What is TF-IDF?

TF-IDF is a natural language processing technique useful for the extraction of important keywords within a set of documents or chapters. The acronym stands for "term frequency-inverse document frequency" .

TF-IDF can actually be used to extract important keywords from a document to get a sense of what characterizes a document. For example, if you are dealing with Wikipedia articles, you can use tf-idf to extract words that are unique to a given article. These keywords can be used as a very simple summary of a document, and for text-analytics when we look at these keywords in aggregate. It has many uses, most importantly in automated text analysis, and is very useful for scoring words in machine learning algorithms for Natural Language Processing(NLP).

How is TF-IDF calculated?

TF-IDF for a word in a document is calculated by multiplying two different metrics:

- The **term frequency** of a word in a document. There are several ways of calculating this frequency, with the simplest being a raw count of instances a word appears in a document. Then, there are ways to adjust the frequency, by length of a document, or by the raw frequency of the most frequent word in a document.

- The **inverse document frequency** of the word across a set of documents. This metric can be calculated by taking the total number of documents, dividing it by the number of documents that contain a word, and calculating the logarithm.
- So, if the word is very common and appears in many documents, this number will approach 0. Otherwise, it will approach 1.

Multiplying these two numbers results in the TF-IDF score of a word in a document. The higher the score, the more relevant that word is in that particular document.

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

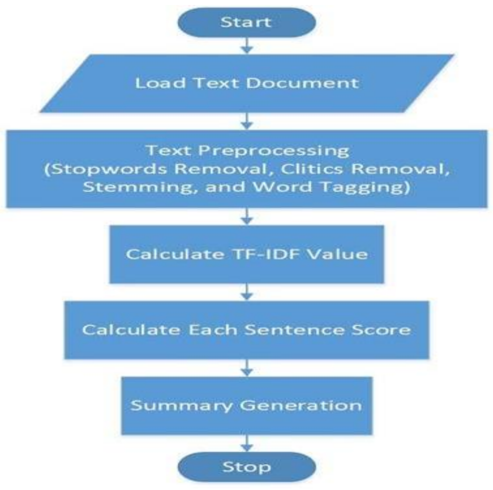
$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Why is TF-IDF used in Machine Learning?

Machine learning with natural language is faced with one major hurdle – its algorithms usually deal with numbers, and natural language well as text. So we need to transform that text into numbers, otherwise it called as text vectorization. It is a fundamental step in the process of machine learning for analysing data, and different vectorization algorithms will drastically affect end results, so you need to choose one that will deliver the results you're hoping for.

Once you've transformed words into numbers, in the way the machine learning algorithms can understand, the TF-IDF score can be fed to algorithms such as naive Bayes and Support Vector Machines.

Flow Chart


Start
Load Text Document
Text Preprocessing (Stopwords Removal, Clitics Removal, Stemming, and Word Tagging)
Calculate TF-IDF Value
Calculate Each Sentence Score
Summary Generation
Stop

The 9 steps implementation

1.Tokenize the sentences

```
import nltk

from nltk.tokenize import sent_tokenize
text='Now, assume we have 10 million documents and the word apple appea
rs in one thousand of these.'
sentences = sent_tokenize(text) # NLTK function
total_documents = len(sentences)
total_documents
```

We’ll tokenize the sentences here instead of words. And we’ll give weight to these sentences.

2.Create the Frequency matrix of the words in each sentence.

```
def _create_frequency_matrix(sentences):
    frequency_matrix = {}
    stopWords = set(stopwords.words("english"))
    ps = PorterStemmer()

    for sent in sentences:
        freq_table = {}
        words = nltk.word_tokenize(sent)
        for word in words:
            word = word.lower()
            word = ps.stem(word)
            if word in stopWords:
                continue

            if word in freq_table:
                freq_table[word] += 1
            else:
                freq_table[word] = 1

        frequency_matrix[sent[:15]] = freq_table

    return frequency_matrix

fm=_create_frequency_matrix(sentences)
fm
```

3. Calculate Term Frequency and generate a matrix

We'll find the Term Frequency for each word in a paragraph.

Now, remember the definition of **TF,**

TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document)

```
def _create_tf_matrix(fm):
    tf_matrix = {}

    for sent, f_table in fm.items():
        tf_table = {}

        count_words_in_sentence = len(f_table)
        for word, count in f_table.items():
            tf_table[word] = count / count_words_in_sentence

        tf_matrix[sent] = tf_table

    return tf_matrix
_create_tf_matrix(fm)
```

4. Creating a table for documents per words

```
def _create_documents_per_words(fm):
    word_per_doc_table = {}

    for sent, f_table in fm.items():
        for word, count in f_table.items():
            if word in word_per_doc_table:
                word_per_doc_table[word] += 1
            else:
                word_per_doc_table[word] = 1

    return word_per_doc_table
cd=_create_documents_per_words(fm)
```

5. Calculate IDF and generate a matrix

We'll find the IDF for each word in a paragraph.

Now, remember the definition of **IDF,**

IDF(t) = log_e(Total number of documents / Number of documents with term t in it)

6. Calculate TF-IDF and generate a matrix

```
def _create_tf_idf_matrix(tf_matrix, idf_matrix):
    tf_idf_matrix = {}

    for (sent1, f_table1), (sent2, f_table2) in zip(tf_matrix.items(),
idf_matrix.items()):

        tf_idf_table = {}

        for (word1, value1), (word2, value2) in zip(f_table1.items(),
                                                    f_table2.items()):
 # here, keys are the same in both the table
            tf_idf_table[word1] = float(value1 * value2)

        tf_idf_matrix[sent1] = tf_idf_table

    return tf_idf_matrix
_create_tf_idf_matrix(t, idf_matrix)
```

7. Score the sentences

```
def _score_sentences(tf_idf_matrix) -> dict:


    sentenceValue = {}

    for sent, f_table in tf_idf_matrix.items():
        total_score_per_sentence = 0

        count_words_in_sentence = len(f_table)
        for word, score in f_table.items():
            total_score_per_sentence += score

        sentenceValue[sent] = total_score_per_sentence / count_words_in
_sentence

    return sentenceValue
```

```
s=_score_sentences(tf_idf_matrix)
```

8. Find the threshold

```
def _find_average_score(sentenceValue) -> int:
    """
    Find the average score from the sentence value dictionary
    :rtype: int
    """
    sumValues = 0
    for entry in sentenceValue:
        sumValues += sentenceValue[entry]

    # Average value of a sentence from original summary_text
    average = (sumValues / len(sentenceValue))

    return average
_find_average_score(s)
```

9. Generate the summary

```
def _generate_summary(sentences, sentenceValue, threshold):
    sentence_count = 0
    summary = ''

    for sentence in sentences:
        if sentence[:15] in sentenceValue and sentenceValue[sentence[:1
5]] >= (threshold):
            summary += " " + sentence
            sentence_count += 1

    return summary
_generate_summary(sentences, sentenceValue, threshold)
```

Advantages

- stop worrying about using the stop-words,
- successfully hunt words with higher search volumes and lower competition,
- be sure to have words that make your content unique and relevant to the user, etc

Disadvantages

- It computes document similarity directly in the word-count space, which may be slow for large vocabularies.
- It assumes that the counts of different words provide independent evidence of similarity.
- It makes no use of semantic similarities between words.