Implement Elias Delta and Elias Gamma and Golomb coding in python

Sample Code :

```
#!/usr/bin/python


                    from math import log,ceil


                    log2 = lambda x: log(x,2)


                    def binary(x,l=1):
                           fmt = '{0:0%db}' % l
                           return fmt.format(x)


                    def unary(x):
                           return x*'1'+'0'


                    def elias_generic(lencoding, x):
                           if x == 0: return '0'

                           l = 1+int(log2(x))
                           a = x - 2**(int(log2(x)))

                           k = int(log2(x))


                           return lencoding(l) + binary(a,k)

                    def golomb(b, x):
                           q = int((x) / b)
                           r = int((x) % b)

                           l = int(ceil(log2(b)))
                           #print q,r,l


                           return unary(q) + binary(r, l)



                    def elias_gamma(x):
```

```
        return elias_generic(unary, x)


def elias_delta(x):
        return elias_generic(elias_gamma,x)

for i in range(11):
        print "%5d: %-10s : %-10s : %-10s" %(i,
elias_gamma(i),elias_delta(i), golomb(3,i))
```

How to process textual data using TF-IDF in Python

-Roy

Computers are good with numbers, but not that much with textual data. One of the most widely used techniques to process textual data is TF-IDF. In this article, we will learn how it works and what are its features.

From our intuition, we think that the words which appear more often should have a greater weight in textual data analysis, but that’s not always the case. Words such as “the”, “will”, and “you”—called **stopwords**—appear the most in a corpus of text, but are of very little significance. Instead, the words which are rare are the ones that actually help in distinguishing between the data, and carry more weight.

An introduction to TF-IDF

TF-IDF stands for "Term Frequenct—Inverse Data Frequency". First, we will learn what this term means mathematically.

Term Frequency (tf): gives us the frequency of the word in each document in the corpus. It is the ratio of number of times the word appears in a document compared to the total number of words in that document. It increases as the number of occurrences of that word within the document increases. Each document has its own tf.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

Inverse Data Frequency (idf): used to calculate the weight of rare words across all documents in the corpus. The words that occur rarely in the corpus have a high IDF score. It is given by the equation below.

$$idf(w) = log(\frac{N}{df_t})$$

Combining these two we come up with the TF-IDF score (w) for a word in a document in the corpus. It is the product of tf and idf:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$$\begin{gathered} tf_{i,j} = \text{number of occurrences of } i \text{ in } j \\ df_i = \text{number of documents containing } i \\ N = \text{total number of documents} \end{gathered}$$

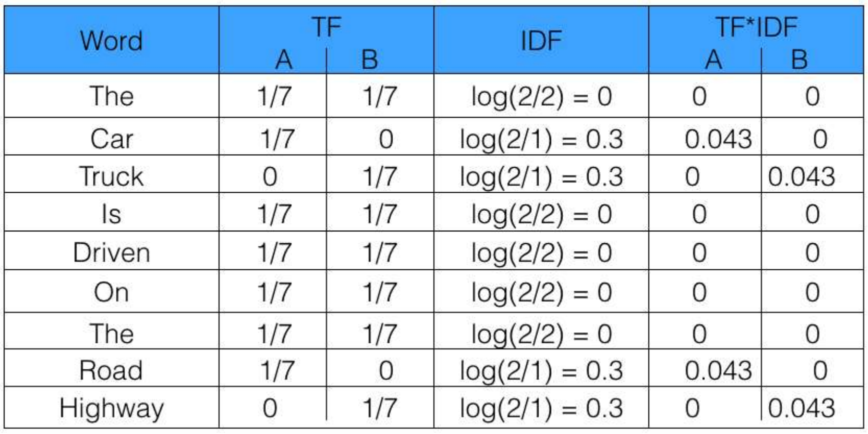
Let’s take an example to get a clearer understanding.

Sentence 1 : The car is driven on the road.

Sentence 2: The truck is driven on the highway.

In this example, each sentence is a separate document.

We will now calculate the TF-IDF for the above two documents, which represent our corpus.



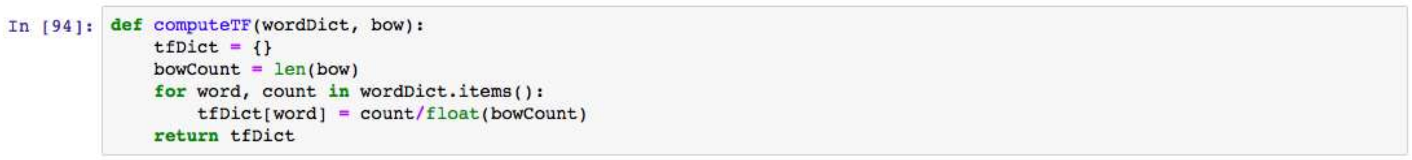
Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

From the above table, we can see that TF-IDF of common words was zero, which shows they are not significant. On the other hand, the TF-IDF of “car” , “truck”, “road”, and “highway” are non-zero. These words have more significance.

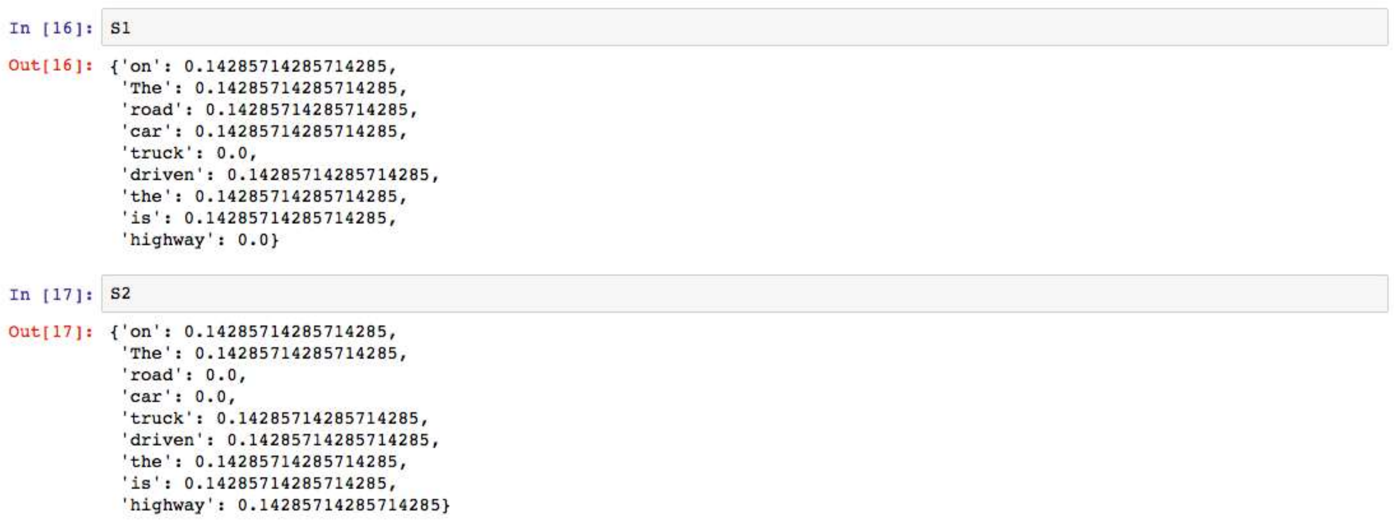
Using Python to calculate TF-IDF

Lets now code TF-IDF in Python from scratch. After that, we will see how we can use sklearn to automate the process.

The function `computeTF` computes the TF score for each word in the corpus, by document.

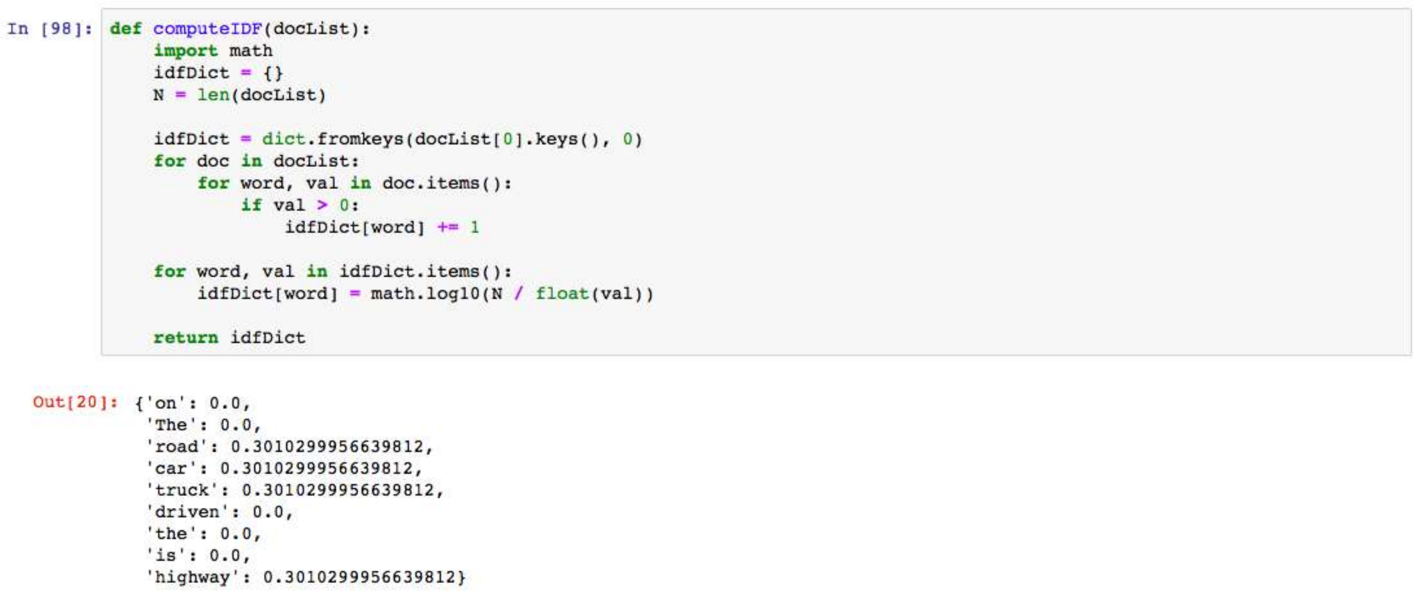


```
In [94]: def computeTF(wordDict, bow):
             tfDict = {}
             bowCount = len(bow)
             for word, count in wordDict.items():
                 tfDict[word] = count/float(bowCount)
             return tfDict
```



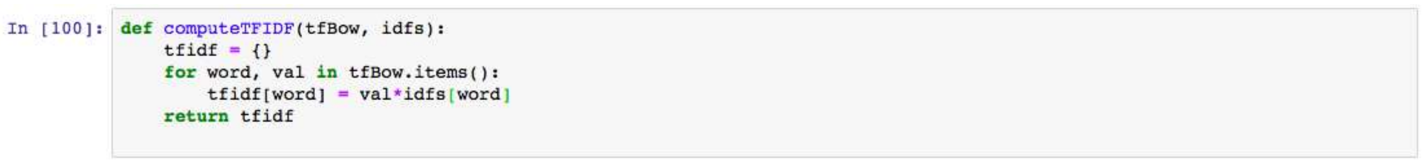
```
In [16]: S1

Out[16]: {'on': 0.14285714285714285,
          'The': 0.14285714285714285,
          'road': 0.14285714285714285,
          'car': 0.14285714285714285,
          'truck': 0.0,
          'driven': 0.14285714285714285,
          'the': 0.14285714285714285,
          'is': 0.14285714285714285,
          'highway': 0.0}

In [17]: S2

Out[17]: {'on': 0.14285714285714285,
          'The': 0.14285714285714285,
          'road': 0.0,
          'car': 0.0,
          'truck': 0.14285714285714285,
          'driven': 0.14285714285714285,
          'the': 0.14285714285714285,
          'is': 0.14285714285714285,
          'highway': 0.14285714285714285}
```

The function `computeIDF` computes the IDF score of every word in the corpus.



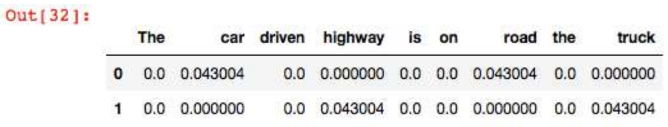
```
In [98]: def computeIDF(docList):
             import math
             idfDict = {}
             N = len(docList)

             idfDict = dict.fromkeys(docList[0].keys(), 0)
             for doc in docList:
                 for word, val in doc.items():
                     if val > 0:
                         idfDict[word] += 1

             for word, val in idfDict.items():
                 idfDict[word] = math.log10(N / float(val))

             return idfDict

Out[20]: {'on': 0.0,
          'The': 0.0,
          'road': 0.3010299956639812,
          'car': 0.3010299956639812,
          'truck': 0.3010299956639812,
          'driven': 0.0,
          'the': 0.0,
          'is': 0.0,
          'highway': 0.3010299956639812}
```

The function `computeTFIDF` below computes the TF-IDF score for each word, by multiplying the TF and IDF scores.



```
In [100]: def computeTFIDF(tfBow, idfs):
              tfidf = {}
              for word, val in tfBow.items():
                  tfidf[word] = val*idfs[word]
              return tfidf
```

The output produced by the above code for the set of documents D1 and D2 is the same as what we manually calculated above in the table.



Out[32]:

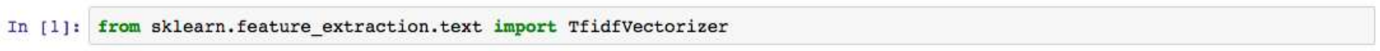
	The	car	driven	highway	is	on	road	the	truck
0	0.0	0.043004	0.0	0.000000	0.0	0.0	0.043004	0.0	0.000000
1	0.0	0.000000	0.0	0.043004	0.0	0.0	0.000000	0.0	0.043004

You can refer to this link for the complete implementation.

sklearn

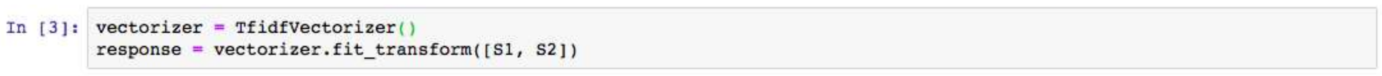
Now we will see how we can implement this using sklearn in Python.

First, we will import `TfidfVectorizer` from `sklearn.feature_extraction.text` :



```
In [1]: from sklearn.feature_extraction.text import TfidfVectorizer
```

Now we will initialise the `vectorizer` and then call fit and transform over it to calculate the TF-IDF score for the text.



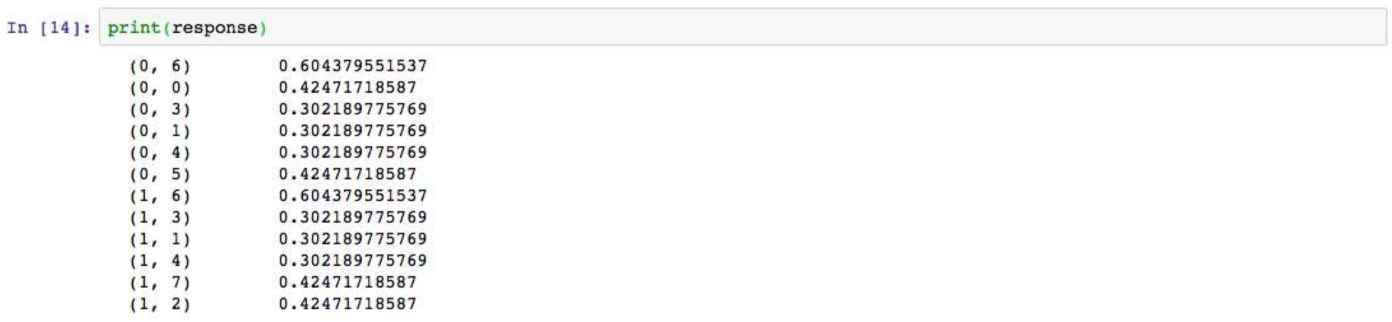
```
In [3]: vectorizer = TfidfVectorizer()
        response = vectorizer.fit_transform([S1, S2])
```

Under the hood, the sklearn fit_transform executes the following `fit` and `transform` functions. These can be found in the official sklearn library at GitHub.

```
    def fit(self, X, y=None):
        """Learn the idf vector (global term weights)
        Parameters
        ----------
        X : sparse matrix, [n_samples, n_features]
            a matrix of term/token counts
        """
        if not sp.issparse(X):
            X = sp.csc_matrix(X)
        if self.use_idf:
            n_samples, n_features = X.shape
            df = _document_frequency(X)

            # perform idf smoothing if required
            df += int(self.smooth_idf)
            n_samples += int(self.smooth_idf)

            # log+1 instead of log makes sure terms with z
            # suppressed entirely.
            idf = np.log(float(n_samples) / df) + 1.0
            self._idf_diag = sp.spdiags(idf, diags=0, m=n_
                                        n=n_features, form

        return self

    def transform(self, X, copy=True):
        """Transform a count matrix to a tf or tf-idf repr
        Parameters
        ----------
        X : sparse matrix, [n_samples, n_features]
            a matrix of term/token counts
        copy : boolean, default True
            Whether to copy X and operate on the copy or p
            operations.
        Returns
        -------
        vectors : sparse matrix, [n_samples, n_features]
        """
        if hasattr(X, 'dtype') and np.issubdtype(X.dtype,
            # preserve float family dtype
            X = sp.csr_matrix(X, copy=copy)
        else:
```

One thing to notice in the above code is that, instead of just the log of n_samples, 1 has been added to n_samples to calculate the IDF score. This ensures that the words with an IDF score of zero don't get suppressed entirely.

The output obtained is in the form of a skewed matrix, which is normalised to get the following result.



```
In [14]: print(response)
  (0, 6)	0.604379551537
  (0, 0)	0.42471718587
  (0, 3)	0.302189775769
  (0, 1)	0.302189775769
  (0, 4)	0.302189775769
  (0, 5)	0.42471718587
  (1, 6)	0.604379551537
  (1, 3)	0.302189775769
  (1, 1)	0.302189775769
  (1, 4)	0.302189775769
  (1, 7)	0.42471718587
  (1, 2)	0.42471718587
```

Thus we saw how we can easily code TF-IDF in just 4 lines using sklearn. Now we understand how powerful TF-IDF is as a tool to process textual data out of a corpus. To learn more about sklearn TF-IDF, you can use this link.

Happy coding!

Thanks for reading this article. Be sure to clap and recommend this article if you find it helpful.

For more about programming, you can follow me, so that you get notified every time I come up with a new post.

Cheers!

Also, Let's get connected on **Twitter**, **Linkedin**, **Github** and **Facebook**.