1. **What are Corpora?**

A corpus is a collection of authentic text or audio organized into datasets. Authentic here means text written or audio spoken by a native of the language or dialect. A corpus can be made up of everything from newspapers, novels, recipes, radio broadcasts to television shows, movies, and tweets. In natural language processing, a corpus contains text and speech data that can be used to train AI and machine learning systems. If a user has a specific problem or objective they want to address, they’ll need a collection of data that supports, or at least is a representation of, what they’re looking to achieve with machine learning and NLP.

**What are the features of a good corpus?**

* **Large corpus size:**Generally, the larger the size of a corpus, the better. Large quantities of specialized datasets are vital to training algorithms designed to perform sentiment analysis.
* **High-quality data:**High quality is crucial when it comes to the data within a corpus. Due to the large volume of data required for a corpus, even minuscule errors in the training data can lead to large-scale errors in the machine learning system’s output.
* **Clean data:**Data cleansing is also vital for creating and maintaining a high-quality corpus. Data cleansing allows identifying and eliminating any errors or duplicate data to create a more reliable corpus for NLP.
* **Balance:**A high-quality corpus is a balanced corpus. While it can be tempting to fill a corpus with everything and anything available, if one doesn’t streamline and structure the data collection process, it could unbalance the relevance of the dataset.

**What are the challenges regarding creating a corpus?**

* Deciding the type of data needed to solve the problem statement
* Availability of data
* Quality of the data
* Adequacy of the data in terms of the amount

1. **What are Tokens?**

**Tokenization in natural language processing (NLP) is a technique that involves dividing a sentence or phrase into smaller units known as tokens. These tokens can encompass words, dates, punctuation marks, or even fragments of words. The article aims to cover the fundamentals of tokenization, it’s types and use case.**

**What is Tokenization in NLP?**

[**Natural Language Processing (NLP)**](https://www.geeksforgeeks.org/natural-language-processing-overview/)**is a subfield of**[**computer science**](https://www.geeksforgeeks.org/computer-science-projects/)**,**[**artificial intelligence**](https://www.geeksforgeeks.org/artificial-intelligence-an-introduction/)**, information engineering, and human-computer interaction. This field focuses on how to program computers to process and analyze large amounts of natural language data. It is difficult to perform as the process of reading and understanding languages is far more complex than it seems at first glance.**[**Tokenization**](https://www.geeksforgeeks.org/tokenize-text-using-nltk-python/)**is a foundation step in NLP pipeline that shapes the entire workflow.**

**Tokenization is the process of dividing a text into smaller units known as tokens. Tokens are typically words or sub-words in the context of natural language processing. Tokenization is a critical step in many NLP tasks, including**[**text processing**](https://www.geeksforgeeks.org/text-preprocessing-in-python-set-1/)**,**[**language modelling**](https://www.geeksforgeeks.org/videos/what-is-language-modelling-in-nlp/)**, and**[**machine translation**](https://www.geeksforgeeks.org/machine-translation-of-languages-in-artificial-intelligence/)**. The process involves splitting a string, or text into a list of tokens. One can think of tokens as parts like a word is a token in a sentence, and a sentence is a token in a paragraph.**

**Tokenization involves using a tokenizer to segment unstructured data and natural language text into distinct chunks of information, treating them as different elements. The tokens within a document can be used as vector, transforming an unstructured text document into a numerical data structure suitable for machine learning. This rapid conversion enables the immediate utilization of these tokenized elements by a computer to initiate practical actions and responses. Alternatively, they may serve as features within a machine learning pipeline, prompting more sophisticated decision-making processes or behaviors.**

**Types of Tokenization**

**Tokenization can be classified into several types based on how the text is segmented. Here are some types of tokenization:**

**Word Tokenization:**

**Word tokenization divides the text into individual words. Many NLP tasks use this approach, in which words are treated as the basic units of meaning.**

**Example:**

**Input: "Tokenization is an important NLP task."  
Output: ["Tokenization", "is", "an", "important", "NLP", "task", "."]**

**Sentence Tokenization:**

**The text is segmented into sentences during sentence tokenization. This is useful for tasks requiring individual sentence analysis or processing.**

**Example:**

**Input: "Tokenization is an important NLP task. It helps break down text into smaller units."  
Output: ["Tokenization is an important NLP task.", "It helps break down text into smaller units."]**

**Subword Tokenization:**

**Subword tokenization entails breaking down words into smaller units, which can be especially useful when dealing with morphologically rich languages or rare words.**

**Example:**

**Input: "tokenization"  
Output: ["token", "ization"]**

**Character Tokenization:**

**This process divides the text into individual characters. This can be useful for modelling character-level language.**

**Example:**

**Input: "Tokenization"  
Output: ["T", "o", "k", "e", "n", "i", "z", "a", "t", "i", "o", "n"]**

**Need of Tokenization**

**Tokenization is a crucial step in**[**text processing**](https://www.geeksforgeeks.org/text-preprocessing-in-python-set-1/)**and natural language processing (NLP) for several reasons.**

* **Effective Text Processing: Tokenization reduces the size of raw text so that it can be handled more easily for processing and analysis.**
* **Feature extraction: Text data can be represented numerically for algorithmic comprehension by using tokens as features in**[**machine learning**](https://www.geeksforgeeks.org/machine-learning/)**models.**
* **Language Modelling: Tokenization in NLP facilitates the creation of organized representations of language, which is useful for tasks like text generation and language modelling.**
* **Information Retrieval: Tokenization is essential for indexing and searching in systems that store and retrieve information efficiently based on words or phrases.**
* **Text Analysis: Tokenization is used in many NLP tasks, including**[**sentiment analysis**](https://www.geeksforgeeks.org/what-is-sentiment-analysis/)**and**[**named entity recognition**](https://www.geeksforgeeks.org/named-entity-recognition/)**, to determine the function and context of individual words in a sentence.**
* **Vocabulary Management: By generating a list of distinct tokens that stand in for words in the dataset, tokenization helps manage a corpus’s vocabulary.**
* **Task-Specific Adaptation: Tokenization can be customized to meet the needs of particular NLP tasks, meaning that it will work best in applications such as summarization and machine translation.**
* **Preprocessing Step: This essential preprocessing step transforms unprocessed text into a format appropriate for additional statistical and computational analysis.**

**Implementation for Tokenization**

**Sentence Tokenization using sent\_tokenize**

**The code snippet uses sent\_tokenize function from NLTK library. The sent\_tokenize function is used to segment a given text into a list of sentences.**

* **Python3**

|  |
| --- |
| **from nltk.tokenize import sent\_tokenize**    **text = "Hello everyone. Welcome to GeeksforGeeks. You are studying NLP article."**  **sent\_tokenize(text)** |

**Output:**

**['Hello everyone.',  
 'Welcome to GeeksforGeeks.',  
 'You are studying NLP article']**

***How sent\_tokenize works ?***

***The sent\_tokenize function uses an instance of PunktSentenceTokenizer from the nltk.tokenize.punkt module, which is already been trained and thus very well knows to mark the end and beginning of sentence at what characters and punctuation.***

**Sentence Tokenization using PunktSentenceTokenizer**

**When we have huge chunks of data then it is efficient to use ‘PunktSentenceTokenizer' from the NLTK library. The Punkt tokenizer is a data-driven sentence tokenizer that comes with NLTK. It is trained on large corpus of text to identify sentence boundaries.**

* **Python3**

|  |
| --- |
| **import nltk.data**    **# Loading PunktSentenceTokenizer using English pickle file**  **tokenizer = nltk.data.load('tokenizers/punkt/PY3/english.pickle')**  **tokenizer.tokenize(text)** |

**Output:**

**['Hello everyone.',  
 'Welcome to GeeksforGeeks.',  
 'You are studying NLP article']**

**Tokenize sentence of different language**

**One can also tokenize sentence from different languages using different pickle file other than English. In the following code snippet, we have used NLTK library to tokenize a Spanish text into sentences using pre-trained Punkt tokenizer for Spanish. The Punkt tokenizer is a data-driven tokenizer that uses machine learning techniques to identify sentence boundaries.**

* **Python3**

|  |
| --- |
| **import nltk.data**    **spanish\_tokenizer = nltk.data.load('tokenizers/punkt/PY3/spanish.pickle')**    **text = 'Hola amigo. Estoy bien.'**  **spanish\_tokenizer.tokenize(text)** |

**Output:**

**['Hola amigo.',   
 'Estoy bien.']**

**Word Tokenization using work\_tokenize**

**The code snipped uses the word\_tokenize function from NLTK library to tokenize a given text into individual words. The word\_tokenize function is helpful for breaking down a sentence or text into its constituent words, facilitating further analysis or processing at the word level in natural language processing tasks.**

* **Python3**

|  |
| --- |
| **from nltk.tokenize import word\_tokenize**    **text = "Hello everyone. Welcome to GeeksforGeeks."**  **word\_tokenize(text)** |

**Output:**

**['Hello', 'everyone', '.', 'Welcome', 'to', 'GeeksforGeeks', '.']**

***How word\_tokenize works?***

***word\_tokenize() function is a wrapper function that calls tokenize() on an instance of the TreebankWordTokenizer class.***

**Word Tokenization Using TreebankWordTokenizer**

**The code snippet uses the TreebankWordTokenizer from the Natural Language Toolkit (NLTK) to tokenize a given text into individual words.**

* **Python3**

|  |
| --- |
| **from nltk.tokenize import TreebankWordTokenizer**    **tokenizer = TreebankWordTokenizer()**  **tokenizer.tokenize(text)** |

**Output:**

**['Hello', 'everyone.', 'Welcome', 'to', 'GeeksforGeeks', '.']**

**These tokenizers work by separating the words using punctuation and spaces. And as mentioned in the code outputs above, it doesn’t discard the punctuation, allowing a user to decide what to do with the punctuations at the time of pre-processing.**

**Word Tokenization using WordPunctTokenizer**

**The WordPunctTokenizer is one of the NLTK tokenizers that splits words based on punctuation boundaries. Each punctuation mark is treated as a separate token.**

* **Python3**

|  |
| --- |
| **from nltk.tokenize import WordPunctTokenizer**    **tokenizer = WordPunctTokenizer()**  **tokenizer.tokenize("Let's see how it's working.")** |

**Output:**

**['Let', "'", 's', 'see', 'how', 'it', "'", 's', 'working', '.']**

**Word Tokenization using Regular Expression**

**The code snippet uses the RegexpTokenizer from the**[**Natural Language Toolkit (NLTK)**](https://www.geeksforgeeks.org/tokenize-text-using-nltk-python/)**to tokenize a given text based on a regular expression pattern.**

* **Python3**

|  |
| --- |
| **from nltk.tokenize import RegexpTokenizer**    **tokenizer = RegexpTokenizer(r'\w+')**  **text = "Let's see how it's working."**  **tokenizer.tokenize(text)** |

**Output:**

**['Let', 's', 'see', 'how', 'it', 's', 'working']**

**Using regular expressions allows for more fine-grained control over tokenization, and you can customize the pattern based on your specific requirements.**

**More Techniques for Tokenization**

**We have discussed the ways to implement how can we perform tokenization using NLTK library. We can also implement tokenization using following methods and libraries:**

* **Spacy: [Spacy](https://www.geeksforgeeks.org/tokenization-using-spacy-library/) is NLP library that provide robust tokenization capabilities.**
* **BERT tokenizer:**[**BERT**](https://www.geeksforgeeks.org/explanation-of-bert-model-nlp/)**uses WordPiece tokenizer is a type of subword tokenizer for tokenizing input text. Using regular expressions allows for more fine-grained control over tokenization, and you can customize the pattern based on your specific requirements.**
* **Byte-Pair Encoding:**[**Byte Pair Encoding (BPE)**](https://www.geeksforgeeks.org/byte-pair-encoding-bpe-in-nlp/)**is a data compression algorithm that has also found applications in the field of natural language processing, specifically for tokenization. It is a [subword tokenization](https://www.geeksforgeeks.org/subword-tokenization-in-nlp/)technique that works by iteratively merging the most frequent pairs of consecutive bytes (or characters) in a given corpus.**
* **Sentence Piece: SentencePiece is another subword tokenization algorithm commonly used for natural language processing tasks. It is designed to be language-agnostic and works by iteratively merging frequent sequences of characters or subwords in a given corpus.**

**Limitations of Tokenization**

* **Tokenization is unable to capture the meaning of the sentence hence, results in ambiguity.**
* **In certain languages like Chinese, Japanese, Arabic, lack distinct spaces between words. Hence, there is an absence of clear boundaries that complicates the process of tokenization.**
* **Text may also include more than one word, for example email address, URLs and special symbols, hence it is difficult to decide how to tokenize such elements.**

**Tokenization – Frequently Asked Questions (FAQs)**

**Q. What is Tokenization in NLP?**

***Tokenization is the process of converting a sequence of text into smaller parts known as tokens in the context of Natural Language Processing (NLP) and machine learning. These tokens can be as short as a character or as long as a sentence.***

**Q. What is Lemmatization in NLP?**

***Lemmatization is a text pre-processing method that helps natural language processing (NLP) models find similarities by reducing a word to its most basic meaning. A lemmatization algorithm, for instance, would reduce the word better to its lemme, or good.***

**Q. Which are most common types of tokenization?**

***Word tokenization, which divides text into words, sentence tokenization, which divides text into sentences, subword tokenization, which divides words into smaller units, and character tokenization, which divides text into individual characters, are common forms of tokenization.***

1. **What are Unigrams, Bigrams, Trigrams?**

N-grams are one of the fundamental concepts every data scientist and computer science professional must know while working with text data. In this beginner-level tutorial, we will learn what n-grams are and explore them on text data in Python. The objective of the blog is to analyze different types of n-grams on the given text data and hence decide which n-gram works the best for our data.

**Learning Objectives**

* Implement n-gram in Python from scratch and using [nltk](https://www.analyticsvidhya.com/blog/2021/07/nltk-a-beginners-hands-on-guide-to-natural-language-processing/)
* Understand n-grams and their importance
* Know the applications of n-grams in [NLP](https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/)

This article was published as a part of the [Data Science Blogathon](https://datahack.analyticsvidhya.com/contest/data-science-blogathon-40/).

What Are N-Grams(ngrams)?

N-grams are continuous sequences of words or symbols, or tokens in a document. In technical terms, they can be defined as the neighboring sequences of items in a document. They come into play when we deal with text data in NLP (Natural Language Processing) tasks. They have a wide range of applications, like language models, semantic features, spelling correction, machine translation, text mining, etc.

How Are N-Grams Classified?

Did you notice the ‘n’ in the term “n-grams”? Can you guess what this ‘n’ possibly is?

Remember when we learned how to input an array by first inputting its size(n) or even a number from the user? Generally, we used to store such values in a variable declared as ‘n’! Apart from programming, you must have extensively encountered ‘n’ in the formulae of the sum of series and so on. What do you think ‘n’ was over there?

Summing up, ‘n’ is just a variable that can have positive integer values, including 1,2,3, and so on.’n’ basically refers to multiple.

Thinking along the same lines, n-grams are classified into the following types, depending on the value that ‘n’ takes.

|  |  |
| --- | --- |
| **n** | **Term** |
| 1 | Unigram |
| 2 | Bigram |
| 3 | Trigram |
| n | n-gram |

As clearly depicted in the table above, when n=1, it is said to be a unigram. When n=2, it is said to be a bigram, and so on.

Now, you must be wondering why we need many different types of n-grams?! This is because different types of n-grams are suitable for different types of applications. You should try different n-grams on your data in order to confidently conclude which one works the best among all for your [text analysis](https://www.analyticsvidhya.com/blog/2018/02/the-different-methods-deal-text-data-predictive-python/). **For instance, research has substantiated that trigrams and 4 grams work the best in the case of spam filtering.**

Example of N-Grams

Let’s understand n-grams practically with the help of the following sample sentence:

“I reside in Bengaluru”.

|  |  |  |
| --- | --- | --- |
| **SL.No.** | **Type of n-gram** | **Generated n-grams** |
| 1 | Unigram | [“I”,”reside”,”in”,”Bengaluru”] |
| 2 | Bigram | [“I reside”,”reside in”,”in Bengaluru”] |
| 3 | Trigram | [“I reside in”, “reside in Bengaluru”] |

**from** nltk **import** ngrams

sentence = 'I reside in Bengaluru.'

n = 1

unigrams = ngrams(sentence.split(), n)

**for** grams **in** unigrams:

print grams

For the time being, let’s not consider the removal of stop-words :

From the table above, it’s clear that unigram means taking only one word at a time, bigram means taking two words at a time, and trigram means taking three words at a time. We will be implementing only till trigrams here in this blog. Feel free to proceed ahead and explore 4 grams, 5 grams, and so on from your takeaways from the blog!

Step-By-Step Implementation of N-Grams in Python

And here comes the most interesting section of the blog! Unless we practically implement what we learn, there is absolutely no fun in learning it! So, let’s proceed to code and generate n-grams on [Google Colab](https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index) in [Python](https://www.analyticsvidhya.com/blog/2016/01/complete-tutorial-learn-data-science-python-scratch-2/). You can also build a simple n-gram language model on top of this code.

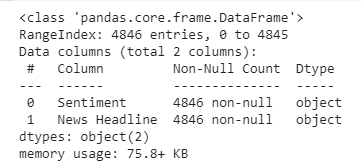
Step 1: Explore the Dataset

I will be using sentiment analysis for the financial news dataset. The sentiments are from the perspective of retail investors. It is an open-source Kaggle dataset. Download it from [here](https://www.kaggle.com/ankurzing/sentiment-analysis-for-financial-news) before moving ahead.

Let’s begin, as usual, by importing the required libraries and reading and understanding the data:

**Python Code:**

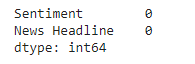
df.info()



You can see that the dataset has 4846 rows and two columns, namely,’ Sentiment’ and ‘News Headline.’

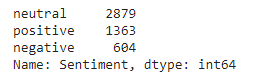
**NOTE:***When you download the dataset from Kaggle directly, you will notice that the columns are nameless! So, I named them later and updated them in the all-data.csv file before reading it using pandas. Ensure that you do not miss this step.*

df.isna().sum()



The data is just perfect, with absolutely no missing values at all! That’s our luck, indeed!

df['Sentiment'].value\_counts()



We can undoubtedly infer that the dataset includes three categories of sentiments:

* Neutral
* Positive
* Negative

Out of 4846 sentiments, 2879 have been found to be neutral, 1363 positive, and the rest negative.

Step 2: Feature Extraction

Our objective is to predict the sentiment of a given news headline. Obviously, the ‘News Headline’ column is our only feature, and the ‘Sentiment’ column is our target variable.

y=df['Sentiment'].values

y.shape

x=df['News Headline'].values

x.shape

Both the outputs return a shape of (4846,) which means 4846 rows and 1 column as we have 4846 rows of data and just 1 feature and a target for x and y, respectively.

Step 3: Train-Test Split

In any machine learning, deep learning, or NLP(Natural Language Processing) task, splitting the data into train and test is indeed a highly crucial step. The train\_test\_split() method provided by sklearn is widely used for the same. So, let’s begin by importing it:

**from** sklearn.model\_selection **import** train\_test\_split

Here’s how I’ve split the data: 60% for the train and the rest 40% for the test. I had started with 20% for the test. I kept on playing with the test\_size parameter only to realize that the 60-40 ratio of split provides more useful and meaningful insights from the trigrams generated. Don’t worry; we will be looking at trigrams in just a while.

(x\_train,x\_test,y\_train,y\_test)=train\_test\_split(x,y,test\_size=0.4)

x\_train.shape

y\_train.shape

x\_test.shape

y\_test.shape

On executing the codes above, you will observe that 2907 rows have been considered as train data, and the rest of the 1939 rows have been considered as test data.

Our next step is to convert these NumPy arrays to Pandas data frames and thus create two data frames, namely,df\_train and df\_test. The former is created by concatenating x\_train and y\_train arrays. The latter data frame is created by concatenating x\_test and y\_test arrays. This is necessary to count the number of positive, negative, and neutral sentiments in both train and test datasets which we will be doing in a while.

df1=pd.DataFrame(x\_train)

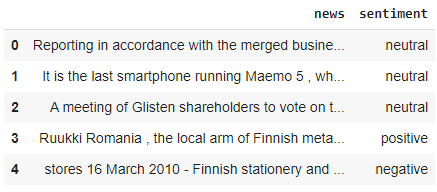
df1=df1.rename(columns={0:'news'})

df2=pd.DataFrame(y\_train)

df2=df2.rename(columns={0:'sentiment'})

df\_train=pd.concat([df1,df2],axis=1)

df\_train.head()



df3=pd.DataFrame(x\_test)

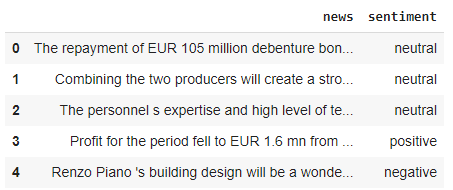
df3=df3.rename(columns={0:'news'})

df4=pd.DataFrame(y\_test)

df4=df2.rename(columns={0:'sentiment'})

df\_test=pd.concat([df3,df4],axis=1)

df\_test.head()



Step 4: Basic Pre-Processing of Train and Test Data

Here, in order to pre-process our text data, we will remove punctuations in train and test data for the ‘news’ column using punctuation provided by the string library.

#removing punctuations

#library that contains punctuation

import string

string.punctuation

#defining the **function** **to** remove punctuation

def remove\_punctuation(**text**):

**if**(type(**text**)==float):

**return** **text**

ans=""

**for** i **in** **text**:

**if** i not **in** string.punctuation:

ans+=i

**return** ans

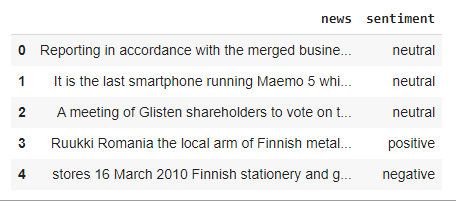
#storing the puntuation free text in a new column called clean\_msg

df\_train['news']= df\_train['news'].**apply**(lambda x:**remove\_punctuation**(x))

df\_test['news']= df\_test['news'].**apply**(lambda x:**remove\_punctuation**(x))

df\_train.head()

#punctuations are removed from news column in train dataset



Compare the above output with the previous output of df\_train. You can observe that punctuations have been successfully removed from the text present in the feature column(news column) of the training dataset. Similarly, from the above codes, punctuations will be removed successfully from the news column of the test data frame as well. You can optionally view *df\_test.head()* as well to note it.

As a next step, we have to remove stopwords from the news column. For this, let’s use the stopwords provided by nltk as follows:

**import** nltk

**from** nltk.corpus **import** stopwords

nltk.**download**('stopwords')

We will be using this to generate n-grams in the very next step.

Step 5: Code to Generate N-grams

Let’s code a custom function to generate n-grams for a given text as follows:

#method **to** generate n-grams:

#params:

#**text**-the **text** **for** which we have **to** generate n-grams

#ngram-number **of** grams **to** be generated **from** the **text**(1,2,3,4 etc., **default** value=1)

def generate\_N\_grams(text,ngram=1):

words=[word for word in text.split(" ") if word not in set(stopwords.words('english'))]

print("Sentence after removing stopwords:",words)

temp=zip(\*[words[i:] for i in range(0,ngram)])

ans=[' '.join(ngram) for ngram in temp]

return ans

The above function inputs two parameters, namely, *text* and *ngram,* which refer to the text data for which we want to generate a given number of *n-grams* and the number of grams to be generated, respectively. Firstly, word tokenization is done where the stop words are ignored, and the remaining words are retained. From the example section, you must have been clear on how to generate n-grams manually for a given text. We have coded the very same logic in the function *generate\_N\_grams()* above. It will thus consider *n* words at a time from the *text* where n is given by the value of the *ngram* parameter of the function.

Let’s check the working of the function with the help of a simple example to create bigrams as follows:

#sample!

generate\_N\_grams("The sun rises in the east",2)

generate n grams

Great! We are now set to proceed.

Step 6: Creating Unigrams

Let’s follow the steps below to create unigrams for the *news* column of the *df\_train* data frame:

1. Create unigrams for each of the news records belonging to each of the three categories of sentiments.
2. Store the word and its count in the corresponding dictionaries.
3. Convert these dictionaries to corresponding data frames.
4. Fetch the top 10 most frequently used words.
5. Visualize the most frequently used words for all the 3 categories-positive, negative and neutral.

Have a look at the codes below to understand the steps better.

**from** collections **import** defaultdict

positiveValues=defaultdict(int)

negativeValues=defaultdict(int)

neutralValues=defaultdict(int)

#get the count of every word in both the columns of df\_train and df\_test dataframes

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="positive"

**for** text **in** df\_train[df\_train.sentiment=="positive"].news:

**for** word **in** generate\_N\_grams(text):

positiveValues[word]+=1

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="negative"

**for** text **in** df\_train[df\_train.sentiment=="negative"].news:

**for** word **in** generate\_N\_grams(text):

negativeValues[word]+=1

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="neutral"

**for** text **in** df\_train[df\_train.sentiment=="neutral"].news:

**for** word **in** generate\_N\_grams(text):

neutralValues[word]+=1

#focus on more frequently occuring words for every sentiment=>

#sort in DO wrt 2nd column in each of positiveValues,negativeValues and neutralValues

df\_positive=pd.DataFrame(sorted(positiveValues.items(),key=lambda x:x[1],reverse=True))

df\_negative=pd.DataFrame(sorted(negativeValues.items(),key=lambda x:x[1],reverse=True))

df\_neutral=pd.DataFrame(sorted(neutralValues.items(),key=lambda x:x[1],reverse=True))

pd1=df\_positive[0][:10]

pd2=df\_positive[1][:10]

ned1=df\_negative[0][:10]

ned2=df\_negative[1][:10]

nud1=df\_neutral[0][:10]

nud2=df\_neutral[1][:10]

plt.figure(1,figsize=(16,4))

plt.bar(pd1,pd2, color ='green',

width = 0.4)

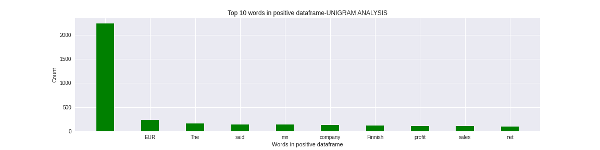
plt.xlabel("Words in positive dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in positive dataframe-UNIGRAM ANALYSIS")

plt.savefig("positive-unigram.png")

plt.show()



plt.figure(1,figsize=(16,4))

plt.bar(ned1,ned2, color ='red',

width = 0.4)

plt.xlabel("Words in negative dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in negative dataframe-UNIGRAM ANALYSIS")

plt.savefig("negative-unigram.png")

plt.show()



plt.figure(1,figsize=(16,4))

plt.bar(nud1,nud2, color ='yellow',

width = 0.4)

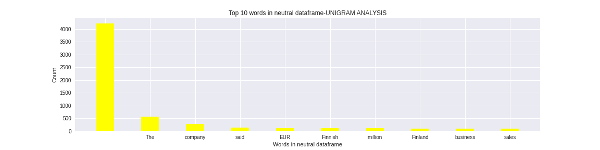
plt.xlabel("Words in neutral dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in neutral dataframe-UNIGRAM ANALYSIS")

plt.savefig("neutral-unigram.png")

plt.show()



Step 7: Creating Bigrams

Repeat the same steps which we followed to analyze our data using unigrams, except that you have to pass parameter 2 while invoking the *generate\_N\_grams()* function. You can optionally consider changing the names of the data frames, which I have done.

positiveValues2=defaultdict(int)

negativeValues2=defaultdict(int)

neutralValues2=defaultdict(int)

#get the count of every word in both the columns of df\_train and df\_test dataframes

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="positive"

**for** text **in** df\_train[df\_train.sentiment=="positive"].news:

**for** word **in** generate\_N\_grams(text,2):

positiveValues2[word]+=1

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="negative"

**for** text **in** df\_train[df\_train.sentiment=="negative"].news:

**for** word **in** generate\_N\_grams(text,2):

negativeValues2[word]+=1

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="neutral"

**for** text **in** df\_train[df\_train.sentiment=="neutral"].news:

**for** word **in** generate\_N\_grams(text,2):

neutralValues2[word]+=1

#focus on more frequently occuring words for every sentiment=>

#sort in DO wrt 2nd column in each of positiveValues,negativeValues and neutralValues

df\_positive2=pd.DataFrame(sorted(positiveValues2.items(),key=lambda x:x[1],reverse=True))

df\_negative2=pd.DataFrame(sorted(negativeValues2.items(),key=lambda x:x[1],reverse=True))

df\_neutral2=pd.DataFrame(sorted(neutralValues2.items(),key=lambda x:x[1],reverse=True))

pd1bi=df\_positive2[0][:10]

pd2bi=df\_positive2[1][:10]

ned1bi=df\_negative2[0][:10]

ned2bi=df\_negative2[1][:10]

nud1bi=df\_neutral2[0][:10]

nud2bi=df\_neutral2[1][:10]

plt.figure(1,figsize=(16,4))

plt.bar(pd1bi,pd2bi, color ='green',width = 0.4)

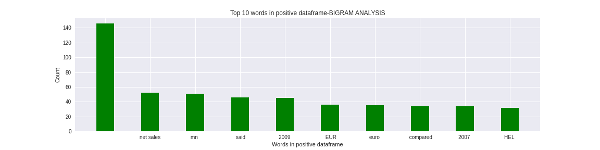
plt.xlabel("Words in positive dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in positive dataframe-BIGRAM ANALYSIS")

plt.savefig("positive-bigram.png")

plt.show()



plt.figure(1,figsize=(16,4))

plt.bar(ned1bi,ned2bi, color ='red',

width = 0.4)

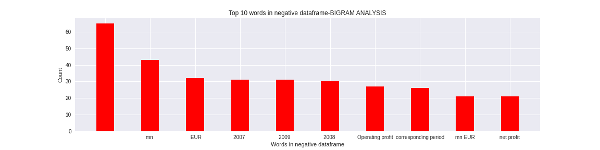
plt.xlabel("Words in negative dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in negative dataframe-BIGRAM ANALYSIS")

plt.savefig("negative-bigram.png")

plt.show()



plt.figure(1,figsize=(16,4))

plt.bar(nud1bi,nud2bi, color ='yellow',

width = 0.4)

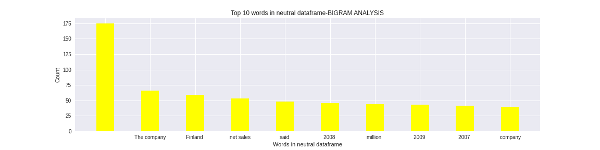
plt.xlabel("Words in neutral dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in neutral dataframe-BIGRAM ANALYSIS")

plt.savefig("neutral-bigram.png")

plt.show()



Step 8: Creating Trigrams

Repeat the same steps which we followed to analyze our data using unigrams, except that you have to pass parameter 3 while invoking the *generate\_N\_grams()* function. You can optionally consider changing the names of the data frames, which I have done.

positiveValues3=defaultdict(int)

negativeValues3=defaultdict(int)

neutralValues3=defaultdict(int)

#get the count of every word in both the columns of df\_train and df\_test dataframes

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="positive"

**for** text **in** df\_train[df\_train.sentiment=="positive"].news:

**for** word **in** generate\_N\_grams(text,3):

positiveValues3[word]+=1

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="negative"

**for** text **in** df\_train[df\_train.sentiment=="negative"].news:

**for** word **in** generate\_N\_grams(text,3):

negativeValues3[word]+=1

#**get** the count **of** **every** word **in** **both** the columns **of** df\_train **and** df\_test dataframes **where** sentiment="neutral"

**for** text **in** df\_train[df\_train.sentiment=="neutral"].news:

**for** word **in** generate\_N\_grams(text,3):

neutralValues3[word]+=1#focus **on** more frequently occuring words **for** **every** sentiment=>

#sort **in** DO wrt 2nd **column** **in** **each** **of** positiveValues,negativeValues **and** neutralValues

df\_positive3=pd.DataFrame(sorted(positiveValues3.items(),key=lambda x:x[1],reverse=True))

df\_negative3=pd.DataFrame(sorted(negativeValues3.items(),key=lambda x:x[1],reverse=True))

df\_neutral3=pd.DataFrame(sorted(neutralValues3.items(),key=lambda x:x[1],reverse=True))

pd1tri=df\_positive3[0][:10]

pd2tri=df\_positive3[1][:10]

ned1tri=df\_negative3[0][:10]

ned2tri=df\_negative3[1][:10]

nud1tri=df\_neutral3[0][:10]

nud2tri=df\_neutral3[1][:10]

plt.figure(1,figsize=(16,4))

plt.bar(pd1tri,pd2tri, color ='green',

width = 0.4)

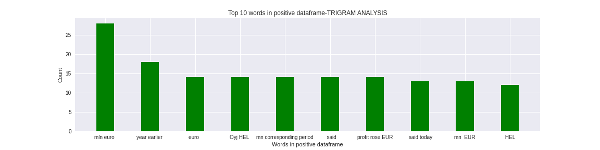
plt.xlabel("Words in positive dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in positive dataframe-TRIGRAM ANALYSIS")

plt.savefig("positive-trigram.png")

plt.show()



plt.figure(1,figsize=(16,4))

plt.bar(ned1tri,ned2tri, color ='red',

width = 0.4)

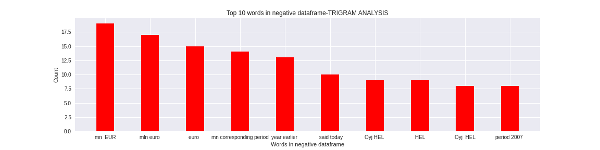
plt.xlabel("Words in negative dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in negative dataframe-TRIGRAM ANALYSIS")

plt.savefig("negative-trigram.png")

plt.show()



plt.figure(1,figsize=(16,4))

plt.bar(nud1tri,nud2tri, color ='yellow',

width = 0.4)

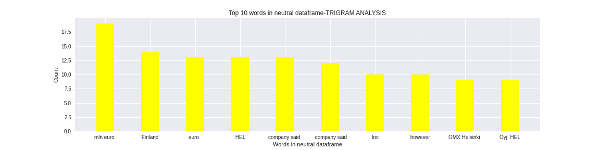
plt.xlabel("Words in neutral dataframe")

plt.ylabel("Count")

plt.title("Top 10 words in neutral dataframe-TRIGRAM ANALYSIS")

plt.savefig("neutral-trigram.png")

plt.show()



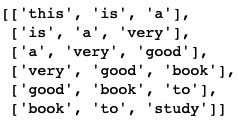
1. **How to generate n-grams from text?**

N-grams are all possible combinations of “N” words from the text. When two words are combined at a time, they are known as **Bigrams**, when three words are combined at a time, they are known as **Trigrams**, so on and so forth. They are very useful when we are trying to do NLP because combinations of words are more meaningful as compared to individual words.

There are two ways to generate N-grams, either by writing the logic yourself or by using the nltk library function.

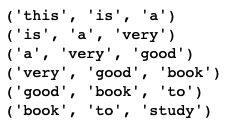
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | # Creating a function to generate N-Grams  def generate\_ngrams(text, WordsToCombine):  words = text.split()  output = []  for i in range(len(words)- WordsToCombine+1):  output.append(words[i:i+WordsToCombine])  return output  # Calling the function  generate\_ngrams(text='this is a very good book to study', WordsToCombine=3) |

**Sample Output**

Generating N-grams in Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # NLTK function to generate ngrams  import nltk  from nltk.util import ngrams  samplText='this is a very good book to study'  NGRAMS=ngrams(sequence=nltk.word\_tokenize(samplText), n=3)  for grams in NGRAMS:  print(grams) |

**Sample Output**

Generate N-grams using nltk in Python

1. **Explain Lemmatization**

**WHAT IS LEMMATIZATION?**

Lemmatization is a text pre-processing technique used in natural language processing (NLP) models to break a word down to its root meaning to identify similarities. For example, a lemmatization algorithm would reduce the word *better* to its root word, or lemme, *good*.

**How Is Lemmatization Different From Stemming?**

In stemming, a part of the word is just chopped off at the tail end to arrive at the stem of the word. There are different algorithms used to find out how many characters have to be chopped off, but the algorithms don’t actually know the meaning of the word in the language it belongs to. In lemmatization, the algorithms do have this knowledge. In fact, you can even say that these algorithms refer to a dictionary to understand the meaning of the word before reducing it to its root word, or lemma.

So, a lemmatization algorithm would know that the word *better* is derived from the word *good*, and hence, the lemme is *good*. But a stemming algorithm wouldn’t be able to do the same. There could be over-stemming or under-stemming, and the word *better* could be reduced to either *bet*, or *bett*, or just retained as *better*. But there is no way in stemming that can reduce better to its root word *good*. This is the difference between stemming and lemmatization.

**Advantages and Disadvantages of Lemmatization**

As you can probably tell by now, the obvious advantage of lemmatization is that it is more accurate than stemming. So, if you’re dealing with an NLP application such as a [chat bot](https://builtin.com/design-ux/chatbot-turing-test-mitsuku-pandorabots) or a [virtual assistant](https://builtin.com/founders-entrepreneurship/virtual-assistant-optimize), where understanding the meaning of the dialogue is crucial, lemmatization would be useful. But this accuracy comes at a cost.

Because lemmatization involves deriving the meaning of a word from something like a dictionary, it’s very time consuming. So most lemmatization algorithms are slower compared to their stemming counterparts. There is also a computation overhead for lemmatization, however, in most machine learning problems, computational resources are rarely a cause of concern.

**Should You Choose Lemmatization Over Stemming?**

Well, I can’t answer that question. Lemmatization and stemming are both much more complex than what I’ve made them appear here. There are a lot more things to consider about both the approaches before making a decision. But I’ve rarely seen any significant improvement in efficiency and accuracy of a product that uses lemmatization over stemming. In most cases, at least according to my knowledge, the overhead that lemmatization demands is not justified. So, it depends on the project in question. But I want to put out a disclaimer here. Most of the work I have done in NLP is for [text classification](https://builtin.com/artificial-intelligence/what-is-natural-language-generation), and that’s where I haven’t seen a significant difference. There are applications where the overhead of lemmatization is perfectly justified, and in fact, lemmatization would be a necessity.

1. **Explain Stemming**

**Stemming** is a method in **text processing** that eliminates prefixes and suffixes from words, transforming them into their fundamental or root form, The main objective of stemming is to streamline and standardize words, enhancing the effectiveness of the**natural language processing** tasks. The article explores more on the stemming technique and how to perform stemming in Python.

## What is Stemming in NLP?

Simplifying words to their most basic form is called stemming, and it is made easier by stemmers or stemming algorithms. For example, “chocolates” becomes “chocolate” and “retrieval” becomes “retrieve.” This is crucial for pipelines for natural language processing, which use tokenized words that are acquired from the first stage of dissecting a document into its constituent words.

Stemming in [natural language processing](https://www.geeksforgeeks.org/natural-language-processing-overview/) reduces words to their base or root form, aiding in text normalization for easier processing. This technique is crucial in tasks like [text classification](https://www.geeksforgeeks.org/rnn-for-text-classifications-in-nlp/),[information retrieval](https://www.geeksforgeeks.org/what-is-information-retrieval/), and [text summarization](https://www.geeksforgeeks.org/python-extractive-text-summarization-using-gensim/). While beneficial, stemming has drawbacks, including potential impacts on text readability and occasional inaccuracies in determining the correct root form of a word.

### Why is Stemming important?

It is important to note that stemming is different from [Lemmatization](https://www.geeksforgeeks.org/python-lemmatization-with-nltk/). Lemmatization is the process of reducing a word to its base form, but unlike stemming, it takes into account the context of the word, and it produces a valid word, unlike stemming which may produce a non-word as the root form.

1. **Explain Part-of-speech (POS) tagging**

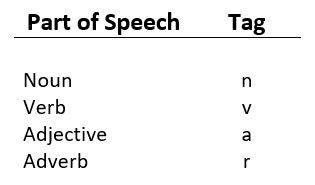
One of the core tasks in **Natural Language Processing (NLP)** is **Parts of Speech (PoS) tagging**, which is giving each word in a text a grammatical category, such as nouns, verbs, adjectives, and adverbs. Through improved comprehension of phrase structure and semantics, this technique makes it possible for machines to study and comprehend human language more accurately.

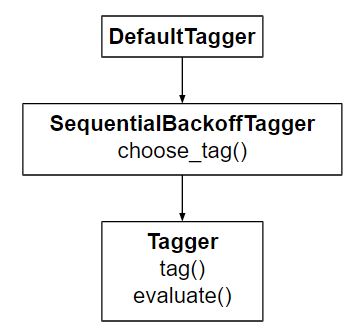
In many [NLP](https://www.geeksforgeeks.org/natural-language-processing-overview/) applications, including machine translation, sentiment analysis, and information retrieval, PoS tagging is essential. PoS tagging serves as a link between language and machine understanding, enabling the creation of complex language processing systems and serving as the foundation for advanced linguistic analysis.

**What is POS(Parts-Of-Speech) Tagging?**

[Parts of Speech tagging](https://www.geeksforgeeks.org/python-pos-tagging-and-lemmatization-using-spacy/) is a linguistic activity in[Natural Language Processing](https://www.geeksforgeeks.org/introduction-to-natural-language-processing/) (NLP) wherein each word in a document is given a particular part of speech (adverb, adjective, verb, etc.) or grammatical category. Through the addition of a layer of syntactic and semantic information to the words, this procedure makes it easier to comprehend the sentence’s structure and meaning.

In NLP applications, POS tagging is useful for machine translation, [named entity recognition](https://www.geeksforgeeks.org/named-entity-recognition/), and information extraction, among other things. It also works well for clearing out ambiguity in terms with numerous meanings and revealing a sentence’s grammatical structure.



**Default tagging**is a basic step for the part-of-speech tagging. It is performed using the DefaultTagger class. The DefaultTagger class takes ‘tag’ as a single argument. **NN** is the tag for a singular noun. DefaultTagger is most useful when it gets to work with most common part-of-speech tag. that’s why a noun tag is recommended.**Example of POS Tagging**

Consider the sentence: “The quick brown fox jumps over the lazy dog.”

**After performing POS Tagging:**

* “The” is tagged as determiner (DT)
* “quick” is tagged as adjective (JJ)
* “brown” is tagged as adjective (JJ)
* “fox” is tagged as noun (NN)
* “jumps” is tagged as verb (VBZ)
* “over” is tagged as preposition (IN)
* “the” is tagged as determiner (DT)
* “lazy” is tagged as adjective (JJ)
* “dog” is tagged as noun (NN)

By offering insights into the grammatical structure, this tagging aids machines in comprehending not just individual words but also the connections between them inside a phrase. For many[NLP](https://www.geeksforgeeks.org/nlp-how-tokenizing-text-sentence-words-works/) applications, like text summarization, sentiment analysis, and machine translation, this kind of data is essential.

**Workflow of POS Tagging in NLP**

The following are the processes in a typical natural language processing (NLP) example of part-of-speech (POS) tagging:

* Tokenization: Divide the input text into discrete tokens, which are usually units of words or subwords. The first stage in NLP tasks is tokenization.
* **Loading Language Models:** To utilize a library such as NLTK or SpaCy, be sure to load the relevant language model. These models offer a foundation for comprehending a language’s grammatical structure since they have been trained on a vast amount of linguistic data.
* **Text Processing**: If required, preprocess the text to handle special characters, convert it to lowercase, or eliminate superfluous information. Correct PoS labeling is aided by clear text.
* **Linguistic Analysis**: To determine the text’s grammatical structure, use linguistic analysis. This entails understanding each word’s purpose inside the sentence, including whether it is an adjective, verb, noun, or other.
* **Part-of-Speech Tagging:** To determine the text’s grammatical structure, use linguistic analysis. This entails understanding each word’s purpose inside the sentence, including whether it is an adjective, verb, noun, or other.
* **Results Analysis:** Verify the accuracy and consistency of the PoS tagging findings with the source text. Determine and correct any possible problems or mistagging.

1. **Explain Chunking or shallow parsing**

**Shallow parsing** (also **chunking** or **light**[**parsing**](https://en.wikipedia.org/wiki/Parsing)) is an analysis of a [sentence](https://en.wikipedia.org/wiki/Sentence_(linguistics)) which first identifies constituent parts of sentences (nouns, verbs, adjectives, etc.) and then links them to higher order units that have discrete grammatical meanings ([noun](https://en.wikipedia.org/wiki/Noun) groups or [phrases](https://en.wikipedia.org/wiki/Noun_phrase), verb groups, etc.). While the most elementary chunking algorithms simply link constituent parts on the basis of elementary search patterns (e.g., as specified by [regular expressions](https://en.wikipedia.org/wiki/Regular_expression)), approaches that use [machine learning techniques](https://en.wikipedia.org/wiki/Machine_learning) (classifiers, [topic modeling](https://en.wikipedia.org/wiki/Topic_modeling), etc.) can take contextual information into account and thus compose chunks in such a way that they better reflect the semantic relations between the basic constituents.[[1]](https://en.wikipedia.org/wiki/Shallow_parsing#cite_note-1) That is, these more advanced methods get around the problem that combinations of elementary constituents can have different higher level meanings depending on the context of the sentence.

It is a technique widely used in [natural language processing](https://en.wikipedia.org/wiki/Natural_language_processing). It is similar to the concept of [lexical analysis](https://en.wikipedia.org/wiki/Lexical_analysis) for computer languages. Under the name "shallow structure hypothesis", it is also used as an explanation for why [second language](https://en.wikipedia.org/wiki/Second_language) learners often fail to parse complex sentences correctly.[[2]](https://en.wikipedia.org/wiki/Shallow_parsing#cite_note-2)

1. **Explain Noun Phrase (NP) chunking**

In the last post, I covered [Part of Speech Tagging](http://localhost:8000/pos.html), which is the process of tagging words with their grammatical parts. Here I will cover **Noun Chunking** or **Noun Phrase Chunking**, or **Base Noun Phrases** [1](http://localhost:8000/nounphrase.html#fn:SPACY). Chunking builds upon these grammatical parts to identify groups of words that go together to form symbolic meaning. This can be an adjective that goes along with a noun or a group of nouns related to each other. Below is a simple example using Spacy to demonstrate how it works.

 import spacy  
  
nlp = spacy.load("en\_core\_web\_sm")  
doc = nlp("Five Words in Orange Neon")  
for chunk in doc.noun\_chunks:  
 print(chunk.text)

 Five Words  
Orange Neon

In this simple five-word sentence, we can see how it identified **Five Words** and **Orange Neon** as the chunks.

For this post, we'll take this blob of text about Chatbots[2](http://localhost:8000/nounphrase.html#fn:WIKI) and compare both Spacy's and NLTK's methods for chunking:

A chatbot (also known as a talkbot, chatterbot, Bot, IM bot, interactive agent, or Artificial Conversational Entity) is a computer program or an artificial intelligence which conducts a conversation via auditory or textual methods. Such programs are often designed to convincingly simulate how a human would behave as a conversational partner, thereby passing the Turing test. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatterbots use sophisticated natural language processing systems, but many simpler systems scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.

1. **Explain Named Entity Recognition**



**What is named entity recognition (NER)?**

Named entity recognition (NER) is a natural language processing ([NLP](https://www.techtarget.com/searchenterpriseai/definition/natural-language-processing-NLP)) method that extracts information from text. NER involves detecting and categorizing important information in text known as [*named entities*](https://www.techtarget.com/searchbusinessanalytics/definition/named-entity). Named entities refer to the key subjects of a piece of text, such as names, locations, companies, events and products, as well as themes, topics, times, monetary values and percentages.

NER is also referred to as *entity extraction*, *chunking* and *identification*. It's used in many fields in artificial intelligence ([AI](https://www.techtarget.com/searchenterpriseai/definition/AI-Artificial-Intelligence)), including machine learning ([ML](https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML)), [deep learning](https://www.techtarget.com/searchenterpriseai/definition/deep-learning-deep-neural-network) and [neural networks](https://www.techtarget.com/searchenterpriseai/definition/neural-network). NER is a key component of NLP systems, such as chatbots, [sentiment analysis](https://www.techtarget.com/searchbusinessanalytics/definition/opinion-mining-sentiment-mining) tools and search engines. It's used in healthcare, finance, human resources (HR), customer support, higher education and social media analysis.

**What is the purpose of NER?**

NER identifies, categorizes and extracts the most important pieces of information from [unstructured text](https://www.techtarget.com/searchbusinessanalytics/definition/unstructured-data) without requiring time-consuming human analysis. It's particularly useful for quickly extracting key information from large amounts of data because it automates the extraction process.

NER delivers [critical insights to organizations](https://www.computerweekly.com/news/252471606/Dutch-group-works-on-digitising-collaborator-archives-created-in-aftermath-of-World-War-2) about their customers, products, competition and market trends. For example, companies use it to detect when they're mentioned in publications. Healthcare providers use it to extract key medical information from patient records.

As NER models improve their ability to correctly identify important information, they are helping improve AI systems in general. These systems are enhancing AI language comprehension capabilities in areas such as summarization and translation systems and the ability of AI systems to analyze text.

**How does NER work?**

NER uses algorithms that function based on grammar, statistical NLP models and [predictive models](https://www.techtarget.com/searchenterpriseai/definition/predictive-modeling). These algorithms are trained on data sets that people label with predefined named entity categories, such as people, locations, organizations, expressions, percentages and monetary values. Categories are identified with abbreviations; for example, LOC is used for location, PER for persons and ORG for organizations.

Named entity recognition can identify and categorize key pieces of information in unstructured text.

Once an NER learning model has been trained on textual data and entity types, it automatically analyzes new unstructured text, categorizing named entities and [semantic](https://www.techtarget.com/searchdatamanagement/definition/semantic-technology) meaning based on its training. When the information category of a piece of text is recognized, an information extraction utility extracts the named entity's related information and constructs a machine-readable document that other tools can process to extract meaning.

**What are the three types of NER?**

The three most commonly used NER systems are the following:

1. **Supervised machine learning-based systems** use ML models trained on texts humans have pre-labeled with named entity categories. [Supervised machine learning](https://www.techtarget.com/searchenterpriseai/definition/supervised-learning) approaches use algorithms such as conditional random fields and maximum entropy, two complex statistical [language models](https://www.techtarget.com/searchenterpriseai/definition/language-modeling). This method is effective for parsing semantic meanings and other complexities, though it requires large volumes of training data.
2. **Rules-based systems**use rules to extract information. Rules can include capitalizations or titles, such as "Dr." This method requires a lot of human intervention to input, monitor and tweak the rules, and it might miss textual variations not included in its training annotations. It's thought that rules-based systems don't handle complexity as well as machine learning models.
3. **Dictionary-based systems**use a dictionary with an extensive vocabulary and synonym collection to cross-check and identify named entities. This method might have trouble classifying named entities with variations in spellings.

There are also several emerging NER methods:

* **Unsupervised machine learning systems** use ML systems not already pre-trained on annotated text data. [Unsupervised learning](https://www.techtarget.com/searchenterpriseai/definition/unsupervised-learning) models are thought to be capable of processing more complex tasks than supervised systems.
* **Bootstrapping systems**, also known as *self-supervised*, predictively categorize named entities based on grammatical characteristics, such as capitalization, parts-of-speech tags and other pre-trained categories. A human then fine-tunes the bootstrap system, labeling the system's predictions as correct or incorrect and adding the correct ones to a new training set.
* **Neural network systems**build an NER model using neural networks, bidirectional architecture learning models, such as [Bidirectional Encoder Representations from Transformers](https://www.techtarget.com/searchenterpriseai/definition/BERT-language-model), and encoding techniques. This approach minimizes human interaction.
* **Statistical systems** use probabilistic models trained on textual patterns and relationships to predict named entities in new text data.
* **Semantic role labeling systems**preprocesses an NER model with semantic learning techniques to teach it the context and relationships between categories.
* **Hybrid systems** use aspects of multiple systems in a combined approach.

**Who uses NER?**

Various industries and applications use NER in different ways:

* **Chatbots.** OpenAI's generative AI, ChatGPT, Google's Bard and other [chatbots](https://www.techtarget.com/searchcustomerexperience/definition/chatbot) use NER models to identify relevant entities mentioned in user queries and conversations. This helps them understand the context of a user's question and improves their responses.
* **Customer support.** Named entity recognition systems can organize customer feedback and complaints by product name and identify common or trending complaints about specific products or branch locations. This helps [customer support](https://www.techtarget.com/searchcustomerexperience/definition/customer-service-and-support) teams prepare for incoming queries, respond faster and establish automated systems that route customers to relevant support desks and sections of FAQ pages.
* **Finance.**NER can extract figures from private markets, loans and earnings reports, increasing the speed and accuracy of analyzing profitability and credit risk. NER can also extract names and companies mentioned in social media and other online posts, helping financial institutions monitor trends and developments that could affect stock prices.
* **Healthcare.** NER tools can extract critical information from lab reports and patients' [electronic health records](https://www.techtarget.com/searchhealthit/definition/electronic-health-record-EHR), helping healthcare providers reduce workloads, analyze data faster and more accurately, and improve care.
* **Higher education.** NER enables students, researchers and professors to quickly summarize volumes of papers and archival material, as well as find relevant subjects, topics and themes.
* **HR.** These systems can streamline recruitment and hiring by summarizing applicants' resumes and extracting information, such as qualifications, education and references. NER can also filter employee complaints and queries to the relevant departments, helping to organize internal workflows.
* **News providers.** News providers use NER to analyze the many articles and social media posts they need to review and to categorize the content into important information and trends. This helps them quickly understand and report on news and current events.
* **Recommendation engines.** Many companies use NER to improve the relevancy of their [recommendation engines](https://www.techtarget.com/whatis/definition/recommendation-engine). For instance, companies like Netflix use NER to analyze users' searches and viewing histories to provide personalized recommendations.
* **Search engines.** NER is critical to [search engines](https://www.techtarget.com/whatis/definition/search-engine), identifying and categorizing subjects mentioned on the web and in searches. This helps search engines understand the relevancy of subjects to a user's search and provide users with accurate results.
* **Sentiment analysis.** NER is a key component of sentiment analysis. It extracts product names, brands and other information mentioned in customer reviews, social media posts and other unstructured text. The sentiment analysis tool then analyzes the information to determine the author's sentiment. NER is also used to analyze employee sentiment in survey responses and complaints.

**NER benefits and challenges**

There are several benefits and challenges to NER.

**Benefits of NER**

Named entity recognition provides a range of advantages when used appropriately:

* Automates the information extraction of large amounts of data.
* Analyzes key information in unstructured text.
* Facilitates the analysis of emerging trends.
* Eliminates human error in analysis.
* Is used in almost all industries.
* Frees up time for employees to perform other tasks.
* Improves the precision of NLP tasks and processes.

**NER challenges**

NER also comes with its own set of issues:

* Has difficulty in analyzing [lexical ambiguities](https://www.techtarget.com/whatis/definition/lexical-ambiguity), semantics and evolving usages of language in text.
* Runs into problems with spelling variations.
* Doesn't know all foreign words.
* Can have issues with spoken word text, such as telephone conversations.
* Leads to many state-of-the-art NER models [reporting limited performance measures](https://arxiv.org/ftp/arxiv/papers/2205/2205.00034.pdf).
* Can require large volumes of training data or a lot of human intervention.
* Can be prone to bias in results if the [ML algorithm has hidden bias](https://www.techtarget.com/searchenterpriseai/feature/Bias-in-machine-learning-examples-Policing-banking-COVID-19).

There are several steps involved with getting the NER process to produce analysis.

**Natural Language Toolkit vs. SpaCy**

Natural Language Toolkit (NLTK) and SpaCy are two NER programs with unique differences. NLTK is based on Python's NLP library and provides several algorithms. Often used for teaching NLP to beginners, as well as researchers building applications from the ground level, NLTK uses [strings](https://www.techtarget.com/whatis/definition/search-string) as inputs and outputs in preprocessing. It provides [tokenization](https://www.techtarget.com/searchsecurity/definition/tokenization), stemming, part-of-speech tagging and parsing and can be trained on customized data.

SpaCy, on the other hand, is open source and uses a single [stemmer](https://www.techtarget.com/searchenterpriseai/definition/stemming) algorithm well suited for concrete tasks. It is often used for building professional NLP applications and is object-oriented in preprocessing. SpaCy is also able to handle large data volumes, extract relationships between entities and offer support for word [vectors](https://www.techtarget.com/whatis/definition/vector). It is considered faster than NLTK.