Basic Concepts of Natural Language Processing (NLP) Models and Python Implementation

Prasun Biswas Jul 1 · 7 min read

Source: Photo by Ogelamar on Unsplash

In the data science domain, Natural Language Processing (NLP) is a very important component for its vast applications in various industries/sectors. For a human it's pretty easy to understand the language but machines are not capable enough to recognize it easily. NLP is the technique that enables the machines to interpret and to understand the way humans communicate.

At present social media is a golden data mine for natural languages, be it any type of reviews from any online sites (Amazon, Google, etc.), or simply posts from Twitter, Facebook, LinkedIn, or emails. The business use cases (Classifications, Text Summarization, Triaging, Interactive voice responses (IVR), Language translation, Chatbots) might be different in each sector but NLP defines the core underlying solution of these use cases.

Natural languages are a free form of text which means it is very much unstructured in nature. So, cleaning and preparing the data to extract the features are very important

packages and develop an NLP-based classification model.

> ***A) Data Cleaning***
>
> ***B) Tokenization***
>
> ***C) Vectorization/Word Embedding***
>
> ***D) Model Development***

A) Data Cleaning

As mentioned above, data cleaning is basic but very important step in NLP. Below are a few ways for data cleaning. Let's consider the below line.

```
line = 'Reaching out for HELP. Please meet me in LONDON at 6 a.m
xyz@abc.com #urgent'
```

1. Remove stopwords: There are a few words which are very commonly used when humans interact, but these words don't make any sense or add any extra value. Additionally, there might be few words which are not required for the business case given in hand. So, these words need to be deleted from the data.

The NLTK package has a defined set of stopwords for different languages like English. Here, we will focus on 'english' stopwords. One can also consider additional stopwords if required.

```
import nltk

import re

from nltk.corpus import stopwords
```

```
stopword = stopwords.words("english")
stopword.extend(extra_list)
line = ' '.join([i for i in line.split() if i not in stopword])
```

2. Make lower case: This is required to make all the words in lowercase just to maintain the uniformity.

```
line = line.lower()
```

3. Lemmatization: This helps to reduce the words into a single form. For example:

```
def lemmatize_text(text):
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()
return [lemmatizer.lemmatize(w) for w in w_tokenizer.tokenize(text)]
lemmatize_text(line)
```

4. Stemming: This helps to reduce the words into their root form. For example:

```
def stem_porter(text):
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
ps = nltk.PorterStemmer()
return [ps.stem(w) for w in w_tokenizer.tokenize(text)]
stem_porter(line)
```

5. Removal of regular expressions: Regular expression helps to identify and to get rid of different patterns which are not required in the text.

```
line = re.sub('\S*@\S*\s?',' ',line) #email remove

line = re.sub('\s+',' ',line) #new line character remove

line = re.sub("\'"," ",line) #single quote remove

line = re.sub('_',' ',line) #underscore remove

line = re.sub('http\S*\s?',' ',line) #link remove

line = ' '.join([i for i in line.split() if i.find('#') < 0])
#hasgtag remove

line = ' '.join([i for i in line.split() if i in
re.findall(r'\w+',line)]) #only keep words and numbers
```

6. Parts-of-Speech (POS) tagging: This helps to identify the parts of speech. Based on the use case one can keep or remove some of them.

```
import spacy

from spacy.tokenizer import Tokenizer

nlp = spacy.load("en_core_web_sm")

tokens_spacy = nlp(line)

for token in tokens_spacy:

print(token.text, ': ', token.pos_, ': ', token.is_stop)
```

7. Named-Entity-Recognition (NER): This helps to identify and categorize the different groups which includes names, places, currency etc.

B) Tokenization

This is one of the common practices while working on text data. This helps to split a phrase, sentence, or paragraph into small units like words or terms. Each unit is called a token. There are different types of tokenization. We have already used this in above examples for stemming, POS tagging, and NER. Below are different ways to tokenize the text.

```
str1 = “I am eating pizza and, coke.”
```

1) Tokenization using split () function: Returns list of strings after breaking the given string by the specified separator. By default, a separator is a space.

```
str1.split()
['I', 'am', 'eating', 'pizza', 'and,', 'coke.']
```

2) Tokenization using Regular expression: Returns list based on the regular expressions.

```
re.findall(“[\w]+”,str1)
['I', 'am', 'eating', 'pizza', 'and', 'coke']
```

3) Tokenization using NLTK: There are different types of tokenizers under NLTK package like word tokenizer (word_tokenize), regex tokenizer (RegexpTokenizer), whitespace tokenizer (WhitespaceTokenizer) etc.

3.1)

```
word_tokenize(str1)
['I', 'am', 'eating', 'pizza', 'and', ',', 'coke', '.']
```

3.2)

```
space_tokenizer = nltk.tokenize.WhitespaceTokenizer()
space _tokenizer.tokenize(str1)
['I', 'am', 'eating', 'pizza', 'and,', 'coke,']
```

3.3)

```
reg_tokenizer = nltk.tokenize.RegexpTokenizer("[A-Za-z]+")
reg_tokenizer.tokenize(str1)
['I', 'am', 'eating', 'pizza', 'and', 'coke']
```

Further we can use these tokenized forms to count the number of words in a text or frequency of the words in a text.

C) Vectorization/Word Embedding

Once cleaning and tokenization is done, extracting features from the clean data is very important as the machine doesn't understand the words but numbers. Vectorization helps to map the words to a vector of real numbers, which further helps into predictions. This helps to extract the important features. Below are few techniques used for the purpose:

1. CountVec: Count number of times a particular word appears in the document. CountVectorizer helps to get this count.

```
count_vec = CountVectorizer(analyzer='word', ngram_range=(1, 3),
stop_words = 'english')

count_vec.fit(str2)

count = count_vec.transform(str2)

vectors = count_vec.get_feature_names()

smatrix = count_vec.transform(str2)

dense = smatrix.todense()

dense_list = dense.tolist()

df_countvec = pd.DataFrame(dense_list,columns=vectors)
```

2. TF-IDF: Term frequency Inverse document frequency (TF-IDF) provides an overall weightage of a word in the document. TfidfVectorizer helps to get this weighted score.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vec = TfidfVectorizer (analyzer='word', ngram_range=(1, 3),
stop_words = 'english')

tfidf_vec.fit(str2)

tfidf = tfidf_vec.transform(str2)

vectors = tfidf_vec.get_feature_names()

smatrix = tfidf_vec.transform(str2)

dense = smatrix.todense()

dense_list = dense.tolist()

df_ tfidf = pd.DataFrame(dense_list,columns=vectors)
```

Comparison: CountVec might provide biased results in favour of most frequent words. This ignores the rare words which might have higher importance. This means we need

```
str2 = ['I am going to test Covid',
'It seems ABC hospital is doing the Covid test',
'Covaxin is still in WIP phase']
```

CountVec Output (Image by Author)

TF-IDF Output (Image by Author)

Both these models provide one number (count or weight) per word. But to understand each word's context and to identify the content, one vector per word is much more appropriate. **Word2Vec** provides one vector per word by skimming through the given documents, which is more useful than a simple bag of words or TF-IDF. But Word2Vec lacks to address 'local understanding' of the relationship, which was answered by **Glove**. Glove is a pre-trained vectorization technique that not only understands the local context but also the relationship with global words. Apart from Glove, **FastText** is another popular technique for word embedding, which works better for rare words or Out-of-vocabulary(OOV).

Having said all these, Bag of words or TF-IDF (mainly) is vastly used till now and very much integrated part of day-to-day NLP problems. As this article intends to cover only basic NLP concepts, Glove or FastText are not covered in details.

D) Model Development

Image by Author

One can use any of the classification models like logistic regression, random forest (RF), support vector machines (SVM) or any deep learning models like RNN, LSTM or state-of-art model like BERT, GPT3 to predict the label. As a measure of accuracy ROC, Recall, F1-score can be used based the problem statement in hand.

Before running the model, let's split the data into train and test.

```
X = df.drop(columns='label')

y = df['label']

X_train, X_test, y_train, y_test = train_test_split(X,
y,test_size=0.2, random_state=420,stratify=y)
```

Now, model the data and check accuracy metrics.

```
from sklearn.linear_model import LogisticRegression,
RandomForestClassifier

model = LogisticRegression()

#model = RandomForestClassifier()

model.fit(X_train, y_train)

y_pred_class = model.predict(X_test)

print('Accuracy: ', metrics.accuracy_score(y_test, y_pred_class))
```

Apart from classification problems, NLP can be leveraged for several use cases like text summarization, Q&A, topic modeling (link), text translation etc.

Hope this article will help you to get the feel how to start with any NLP based problem. Till then, Happy Learning!

Disclaimer: The views expressed in this article is the opinion of the author in his personal capacity and not of his employer

NLP Tokenization Python Text Mining Data Science