the following command:

from nltk.corpus import wordnet as guru

Stats reveal that there are **155287 words and 117659 synonym** sets included with English WordNet.

Different methods available with WordNet can be found by typing dir(guru)

['\_LazyCorpusLoader\_\_args', '\_LazyCorpusLoader\_\_kwargs', '\_LazyCorpusLoader\_\_load', '\_LazyCorpusLoader\_\_name', '\_LazyCorpusLoader\_\_reader\_cls', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattr\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_unicode\_\_', '\_\_weakref\_\_', '\_unload', 'subdir', 'unicode\_repr']

Let us understand some of the features available with the wordnet:

**Synset**: It is also called as synonym set or collection of synonym words. Let us check a example

from nltk.corpus import wordnet

syns = wordnet.synsets("dog")

print(syns)

Output:

[Synset('dog.n.01'), Synset('frump.n.01'), Synset('dog.n.03'), Synset('cad.n.01'), Synset('frank.n.02'), Synset('pawl.n.01'), Synset('andiron.n.01'), Synset('chase.v.01')]

**Lexical Relations**: These are semantic relations which are reciprocated. If there is a relationship between {x1,x2,...xn} and {y1,y2,...yn} then there is also relation between {y1,y2,...yn} and {x1,x2,...xn}. For example Synonym is the opposite of antonym or hypernyms and hyponym are type of lexical concept.

Let us write a program using python to find synonym and antonym of word "active" using Wordnet.

from nltk.corpus import wordnet

synonyms = []

antonyms = []

for syn in wordnet.synsets("active"):

for l in syn.lemmas():

synonyms.append(l.name())

if l.antonyms():

antonyms.append(l.antonyms()[0].name())

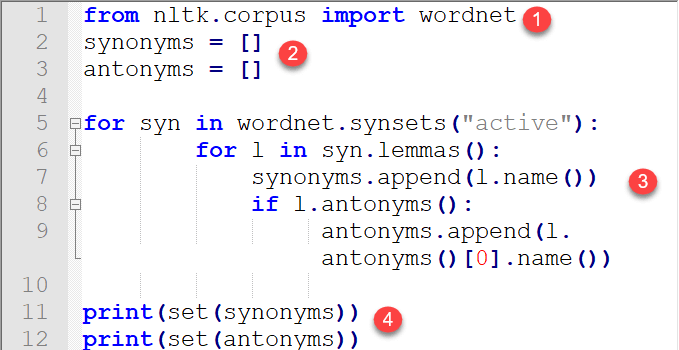
print(set(synonyms))

print(set(antonyms))

**The output of the code:**

{'dynamic', 'fighting', 'combat-ready', 'active\_voice', 'active\_agent', 'participating', 'alive', 'active'} -- Synonym

{'stative', 'passive', 'quiet', 'passive\_voice', 'extinct', 'dormant', 'inactive'} -- Antonym



**Explanation of the code**

1. Wordnet is a corpus, so it is imported from the ntlk.corpus
2. List of both synonym and antonym is taken as empty which will be used for appending
3. Synonyms of the word active are searched in the module synsets and are appended in the list synonyms. The same process is repeated for the second one.
4. Output is printed

**Conclusion:**

WordNet is a lexical database that has been used by a major search engine. From the WordNet, information about a given word or phrase can be calculated such as

* synonym (words having the same meaning)
* hypernyms (The generic term used to designate a class of specifics (i.e., meal is a breakfast), hyponyms (rice is a meal)
* holonyms (proteins, carbohydrates are part of meal)
* meronyms (meal is part of daily food intake)

WordNet also provides information on co-ordinate terms, derivates, senses and more. It is used to find the similarities between any two words. It also holds information on the results of the related word. In short or nutshell one can treat it as Dictionary or Thesaurus. Going deeper in wordnet, it is divided into four total subnets such as

1. Noun
2. Verb
3. Adjective
4. Adverb

It can be used in the area of artificial intelligence for text analysis. With the help of Wordnet, you can create your corpus for spelling checking, language translation, Spam detection and many more.

In the same way, you can use this corpus and mold it to work some dynamic functionality. This is just like ready to made corpus for you. You can use it in your way.

# Tagging Problems and Hidden Markov Model

## Tagging Sentences

Tagging Sentence in a broader sense refers to the addition of labels of the verb, noun,etc.by the context of the sentence. Identification of POS tags is a complicated process. Thus generic tagging of POS is manually not possible as some words may have different (ambiguous) meanings according to the structure of the sentence. Conversion of text in the form of list is an important step before tagging as each word in the list is looped and counted for a particular tag. Please see the below code to understand it better

import nltk

text = "Hello Guru99, You have to build a very good site, and I love visiting your site."

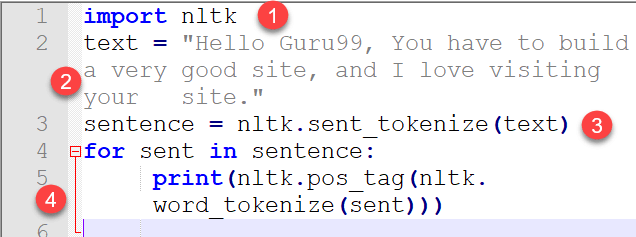
sentence = nltk.sent\_tokenize(text)

for sent in sentence:

print(nltk.pos\_tag(nltk.word\_tokenize(sent)))

OUTPUT

[('Hello', 'NNP'), ('Guru99', 'NNP'), (',', ','), ('You', 'PRP'), ('have', 'VBP'), ('build', 'VBN'), ('a', 'DT'), ('very', 'RB'), ('good', 'JJ'), ('site', 'NN'), ('and', 'CC'), ('I', 'PRP'), ('love', 'VBP'), ('visiting', 'VBG'), ('your', 'PRP$'), ('site', 'NN'), ('.', '.')]



**Code Explanation**

1. Code to import nltk (Natural language toolkit which contains submodules such as sentence tokenize and word tokenize.)
2. Text whose tags are to be printed.
3. Sentence Tokenization
4. For loop is implemented where words are tokenized from sentence and tag of each word is printed as output.

In Corpus there are two types of POS taggers:

* Rule-Based
* Stochastic POS Taggers

**1.Rule-Based POS Tagger:**For the words having ambiguous meaning, rule-based approach on the basis of contextual information is applied. It is done so by checking or analyzing the meaning of the preceding or the following word. Information is analyzed from the surrounding of the word or within itself. Therefore words are tagged by the grammatical rules of a particular language such as capitalization and punctuation. e.g., Brill's tagger.

**2.Stochastic POS Tagger:**Different approaches such as frequency or probability are applied under this method. If a word is mostly tagged with a particular tag in training set then in the test sentence it is given that particular tag. The word tag is dependent not only on its own tag but also on the previous tag. This method is not always accurate. Another way is to calculate the probability of occurrence of a specific tag in a sentence. Thus the final tag is calculated by checking the highest probability of a word with a particular tag.

**Hidden Markov Model:**

Tagging Problems can also be modeled using HMM. It treats input tokens to be observable sequence while tags are considered as hidden states and goal is to determine the hidden state sequence. For example **x = x1,x2,............,xn** where x is a sequence of tokens while **y = y1,y2,y3,y4.........yn**is the hidden sequence.

**How HMM Model Works?**

HMM uses join distribution which is P(x, y) where x is the input sequence/ token sequence and y is tag sequence.

Tag Sequence for x will be argmaxy1....ynp(x1,x2,....xn,y1,y2,y3,.....). We have categorized tags from the text, but stats of such tags are vital. So the next part is counting these tags for statistical study.

# Counting POS Tags, Frequency Distribution & Collocations in NLTK

## COUNTING POS TAGS

We have discussed various **pos\_tag** in the previous section. In this particular tutorial, you will study how to count these tags. Counting tags are crucial for text classification as well as preparing the features for the Natural language-based operations. I will be discussing with you the approach which guru99 followed while preparing code along with a discussion of output. Hope this will help you.

How to count Tags:

Here first we will write working code and then we will write different steps to explain the code.

from collections import Counter

import nltk

text = " Guru99 is one of the best sites to learn WEB, SAP, Ethical Hacking and much more online."

lower\_case = text.lower()

tokens = nltk.word\_tokenize(lower\_case)

tags = nltk.pos\_tag(tokens)

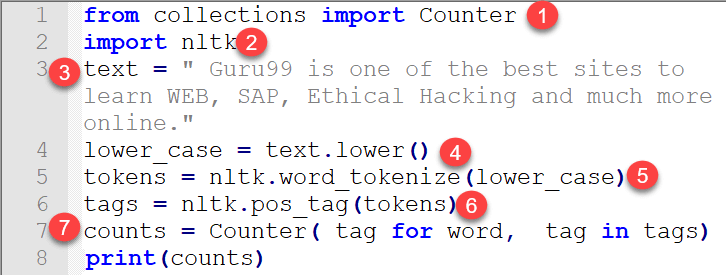
counts = Counter( tag for word, tag in tags)

print(counts)

**Output:**

Counter({'NN': 5, ',': 2, 'TO': 1, 'CC': 1, 'VBZ': 1, 'NNS': 1, 'CD': 1, '.': 1, 'DT': 1, 'JJS': 1, 'JJ': 1, 'JJR': 1, 'IN': 1, 'VB': 1, 'RB': 1})

**Elaboration of the code**



1. To count the tags, you can use the package Counter from the collection's module. A counter is a dictionary subclass which works on the principle of key-value operation. It is an unordered collection where elements are stored as a dictionary key while the count is their value.
2. Import nltk which contains modules to tokenize the text.
3. Write the text whose pos\_tag you want to count.
4. Some words are in upper case and some in lower case, so it is appropriate to transform all the words in the lower case before applying tokenization.
5. Pass the words through word\_tokenize from nltk.
6. Calculate the pos\_tag of each token

Output = [('guru99', 'NN'), ('is', 'VBZ'), ('one', 'CD'), ('of', 'IN'), ('the', 'DT'), ('best', 'JJS'), ('site', 'NN'), ('to', 'TO'), ('learn', 'VB'), ('web', 'NN'), (',', ','), ('sap', 'NN'), (',', ','), ('ethical', 'JJ'), ('hacking', 'NN'), ('and', 'CC'), ('much', 'RB'), ('more', 'JJR'), ('online', 'JJ')]

1. Now comes the role of dictionary counter. We have imported in the code line 1. Words are the key and tags are the value and counter will count each tag total count present in the text.

**Frequency Distribution**

Frequency Distribution is referred to as the number of times an outcome of an experiment occurs. It is used to find the frequency of each word occurring in a document. It uses **FreqDistclass** and defined by **the nltk.probabilty**module.

A frequency distribution is usually created by counting the samples of repeatedly running the experiment. The no of counts is incremented by one, each time. E.g.

freq\_dist = FreqDist()

for the token in the document:

freq\_dist.inc(token.type())

For any word, we can check how many times it occurred in a particular document. E.g.

1. **Count Method:**freq\_dist.count('and')This expression returns the value of the number of times 'and' occurred. It is called the count method.
2. **Frequency Method:**freq\_dist.freq('and')This the expression returns frequency of a given sample.

We will write a small program and will explain its working in detail. We will write some text and will calculate the frequency distribution of each word in the text.

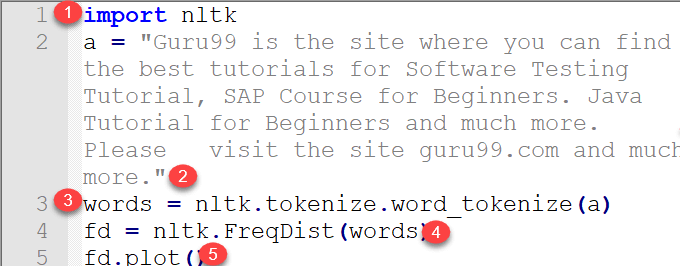
import nltk

a = "Guru99 is the site where you can find the best tutorials for Software Testing Tutorial, SAP Course for Beginners. Java Tutorial for Beginners and much more. Please visit the site guru99.com and much more."

words = nltk.tokenize.word\_tokenize(a)

fd = nltk.FreqDist(words)

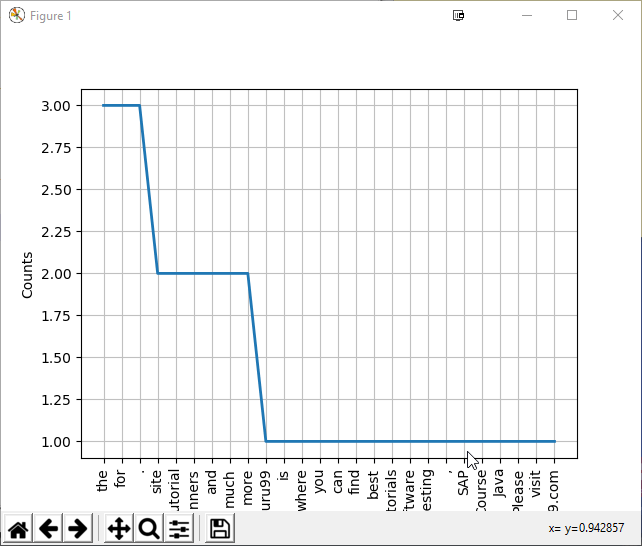
fd.plot()



**Explanation of code:**

1. Import nltk module.
2. Write the text whose word distribution you need to find.
3. Tokenize each word in the text which is served as input to FreqDist module of the nltk.
4. Apply each word to nlk.FreqDist in the form of a list
5. Plot the words in the graph using plot()

Please visualize the graph for a better understanding of the text written



**Frequency distribution of each word in the graph**

NOTE: You need to have matplotlib installed to see the above graph

Observe the graph above. It corresponds to counting the occurrence of each word in the text. It helps in the study of text and further in implementing text-based sentimental analysis. In a nutshell, it can be concluded that nltk has a module for counting the occurrence of each word in the text which helps in preparing the stats of natural language features. It plays a significant role in finding the keywords in the text. You can also extract the text from the pdf using libraries like extract, PyPDF2 and feed the text to nlk.FreqDist.

The key term is "tokenize." After tokenizing, it checks for each word in a given paragraph or text document to determine that number of times it occurred. You do not need the NLTK toolkit for this. You can also do it with your own python programming skills. NLTK toolkit only provides a ready-to-use code for the various operations.

Counting each word may not be much useful. Instead one should focus on collocation and bigrams which deals with a lot of words in a pair. These pairs identify useful keywords to better natural language features which can be fed to the machine. Please look below for their details.

**Collocations: Bigrams and Trigrams**

**What is Collocations?**

Collocations are the pairs of words occurring together many times in a document. It is calculated by the number of those pair occurring together to the overall word count of the document.

Consider electromagnetic spectrum with words like ultraviolet rays, infrared rays.

The words ultraviolet and rays are not used individually and hence can be treated as Collocation. Another example is the CT Scan. We don't say CT and Scan separately, and hence they are also treated as collocation.

We can say that finding collocations requires calculating the frequencies of words and their appearance in the context of other words. These specific collections of words require filtering to retain useful content terms. Each gram of words may then be scored according to some association measure, to determine the relative likelihood of each Ingram being a collocation.

Collocation can be categorized into two types-

* **Bigrams c**ombination of two words
* **Trigrams**combinationof three words

Bigrams and Trigrams provide more meaningful and useful features for the feature extraction stage. These are especially useful in text-based sentimental analysis.

**Bigrams Example Code**

import nltk

text = "Guru99 is a totally new kind of learning experience."

Tokens = nltk.word\_tokenize(text)

output = list(nltk.bigrams(Tokens))

print(output)

Output

[('Guru99', 'is'), ('is', 'totally'), ('totally', 'new'), ('new', 'kind'), ('kind', 'of'), ('of', 'learning'), ('learning', 'experience'), ('experience', '.')]

**Trigrams Example Code**

Sometimes it becomes important to see a pair of three words in the sentence for statistical analysis and frequency count. This again plays a crucial role in forming NLP (natural language processing features) as well as text-based sentimental prediction.

The same code is run for calculating the trigrams.

import nltk

text = “Guru99 is a totally new kind of learning experience.”

Tokens = nltk.word\_tokenize(text)

output = list(nltk.trigrams(Tokens))

print(output)

Output

[('Guru99', 'is', 'totally'), ('is', 'totally', 'new'), ('totally', 'new', 'kind'), ('new', 'kind', 'of'), ('kind', 'of', 'learning'), ('of', 'learning', 'experience'), ('learning', 'experience', '.')]

**Word Embedding Tutorial: word2vec using Gensim [EXAMPLE]**

**What is Word Embedding?**

Word Embedding is a type of word representation that allows words with similar meaning to be understood by machine learning algorithms. Technically speaking, it is a mapping of words into vectors of real numbers using the neural network, probabilistic model, or dimension reduction on word co-occurrence matrix. It is language modeling and feature learning technique. Word embedding is a way to perform mapping using a neural network. There are various word embedding models available such as word2vec (Google), Glove (Stanford) and fastest (Facebook).

Word Embedding is also called as distributed semantic model or distributed represented or semantic vector space or vector space model. As you read these names, you come across the word semantic which means categorizing similar words together. For example fruits like apple, mango, banana should be placed close whereas books will be far away from these words. In a broader sense, word embedding will create the vector of fruits which will be placed far away from vector representation of books.

In this tutorial, you will learn

* [What is Word Embedding?](https://www.guru99.com/word-embedding-word2vec.html#1)
* [Where is Word Embedding used?](https://www.guru99.com/word-embedding-word2vec.html#2)
* [What is word2vec?](https://www.guru99.com/word-embedding-word2vec.html#3)
* [What word2vec does?](https://www.guru99.com/word-embedding-word2vec.html#4)
* [Why Word2vec?](https://www.guru99.com/word-embedding-word2vec.html#5)
* [Word2vec Architecture](https://www.guru99.com/word-embedding-word2vec.html#6)
* [Continuous Bag of Words.](https://www.guru99.com/word-embedding-word2vec.html#7)
* [Skip-Gram Model](https://www.guru99.com/word-embedding-word2vec.html#8)
* [The relation between Word2vec and NLTK](https://www.guru99.com/word-embedding-word2vec.html#9)
* [Activators and Word2Vec](https://www.guru99.com/word-embedding-word2vec.html#10)
* [What is Gensim?](https://www.guru99.com/word-embedding-word2vec.html#11)
* [Code Implementation of word2vec using Gensim](https://www.guru99.com/word-embedding-word2vec.html#12)

**Where is Word Embedding used?**

Word embedding helps in feature generation, document clustering, text classification, and natural language processing tasks. Let us list them and have some discussion on each of these applications.

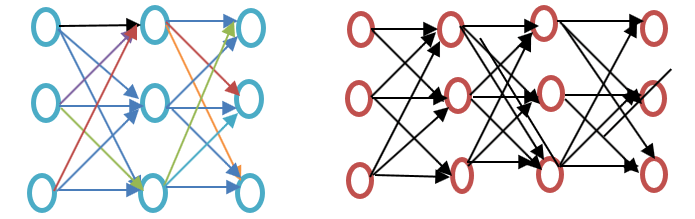
* **Compute similar words:** Word embedding is used to suggest similar words to the word being subjected to the prediction model. Along with that it also suggests dissimilar words, as well as most common words.
* **Create a group of related words:** It is used for semantic grouping which will group things of similar characteristic together and dissimilar far away.
* **Feature for text classification:** Text is mapped into arrays of vectors which is fed to the model for training as well as prediction. Text-based classifier models cannot be trained on the string, so this will convert the text into machine trainable form. Further its features of building semantic help in text-based classification.
* **Document clustering** is another application where word embedding is widely used
* **Natural language processing:** There are many applications where word embedding is useful and wins over feature extraction phases such as parts of speech tagging, sentimental analysis, and syntactic analysis.

Now we have got some knowledge of word embedding. Some light is also thrown on different models to implement word embedding. This whole tutorial is focused on one of the models (word2vec).

**What is word2vec?**

Word2vec is the technique/model to produce word embedding for better word representation. It captures a large number of precise syntactic and semantic word relationship. It is a shallow two-layered neural network. Before going further, please see the difference between shallow and deep neural network:

The shallow neural network consists of the only a hidden layer between input and output whereas deep neural network contains multiple hidden layers between input and output. Input is subjected to nodes whereas the hidden layer, as well as the output layer, contains neurons.



**Figure: Shallow vs. Deep learning**

word2vec is a two-layer network where there is input one hidden layer and output.

Word2vec was developed by a group of researcher headed by Tomas Mikolov at Google. Word2vec is better and more efficient that latent semantic analysis model.

## What word2vec does?

Word2vec represents words in vector space representation. Words are represented in the form of vectors and placement is done in such a way that similar meaning words appear together and dissimilar words are located far away. This is also termed as a semantic relationship. Neural networks do not understand text instead they understand only numbers. Word Embedding provides a way to convert text to a numeric vector.

Word2vec reconstructs the linguistic context of words. Before going further let us understand, what is linguistic context? In general life scenario when we speak or write to communicate, other people try to figure out what is objective of the sentence. For example, "What is the temperature of India", here the context is the user wants to know "temperature of India" which is context. In short, the main objective of a sentence is context. Word or sentence surrounding spoken or written language (disclosure) helps in determining the meaning of context. Word2vec learns vector representation of words through the contexts.

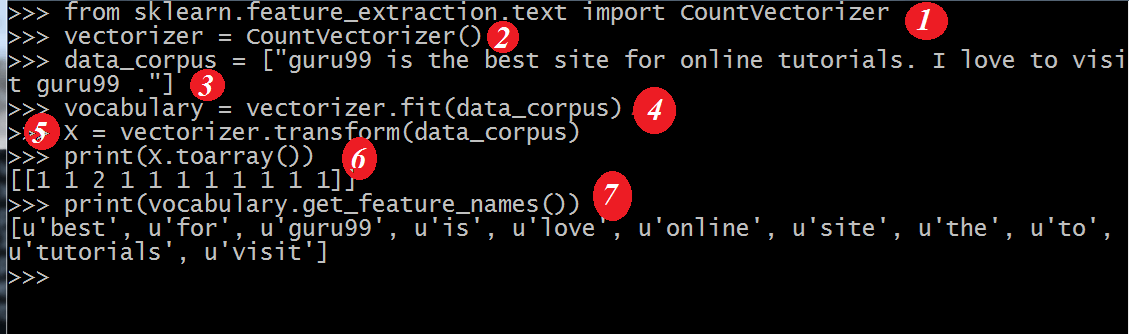
## Why Word2vec?

### Before Word Embedding

It is important to know which approach is used before word embedding and what are its demerits and then we will move to the topic of how demerits are overcome by Word embedding using word2vec approach. Finally, we will move how word2vec works because it is important to understand it's working.

### Approach for Latent Semantic Analysis

This is the approach which was used before word embedding. It used the concept of Bag of words where words are represented in the form of encoded vectors. It is a sparse vector representation where the dimension is equal to the size of vocabulary. If the word occurs in the dictionary, it is counted, else not. To understand more, please see the below program.



from sklearn.feature\_extraction.text import CountVectorizer

vectorizer=CountVectorizer()

data\_corpus=["guru99 is the best sitefor online tutorials. I love to visit guru99."]

vocabulary=vectorizer.fit(data\_corpus)

X= vectorizer.transform(data\_corpus)

print(X.toarray())

print(vocabulary.get\_feature\_names())

Output:

[[1 2 1 1 1 1 1 1 1 1]] [u'best', u'guru99', u'is', u'love', u'online', u'sitefor', u'the', u'to', u'tutorials', u'visit']

**Code Explanation**

1. CountVectorizer is the module which is used to store the vocabulary based on fitting the words in it. This is imported from the sklearn
2. Make the object using the class CountVectorizer.
3. Write the data in the list which is to be fitted in the CountVectorizer.
4. Data is fit in the object created from the class CountVectorizer.
5. Apply a bag of word approach to count words in the data using vocabulary. If word or token is not available in the vocabulary, then such index position is set to zero.
6. Variable in line 5 which is x is converted to an array (method available for x). This will provide the count of each token in the sentence or list provided in Line 3.
7. This will show the features which are part of the vocabulary when it is fitted using the data in Line 4.

In Latent Semantic approach, the row represents unique words whereas the column represents the number of time that word appears in the document. It is a representation of words in the form of the document matrix. Term-Frequency inverse document frequency (TFIDF) is used to count the frequency of words in the document which is the frequency of the term in the document/ frequency of the term in the entire corpus.

**Shortcoming of Bag of Words method**

* It ignores the order of the word, for example, this is bad = bad is this.
* It ignores the context of words. Suppose If I write the sentence "He loved books. Education is best found in books". It would create two vectors one for "He loved books" and other for "Education is best found in books." It would treat both of them orthogonal which makes them independent, but in reality, they are related to each other

To overcome these limitation word embedding was developed and word2vec is an approach to implement such.

**How Word2vec works?**

Word2vec learns word by predicting its surrounding context. For example, let us take the word "He **loves**Football."

We want to calculate the word2vec for the word: loves.

Suppose

loves = Vin. P(Vout / Vin) is calculated

where,

Vin is the input word.

P is the probability of likelihood.

Vout is the output word.

Word **loves**moves over each word in the corpus. Syntactic as well as the Semantic relationship between words is encoded. This helps in finding similar and analogies words.

All random features of the word **loves** is calculated. These features are changed or update concerning neighbor or context words with the help of a back propagation method.

Another way of learning is that if the context of two words are similar or two words have similar features, then such words are related.

**Word2vec Architecture**

There are two architectures used by word2vec

1. Continuous Bag of words (CBOW)
2. skip gram

Before going further, let us discuss why these architectures or models are important from word representation point of view. Learning word representation is essentially unsupervised, but targets/labels are needed to train the model. Skip-gram and CBOW convert unsupervised representation to supervised form for model training.

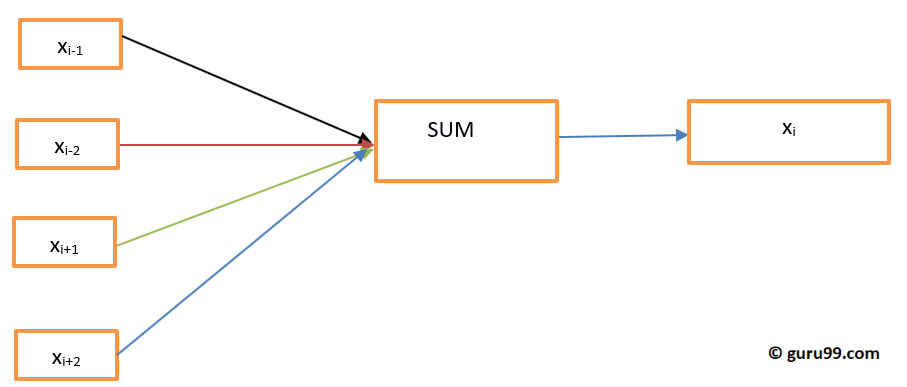
In CBOW, the current word is predicted using the window of surrounding context windows. For example, if wi-1,wi-2,wi+1,wi+2are given words or context, this model will provide wi

Skip-Gram performs opposite of CBOW which implies that it predicts the given sequence or context from the word. You can reverse the example to understand it. If wiis given, this will predict the context or wi-1,wi-2,wi+1,wi+2.

Word2vec provides an option to choose between CBOW (continuous Bag of words) and skim-gram. Such parameters are provided during training of the model. One can have the option of using negative sampling or hierarchical softmax layer.

**Continuous Bag of Words.**

Let us draw a simple diagram to understand the continuous bag of word architecture.



**Figure Continuous Bag of Word Architecture**

Let us calculate the equations mathematically. Suppose V is the vocabulary size and N is the hidden layer size. Input is defined as { xi-1, xi-2,xi+1,xi+2}. We obtain the weight matrix by multiplying V \* N. Another matrix is obtained by multiplying input vector with the weight matrix. This can also be understood by the following equation.

h=xitW

where xit∧ W are the input vector and weight matrix respectively,

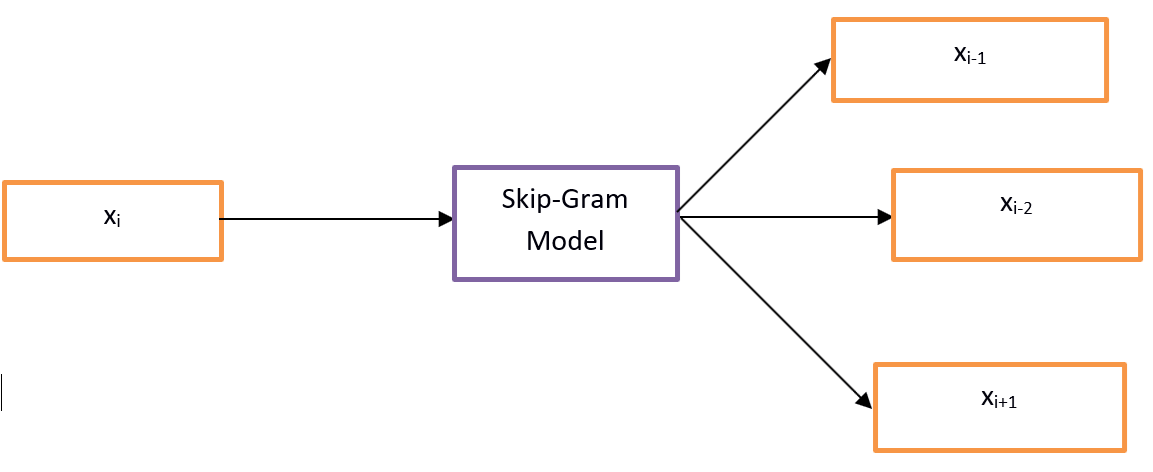
To calculate the match between context and the next word, please refer to the below equation

u=predictedrepresentation\*h

where predictedrepresentation is obtained model∧h in the above equation.

### Skip-Gram Model

Skip-Gram approach is used to predict a sentence given an input word. To understand it better let us draw the diagram.



**Figure Skip-Gram Model**

One can treat it as the reverse of the Continuous bag of word model where the input is the word and model provides the context or the sequence. We can also conclude that the target is fed to the input and output layer is replicated multiple times to accommodate the chosen number of context words. Error vector from all the output layer is summed up to adjust weights via a backpropagation method.

### Which model to choose?

CBOW is several times faster than skip gram and provides a better frequency for frequent words whereas skip gram needs a small amount of training data and represents even rare words or phrases.

## The relation between Word2vec and NLTK

[NLTK](https://www.guru99.com/nltk-tutorial.html) is natural Language toolkit. It is used for preprocessing of the text. One can do different operations such as parts of speech tagging, lemmatizing, stemming, stop words removal, removing rare words or least used words. It helps in cleaning the text as well as helps in preparing the features from the effective words. In the other way, word2vec is used for semantic (closely related items together) and syntactic (sequence) matching. Using word2vec, one can find similar words, dissimilar words, dimensional reduction, and many others. Another important feature of word2vec is to convert the higher dimensional representation of the text into lower dimensional of vectors.

### Where to use NLTK and Word2vec?

If one has to accomplish some general-purpose tasks as mentioned above like tokenization, POS tagging and parsing one must go for using NLTK whereas for predicting words according to some context, topic modeling, or document similarity one must use Word2vec.

### Relation of NLTK and Word2vec with the help of code

NLTK and Word2vec can be used together to find similar words representation or syntactic matching. NLTK toolkit can be used to load many packages which come with NLTK and model can be created using word2vec. It can be then tested on the real time words. Let us see the combination of both in the following code. Before processing further, please have a look on the corpora which NLTK provides. You can download using the command nltk(nltk.download('all'))

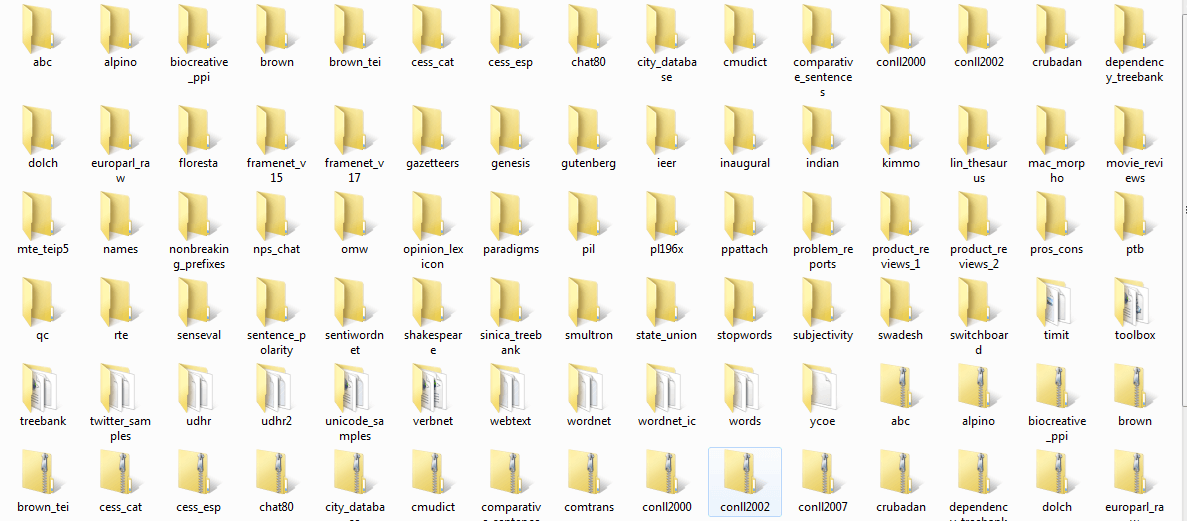


Figure Corpora downloaded using NLTK

Please see the screenshot for the code.

import nltk

import gensim

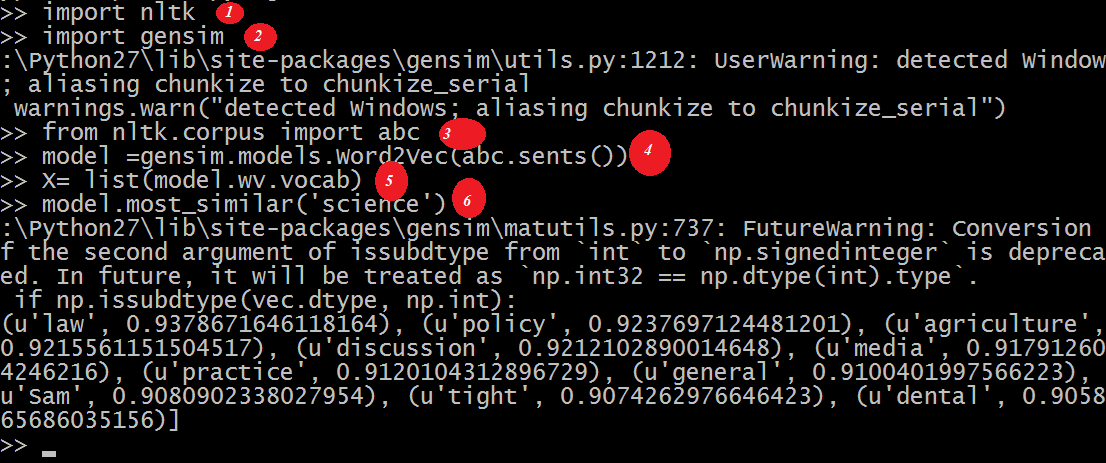
from nltk.corpus import abc

model= gensim.models.Word2Vec(abc.sents())

X= list(model.wv.vocab)

data=model.most\_similar('science')

print(data)



Output:

[('law', 0.9415997266769409), ('practice', 0.9276568293571472), ('discussion', 0.9259148836135864), ('agriculture', 0.9257254004478455), ('media', 0.9232194423675537), ('policy', 0.922248125076294), ('general', 0.9166069030761719), ('undertaking', 0.916458249092102), ('tight', 0.9129181504249573), ('board', 0.9107444286346436)]

**Explanation of Code**

1. nltk library is imported which from where you can download the abc corpus which we will use in the next step.
2. Gensim is imported. If Gensim is not installed, please install it using the command " pip3 install gensim". Please see the below screenshot.

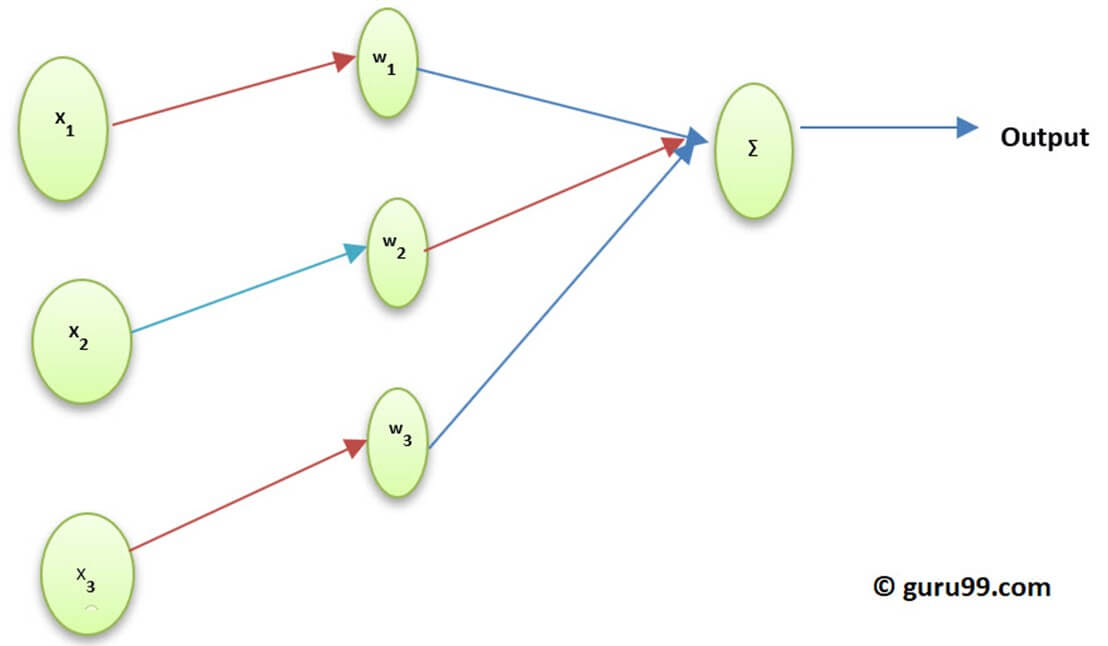


**Figure Installing Gensim using PIP**

1. import the corpus abc which has been downloaded using nltk.download('abc').
2. Pass the files to the model word2vec which is imported using Gensim as sentences.
3. Vocabulary is stored in the form of the variable.
4. Model is tested on sample word science as these files are related to science.
5. Here the similar word of "science" is predicted by the model.

**Activators and Word2Vec**

The activation function of the neuron defines the output of that neuron given a set of inputs. Biologically inspired by an activity in our brains where different neurons are activated using different stimuli. Let us understand the activation function through the following diagram.



**Figure Understanding Activation function**

Here x1,x2,..x4 is the node of the neural network.

w1, w2, w3 is the weight of the node,

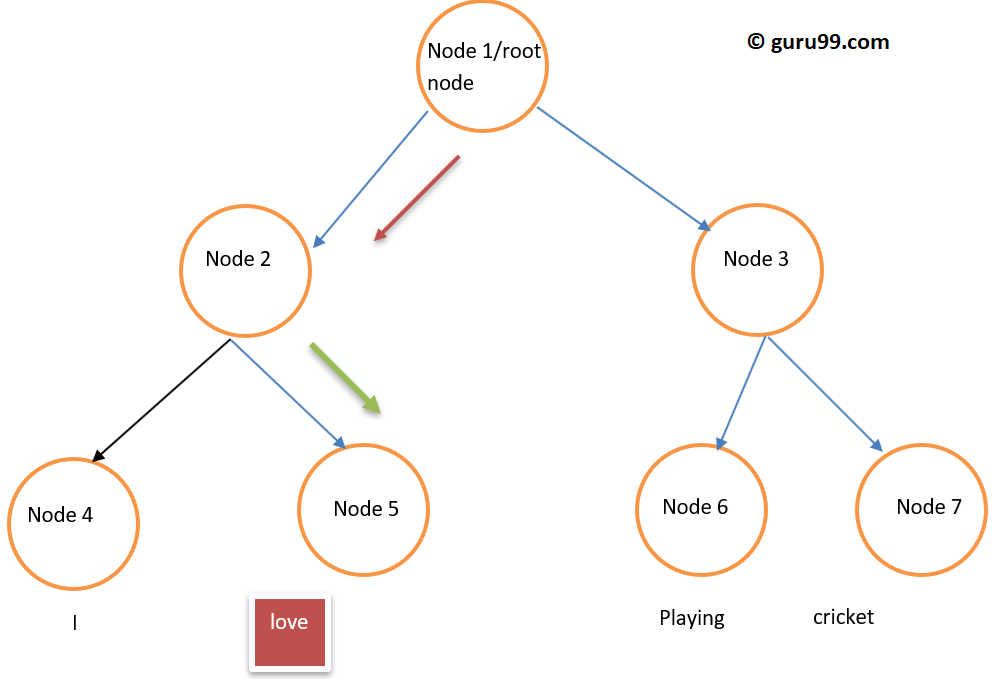
∑ is the summation of all weight and node value which work as the activation function.

### Why Activation function?

If no activation function is used output would be linear but the functionality of linear function is limited. To achieve complex functionality such as object detection, image classification, typing text using voice and many other non-linear outputs is needed which is achieved using activation function.

### How the activation layer is computed in the word embedding (word2vec)

Softmax Layer (normalized exponential function) is the output layer function which activates or fires each node. Another approach used is Hierarchical softmax where the complexity is calculated by O(log2V) wherein the softmax it is O(V) where V is the vocabulary size. The difference between these is the reduction of the complexity in hierarchical softmax layer. To understand its (Hierarchical softmax) functionality, please look at the below example:



**Figure Hierarchical softmax tree like structure**

Suppose we want to compute the probability of observing the word **love** given a certain context. The flow from the root to the leaf node will be the first move to node 2 and then to node 5. So if we have had the vocabulary size of 8, only three computations are needed. So it allows decomposing, calculation of the probability of one word (**love**).

**What other options are available other than Hierarchical Softmax?**

If speaking in a general sense for word embedding options available are Differentiated Softmax, CNN-Softmax, Importance Sampling, Adaptive Importance sampling, Noise Contrastive Estimations, Negative Sampling, Self-Normalization, and infrequent Normalization.

Speaking specifically about Word2vec we have negative sampling available.

Negative Sampling is a way to sample the training data. It is somewhat like stochastic gradient descent, but with some difference. Negative sampling looks only for negative training examples. It is based on noise contrastive estimation and randomly samples words, not in the context. It is a fast training method and chooses the context randomly. If the predicted word appears in the randomly chosen context both the vectors are close to each other.

**What conclusion can be drawn?**

Activators are firing the neurons just like our neurons are fired using the external stimuli. Softmax layer is one of the output layer function which fires the neurons in case of word embedding. In word2vec we have options such as hierarchical softmax and negative sampling. Using activators, one can convert the linear function into the nonlinear function, and a complex machine learning algorithm can be implemented using such.

**What is Gensim?**

Gensim is a topic modeling toolkit which is implemented in python. Topic modeling is discovering hidden structure in the text body. Word2vec is imported from Gensim toolkit. Please note that Gensim not only provides an implementation of word2vec but also Doc2vec and FastText but this tutorial is all about word2vec so we will stick to the current topic.

**Implementation of word2vec using Gensim**

Till now we have discussed what word2vec is, its different architectures, why there is a shift from a bag of words to word2vec, the relation between word2vec and NLTK with live code and activation functions. In this section, will implement word2vec using Gensim

Step 1) Data Collection

The first Step to implement any machine learning model or implementing natural language processing is data collection

Please observe the data to build an intelligent chatbot.

[{"tag": "welcome",

"patterns": ["Hi", "How are you", "Is any one to talk?", "Hello", "hi are you available"],

"responses": ["Hello, thanks for contacting us", "Good to see you here"," Hi there, how may I assist you?"]

},

{"tag": "goodbye",

"patterns": ["Bye", "See you later", "Goodbye", "I will come back soon"],

"responses": ["See you later, thanks for visiting", "have a great day ahead", "Wish you Come back again soon."]

},

{"tag": "thankful",

"patterns": ["Thanks for helping me", "Thank your guidance", "That's helpful and kind from you"],

"responses": ["Happy to help!", "Any time!", "My pleasure", "It is my duty to help you"]

},

{"tag": "hoursopening",

"patterns": ["What hours are you open?", "Tell your opening time?", "When are you open?", "Just your timing please"],

"responses": ["We're open every day 8am-7pm", "Our office hours are 8am-7pm every day", "We open office at 8 am and close at 7 pm"]

},

{"tag": "payments",

"patterns": ["Can I pay using credit card?", " Can I pay using Mastercard?", " Can I pay using cash only?" ],

"responses": ["We accept VISA, Mastercard and credit card", "We accept credit card, debit cards and cash. Please don’t worry"]

}

]

Here is what we understand from the data

* This data contains three things tag, pattern, and responses. The tag is the intent (what is the topic of discussion).
* The data is in JSON format.
* A pattern is a question users will ask to the bot
* Responses is the answer that chatbot will provide to the corresponding question/pattern.

Step 2) Data preprocessing.

It is very important to process the raw data. If cleaned data is fed to the machine, then the model will respond more accurately and will learn the data more efficiently.

This step involves removing stop words, stemming, unnecessary words, etc. Before going ahead, it is important to load data and convert it into a data frame. Please see the below code for such

import json

json\_file =’intents.json'

with open('intents.json','r') as f:

data = json.load(f)

**Explanation of CODE.**

1. As data is in the form of json format hence json is imported
2. File is stored in the variable
3. File is open and loaded in data variable

Now data is imported and it is time to convert data into data frame. Please see the below code to see the next step

import pandas as pd

df = pd.DataFrame(data)

df['patterns'] = df['patterns'].apply(', '.join)

**Explanation of CODE**

1. Data is converted into data frame using pandas which was imported above.

2. It will convert the list in column patterns to string.

from nltk.corpus import stopwords

from textblob import Word

stop = stopwords.words('english')

df['patterns'] = df['patterns'].apply(lambda x:' '.join(x.lower() for x in x.split()))

df['patterns't']= df['patterns''].apply(lambda x: ' '.join(x for x in x.split() if x not in string.punctuation)

df['patterns']= df['patterns'].str.replace('[^\w\s]','')

df['patterns']= df['patterns'].apply(lambda x: ' '.join(x for x in x.split() if not x.isdigit()))

df['patterns'] = df['patterns'].apply(lambda x:' '.join(x for x in x.split() if not x in stop))

df['patterns'] = df['patterns'].apply(lambda x: " ".join([Word(word).lemmatize() for word in x.split()]))

**Code Explanation**

1. English stop words are imported using stop word module from nltk toolkit

2. All the words of the text is converted into lower case using for condition and lambda function. Lambda function is an anonymous function.

3. All the rows of the text in the data frame is checked for string punctuations, and these are filtered.

4. Characters such as numbers or dot are removed using a regular expression.

5. Digits are removed from the text.

6. Stop words are removed at this stage.

7. Words are filtered now, and different form of the same word is removed using lemmatization. With these, we have finished the data preprocessing.

**Output:**

, patterns, responses, tag

0,hi one talk hello hi available,"['Hello, thanks for contacting us', 'Good to see you here', ' Hi there, how may I assist you?']",welcome

1,bye see later goodbye come back soon,"['See you later, thanks for visiting', 'have a great day ahead', 'Wish you Come back again soon.']",goodbye

2,thanks helping thank guidance thats helpful kind,"['Happy to help!', 'Any time!', 'My pleasure', 'It is my duty to help you']",thankful

3,hour open tell opening time open timing please,"[""We're open every day 8am-7pm"", 'Our office hours are 8am-7pm every day', 'We open office at 8 am and close at 7 pm']",hoursopening

4,pay using credit card pay using mastercard pay using cash,"['We accept VISA, Mastercard and credit card', 'We accept credit card, debit cards and cash. Please donâ€™t worry']",payments

**Step 3) Neural Network building using word2vec**

Now it is time to build a model using Gensim module word2vec. We have to import word2vec from Gensim. Let us do this, and then we will build and in the final stage we will check the model on real time data.

from gensim.models import Word2Vec

Now, we can successfully build the model using Word2Vec. Please refer to the next line of code to learn how to create the model using Word2Vec. Text is provided to the model in the form of a list so we will convert the text from data frame to list using the below code

Bigger\_list=[]

for i in df['patterns']

li = list(i.split(""))

Bigger\_list.append(li)

Model= Word2Vec(Bigger\_list,min\_count=1,size=300,workers=4)

**Explanations of Code**

1. Created the bigger\_list where the inner list is appended. This is the format which is fed to the model Word2Vec.

2. Loop is implemented, and each entry of the patterns column of the data frame is iterated.

3. Each element of the column patterns is split and stored in the inner list li

4. the Inner list is appended with the outer list.

5. This list is provided to the Word2Vec model. Let us understand some of the parameters provided here

**Min\_count:** It will ignore all the words with a total frequency lower than this.

**Size:** It tells the dimensionality of the word vectors.

**Workers:** These are the threads to train the model

There are also others options available, and some important ones are explained below

**Window:** Maximum distance between the current and predicted word within a sentence.

**Sg:** It is a training algorithm and 1 for skip-gram and 0 for a Continuous bag of words. We have discussed these in details in above.

**Hs:** If this is 1 then we are using hierarchical softmax for training and if 0 then negative sampling is used.

**Alpha:** Initial learning rate

Let us display the final code below

#list of libraries used by the code

import string

from gensim.models import Word2Vec

import logging

from nltk.corpus import stopwords

from textblob import Word

import json

import pandas as pd

#data in json format

json\_file = 'intents.json'

with open('intents.json','r') as f:

data = json.load(f)

#displaying the list of stopwords

stop = stopwords.words('english')

#dataframe

df = pd.DataFrame(data)

df['patterns'] = df['patterns'].apply(', '.join)

# print(df['patterns'])

#print(df['patterns'])

#cleaning the data using the NLP approach

print(df)

df['patterns'] = df['patterns'].apply(lambda x:' '.join(x.lower() for x in x.split()))

df['patterns']= df['patterns'].apply(lambda x: ' '.join(x for x in x.split() if x not in string.punctuation))

df['patterns']= df['patterns'].str.replace('[^\w\s]','')

df['patterns']= df['patterns'].apply(lambda x: ' '.join(x for x in x.split() if not x.isdigit()))

df['patterns'] = df['patterns'].apply(lambda x:' '.join(x for x in x.split() if not x in stop))

df['patterns'] = df['patterns'].apply(lambda x: " ".join([Word(word).lemmatize() for word in x.split()]))

#taking the outer list

bigger\_list=[]

for i in df['patterns']:

li = list(i.split(" "))

bigger\_list.append(li)

#structure of data to be taken by the model.word2vec

print("Data format for the overall list:",bigger\_list)

#custom data is fed to machine for further processing

model = Word2Vec(bigger\_list, min\_count=1,size=300,workers=4)

#print(model)

**Step 4) Model saving**

Model can be saved in the form of bin and model form. Bin is the binary format. Please see the below lines to save the model

model.save("word2vec.model")

model.save("model.bin")

Explanation of the above code

1. Model is saved in the form of a .model file.

2. model is saved in the form of .bin file

We will use this model to do real time testing such as Similar words, dissimilar words, and most common words.

**Step 5) Loading model and performing real time testing**

Model is loaded using below code

model = Word2Vec.load('model.bin')

If you want to print the vocabulary from it is done using below command vocab = list(model.wv.vocab)

Please see the result

['see', 'thank', 'back', 'thanks', 'soon', 'open', 'mastercard', 'card', 'time', 'pay', 'talk', 'cash', 'one', 'please', 'goodbye', 'thats', 'helpful', 'hour', 'credit', 'hi', 'later', 'guidance', 'opening', 'timing', 'hello', 'helping', 'bye', 'tell', 'come', 'using', 'kind', 'available']

**Step 6) Most Similar words checking**

Let us implement the things practically

similar\_words = model.most\_similar('thanks')

print(similar\_words)

Please see the result

[('kind', 0.16104359924793243), ('using', 0.1352398842573166), ('come', 0.11500970274209976), ('later', 0.09989878535270691), ('helping', 0.04855936020612717), ('credit', 0.04659383371472359), ('pay', 0.0329081267118454), ('thank', 0.02484947443008423), ('hour', 0.0202352125197649), ('opening', 0.018177658319473267)]

**Step 7) Does not match word from words supplied**

dissimlar\_words = model.doesnt\_match('See you later, thanks for visiting'.split())

print(dissimlar\_words)

We have supplied the words **'See you later, thanks for visiting'.**This willprint the most dissimilar words from these words. Let us run this code and find the result

The result after execution of the above code.

Thanks

**Step 8) Finding the similarity between two words**

This will tell result in probability of similarity between two words. Please see the below code how to execute this section.

similarity\_two\_words = model.similarity('please','see')

print("Please provide the similarity between these two words:")

print(similarity\_two\_words)

The result of the above code is as below

0.13706

You can further find similar words by executing the below code

similar = model.similar\_by\_word('kind')

print(similar)

**Output of above code**

[('credit', 0.11764447391033173), ('cash', 0.11440904438495636), ('one', 0.11151769757270813), ('hour', 0.0944807156920433), ('using', 0.0705675333738327), ('thats', 0.05206916481256485), ('later', 0.04502468928694725), ('bye', 0.03960943967103958), ('back', 0.03837274760007858), ('thank', 0.0380823090672493)]

**Conclusion**

* Word Embedding is a type of word representation that allows words with similar meaning to be understood by machine learning algorithms
* Word Embedding is used to compute similar words, Create a group of related words, Feature for text classification, Document clustering, Natural language processing
* Word2vec is a shallow two-layered neural network model to produce word embedding for better word representation
* Word2vec represents words in vector space representation. Words are represented in the form of vectors and placement is done in such a way that similar meaning words appear together and dissimilar words are located far away
* Word2vec used 2 architectures Continuous Bag of words (CBOW) and skip gram
* CBOW is several times faster than skip gram and provides a better frequency for frequent words whereas skip gram needs a small amount of training data and represents even rare words or phrases.
* NLTK and word2vec can be used together create powerful applications
* The activation function of the neuron defines the output of that neuron given a set of inputs. In word2vec. Softmax Layer (normalized exponential function) is the output layer function which activates or fires each node. Word2vec also has negative sampling available
* Gensim is a topic modeling toolkit which is implemented in python

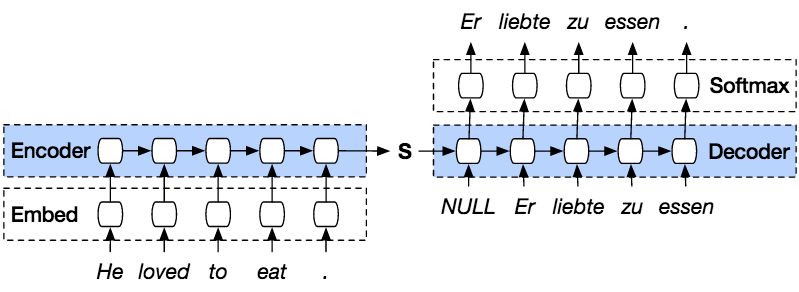
# seq2seq (Sequence to Sequence) Model for Deep Learning with PyTorch

## What is NLP?

NLP or Natural Language Processing is one of the popular branches of Artificial Intelligence that helps computers understands, manipulate or respond to a human in their natural language. NLP is the engine behind Google Translate that helps us understand other languages.

## What is Seq2Seq?

Seq2Seq is a method of encoder-decoder based machine translation that maps an input of sequence to an output of sequence with a tag and attention value. The idea is to use 2 RNN that will work together with a special token and trying to predict the next state sequence from the previous sequence.



**Step 1)** Loading our Data

For our dataset, you will use a dataset from [Tab-delimited Bilingual Sentence Pairs](http://www.manythings.org/anki/). Here I will use the English to Indonesian dataset. You can choose anything you like but remember to change the file name and directory in the code.

from \_\_future\_\_ import unicode\_literals, print\_function, division

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

import numpy as np

import pandas as pd

import os

import re

import random

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

**Step 2)**Data Preparation

You can't use the dataset directly. You need to split the sentences into words and convert it into One-Hot Vector. Every word will be uniquely indexed in the Lang class to make a dictionary. The Lang Class will store every sentence and split it word by word with the addSentence. Then create a dictionary by indexing every unknown word.

SOS\_token = 0

EOS\_token = 1

MAX\_LENGTH = 20

#initialize Lang Class

class Lang:

def \_\_init\_\_(self):

#initialize containers to hold the words and corresponding index

self.word2index = {}

self.word2count = {}

self.index2word = {0: "SOS", 1: "EOS"}

self.n\_words = 2 # Count SOS and EOS

#split a sentence into words and add it to the container

def addSentence(self, sentence):

for word in sentence.split(' '):

self.addWord(word)

#If the word is not in the container, the word will be added to it,

#else, update the word counter

def addWord(self, word):

if word not in self.word2index:

self.word2index[word] = self.n\_words

self.word2count[word] = 1

self.index2word[self.n\_words] = word

self.n\_words += 1

else:

self.word2count[word] += 1

The Lang Class is a class that will help us make a dictionary. For each language, every sentence will be split into words and then added to the container. Each container will store the words in the appropriate index, count the word, and add the index of the word so we can use it to find the index of a word or finding a word from its index.

Because our data is separated by TAB, you need to use pandas as our data loader. Pandas will read our data as dataFrame and split it into our source and target sentence. For every sentence that you have,

* you will normalize it to lower case,
* remove all non-character
* convert to ASCII from Unicode
* split the sentences, so you have each word in it.

#Normalize every sentence

def normalize\_sentence(df, lang):

sentence = df[lang].str.lower()

sentence = sentence.str.replace('[^A-Za-z\s]+', '')

sentence = sentence.str.normalize('NFD')

sentence = sentence.str.encode('ascii', errors='ignore').str.decode('utf-8')

return sentence

def read\_sentence(df, lang1, lang2):

sentence1 = normalize\_sentence(df, lang1)

sentence2 = normalize\_sentence(df, lang2)

return sentence1, sentence2

def read\_file(loc, lang1, lang2):

df = pd.read\_csv(loc, delimiter='\t', header=None, names=[lang1, lang2])

return df

def process\_data(lang1,lang2):

df = read\_file('text/%s-%s.txt' % (lang1, lang2), lang1, lang2)

print("Read %s sentence pairs" % len(df))

sentence1, sentence2 = read\_sentence(df, lang1, lang2)

source = Lang()

target = Lang()

pairs = []

for i in range(len(df)):

if len(sentence1[i].split(' ')) < MAX\_LENGTH and len(sentence2[i].split(' ')) < MAX\_LENGTH:

full = [sentence1[i], sentence2[i]]

source.addSentence(sentence1[i])

target.addSentence(sentence2[i])

pairs.append(full)

return source, target, pairs

Another useful function that you will use is the converting pairs into Tensor. This is very important because our network only reads tensor type data. It's also important because this is the part that at every end of the sentence there will be a token to tell the network that the input is finished. For every word in the sentence, it will get the index from the appropriate word in the dictionary and add a token at the end of the sentence.

def indexesFromSentence(lang, sentence):

return [lang.word2index[word] for word in sentence.split(' ')]

def tensorFromSentence(lang, sentence):

indexes = indexesFromSentence(lang, sentence)

indexes.append(EOS\_token)

return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)

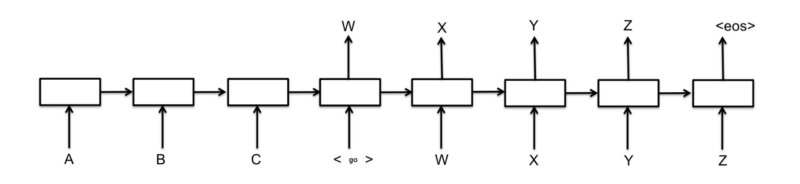
def tensorsFromPair(input\_lang, output\_lang, pair):

input\_tensor = tensorFromSentence(input\_lang, pair[0])

target\_tensor = tensorFromSentence(output\_lang, pair[1])

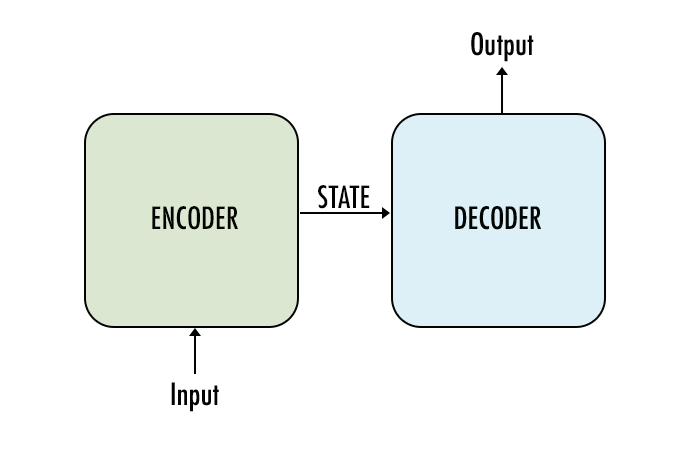
return (input\_tensor, target\_tensor)

**Seq2Seq Model**



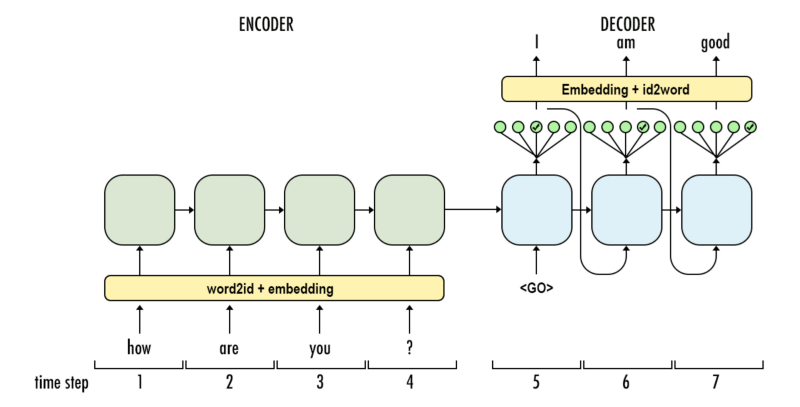
[Source: Seq2Seq](https://medium.com/@Aj.Cheng/seq2seq-18a0730d1d77)

Seq2Seq Model is a kind of model that use Encoder and a Decoder on top of the model. The Encoder will encode the sentence word by words into an indexed of vocabulary or known words with index, and the decoder will predict the output of the coded input by decoding the input in sequence and will try to use the last input as the next input if its possible. With this method, it is also possible to predict the next input to create a sentence. Each sentence will be assigned a token to mark the end of the sequence. At the end of prediction, there will also be a token to mark the end of the output. So, from the encoder, it will pass a state to the decoder to predict the output.



[Source: Seq2Seq Model](https://towardsdatascience.com/sequence-to-sequence-model-introduction-and-concepts-44d9b41cd42d)

The Encoder will encode our input sentence word by word in sequence and in the end there will be a token to mark the end of a sentence. The encoder consists of an Embedding layer and a GRU layers. The Embedding layer is a lookup table that stores the embedding of our input into a fixed sized dictionary of words. It will be passed to a GRU layer. GRU layer is a Gated Recurrent Unit that consists of multiple layer type of RNN that will calculate the sequenced input. This layer will calculate the hidden state from the previous one and update the reset, update, and new gates.



[Source: Seq2Seq](https://medium.com/@Aj.Cheng/seq2seq-18a0730d1d77)

The Decoder will decode the input from the encoder output. It will try to predict the next output and try to use it as the next input if it's possible. The Decoder consists of an Embedding layer, GRU layer, and a Linear layer. The embedding layer will make a lookup table for the output and passed into a GRU layer to calculate the predicted output state. After that, a Linear layer will help to calculate the activation function to determine the true value of the predicted output.

class Encoder(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, embbed\_dim, num\_layers):

super(Encoder, self).\_\_init\_\_()

#set the encoder input dimesion , embbed dimesion, hidden dimesion, and number of layers

self.input\_dim = input\_dim

self.embbed\_dim = embbed\_dim

self.hidden\_dim = hidden\_dim

self.num\_layers = num\_layers

#initialize the embedding layer with input and embbed dimention

self.embedding = nn.Embedding(input\_dim, self.embbed\_dim)

#intialize the GRU to take the input dimetion of embbed, and output dimention of hidden and

#set the number of gru layers

self.gru = nn.GRU(self.embbed\_dim, self.hidden\_dim, num\_layers=self.num\_layers)

def forward(self, src):

embedded = self.embedding(src).view(1,1,-1)

outputs, hidden = self.gru(embedded)

return outputs, hidden

class Decoder(nn.Module):

def \_\_init\_\_(self, output\_dim, hidden\_dim, embbed\_dim, num\_layers):

super(Decoder, self).\_\_init\_\_()

#set the encoder output dimension, embed dimension, hidden dimension, and number of layers

self.embbed\_dim = embbed\_dim

self.hidden\_dim = hidden\_dim

self.output\_dim = output\_dim

self.num\_layers = num\_layers

# initialize every layer with the appropriate dimension. For the decoder layer, it will consist of an embedding, GRU, a Linear layer and a Log softmax activation function.

self.embedding = nn.Embedding(output\_dim, self.embbed\_dim)

self.gru = nn.GRU(self.embbed\_dim, self.hidden\_dim, num\_layers=self.num\_layers)

self.out = nn.Linear(self.hidden\_dim, output\_dim)

self.softmax = nn.LogSoftmax(dim=1)

def forward(self, input, hidden):

# reshape the input to (1, batch\_size)

input = input.view(1, -1)

embedded = F.relu(self.embedding(input))

output, hidden = self.gru(embedded, hidden)

prediction = self.softmax(self.out(output[0]))

return prediction, hidden

class Seq2Seq(nn.Module):

def \_\_init\_\_(self, encoder, decoder, device, MAX\_LENGTH=MAX\_LENGTH):

super().\_\_init\_\_()

#initialize the encoder and decoder

self.encoder = encoder

self.decoder = decoder

self.device = device

def forward(self, source, target, teacher\_forcing\_ratio=0.5):

input\_length = source.size(0) #get the input length (number of words in sentence)

batch\_size = target.shape[1]

target\_length = target.shape[0]

vocab\_size = self.decoder.output\_dim

#initialize a variable to hold the predicted outputs

outputs = torch.zeros(target\_length, batch\_size, vocab\_size).to(self.device)

#encode every word in a sentence

for i in range(input\_length):

encoder\_output, encoder\_hidden = self.encoder(source[i])

#use the encoder’s hidden layer as the decoder hidden

decoder\_hidden = encoder\_hidden.to(device)

#add a token before the first predicted word

decoder\_input = torch.tensor([SOS\_token], device=device) # SOS

#topk is used to get the top K value over a list

#predict the output word from the current target word. If we enable the teaching force, then the #next decoder input is the next word, else, use the decoder output highest value.

for t in range(target\_length):

decoder\_output, decoder\_hidden = self.decoder(decoder\_input, decoder\_hidden)

outputs[t] = decoder\_output

teacher\_force = random.random() < teacher\_forcing\_ratio

topv, topi = decoder\_output.topk(1)

input = (target[t] if teacher\_force else topi)

if(teacher\_force == False and input.item() == EOS\_token):

break

return outputs

**Step 3)**Training the Model

The training process is started with converting each pair of sentences into Tensors from their Lang index. Our model will use SGD as the optimizer and NLLLoss function to calculate the losses. The training process begins with feeding the pair of a sentence to the model to predict the correct output. At each step, the output from the model will be calculated with the true words to find the losses and update the parameters. So because you will use 75000 iterations, our model will generate random 75000 pairs from our dataset.

teacher\_forcing\_ratio = 0.5

def clacModel(model, input\_tensor, target\_tensor, model\_optimizer, criterion):

model\_optimizer.zero\_grad()

input\_length = input\_tensor.size(0)

loss = 0

epoch\_loss = 0

# print(input\_tensor.shape)

output = model(input\_tensor, target\_tensor)

num\_iter = output.size(0)

print(num\_iter)

#calculate the loss from a predicted sentence with the expected result

for ot in range(num\_iter):

loss += criterion(output[ot], target\_tensor[ot])

loss.backward()

model\_optimizer.step()

epoch\_loss = loss.item() / num\_iter

return epoch\_loss

def trainModel(model, source, target, pairs, num\_iteration=20000):

model.train()

optimizer = optim.SGD(model.parameters(), lr=0.01)

criterion = nn.NLLLoss()

total\_loss\_iterations = 0

training\_pairs = [tensorsFromPair(source, target, random.choice(pairs))

for i in range(num\_iteration)]

for iter in range(1, num\_iteration+1):

training\_pair = training\_pairs[iter - 1]

input\_tensor = training\_pair[0]

target\_tensor = training\_pair[1]

loss = clacModel(model, input\_tensor, target\_tensor, optimizer, criterion)

total\_loss\_iterations += loss

if iter % 5000 == 0:

avarage\_loss= total\_loss\_iterations / 5000

total\_loss\_iterations = 0

print('%d %.4f' % (iter, avarage\_loss))

torch.save(model.state\_dict(), 'mytraining.pt')

return model

**Step 4)**Test the Model

The evaluation process is to check the model output. Each pair of sentence will be feed into the model and generate the predicted words. After that you will look the highest value at each output to find the correct index. And in the end, you will compare to see our model prediction with the true sentence

def evaluate(model, input\_lang, output\_lang, sentences, max\_length=MAX\_LENGTH):

with torch.no\_grad():

input\_tensor = tensorFromSentence(input\_lang, sentences[0])

output\_tensor = tensorFromSentence(output\_lang, sentences[1])

decoded\_words = []

output = model(input\_tensor, output\_tensor)

# print(output\_tensor)

for ot in range(output.size(0)):

topv, topi = output[ot].topk(1)

# print(topi)

if topi[0].item() == EOS\_token:

decoded\_words.append('<EOS>')

break

else:

decoded\_words.append(output\_lang.index2word[topi[0].item()])

return decoded\_words

def evaluateRandomly(model, source, target, pairs, n=10):

for i in range(n):

pair = random.choice(pairs)

print(‘source {}’.format(pair[0]))

print(‘target {}’.format(pair[1]))

output\_words = evaluate(model, source, target, pair)

output\_sentence = ' '.join(output\_words)

print(‘predicted {}’.format(output\_sentence))

Now, let's start our training, with the number of iterations of 75000 and num of RNN layer of 1 with the hidden size of 512.

lang1 = 'eng'

lang2 = 'ind'

source, target, pairs = process\_data(lang1, lang2)

randomize = random.choice(pairs)

print('random sentence {}'.format(randomize))

#print number of words

input\_size = source.n\_words

output\_size = target.n\_words

print('Input : {} Output : {}'.format(input\_size, output\_size))

embed\_size = 256

hidden\_size = 512

num\_layers = 1

num\_iteration = 100000

#create encoder-decoder model

encoder = Encoder(input\_size, hidden\_size, embed\_size, num\_layers)

decoder = Decoder(output\_size, hidden\_size, embed\_size, num\_layers)

model = Seq2Seq(encoder, decoder, device).to(device)

#print model

print(encoder)

print(decoder)

model = trainModel(model, source, target, pairs, num\_iteration)

evaluateRandomly(model, source, target, pairs)

As you can see, our predicted sentence is not matched very well, so in order to get higher accuracy, you need to train with a lot more data and try to add more iterations and number of layers.

random sentence ['tom is finishing his work', 'tom sedang menyelesaikan pekerjaannya']

Input : 3551 Output : 4253

Encoder(

(embedding): Embedding(3551, 256)

(gru): GRU(256, 512)

)

Decoder(

(embedding): Embedding(4253, 256)

(gru): GRU(256, 512)

(out): Linear(in\_features=512, out\_features=4253, bias=True)

(softmax): LogSoftmax()

)

Seq2Seq(

(encoder): Encoder(

(embedding): Embedding(3551, 256)

(gru): GRU(256, 512)

)

(decoder): Decoder(

(embedding): Embedding(4253, 256)

(gru): GRU(256, 512)

(out): Linear(in\_features=512, out\_features=4253, bias=True)

(softmax): LogSoftmax()

)

)

5000 4.0906

10000 3.9129

15000 3.8171

20000 3.8369

25000 3.8199

30000 3.7957

35000 3.8037

40000 3.8098

45000 3.7530

50000 3.7119

55000 3.7263

60000 3.6933

65000 3.6840

70000 3.7058

75000 3.7044

> this is worth one million yen

= ini senilai satu juta yen

< tom sangat satu juta yen <EOS>

> she got good grades in english

= dia mendapatkan nilai bagus dalam bahasa inggris

< tom meminta nilai bagus dalam bahasa inggris <EOS>

> put in a little more sugar

= tambahkan sedikit gula

< tom tidak <EOS>

> are you a japanese student

= apakah kamu siswa dari jepang

< tom kamu memiliki yang jepang <EOS>

> i apologize for having to leave

= saya meminta maaf karena harus pergi

< tom tidak maaf karena harus pergi ke

> he isnt here is he

= dia tidak ada di sini kan

< tom tidak <EOS>

> speaking about trips have you ever been to kobe

= berbicara tentang wisata apa kau pernah ke kobe

< tom tidak <EOS>

> tom bought me roses

= tom membelikanku bunga mawar

< tom tidak bunga mawar <EOS>

> no one was more surprised than tom

= tidak ada seorangpun yang lebih terkejut dari tom

< tom ada orang yang lebih terkejut <EOS>

> i thought it was true

= aku kira itu benar adanya

< tom tidak <EOS>