**Transforming tokens into useful features (BOW,TF-IDF)**

## Bag of words (BOW)

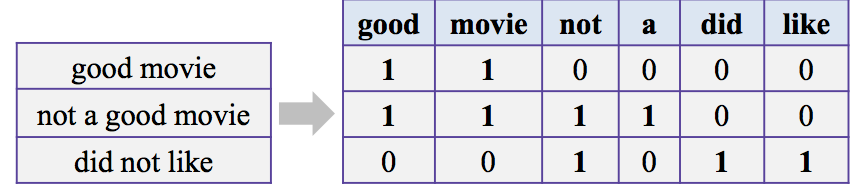
One of the most common techniques to transform tokens into features is Bag of Words. What doe we do? We simply count the occurrences of a particular token in our text.

* Motivation: we’re actually looking for **marker words** like “excellent” or “disappointed”, and **make decisions based on absence or presence** **of that particular word.**
* For each token, we will have a feature column and this is called text vectorization.

Let's take an example of three reviews like: "good movie", "not a good movie", "did not like". Let's take all the possible words or tokens and introduce a new feature or column that will correspond to that particular word.

So, let's take for example "good movie". We have the word "good", which is present in our text so we put one in the column that corresponds to that word, then comes word "movie" and we put one in the second column just to show that that word is actually seen in our text. We don't have any other words, so all the rest are zeroes.

This will result in a really long vector which is sparse in the sense that it has a lot of zeroes. This process is called text vectorization because we actually replace the text with a huge vector of numbers, and each dimension of that vector corresponds to a certain token in our database.

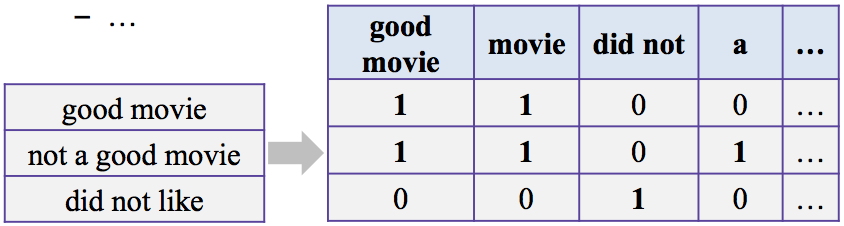


It is obvious that it has some problems:

* We lose word order, hence the name “bag of words” as they're not ordered, and so they can come up in any order
* The counters are not normalized.

## Preserve some ordering

How can we do that? Actually you can easily come to an idea that you should look at token pairs, triplets, or different combinations (extracting n-grams). One gram stands for tokens, two gram stands for a token pair and so forth.



We have the same three reviews, and now we don't only have columns that correspond to tokens, but we have also columns that correspond to let's say token pairs ("good movie"). In this way, we preserve some local word order, and we hope that that will help us to analyze this text better.

The problems are obvious though. This representation can have too many features, because let's say you have 100,000 words in your database, and if you try to take the pairs of those words, then you can actually come up with a huge number that can exponentially grow with the number of consecutive words that you want to analyze.

## ❌ Remove some n-grams

To overcome the above problem, let's remove n-grams from features based on their occurrence frequency in documents of our corpus. Actually, high and low frequency n-grams are not so useful so we proceed and remove them.  Let's understand first why they are not useful:

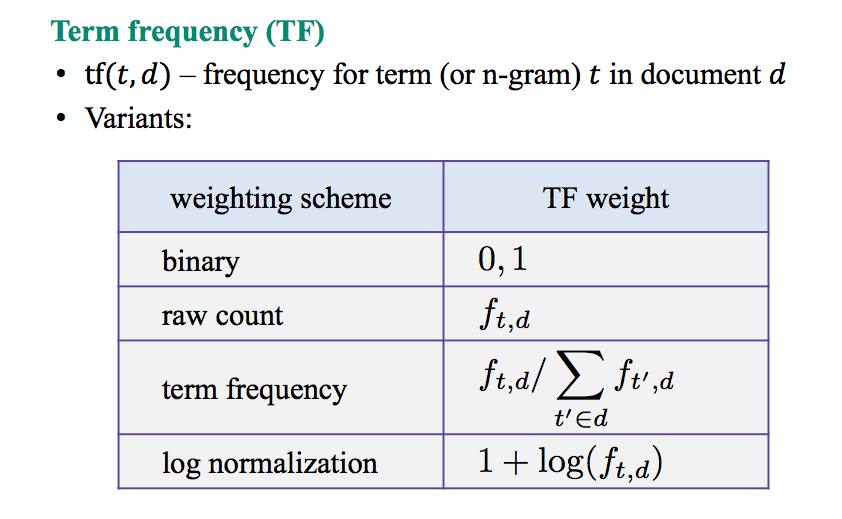
* **High frequency** n-grams that is seen in almost all of the documents are **articles**, and **preposition**, and stuff like that. Because they're just there for grammatical structure and they don't have much meaning (**stop-words**).
* **Low frequency** n-grams can be **typos** or **rare n-grams** which are bad for our model. Because if we don't remove these tokens would be a very good feature for our future classifier (overfit) as it will learn some dependencies that we don't really need them.
* **Medium frequency**n-grams, and those are really good n-grams, because they contain n-grams that are not stop-words, that are not typos and we actually look at them.

## Term Frequency (TF)

As we saw before the **Medium frequency**n-grams are really good n-grams but in a document can be a lot of them. To tackle this we can use n-gram frequency in our corpus for filtering out bad n-grams and also for ranking medium frequency n-grams.

The idea is the following that out of the medium n-gram the one with the**smaller frequency** can be more **discriminating** because it can capture a specific issue in the review. Let's say, somebody is not happy with the Wi-Fi and let's say it says, "Wi-Fi breaks often", and that n-gram, "Wi-Fi breaks" it can not be very frequent in our database but it highlights a specific issue that we need to look closer at.

And to utilize that idea, we will have to introduce some notions first like term frequency. TF is the frequency for term t while as term we can denote any n-gram, token, or anything like that in a document d.

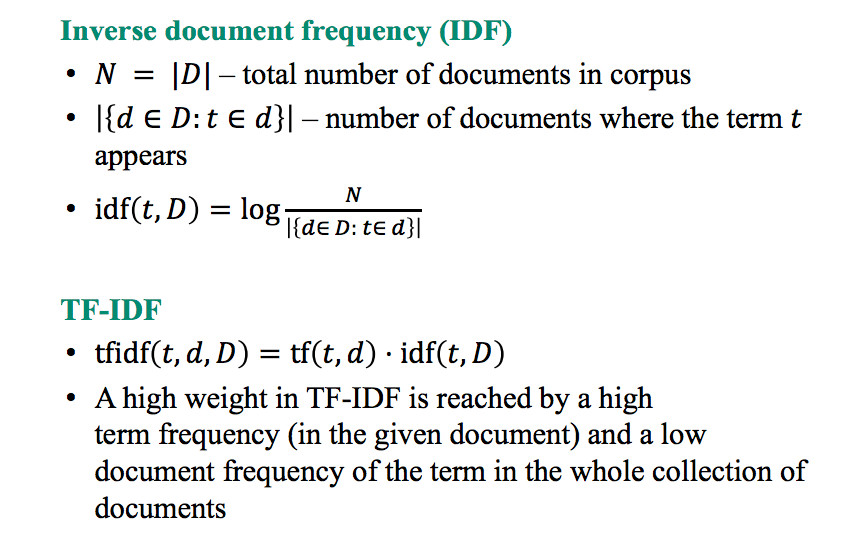


There are different options how you can count that term frequency:

* The easiest one is binary. You can actually take zero or one based on the fact whether that token is absent in our text or it is present.
* A different option is to take just a raw count of how many times we've seen that term in our document and let's denote that by f.
* Then, you can take a term frequency, so you can actually look at all the counts of all the terms that you have seen in your document and you can normalize those counters to have a sum of one. So there is a kind of a probability distribution on those tokens.
* And, one more useful scheme is logarithmic normalization. You take the logarithm of those counts and it actually introduces a logarithmic scale for your counters and that might help you to solve the task better.

## TF-IDF

Before looking at TF-IDF let's first look the inverse document frequency(IDF).



If you think about document frequency(DF), you simply take the number of documents where the term appears and divide by the total number of documents, and you have a frequency. But if you want to take inverse argument frequency then you just swap the up and down of that ratio and you take the logarithm of that thing, we will call inverse document frequency (IDF). So, it is just the logarithm of N over the number of documents where the term appears.

Using these two things, IDF and term frequency(TF), we can actually come up with TF-IDF value which is just their product and needs a term, a document, and a corpus to be calculated. Let's see why it actually makes sense to do something like this. A high TF-IDF is reached when we have high term frequency in the given document and a low document frequency of the term in the whole collection of documents. That is precisely the idea that we wanted to follow. We wanted to find frequent issues in the reviews that are not so frequent in the whole data-set (highlight specific issues).

Using TF-IDF we can even improve bag of words representation by using TF-IDF values and also normalize each result row-wise (dividing by L2 norm).

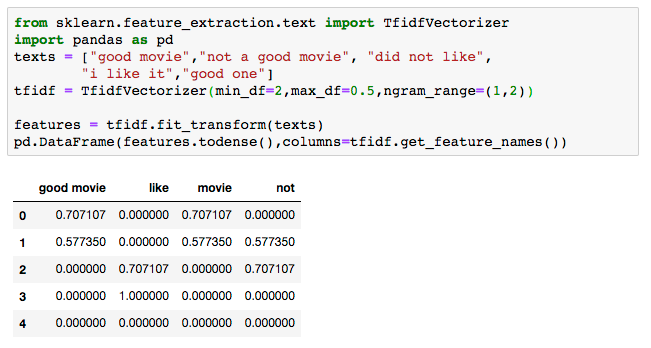
|  |
| --- |
| from sklearn.feature\_extraction.text import TfidfVectorizer  import pandas as pd  texts = ["good movie","not a good movie", "did not like",  "i like it","good one"]  tfidf = TfidfVectorizer(min\_df=2,max\_df=0.5,ngram\_range=(1,2))  features = tfidf.fit\_transform(texts)  pd.DataFrame(features.todense(),columns=tfidf.get\_feature\_names()) |

**Hyperparameters of TfidfVectorizer:**

**ngram\_range** : tuple (1, 2) represent the lower and upper boundary of the range of n-values for different n-grams to be extracted. In our example, it will extract all one gram and two gram.

**max\_df** : 0.5 ignore terms that have a document frequency strictly higher than the given threshold. For example "good" gram is ignored as it appears in 3 out of the five documents.

**min\_df :**2 When building the vocabulary ignore terms that have a document     occurrence strictly lower than the given threshold (cut-off). That's the reason why the token "one" was dropped as it appeared only in one out of the five documents.



Let's understand why "good movie" and "movie" column have both 0.7017 in the first row. Let's first find their TF-IDF value:

* TF: the first sentence has two tokens "good movie" and "movie" so the term frequency is 1/2=0.5
* IDF: "good movie" appear in 2 out of the five documents so: log(5/2)=0.92
* TF-IDF: (TF)\*(IDF)=0.5\*0.92=0.46

Let's found the TF-IDF value for "movie":

* TF: the first sentence has two tokens "good movie" and "movie" so the term frequency is 1/2=0.5
* IDF: "movie" appear in 2 out of the five documents so: log(5/2)=0.92
* TF-IDF: (TF)\*(IDF)=0.5\*0.92=0.46

So now we need to normalize by dividing with the L2-norm so:

* "good movie": (TF-IDF "good movie")/sqrt((TF-IDF "good movie")^2 +(TF-IDF "movie")^2) = 0.46/sqrt(0.46\*0.46+0.46\*0.46) = 0.707
* "movie": (TF-IDF "movie")/sqrt((TF-IDF "good movie")^2 +(TF-IDF "movie")^2) = 0.46/sqrt(0.46\*0.46+0.46\*0.46) = 0.707