NLP Text Encoding - A Beginner's Guide

By Sunil Ghimire- "Your legacy will never be erased"

Welcome to the second tutorial on Natural Language Processing Basic Course. In our previous tutorial, we had covered about Data Cleaning, Splitting, Normalizing, and Stemming. In this session, I will be covering Text Encoding methods like Index based encoding, Bag of Words, Tf-IDF, and so on.

TABLE OF CONTENTS

Introduction to Text Encoding

Let's try to understand a few basic facts first…

1. Machine doesn't understand characters, words, or sentences.
2. Machines can only process numbers
3. Text data must be encoded as numbers for input or output for any machine.

As mentioned in the above points we cannot pass raw text into machines as input until and unless we convert them into numbers, hence we need to perform text encoding. Text encoding is a process to convert meaningful text into number/vector representation so as to preserve the context and relationship between words and sentences, such that a machine can understand the pattern associated with any text and can make out the context of sentences.

There are a lot of methods to convert Text into numerical vectors, they are:

1. Index-Based Encoding
2. Bag of Words (BOW)
3. TF-IDF Encoding
4. Word2Vector Encoding
5. BERT Encoding

As this is a basic explanation of NLP text encoding hence we will be skipping the last 2 methods, i.e. Word2Vector and BERT as they are quite complex and powerful implementations of Deep Learning method-based Text Embedding to convert text into vector encoding.

Before we deep dive into each method let's set some ground examples so as to make it easier to follow through.

1. Document Corpus: This is the whole set of text we have, basically our text corpus, which can be anything like news articles, blogs, etc.

   Example: We have 5 sentences namely, [*"this is a good phone", "this is a bad mobile", "she is a good cat", "he has a bad temper", "this mobile phone is not good"*]
2. Data Corpus: It is the collection of unique words in our document corpus. Example: [*"a", "bad", "cat", "good", "has", "he", "is", "mobile", "not", "phone", "she", "temper", "this"*]

```
document_corpus = ["this is good phone phone" ,
                   "this is bad mobile mobile" ,
                   "she is good good cat" ,
                   "he has bad temper temper" ,
                   "this mobile phone phone is not good good"]
data_corpus = set()
for row in document_corpus:
    for word in row.split(" "):
        if word not in data_corpus:
            data_corpus.add(word)

data_corpus=sorted(data_corpus)
print(data_corpus)

Output:
  ['bad', 'cat', 'good', 'has', 'he', 'is', 'mobile', 'not', 'phone',
'she', 'temper', 'this']
```

TYPES OF TEXT ENCODING

1. Index-Based Indexing

As the name mentions, Index-based, we surely need to give all the unique words an index, like we have separated out our Data Corpus, now we can index them individually, like - bad: 1, cat: 2, …, this: 13, and so on.

Now we have assigned a unique index to all the words so that based on the index we can uniquely identify them, we can convert our sentences using this index-based method.

```
res = len(max(document_corpus, key = len).split(" "))
index_based_encoding=[]
for row in document_corpus:
```

```
    row_encoding = []
    split = row.split(" ")
    for i in range(res):
        if i <= len(split)-1:
            row_encoding.append(data_corpus.index(split[i])+1)
        else:
            row_encoding.append(0)
    index_based_encoding.append(row_encoding)
print(index_based_encoding)

Output:
  [[12, 6, 3, 9, 9, 0, 0, 0], [12, 6, 1, 7, 7, 0, 0, 0], [10, 6, 3, 3, 2,
0, 0, 0], [5, 4, 1, 11, 11, 0, 0, 0], [12, 7, 9, 9, 6, 8, 3, 3]]
```

It is very trivial to understand that we are just replacing the words in each sentence with their respective indexes. But there is a tiny bit of issue which needs to be addressed first and that is the consistency of the input. Our input needs to be of the same length as our model, it cannot vary. It might vary in the real world but needs to be taken care of when we are using it as input to our model.

Now as we can see the first sentence has 5 words, but the last sentence has 6 words, this will cause an imbalance in our model. So to take care of that issue what we do is max padding, which means we take the longest sentence from our document corpus and we pad the other sentence to be as long. This means if all of our sentences are 5 words and one sentence is 6 words we will make all the sentences of 6 words.

HOW WE ADD EXTRA INDEX??

If you have noticed, we didn't use 0 as an index number, and preferably that will not be used anywhere even if we have 100000 words long data corpus, hence we use 0 as our padding index. This also means that we are appending nothing to our actual sentence as 0 doesn't represent any specific word, hence the integrity of our sentences is intact.

2. BAG OF WORDS (BOW)

Bag of Words or BoW is another form of encoding where we use the whole data corpus to encode our sentences.

Data Corpus: ["a", "bad", "cat", "good", "has", "he", "is", "mobile", "not", "phone", "she", "temper", "this"]

As we know that our data corpus will never change, so if we use this as a baseline to create encodings for our sentences, then we will be on the upper hand to not pad any extra words. Now, 1st sentence we have is "this is a

good phone". This is how our first sentence is represented.

a	bad	cat	good	has	he	is	mobile	not	phone	she	temper	this
1	0	0	1	0	0	1	0	0	1	0	0	1

Now, there are 2 kinds of BOW:

1. **Binary BOW**

```
#BINARY BOW
one_hot_encoding = []
for row in document_corpus:
    row_encoding = []
    split = row.split(" ")
    for word in data_corpus:
        if word in split:
            row_encoding.append(1)
        else:
            row_encoding.append(0)
    one_hot_encoding.append(row_encoding)
print(one_hot_encoding)

OUTPUT:
  [[0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1], [1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
1], [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0, 0, 0,
1, 0], [0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1]]
```

2. **BOW**

```
#BOW ENCODING
one_hot_encoding = []
for row in document_corpus:
    row_encoding = []
    split = row.split(" ")
    for word in data_corpus:
        count = split.count(word)
        if word in split:
            row_encoding.append(count)
        else:
            row_encoding.append(count)
    one_hot_encoding.append(row_encoding)
print(one_hot_encoding)
```

```
OUTPUT:
  [[0, 0, 1, 0, 0, 1, 0, 0, 2, 0, 0, 1], [1, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0,
1], [0, 1, 2, 0, 0, 1, 0, 0, 0, 1, 0, 0], [1, 0, 0, 1, 1, 0, 0, 0, 0, 0,
2, 0], [0, 0, 2, 0, 0, 1, 1, 1, 2, 0, 0, 1]]
```

The difference between them is, in Binary BoW we encode 1 or 0 for each word appearing or non-appearing in the sentence. We do not take into consideration the frequency of the word appearing in that sentence. In BoW we also take into consideration the frequency of each word occurring in that sentence.

SCIKIT IMPLEMENTATION OF BOW

```
from sklearn.feature_extraction.text import CountVectorizer

# list of text documents
text = ["The quick brown fox jumped over the lazy dog."]

# create the transform
vectorizer = CountVectorizer()

# tokenize and build vocab
vectorizer.fit(text)

# summarize
print(vectorizer.vocabulary_)

# encode document
vector = vectorizer.transform(text)

# summarize encoded vector
print(vector.shape)
print(vector.toarray())

OUTPUT:
  {'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5,
'lazy': 4, 'dog': 1}
  (1, 8)
  [[1 1 1 1 1 1 1 2]]
```

In this cell, we can see how the application of the fit() and transform() functions enabled us to create a vocabulary of 8 words for the document. The vectorizer fit allowed us to build the vocab and the transform encoded it into the number of appearances of each word. The indexing is done alphabetically.

3. TF-IDF ENCODING

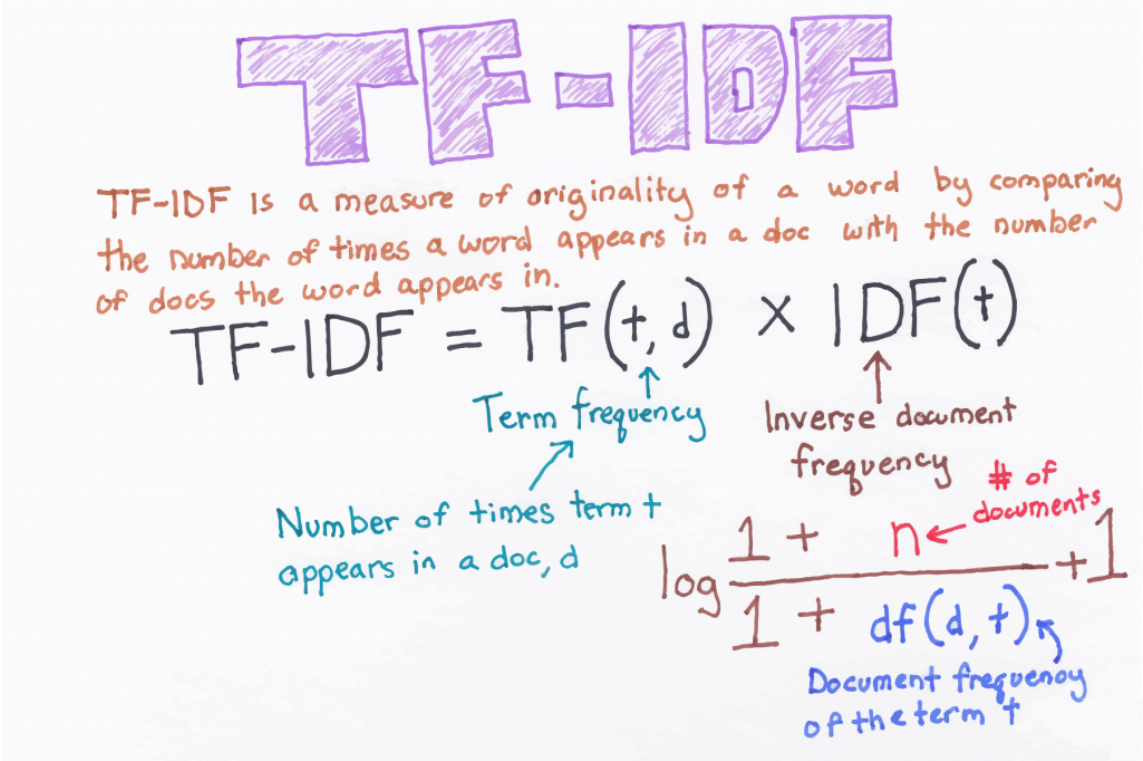

Figure 01:- text encoding tf idf

Term Frequency-Inverse Data Frequency. As the name suggests, here we give every word a relative frequency coding with respect to the current sentence and the whole document.

1. **Term Frequency**: Is the occurrence of the current word in the current sentence w.r.t the total number of words in the current sentence.
2. **Inverse Data Frequency**: Log of the Total number of words in the whole data corpus w.r.t the total number of sentences containing the current word.

$$TF_{i,j} = \frac{n_{i,j}}{\Sigma_k n_{i,j}}$$

Figure 02: TF Formula

$$idf_t = log\frac{N}{df_t}$$

Figure 03: IDF Formula

One thing to note here is we have to calculate the word frequency of each word for that particular sentence, because depending on the number of times a word occurs in a sentence the TF value can change, whereas the IDF value remains constant, until and unless new sentences are getting added.

Data Corpus: [“bad”, “cat”, “good”, “has”, “he”, “is”, “mobile”, “not”, “phone”, “she”, “temper”, “this”]

Here, we will calculate the TF-IDF of the word “this" from the first sentence “this is a good phone phone”. TF is the ratio of a number of “this” words in a sentence1 to a total number of words in a sentence whereas IDF is a logarithmic ratio of the total number of words in the whole data corpus to the total number of sentences having “this” word.

TF : 1 / 5 = 0.2

IDF : log(13 / 3) = 1.4663

TF-IDF : 0.2 * 1.4663 = 0.3226

```
from sklearn.feature_extraction.text import TfidfVectorizer

# list of text documents
text = ["The quick brown fox jumped over the lazy dog.",
          "The dog.",
          "The fox"]
# create the transform
vectorizer = TfidfVectorizer()

# tokenize and build vocab
vectorizer.fit(text)

# summarize
print(vectorizer.vocabulary_)
print(vectorizer.idf_)

# encode document
vector = vectorizer.transform([text[0]])

# summarize encoded vector
print(vector.shape)
print(vector.toarray())

OUTPUT:
  {'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5,
'lazy': 4, 'dog': 1}
[1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
 1.69314718 1.        ]
(1, 8)
[[0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646
  0.36388646 0.42983441]]
```

You can tell now, for example, the word ‘brown’ has a higher multiplier than ‘the’ even though the latter has two

appearances in the first document. This is due to the fact ‘the’ appears in the other two documents which reduces its uniqueness.

```
text = ["The quick quick quick fox jumped over a big dog"]

# encode document
vector = vectorizer.transform(text)

# summarize encoded vector
print(vector.shape)
print(vector.toarray())

OUTPUT:
  (1, 8)
  [[0. 0.21506078 0.21506078 0.28277908 0.  0.28277908 0.84833724
0.16701388]]
```

☺ Thanks for your time ☺

What do you think of this “**NLP Text Encoding - A Beginner’s Guide**“? (Appreciation, Suggestions, and Questions are highly appreciated).