Marius Borcan Follow
Jun 8, 2020 · 6 min read ★ · Listen

TF-IDF Explained And Python Sklearn Implementation

What is TF-IDF and how you can implement it in Python and Scikit-Learn.

TF-IDF is an information retrieval and information extraction subtask which aims to express the importance of a word to a document which is part of a colection of documents which we usually name a corpus. It is usually used by some search engines to help them obtain better results which are more relevant to a specific query. In this article we are going to discuss what exactly is TF-IDF, explain the math behind it and then we will see how we can implement it in Python by using the Scikit-Learn library.

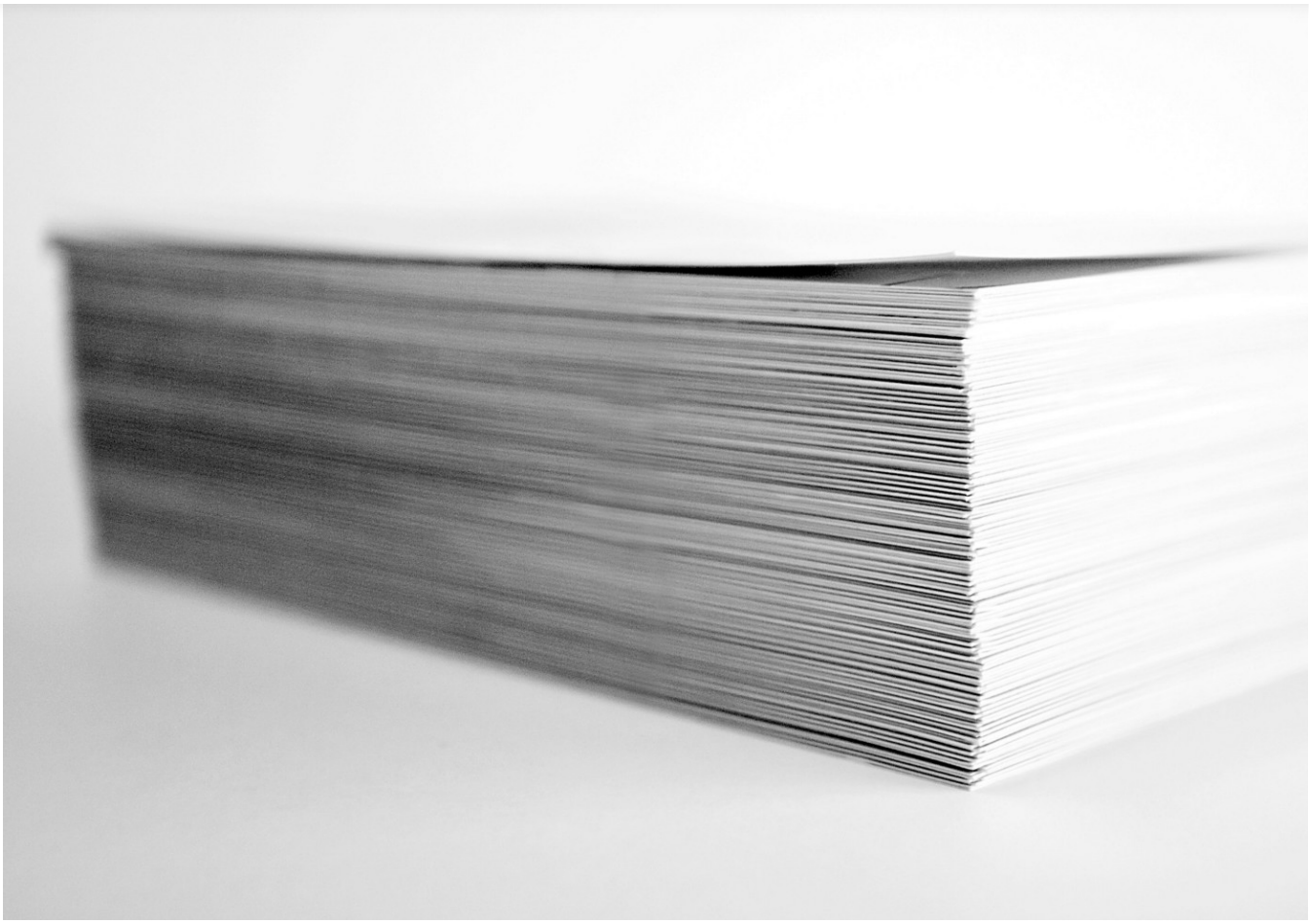

Photo by ron dyar on Unsplash

This article was originally published on the Programmer Backpack Blog. Make sure to visit this blog if you want to read more stories of this kind.

Thank you so much for reading this! Interested in more stories like this? Follow me on Twitter at @b_dmarius and I'll post there every new article.

Article Overview

- What is TF-IDF
- TF-IDF formula explained
- TF-IDF sklearn python implementation
- TfIdfVectorizer vs TfIdfTransformer — what is the difference
- TF-IDF Applications

TF-IDF stands for Term Frequency — Inverse Document Frequency and is a statistic that aims to better define how important a word is for a document, while also taking into account the relation to other documents from the same corpus.

This is performed by looking at how many times a word appears into a document while also paying attention to how many times the same word appears in other documents in the corpus.

The rationale behind this is the following:

- a word that frequently appears in a document has more relevancy for that document, meaning that there is higher probability that the document is about or in relation to that specific word
- a word that frequently appears in more documents may prevent us from finding the right document in a collection; the word is relevant either for all documents or for none. Either way, it will not help us filter out a single document or a small subset of documents from the whole set.

So then **TF-IDF is a score** which is applied to every word in every document in our dataset. And for every word, the TF-IDF value increases with every appearance of the word in a document, but is gradually decreased with every appearance in other documents. And the maths for that is in the next section.

TF-IDF Formula Explained

Now let's take a look at the simple formula behind the TF-IDF statistical measure. First let's define some notations:

- N is the number of documents we have in our dataset
- d is a given document from our dataset
- D is the collection of all documents
- w is a given word in a document

First step is to calculate the term frequency, our first measure if the score.

$$tf(w,d) = log(1 + f(w,d))$$

Term Frequency Formula

Here *f(w,d)* is the frequency of word *w* in document *d*.

Second step is to calculate the inverse term frequency.

$$idf(w,D) = log(\frac{N}{f(w,D)})$$

Inverse Document Frequency Formula

With *N* documents in the dataset and *f(w, D)* the frequency of word *w* in the whole dataset, this number will be lower with more appearances of the word in the whole dataset.

Final step is to compute the TF-IDF score by the following formula:

$$tfidf(w,d,D) = tf(w,d) * idf(w,D)$$

Term Frequency — Inverse Document Frequency — Formula

TF-IDF Sklearn Python Implementation

With such awesome libraries like scikit-learn implementing TD-IDF is a breeze. First off we need to install 2 dependencies for our project, so let's do that now.

In order to see the full power of TF-IDF we would actually require a proper, larger dataset. But for the purpose of our article, we only want to focus on implementation, so let's import our dependencies into our project and build our mini-dataset.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfTransformer

dataset = [
    "I enjoy reading about Machine Learning and Machine Learning is
my PhD subject",
    "I would enjoy a walk in the park",
    "I was reading in the library"
]
```

Let's now calculate the TF-IDF score and print out our results.

```
tfIdfVectorizer=TfidfVectorizer(use_idf=True)
tfIdf = tfIdfVectorizer.fit_transform(dataset)
df = pd.DataFrame(tfIdf[0].T.todense(),
index=tfIdfVectorizer.get_feature_names(), columns=["TF-IDF"])
df = df.sort_values('TF-IDF', ascending=False)
print (df.head(25))
```

Let's see our results now.

```
TF-IDF
machine   0.513720
learning  0.513720
about     0.256860
subject   0.256860
phd       0.256860
and       0.256860
my        0.256860
is        0.256860
reading   0.195349
enjoy     0.195349
library   0.000000
park      0.000000
in        0.000000
```


TfidfVectorizer vs TfidfTransformer — what is the difference

If you've ever seen other implementations of TF-IDF you may have seen that there are 2 different ways of implementing TF-IDF using Scikit-Learn. One is using the TfidfVectorizer class(like we just did) and the other one is by using the TfidfTransformer class. You may have wondered what's the difference between the 2 of them, so let's discuss that.

Theoretically speaking, there is actually no difference between the 2 implementations. Practically speaking, we need to write some more code if we want to use TfidfTransformer. The main difference between the 2 implementations is that TfidfVectorizer performs both term frequency and inverse document frequency for you, while using TfidfTransformer will require you to use the CountVectorizer class from Scikit-Learn to perform Term Frequency.

So let's see an alternative TF-IDF implementation and validate the results are the same. We will first need to import 2 additional dependencies to our project.

```
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import CountVectorizer
```

We will use the same mini-dataset we used with the other implementation. Let's write the alternative implementation and print out the results.

```
tfIdfTransformer = TfidfTransformer(use_idf=True)
countVectorizer = CountVectorizer()
wordCount = countVectorizer.fit_transform(dataset)
newTfIdf = tfIdfTransformer.fit_transform(wordCount)
df = pd.DataFrame(newTfIdf[0].T.todense(),
index=countVectorizer.get_feature_names(), columns=["TF-IDF"])
df = df.sort_values('TF-IDF', ascending=False)
print (df.head(25))
```

You can look at the results and see they are the same as the above.


```
phd        0.256860
and        0.256860
my         0.256860
is         0.256860
reading    0.195349
enjoy      0.195349
library    0.000000
park       0.000000
in         0.000000
the        0.000000
walk       0.000000
was        0.000000
would      0.000000
```

TF-IDF Applications

So we've seen that implementing TF-IDF using the right tools is very easy, yet the applications of this algorithm are very powerful. The 2 most common use cases for TF-IDF are:

- **Information retrieval:** by calculating the TF-IDF score of a user query against the whole document set we can figure out how relevant a document is to that given query. Rumour has it that most search engines around use some sort of TF-IDF implementation, but I couldn't verify that information myself, so take this with a grain of salt.
- **Keywords extraction**: The highest ranking words for a document in terms of TF-IDF score can very well represent the keywords of that document(as they make that document stand out from the other documents). So we can very easily use some sort of TF-IDF score computation to extract the keywords from a text.

Conclusions

So in this article we've seen an explanation of what is TF-IDF and how we can explain it mathematically. Then we've seen two alternative implementations using the Scikit-Learn Python library. We've then discussed some possible applications of this algorithm. I hope you enjoyed this!

Thank you so much for reading this article! Interested in more? Follow me on Twitter at @b_dmarius and I'll post there every new article