Concept of Tokenization, Stemming, and Lemmatization

In this section, we will understand what is tokenization, stemming, and lemmatization.

**Tokenization**

It may be defined as the process of breaking the given text i.e. the character sequence into smaller units called tokens. The tokens may be the words, numbers or punctuation marks. It is also called word segmentation. Following is a simple example of tokenization −

**Input** − Mango, banana, pineapple and apple all are fruits.

**Output** −Tokenization

The process of breaking the given text can be done with the help of locating the word boundaries. The ending of a word and the beginning of a new word are called word boundaries. The writing system and the typographical structure of the words influence the boundaries.

In the Python NLTK module, we have different packages related to tokenization which we can use to divide the text into tokens as per our requirements. Some of the packages are as follows −

sent\_tokenize package

As the name suggest, this package will divide the input text into sentences. We can import this package with the help of the following Python code −

from nltk.tokenize import sent\_tokenize

word\_tokenize package

This package divides the input text into words. We can import this package with the help of the following Python code −

from nltk.tokenize import word\_tokenize

WordPunctTokenizer package

This package divides the input text into words as well as the punctuation marks. We can import this package with the help of the following Python code −

from nltk.tokenize import WordPuncttokenizer

Stemming

While working with words, we come across a lot of variations due to grammatical reasons. The concept of variations here means that we have to deal with different forms of the same words like ***democracy, democratic,*** and ***democratization***. It is very necessary for machines to understand that these different words have the same base form. In this way, it would be useful to extract the base forms of the words while we are analyzing the text.

We can achieve this by stemming. In this way, we can say that stemming is the heuristic process of extracting the base forms of the words by chopping off the ends of words.

In the Python NLTK module, we have different packages related to stemming. These packages can be used to get the base forms of word. These packages use algorithms. Some of the packages are as follows −

PorterStemmer package

This Python package uses the Porter’s algorithm to extract the base form. We can import this package with the help of the following Python code −

from nltk.stem.porter import PorterStemmer

For example, if we will give the word **‘writing’** as the input to this stemmer them we will get the word **‘write’** after stemming.

LancasterStemmer package

This Python package will use the Lancaster’s algorithm to extract the base form. We can import this package with the help of the following Python code −

from nltk.stem.lancaster import LancasterStemmer

For example, if we will give the word **‘writing’** as the input to this stemmer them we will get the word **‘write’** after stemming.

SnowballStemmer package

This Python package will use the snowball’s algorithm to extract the base form. We can import this package with the help of the following Python code −

from nltk.stem.snowball import SnowballStemmer

For example, if we will give the word **‘writing’** as the input to this stemmer them we will get the word **‘write’** after stemming.

All of these algorithms have different level of strictness. If we compare these three stemmers then the Porter stemmers is the least strict and Lancaster is the strictest. Snowball stemmer is good to use in terms of speed as well as strictness.

Lemmatization

We can also extract the base form of words by lemmatization. It basically does this task with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only. This kind of base form of any word is called lemma.

The main difference between stemming and lemmatization is the use of vocabulary and morphological analysis of the words. Another difference is that stemming most commonly collapses derivationally related words whereas lemmatization commonly only collapses the different inflectional forms of a lemma. For example, if we provide the word saw as the input word then stemming might return the word ‘s’ but lemmatization would attempt to return the word either see or saw depending on whether the use of the token was a verb or a noun.

In the Python NLTK module, we have the following package related to lemmatization process which we can use to get the base forms of word −

WordNetLemmatizer package

This Python package will extract the base form of the word depending upon whether it is used as a noun or as a verb. We can import this package with the help of the following Python code −

from nltk.stem import WordNetLemmatizer

Chunking: Dividing Data into Chunks

It is one of the important processes in natural language processing. The main job of chunking is to identify the parts of speech and short phrases like noun phrases. We have already studied the process of tokenization, the creation of tokens. Chunking basically is the labeling of those tokens. In other words, chunking will show us the structure of the sentence.

In the following section, we will learn about the different types of Chunking.

Types of chunking

There are two types of chunking. The types are as follows −

Chunking up

In this process of chunking, the object, things, etc. move towards being more general and the language gets more abstract. There are more chances of agreement. In this process, we zoom out. For example, if we will chunk up the question that “for what purpose cars are”? We may get the answer “transport”.

Chunking down

In this process of chunking, the object, things, etc. move towards being more specific and the language gets more penetrated. The deeper structure would be examined in chunking down. In this process, we zoom in. For example, if we chunk down the question “Tell specifically about a car”? We will get smaller pieces of information about the car.

**Example**

In this example, we will do Noun-Phrase chunking, a category of chunking which will find the noun phrases chunks in the sentence, by using the NLTK module in Python −

**Follow these steps in python for implementing noun phrase chunking −**

**Step 1** − In this step, we need to define the grammar for chunking. It would consist of the rules which we need to follow.

**Step 2** − In this step, we need to create a chunk parser. It would parse the grammar and give the output.

**Step 3** − In this last step, the output is produced in a tree format.

Let us import the necessary NLTK package as follows −

import nltk

Now, we need to define the sentence. Here, DT means the determinant, VBP means the verb, JJ means the adjective, IN means the preposition and NN means the noun.

sentence=[("a","DT"),("clever","JJ"),("fox","NN"),("was","VBP"),

("jumping","VBP"),("over","IN"),("the","DT"),("wall","NN")]

Now, we need to give the grammar. Here, we will give the grammar in the form of regular expression.

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

We need to define a parser which will parse the grammar.

parser\_chunking = nltk.RegexpParser(grammar)

The parser parses the sentence as follows −

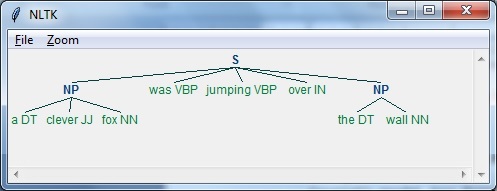
parser\_chunking.parse(sentence)

Next, we need to get the output. The output is generated in the simple variable called **output\_chunk**.

Output\_chunk = parser\_chunking.parse(sentence)

Upon execution of the following code, we can draw our output in the form of a tree.

output.draw()



Bag of Word (BoW) Model

Bag of Word (BoW), a model in natural language processing, is basically used to extract the features from text so that the text can be used in modeling such that in machine learning algorithms.

Now the question arises that why we need to extract the features from text. It is because the machine learning algorithms cannot work with raw data and they need numeric data so that they can extract meaningful information out of it. The conversion of text data into numeric data is called feature extraction or feature encoding.

How it works

This is very simple approach for extracting the features from text. Suppose we have a text document and we want to convert it into numeric data or say want to extract the features out of it then first of all this model extracts a vocabulary from all the words in the document. Then by using a document term matrix, it will build a model. In this way, BoW represents the document as a bag of words only. Any information about the order or structure of words in the document is discarded.

Concept of document term matrix

The BoW algorithm builds a model by using the document term matrix. As the name suggests, the document term matrix is the matrix of various word counts that occur in the document. With the help of this matrix, the text document can be represented as a weighted combination of various words. By setting the threshold and choosing the words that are more meaningful, we can build a histogram of all the words in the documents that can be used as a feature vector. Following is an example to understand the concept of document term matrix −

**Example**

Suppose we have the following two sentences −

* **Sentence 1** − We are using the Bag of Words model.
* **Sentence 2** − Bag of Words model is used for extracting the features.

Now, by considering these two sentences, we have the following 13 distinct words −

* we
* are
* using
* the
* bag
* of
* words
* model
* is
* used
* for
* extracting
* features

Now, we need to build a histogram for each sentence by using the word count in each sentence −

* **Sentence 1** − [1,1,1,1,1,1,1,1,0,0,0,0,0]
* **Sentence 2** − [0,0,0,1,1,1,1,1,1,1,1,1,1]

In this way, we have the feature vectors that have been extracted. Each feature vector is 13-dimensional because we have 13 distinct words.

Concept of the Statistics

The concept of the statistics is called TermFrequency-Inverse Document Frequency (tf-idf). Every word is important in the document. The statistics help us nderstand the importance of every word.

Term Frequency(tf)

It is the measure of how frequently each word appears in a document. It can be obtained by dividing the count of each word by the total number of words in a given document.

Inverse Document Frequency(idf)

It is the measure of how unique a word is to this document in the given set of documents. For calculating idf and formulating a distinctive feature vector, we need to reduce the weights of commonly occurring words like the and weigh up the rare words.

Building a Bag of Words Model in NLTK

In this section, we will define a collection of strings by using CountVectorizer to create vectors from these sentences.

Let us import the necessary package −

from sklearn.feature\_extraction.text import CountVectorizer

Sentences = ['We are using the Bag of Word model', 'Bag of Word model is

used for extracting the features.']

vectorizer\_count = CountVectorizer()

features\_text = vectorizer.fit\_transform(Sentences).todense()

**print(vectorizer.vocabulary\_)**

The above program generates the output as shown below. It shows that we have 13 distinct words in the above two sentences −

{'we': 11, 'are': 0, 'using': 10, 'the': 8, 'bag': 1, 'of': 7,

'word': 12, 'model': 6, 'is': 5, 'used': 9, 'for': 4, 'extracting': 2, 'features': 3}

These are the feature vectors (text to numeric form) which can be used for machine learning.

Solving Problems

In this section, we will solve a few related problems.

Category Prediction

In a set of documents, not only the words but the category of the words is also important; in which category of text a particular word falls. For example, we want to predict whether a given sentence belongs to the category email, news, sports, computer, etc. In the following example, we are going to use tf-idf to formulate a feature vector to find the category of documents. We will use the data from 20 newsgroup dataset of sklearn.

We need to import the necessary packages −

from sklearn.datasets import fetch\_20newsgroups

from sklearn.naive\_bayes import MultinomialNB

from sklearn.feature\_extraction.text import TfidfTransformer

from sklearn.feature\_extraction.text import CountVectorizer

Define the category map. We are using five different categories named Religion, Autos, Sports, Electronics and Space.

category\_map = {'talk.religion.misc':'Religion','rec.autos''Autos',

'rec.sport.hockey':'Hockey','sci.electronics':'Electronics', 'sci.space': 'Space'}

Create the training set −

training\_data = fetch\_20newsgroups(subset = 'train',

categories = category\_map.keys(), shuffle = True, random\_state = 5)

Build a count vectorizer and extract the term counts −

vectorizer\_count = CountVectorizer()

train\_tc = vectorizer\_count.fit\_transform(training\_data.data)

print("\nDimensions of training data:", train\_tc.shape)

The tf-idf transformer is created as follows −

tfidf = TfidfTransformer()

train\_tfidf = tfidf.fit\_transform(train\_tc)

Now, define the test data −

input\_data = [

'Discovery was a space shuttle',

'Hindu, Christian, Sikh all are religions',

'We must have to drive safely',

'Puck is a disk made of rubber',

'Television, Microwave, Refrigrated all uses electricity'

]

The above data will help us train a Multinomial Naive Bayes classifier −

classifier = MultinomialNB().fit(train\_tfidf, training\_data.target)

Transform the input data using the count vectorizer −

input\_tc = vectorizer\_count.transform(input\_data)

Now, we will transform the vectorized data using the tfidf transformer −

input\_tfidf = tfidf.transform(input\_tc)

We will predict the output categories −

predictions = classifier.predict(input\_tfidf)

The output is generated as follows −

for sent, category in zip(input\_data, predictions):

print('\nInput Data:', sent, '\n Category:', \

category\_map[training\_data.target\_names[category]])

The category predictor generates the following output −

Dimensions of training data: (2755, 39297)

Input Data: Discovery was a space shuttle

Category: Space

Input Data: Hindu, Christian, Sikh all are religions

Category: Religion

Input Data: We must have to drive safely

Category: Autos

Input Data: Puck is a disk made of rubber

Category: Hockey

Input Data: Television, Microwave, Refrigrated all uses electricity

Category: Electronics

Gender Finder

In this problem statement, a classifier would be trained to find the gender (male or female) by providing the names. We need to use a heuristic to construct a feature vector and train the classifier. We will be using the labeled data from the scikit-learn package. Following is the Python code to build a gender finder −

Let us import the necessary packages −

import random

from nltk import NaiveBayesClassifier

from nltk.classify import accuracy as nltk\_accuracy

from nltk.corpus import names

Now we need to extract the last N letters from the input word. These letters will act as features −

def extract\_features(word, N = 2):

last\_n\_letters = word[-N:]

return {'feature': last\_n\_letters.lower()}

if \_\_name\_\_=='\_\_main\_\_':

Create the training data using labeled names (male as well as female) available in NLTK −

male\_list = [(name, 'male') for name in names.words('male.txt')]

female\_list = [(name, 'female') for name in names.words('female.txt')]

data = (male\_list + female\_list)

random.seed(5)

random.shuffle(data)

Now, test data will be created as follows −

namesInput = ['Rajesh', 'Gaurav', 'Swati', 'Shubha']

Define the number of samples used for train and test with the following code

train\_sample = int(0.8 \* len(data))

Now, we need to iterate through different lengths so that the accuracy can be compared −

for i in range(1, 6):

print('\nNumber of end letters:', i)

features = [(extract\_features(n, i), gender) for (n, gender) in data]

train\_data, test\_data = features[:train\_sample],

features[train\_sample:]

classifier = NaiveBayesClassifier.train(train\_data)

The accuracy of the classifier can be computed as follows −

accuracy\_classifier = round(100 \* nltk\_accuracy(classifier, test\_data), 2)

print('Accuracy = ' + str(accuracy\_classifier) + '%')

Now, we can predict the output −

for name in namesInput:

print(name, '==>', classifier.classify(extract\_features(name, i)))

The above program will generate the following output −

Number of end letters: 1

Accuracy = 74.7%

Rajesh -> female

Gaurav -> male

Swati -> female

Shubha -> female

Number of end letters: 2

Accuracy = 78.79%

Rajesh -> male

Gaurav -> male

Swati -> female

Shubha -> female

Number of end letters: 3

Accuracy = 77.22%

Rajesh -> male

Gaurav -> female

Swati -> female

Shubha -> female

Number of end letters: 4

Accuracy = 69.98%

Rajesh -> female

Gaurav -> female

Swati -> female

Shubha -> female

Number of end letters: 5

Accuracy = 64.63%

Rajesh -> female

Gaurav -> female

Swati -> female

Shubha -> female

In the above output, we can see that accuracy in maximum number of end letters are two and it is decreasing as the number of end letters are increasing.

Topic Modeling: Identifying Patterns in Text Data

We know that generally documents are grouped into topics. Sometimes we need to identify the patterns in a text that correspond to a particular topic. The technique of doing this is called topic modeling. In other words, we can say that topic modeling is a technique to uncover abstract themes or hidden structure in the given set of documents.

We can use the topic modeling technique in the following scenarios −

Text Classification

With the help of topic modeling, classification can be improved because it groups similar words together rather than using each word separately as a feature.

Recommender Systems

With the help of topic modeling, we can build the recommender systems by using similarity measures.

Algorithms for Topic Modeling

Topic modeling can be implemented by using algorithms. The algorithms are as follows −

Latent Dirichlet Allocation(LDA)

This algorithm is the most popular for topic modeling. It uses the probabilistic graphical models for implementing topic modeling. We need to import gensim package in Python for using LDA slgorithm.

Latent Semantic Analysis(LDA) or Latent Semantic Indexing(LSI)

This algorithm is based upon Linear Algebra. Basically it uses the concept of SVD (Singular Value Decomposition) on the document term matrix.

Non-Negative Matrix Factorization (NMF)

It is also based upon Linear Algebra.

All of the above mentioned algorithms for topic modeling would have the **number of topics** as a parameter, **Document-Word Matrix** as an input and **WTM (Word Topic Matrix)** & **TDM (Topic Document Matrix)** as output.