Working with word frequencies

Natural Language Processing

Piotr Fulmański

WYDZIAŁ
MATEMATYKI
i INFORMATYKI
Uniwersytet Łódzki

Lecture goals

- Counting term frequencies
- Represent document with vectors of term frequencies
- Finding relevant documents from a corpus using inverse document frequencies
- Estimating the similarity of pairs of documents with cosine similarity

Zipf's law

Zipf’s law ([*zif*] or [*tsipf*] named after the American linguist George Kingsley Zipf although the French stenographer Jean-Baptiste Estoup appears to have noticed the regularity before Zipf) states that given some corpus of natural language utterances, **the frequency of any word is inversely proportional to its rank in the frequency table**. Thus the most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

If frequency is inversely proportional to rank, then the product of frequency and rank should be a constant:

$$r \cdot f = c$$

where r is the rank of a word in a text or group of texts, f the frequency of its occurrence and c *is a* constant value.

As a lot of real life things, not only linguistic, are governed by this law, so it usually refers to the "size" y of an occurrence of an *event* relative to it's rank r. Zipf's law states that **the "size" y of the r'th largest occurrence of the event is inversely proportional to it's rank**:

$$y \sim r^{-b}$$

where *b* is close to 1.0.

WYDZIAŁ
MATEMATYKI
i INFORMATYKI
Uniwersytet Łódzki

Zipf's law

CODE: Test Zipf's law

- `lecture_06_01.py`

Number of document	Number of all words	Number of different words	Word count from the most frequent to the less frequent
1	1379	123	count: w1:234, w2:63, w3:33, ...
			percentage: w1:100%, w2:(63/234)*100%, w3:(33/63)*100%, ...
			frequency: w1: 234/1379, w2: 63/1379, w3: 33/1379, ...
2	8237	...	...
			...
			...

Check for which constant c equality $r \cdot f = c$ holds in your case.

Zipf's law

If you complete all the calculations, please make a plot of frequency as a function of a rank (rank on x-axis, frequency on y-axis).

Herdan–Heaps law

Doing a test related to Zipf's law you may also verify another law, the Heaps' law (also called Herdan's law), which is also an empirical law. It describes the *number of distinct words* in a document (or set of documents) *as a function of the document length* (so called type-token relation) and is formulated as:

$$V = kn^{\beta}$$

where:

V – the number of unique words,

n – the number of all words,

k and β are free parameters determined empirically for a given (corpora) language. Typically k is between 10 and 100, and β is between 0.4 and 0.6 (approximately is equal to square root of n).

Herdan–Heaps law

Please make a plot of number of distinct words in a document as a function of the document length (document length on x-axis, number of distinct words on y-axis).

Then empirically please find such k and β so the curve:

$$V(x) = kx^{\beta}$$

best fits your data.

Word counting

BOG (Bag Of Words) - quick remainder from last lecture

```
import pandas as pd

sentences = ["a b c d", "c d e f", "a b e f"]

tokens_of_sentences = [sentence.split() for sentence in sentences]
print(tokens_of_sentences)

bow = {}

for tokens in tokens_of_sentences:
    for token in tokens:
        bow[token] = 1

bow_sorted = sorted(bow.items())
print(bow_sorted)

corpus = {}

for index, tokens in enumerate(tokens_of_sentences):
    corpus['sentence_{}'.format(index)] = dict(
        (token, 1) for token in tokens
    )

df = pd.DataFrame.from_records(corpus).fillna(0).astype(int).T
print(df)
```

```
[['a', 'b', 'c', 'd'],
['c', 'd', 'e', 'f'],
['a', 'b', 'e', 'f']]

[('a', 1), ('b', 1), ('c', 1),
('d', 1), ('e', 1), ('f', 1)]

            a  b  c  d  e  f
sentence_0  1  1  1  1  0  0
sentence_1  0  0  1  1  1  1
sentence_2  1  1  0  0  1  1
```

Word counting

Counter → term frequency (TF)

The number of times a word occurs in a given document is called the *term frequency*, commonly abbreviated TF.

Saying the truth, number of occurrences is not a frequency. For this reason, in some examples the count of word occurrences is *normalized* (divided) by the number of all terms in the document.

Normalized frequency should rather be called a *probability*, but you will use term TF which is a common practice.

Anyway, regardless of the terminology, with both (simple counter or normalized counter) you can infer importance.

Term frequency (TF)
Simple case – single document

CODE: `lecture_06_02_01.py`

Policzyć TF dla jednego dokumentu.

Term frequency (TF)
Multiple documents

CODE: `lecture_06_02_02.py`

Podobnie jak poprzednio, ale liczymy dla każdego dokumentu TF na dwa sposoby:

- tak jak poprzednio (traktując każdy dokument samodzielnie);
- traktując dokument jako część pewnego dużego korpusu.

Inverse document frequency (IDF)

Word counts for sure are useful, but pure word count, even when normalized by the length of the document, doesn't tell us much about the **importance of that word in that document *relative* to the rest of the documents** in the corpus.

For example, if we have a corpus of many books focused on the same topic or discipline, some words may occur many times in every document – that doesn't provide any new information as it doesn't help distinguish between those documents. On the other hand for sure there will be some words which are not so common across the entire corpus – they may exists in just a few of them and this is how we may know more about each document's nature.

So **we need another tool, different than TF. Term frequencies must be *weighted* by *something* to ensure the most important, most meaningful words are given the highest value**.

WYDZIAŁ
MATEMATYKI
i INFORMATYKI
Uniwersytet Łódzki

Inverse document frequency (IDF)

Inverse document frequency is the way we look in topic analysis through Zipf's law to bring out the most important details.

A good way to think of a term's inverse document frequency is this:

If a *term* appears in one document a lot of times and occurs rarely in the rest of the corpus, one could assume it's important to that document specifically. It could mean that **this document is about this *term***.

Other words: **if a *term* is rare among documents, but concentrate in one or few of them, it may be important.**

So, ***term* importance is inversely proportional to its presence in all documents**.

This way of thinking lead us to new definition. You define ***inverse document frequency***, IDF in short, as the **ratio of the total number of documents to the number of documents the term appears in**.

This is how you can start very basic **topic analysis**.

Inverse document frequency (IDF)

So you can think about IDF parameter as a way to strengthen or weaken a frequency parameter TF depending on importance of a given term.

If term is important being characteristic for a given type of documents, then it should be strengthen. Otherwise, when term is so common among various classes that it does not allow to discriminate them, it should be weaken.

TF + IDF = TF-IDF

Rule for TF:
The **more times a word appears** in the document, the TF (and hence the TF-IDF) will **go up**.

Rule for IDF:
As the **number of documents that contain a word goes up**, the IDF (and hence the TF-IDF) for that word will **go down**.

For a given term t, in a given document d, in a corpus (collection of documents) D we calculate TF-IDF as

$$tf(t,d) = \frac{count(t,d)}{count(d)} = \frac{\text{num of term } t \text{ in doc } d}{\text{num of all terms in doc } d}$$

$$idf(t,D) = \frac{count(D)}{count(t,D)} = \frac{\text{num of all docs in } D}{\text{num of docs from } D \text{ containing term } t}$$

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

TF + IDF = TF-IDF

For a given term t, in a given document d, in a corpus (collection of documents) D we calculate TF-IDF as

$$tf(t,d) = \frac{count(t,d)}{count(d)} = \frac{\text{num of term } t \text{ in doc } d}{\text{num of all terms in doc } d}$$

$$idf(t,D) = \frac{count(D)}{count(t,D)} = \frac{\text{num of all docs in } D}{\text{num of docs from } D \text{ containing term } t}$$

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

Note that:

- $tf(t,d) \in [0,1]$
- $idf(t,D) \in [1, |D|]$, $|D| =$ num of all docs in D

TF + IDF = TF-IDF

Why we need log()

Let's say, you have a huge collection of documents; for example 1000000 (1 million).

Now imagine that term *T1* is present in only 1 document, while *T2* in 10. Both 1 and 10 is a tiny drop compared to 1 million. When you count IDF for both terms you get:

$$IDF_{T1} = \frac{1000000}{1} = 1000000$$

$$IDF_{T2} = \frac{1000000}{10} = 100000$$

That's a big difference in terms of Zipf's law. According to this law, when you compare the frequencies of two terms, like IDF's you have just calculated for *T1* and *T2*, even if they occur a similar number of times (which is in our case: 1 and 10 is quite similar, or close to each other, compared to 1 milion), the more frequent word (ranked higher) will have an *exponentially* higher frequency than the less frequent one

1 is 0.0001% of 1 million, 1000000 is 100% of 1 million

10 is 0.001% of 1 million, 100000 is 10% of 1 million

You may say that drawing all four percentage values on a number line, for fixed unit, 0.0001 is much closer to 0.001 than 10 to 100.

TF + IDF = TF-IDF

Why we need log()

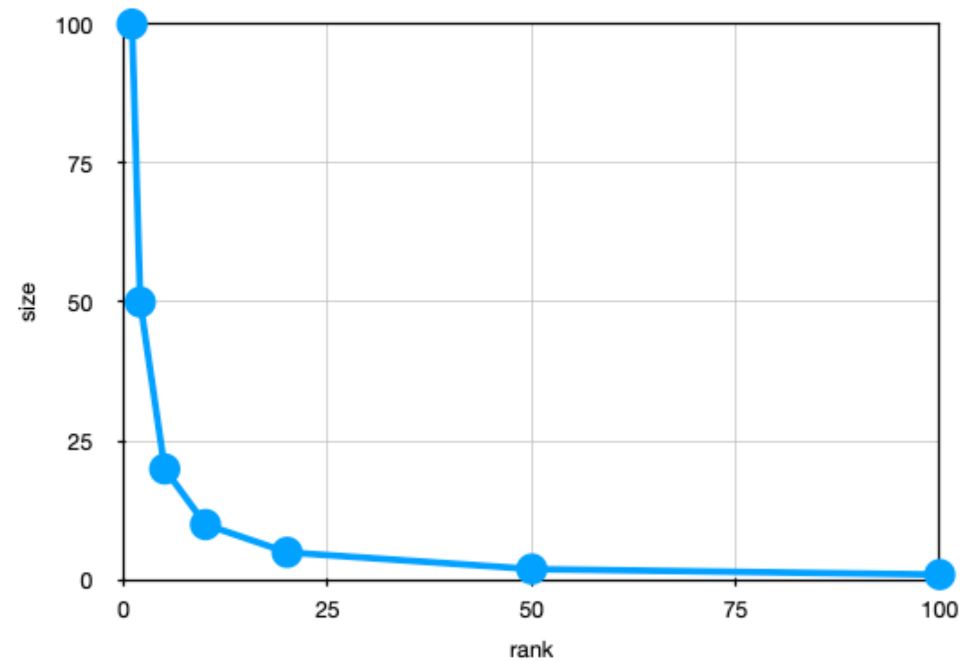
Do you remember? **The frequency of any word is inversely proportional to its rank in the frequency table.** The most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

rank	freq or size	log(rank)	log(freq or size)
1	100	0	4.6
2	50	0.693	3.91
5	20	1.6	2.99
10	10	2.3	2.33
20	5	2.99	1.6
50	2	3.91	0.693
100	1	4.6	0

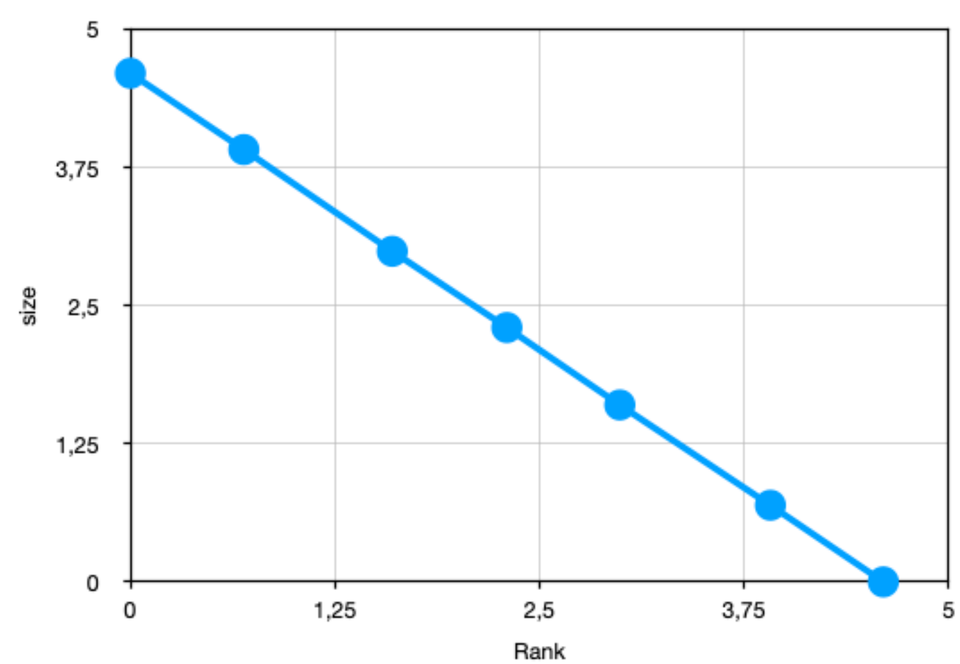
TF + IDF = TF-IDF

Why we need log()

rank	freq or size
1	100
2	50
5	20
10	10
20	5
50	2
100	1



log(rank)	log(freq or size)
0	4.6
0.693	3.91
1.6	2.99
2.3	2.33
2.99	1.6
3.91	0.693
4.6	0



TF + IDF = TF-IDF

Why we need log()

So Zipf's Law suggests that you scale all your frequencies (both for words and document) with the $\log()$ function which is the inverse of $\exp()$. This ensures that terms such as *T1* and *T2* which have similar counts, aren't exponentially different in frequency. And this **(log-log) distribution of word frequencies will ensure that your TF-IDF scores are more uniformly distributed**.

For this reason, TF-IDF is calculated as (note that in this case IDF part is defined with directly included logarithm):

$$tf(t,d) = \frac{count(t,d)}{count(d)} = \frac{\text{num of term } t \text{ in doc } d}{\text{num of all terms in doc } d}$$

$$idf(t,D) = log\left(\frac{count(D)}{count(t,D)}\right) = log\left(\frac{\text{num of all docs in } D}{\text{num of docs from } D \text{ containing term } t}\right)$$

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

TF + IDF = TF-IDF

Why we need log()

Sometimes we make all the calculations in log space (below IDF part itself is defined without logarithm):

$$tfidf(t, d, D) = log(tf(t, d)) \cdot log(idf(t, D))$$

If you use logarithm only for IDF part (as it is given on previous slide) then both TF and IDF are positive. Otherwise, when logarithm is also applied to TF, the term frequency component would be negative.

TF + IDF = TF-IDF

Some notes

TF-IDF relates a specific word or token t to a specific document d in a specific corpus D, and then **it assigns a numeric value to the *importance* of that word in the given document, given its usage across the entire corpus**.

TF + IDF = TF-IDF

CODE: `lecture_06_03_01.py`

Create K-dimensional vector representation for each document in the corpus.

TF + IDF = TF-IDF

Base on `lecture_06_03_01.py`:

Test behaviour of TF-IDF vector: when and how it changes, how it reflects structure of documents and whole corpus.

CODE (to do as an exercise)

- `lecture_06_03_02.py`
- `lecture_06_03_02_results.py`

TF + IDF = TF-IDF

```
documents = ['a a b c',
             'a a a a b b c c',
             'a a b c d e',
             'a a a a b b c c d e',
             ]

=== TF ===
[('a', 0.5),   ('b', 0.25),  ('c', 0.25),  ('d', 0),     ('e', 0)]
[('a', 0.5),   ('b', 0.25),  ('c', 0.25),  ('d', 0),     ('e', 0)]
[('a', 0.333), ('b', 0.167), ('c', 0.167), ('d', 0.167), ('e', 0.167)]
[('a', 0.4),   ('b', 0.2),   ('c', 0.2),   ('d', 0.1),   ('e', 0.1)]

=== IDF ===
[('a', 0.0),   ('b', 0.0),   ('c', 0.0),   ('d', 0.693), ('e', 0.693)]

=== TF-IDF ===
[('a', 0.0),   ('b', 0.0),   ('c', 0.0),   ('d', 0),     ('e', 0)]
[('a', 0.0),   ('b', 0.0),   ('c', 0.0),   ('d', 0),     ('e', 0)]
[('a', 0.0),   ('b', 0.0),   ('c', 0.0),   ('d', 0.116), ('e', 0.116)]
[('a', 0.0),   ('b', 0.0),   ('c', 0.0),   ('d', 0.069), ('e', 0.069)]
```

How to measure similarity

- With euclidean distance
- With cosine similarity

How to measure similarity

Cosine similarity

$$\cos\Theta = \frac{A \cdot B}{|A||B|}$$

A cosine similarity of **1 represents vectors that point in exactly the same direction**; the **vectors may have different lengths or magnitudes**.

When a cosine similarity is close to 1, you know that the documents are **using similar words in similar proportion**. So the documents whose document vectors are close to each other **are likely talking about the same thing**.

Note: A cosine similarity of 0 represents orthogonal vectors. When cosine similarity is equal to -1 vectors are opposite – vectors point in opposite directions.

How to measure similarity

Cosine similarity

$$\cos\Theta = \frac{A \cdot B}{|A||B|}$$

```
A = [1,2]
B = [2,4]
AB = 2 + 8 = 10
|A| = sqrt(5) = 2.2361
|B| = sqrt(20) = 2 * sqrt(5)
cos(theta) = 10/(2*sqrt(5)*sqrt(5)) = 1

A = [1,2]
B = [-2,-4]
AB = (-2) + (-8) = -10
|A| = sqrt(5) = 2.2361
|B| = sqrt(20) = 2 * sqrt(5)
cos(theta) = -10/(2*sqrt(5)*sqrt(5)) = -1

A = [1,2]
B = [4,-2]
AB = 4 - 4 = 0
|A| = sqrt(5) = 2.2361
|B| = sqrt(20) = 2 * sqrt(5)
cos(theta) = 10/(2*sqrt(5)*sqrt(5)) = 0
```

WYDZIAŁ
MATEMATYKI
i INFORMATYKI
Uniwersytet Łódzki

Documents relevance

In the last lecture you used BOW (bag-of-words) vectors to find documents overlap.

Extension of BOW with simple words counting (and even words frequency - TF) isn't a big step forward.

You get a new value replacing each word’s counter (TF) with the word’s TF-IDF. With this your vectors will more thoroughly reflect the meaning, or topic, of the document

Documents relevance

Compute a new document relevance in context of corpus we have.

CODE (to do as an exercise)

- `lecture_06_04.py`
- `lecture_06_04_results.py`

Documents relevance

```
documents = ['a',
             'a b',
             'a b c',
             'a b c d',
             'a b c d e',
             'a b c d e f',
             'a b c d e f g']

doc_test = 'a b c g h'
```

```
Skip token "h"
Compare with document 0:
None
Compare with document 1:
0.0778226964529786l
Compare with document 2:
0.18684518797064792
Compare with document 3:
0.10276244576838443
Compare with document 4:
0.06392858530379265
Compare with document 5:
0.041787650286784016
Compare with document 6:
0.7754668111919074
```

Speedup with indexing

Technical remarks: forward and inverse index

Calculating TF and IDF requires a lot of counting which could be speed-up with proper indexing.

Speedup with indexing

Technical remarks: forward and inverse index

In computer science, an *inverted index* is a database index storing a mapping from content, such as words or numbers, to its locations in a table, or in a document or a set of documents. It is named *inverted* in contrast to a *forward index*, which maps from documents to content.

There is no real technical distinction between a forward index and an inverted index. An inverted index is just an index... but backwards. The concept of an inverted index only makes sense if the concept of a regular (forward) index already exists. Other words, first you need to have something to be able to talk about inverting (it).

"Forward" and "inverted", in the context of a search engine, are just descriptive terms to distinguish between:

- A list of words contained in a document.
- A list of documents containing a word.

For example, forward index would store

```
{ Document1: ["Text", "from", "a", "document", "number", "1"],
  ...
},
```

an inverted index would store:

```
{ "Text": [Document1, Document100, ...],
  "from": [Document1, Document2, ...],
  ...
}
```

One lets you look up a document and find the contents, the other lets you look up a word and get a list of documents.

Speedup with indexing

Technical remarks: forward and inverse index

Advantage of inverted index is:

- Inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database.

Disadvantage of inverted index is:

- Large storage overhead and high maintenance costs on update, delete and insert.

You can say this:

- **Forward index**: fast indexing, less efficient query's
- **Inverted index**: fast query, slower indexing

Another kind of speedup

Technical remarks: forward and inverse index

What about vectors with only relevant words?

How we can use it in chatbot

Technical remarks: forward and inverse index

But most chatbots rely heavily on a search engine. And some chatbots rely exclusively on a search engine as their only algorithm for generating responses. You need to take one additional step to turn your simple search index (TF-IDF) into a chatbot. You need to store your training data in pairs of questions (or statements) and appropriate responses. Then you can use TF-IDF to search for a question (or statement) most like the user input text. Instead of returning the most similar statement in your database, you return the response associated with that statement.