TF-IDF from scratch in python on real world dataset.

William Scott
Feb 15, 2019 · 19 min read

Table of Contents:

Photo by Sanwal Deen on Unsplash

Introduction: TF-IDF

TF-IDF stands for **"Term Frequency — Inverse Document Frequency"**. This is a technique to quantify a word in documents, we generally compute a weight to each word which signifies the importance of the word in the document and corpus. This method is a widely used technique in Information Retrieval and Text Mining.

If i give you a sentence for example "This building is so tall". Its easy for us to understand the sentence as we know the semantics of the words and the sentence. But how will the computer understand this sentence? The computer can understand any data only in the form of numerical value. So, for this reason we vectorize all of the text so that the computer can understand the text better.

By vectorizing the documents we can further perform multiple tasks such as finding the relevant documents, ranking, clustering and so on. This is the same thing that happens when you perform a google search. The web pages are called documents and the search text with which you search is called a query. google maintains a fixed representation for all of the documents. When you search with a query, google will find the relevance of the query with all of the documents, ranks them in the order of relevance and shows you the top k documents, all of this process is done using the vectorized form of query and documents. Although Googles algorithms are highly sophisticated and optimized, this is their underlying structure.

Now coming back to our TF-IDF,

TF-IDF = Term Frequency (TF) * Inverse Document Frequency (IDF)

Terminology

- t — term (word)
- d — document (set of words)
- N — count of corpus
- corpus — the total document set

Term Frequency

This measures the frequency of a word in a document. This highly depends on the length of the document and the generality of word, for example a very common word such as “was” can appear multiple times in a document. but if we take two documents one which have 100 words and other which have 10,000 words. There is a high probability that the common word such as “was” can be present more in the 10,000 worded document. But we cannot say that the longer document is more important than the shorter document. For this exact reason, we perform a normalization on the frequency value. we divide the the frequency with the total number of words in the document.

Recall that we need to finally vectorize the document, when we are planning to vectorize the documents, we cannot just consider the words that are present in that particular document. If we do that, then the vector length will be different for both the documents, and it will not be feasible to compute the similarity. So, what we do is that we vectorize the documents on the **vocab**. vocab is the list of all possible words in the corpus.

When we are vectorizing the documents, we check for each words count. In worst case if the term doesn’t exist in the document, then that particular TF value will be 0 and in other extreme case, if all the words in the document are same, then it will be 1. The final value of the normalised TF value will be in the range of [0 to 1]. 0, 1 inclusive.

TF is individual to each document and word, hence we can formulate TF as follows.

$$tf(t,d) = \text{count of t in d} \,/\, \text{number of words in d}$$

If we already computed the TF value and if this produces a vectorized form of the document, why not use just TF to find the relevance between documents? why do we need IDF?

Let me explain, though we calculated the TF value, still there are few problems, for example, words which are the most common words such as “is, are” will have very high values, giving those words a very high importance. But using these words to compute the relevance produces bad results. These kind of common words are called stop-words, although we will remove the stop words later in the preprocessing step, finding the importance of the word across all the documents and normalizing using that value represents the documents much better.

Document Frequency

This measures the importance of document in whole set of corpus, this is very similar to TF. The only difference is that TF is frequency counter for a term t in document d, where as DF is the count of **occurrences** of term t in the document set N. In other words, DF is the number of documents in which the word is present. We consider one occurrence if the term consists in the document at least once, we do not need to know the number of times the term is present.

$$df(t) = \text{occurrence of } t \text{ in documents}$$

To keep this also in a range, we normalize by dividing with the total number of documents. Our main goal is to know the informativeness of a term, and DF is the exact inverse of it. that is why we inverse the DF

Inverse Document Frequency

IDF is the inverse of the document frequency which measures the informativeness of term t. When we calculate IDF, it will be very low for the most occurring words such as stop words (because stop words such as “is” is present in almost all of the documents, and N/df will give a very low value to that word). This finally gives what we want, a relative weightage.

$$idf(t) = N/df$$

Now there are few other problems with the IDF, in case of a large corpus, say 10,000, the IDF value explodes. So to dampen the effect we take log of IDF.

During the query time, when a word which is not in vocab occurs, the df will be 0. As we cannot divide by 0, we smoothen the value by adding 1 to the denominator.

$$idf(t) = \log(N/(df+1))$$

Finally, by taking a multiplicative value of TF and IDF, we get the TF-IDF score, there are many different variations of TF-IDF but for now let us concentrate on the this basic version.

$$\text{tf-idf}(t, d) = \text{tf}(t, d) * \log(N/(\text{df}+1))$$

Implementing on a real world dataset

Now that we learnt what is TF-IDF let us try to find out the relevance of documents that are available online.

The dataset we are going to use are archives of few stories, this dataset has lots of documents in different formats. Download the dataset and open your notebooks, Jupyter Notebooks i mean 😜.

Dataset Link: http://archives.textfiles.com/stories.zip

Step 1: Analysing Dataset

The first step in any of the Machine Learning tasks is to analyse the data. So if we look at the dataset, at first glance, we see all the documents with words in English. Each document has different names and there are two folders in it.

Now one of the important tasks is to identify the title in the body, if we analyse the documents, there are different patterns of alignment of title. But most of the titles are centre aligned. Now we need to figure out a way to extract the title. But before we get all pumped up and start coding, let us analyse the dataset little deep.

Take few minutes to analyse the dataset yourself. Try to explore…

Upon more inspection, we can notice that there's an index.html in each folder (including the root), which contains all the document names and their titles. So, let us consider ourselves lucky as the titles are given to us, without exhaustively extracting titles from each document.

Step 2: Extracting Title & Body:

There is no specific way to do this, this totally depends on the problem statement at hand and on the analysis we do on the dataset.

As we have already found that the titles and the document names are in the index.html, we need to extract those names and titles. We are lucky that html has tags which we can use as patterns to extract our required content.

Before we start extracting the titles and file names, as we have different folders, first let's crawl to the folders to later read all the index.html files at once.

```
[x[0] for x in os.walk(str(os.getcwd())+'/stories/')]
```

os.walk gives us the files in the directory, os.getcwd gives us the current directory and title and we are going to search in the current directory + stories folder as our data files are in the stories folder.

> Always assume that you are dealing with a huge dataset, this helps in automating the code.

Now we can find that **folders** give extra / for the root folder, so we are going to remove it

```
folders[0] = folders[0][:len(folders[0])-1]
```

the above code removes the last character for the 0th index in folders, which is the root folder

Now, let's crawl through all the index.html to extract their titles. To do that we need to find a pattern to take out the title. As this is in html, our job will be little simpler.

let's see…

We can clearly observe that each file name is enclosed between (**><A HREF=”**) and (”) and each title is between (**
<TD>**) and (**\n**)

We will use simple regular expression to retrieve the name and title. The following code gives the list of all the values that match that pattern. so names and titles variables have the list of all names and titles.

```
names = re.findall('><A HREF="(.*)">', text)
titles = re.findall('<BR><TD> (.*)\n', text)
```

Now that we have code to retrieve the values from index, we just need to iterate to all the folders and get the title and file name from all the index.html files

> *- read the file from index files*
>
> *- extract title and names*
>
> *- iterate to next folder*

```
dataset = []

for i in folders:
    file = open(i+"/index.html", 'r')
    text = file.read().strip()
    file.close()

    file_name = re.findall('><A HREF="(.*)">', text)
    file_title = re.findall('<BR><TD> (.*)\n', text)
```

```
        for j in range(len(file_name)):
            dataset.append((str(i) + str(file_name[j]), file_title[j]))
```

This prepares the indexes of the dataset, which is a tuple of location of file and its title. There is a small issue, the root folder index.html also has folders and its links, we need to remove those

simply use a conditional checker to remove it.

```
if c == False:
    file_name = file_name[2:]
    c = True
```

Step 3: Preprocessing

Preprocessing is one of the major steps when we are dealing with any kind of text models. During this stage we have to look at the distribution of our data, what techniques are needed and how deep we should clean.

This step never has a one hot rule, and totally depends on the problem statement. Few mandatory preprocessing are converting to lowercase, removing punctuation, removing stop words and lemmatization/stemming. In our problem statement it seems like the basic preprocessing steps will be sufficient.

Lowercase

During the text processing each sentence is split to words and each word is considered as a token after preprocessing. Programming languages consider textual data as sensitive, which means that **The** is different from **the.** we humans know that those both belong to same token but due to the character encoding those are considered as different tokens. Converting to lowercase is a very mandatory preprocessing step. As we have all our data in list, numpy has a method which can convert the list of lists to lowercase at once.

```
np.char.lower(data)
```

Stop words

Stop words are the most commonly occurring words which don't give any additional value to the document vector. in-fact removing these will increase computation and space efficiency. nltk library has a method to download the stopwords, so instead of explicitly mentioning all the stopwords ourselves we can just use the nltk library and iterate over all the words and remove the stop words. There are many efficient ways to do this, but ill just give a simple method.

we are going to iterate over all the stop words and not append to the list if it's a stop word

```
new_text = ""
for word in words:
    if word not in stop_words:
        new_text = new_text + " " + word
```

Punctuation

Punctuation are the unnecessary symbols that are in our corpus documents, we should be little careful with what we are doing with this. There might be few problems such as U.S — us “United Stated” being converted to “us” after the preprocessing. hyphen and should usually be dealt little carefully. but for this problem statement we are just going to remove these

```
symbols = "!\"#$%&()*+-./:;<=>?@[\]^_`{|}~\n"
for i in symbols:
    data = np.char.replace(data, i, ' ')
```

We are going to store all our symbols in a variable and iterate that variable removing that particular symbol in the whole dataset. we are using numpy here because our data is stored in list of lists, and numpy is our best best.

Apostrophe

Note that there is no ‘ apostrophe in the punctuation symbols. Because when we remove punctuation first it will convert don’t to dont, and it is a stop word which wont be removed. so what we are doing is we are first removing the stop words, and then symbols and then finally stopwords because few words might still have a apostrophe which are not stop words.

```
return np.char.replace(data, "'", "")
```

Single Characters

Single characters are not much useful in knowing the importance of the document and few final single characters might be irrelevant symbols, so it is always good be remove the single characters.

```
new_text = ""
for w in words:
    if len(w) > 1:
       new_text = new_text + " " + w
```

We just need to iterate to all the words and not append the word if the length is not greater than 1.

Stemming

This is the final and most important part of the preprocessing. stemming converts words to its stem.

For example **playing and played** are the same type of words which basically indicate an action **play.** Stemmer does exactly this, it reduces the word to its stem. we are going to use a library called porter-stemmer which is a rule based stemmer. Porter-Stemmer identifies and removes the suffix or affix of a word. The words given by the stemmer need note be meaningful few times, but it will be identified as the same for the model.

Lemmatisation

Lemmatisation is a way to reduce the word to root synonym of a word. Unlike Stemming, Lemmatisation makes sure that the reduced word is again a dictionary word (word present in the same language). WordNetLemmatizer can be used to lemmatize any word.

Stemming vs Lemmatization

stemming — need not be a dictionary word, removes prefix and affix based on few rules

lemmatization — will be a dictionary word. reduces to a root synonym.

A better efficient way to proceed is to first lemmatise and then stem, but stemming alone is also fine for few problems statements, in this problem statement we are not

going to lemmatise.

Converting Numbers

When user gives a query such as **100 dollars** or **hundred dollars.** For the user both those search terms are same. but out IR model treats them separately, as we are storing 100, dollar, hundred as different tokens. so to make our IR mode little better we need to convert 100 to hundred. To achieve this we are going to use a library called **num2word**.

If we look a little close to the above output, it is giving us few symbols and sentences such as “one hundred ***and*** two”, but damn we just cleaned our data, then how do we handle this? No worries, we will just run the punctuation and stop words again after converting numbers to words.

Preprocessing

Finally, we are going to put in all those preprocessing methods above in another method and we will call that preprocess method.

```
def preprocess(data):
    data = convert_lower_case(data)
    data = remove_punctuation(data)
    data = remove_apostrophe(data)
    data = remove_single_characters(data)
    data = convert_numbers(data)
    data = remove_stop_words(data)
    data = stemming(data)
    data = remove_punctuation(data)
    data = convert_numbers(data)
```

If you look closely, few of the preprocessing methods are repeated again. As discussed, this just helps clean the data little deep. Now we need to read the documents and store their title and the body separately as we are going to use it later. In our problem statement we have very different types of documents, this can cause few errors in reading the documents due to encoding compatibility. to resolve this, just use encoding=”utf8", errors=’ignore’ in the open() method.

Step 3: Calculating TF-IDF

Recall that we need to give different weighs to title and body. Now how are we going to handle that issue? how will the calculation of TF-IDF work in this case?

There is nothing to be worried. Giving different weighs to title and body is a very common approach. We just need to consider the document as body + title, using this we can find the vocab. And we need to give different weights to words in title and different weights to the words in body. To better explain this, let us consider an example.

title = “This is a novel paper”

body = “This paper consists of survey of many papers”

Now, we need to calculate the TF-IDF for body and for title. For the time being let us consider only the word **paper**, and forget about removing stop words.

What is the TF of word **paper** in title? 1/4?

No, it’s 3/13. How? word paper appears in title and body 3 times and the total number of words in title and body is 13 As i mentioned before, we just **consider** the word in title to have different weights, but still we consider the whole document when calculating TF-IDF

Then the TF of **paper** in both title and body is same? Yes, it’s same! it’s just the difference in weights that we are going to give. If the word is present in both title and body, then there wouldn't be any reduction in the TF-IDF value. If the word is present only in title, then the weighs of body for that particular word will not add to the TF of that word, and vice versa.

> *document = body + title*
>
> *TF-IDF(document) = TF-IDF(title) * alpha + TF-IDF(body) * (1-alpha)*

Calculating DF

Let us be smart and calculate DF before hand. We need to iterate through all the words in all the documents and store the document id’s for each word. For this we will use a dictionary as we can use the word as the key and set of documents as the value. I

mentioned set because, even if we are trying to add the document multiple times, set will not just take duplicate values.

```
DF = {}
for i in range(len(processed_text)):
    tokens = processed_text[i]
    for w in tokens:
        try:
            DF[w].add(i)
        except:
            DF[w] = {i}
```

We are going to create a set if the word doesn't have a set yet else add to the set. This condition is checked by the try block. Here processed_text is the body of the document, and we are going to repeat the same for title as well, as we need to consider the DF of whole document.

> *len(DF) will give the unique words*

DF will have the word as the key and list of doc id's as the value. but for DF we don't actually need the list of docs, we just need the count. so we are going to replace the list with its count.

There we have it, the count we need for all the words. To find the total unique words in our vocabulary, we need to take all the keys of DF.

Calculating TF-IDF

Recall that we need to maintain different weights for title and body. To calculate TF-IDF of body or title we need to consider both the title and body. To make our job little easier, let's use dictionary as with *(document, token)* pair as key and any TF-IDF score as the value. We just need to iterate over all the documents, we can use the Coutner which can give us the frequency of the tokens, calculate tf and idf and finally store as a (doc, token) pair in tf_idf. tf_idf dictionoary is for body, we will use the same logic for to build a dictionary tf_idf_title for the words in title.

```
tf_idf = {}
for i in range(N):
    tokens = processed_text[i]
    counter = Counter(tokens + processed_title[i])
    for token in np.unique(tokens):
        tf = counter[token]/words_count
        df = doc_freq(token)
        idf = np.log(N/(df+1))
        tf_idf[doc, token] = tf*idf
```

Coming to the calculation of different weights. Firstly, we need to maintain a value alpha, which is the weight for body, then obviously 1-alpha will be the weight for title. Now let us delve into a little math, we discussed that TF-IDF value of a word will be same for both body and title if the word is present in both the places. We will maintained two different tf-idf dictionaries, one for body and one for title. What we are going to do is little smart, we are going to calculate TF-IDF for body, multiply the whole body TF-IDF values with alpha, iterate the tokens in title, replace the title TF-IDF value in the body TF-IDF value of the (document, token) pair exists. Take some time to process this :P

> *Flow:*
>
> *- Calculate TF-IDF for Body for all docs*

> *- Calculate TF-IDF for title for all docs*
>
> *- multiply the Body TF-IDF with alpha*
>
> *- Iterate Title IF-IDF for every (doc, token)*
>
> *— if token is in body, replace the Body(doc, token) value with the value in Title(doc, token)*

I know this is not easy at first to understand, but still let me explain why the above flow works, as we know that the tf-idf for body and title will be same if the token is in both places, The weights that we use for body and title sum up to one

TF-IDF = body_tf-idf * body_weight + title_tf-idf*title_weight

body_weight + title_weight = 1

When a token is in both the places, then the final TF-IDF will be same as taking either body or title tf_idf. That is exactly what we are doing in the above flow. So, finally we have a dictionary tf_idf which has the values as a (doc, token) pair.

Step 4: Ranking using Matching Score

Matching score is the most simplest way to calculate the similarity, in this method, we **add tf_idf values of the tokens that are in query for every document**. for example, if the query "hello world", we need to check in every document if these words exists and if the word exists, then the tf_idf value is added to the matching score of that particular doc_id. in the end we will sort and take the top k documents.

Mentioned above is the theoretical concept, but as we are using dictionary to hold our dataset, what we are going to do is we will iterate over all of values in the dictionary and check if the value is present in the token. As our dictionary is a (document, token) key, when we find a token which is in query we will add the document id to another dictionary along with the tf-idf value. Finally we will just take the top k documents again.

```
def matching_score(query):
    query_weights = {}
    for key in tf_idf:
        if key[1] in tokens:
            query_weights[key[0]] += tf_idf[key]
```

key[0] is the documentid, key[1] is the token.

Step 5: Ranking using Cosine Similarity

When we have a perfectly working **Matching Score**, why do we need cosine similarity again? though **Matching Score** gives relevant documents, it quite fails when we give long queries, it will not be able to rank them properly. what cosine similarly does is that it will mark all the documents as vectors of tf-idf tokens and plots them from the centre. what will happen is that, few times the query length would be small but it might be closely related to the document, in these cases cosine similarity is the best want to find relevance.

Observe the above plot, the blue vectors are the documents and the red vector is the query, as we can clearly see, though the manhattan distance (green line) is very high for document d1, the query is still close to document d1. In these kind of cases cosine similarity would be better as it considers the angle between those two vectors. But Matching Score will return document d3 but that is not very closely related.

Vectorization

To compute any of the above, the simplest way is to convert everything to a vector and then compute the cosine similarity. So, let's convert the query and documents to vectors. We are going to use total_vocab variable which has all the list of unique tokens to generate a index for each token, and we will use numpy of shape (docs, total_vocab) to store the document vectors.

```
# Document Vectorization
D = np.zeros((N, total_vocab_size))
for i in tf_idf:
    ind = total_vocab.index(i[1])
    D[i[0]][ind] = tf_idf[i]
```

For vector, we need to calculate the TF-IDF values, TF we can calculate from the query itself, and we can make use of DF that we created for the document frequency, and finally we will store in a (1,vocab_size) numpy array to store the tf-idf values, index of the token will be decided from the total_voab list

```
Q = np.zeros((len(total_vocab)))
counter = Counter(tokens)
words_count = len(tokens)
query_weights = {}
for token in np.unique(tokens):
    tf = counter[token]/words_count
    df = doc_freq(token)
    idf = math.log((N+1)/(df+1))
```

Now, all we have to do is calculate the cosine similarity for all the documents and return the maximum k documents. Cosine similarity is defined as follows.

$$np.dot(a, b)/(norm(a)*norm(b))$$

Analysis

I took the text from doc_id **200** (for me) and pasted some content with long query and short query in both matching score and cosine similarity.

Short Query

Long Query

As we can see from above document 200 is always highly rated in cosine method than the matching method, this is because cosine similarity learns the context more.

Try out yourself. Click here for git repo.

Libraries Used

- nltk, numpy, re, mat, num2words

Information Retrieval Series:

- 1. Introduction
- 2. Unigram Indexing & Positional Indexing
- 3. TF-IDF
- More to come…

Data Science Tf Idf Data Preprocessing Information Retrieval NLP