Topic Modeling in NLP - TfIdf - 2

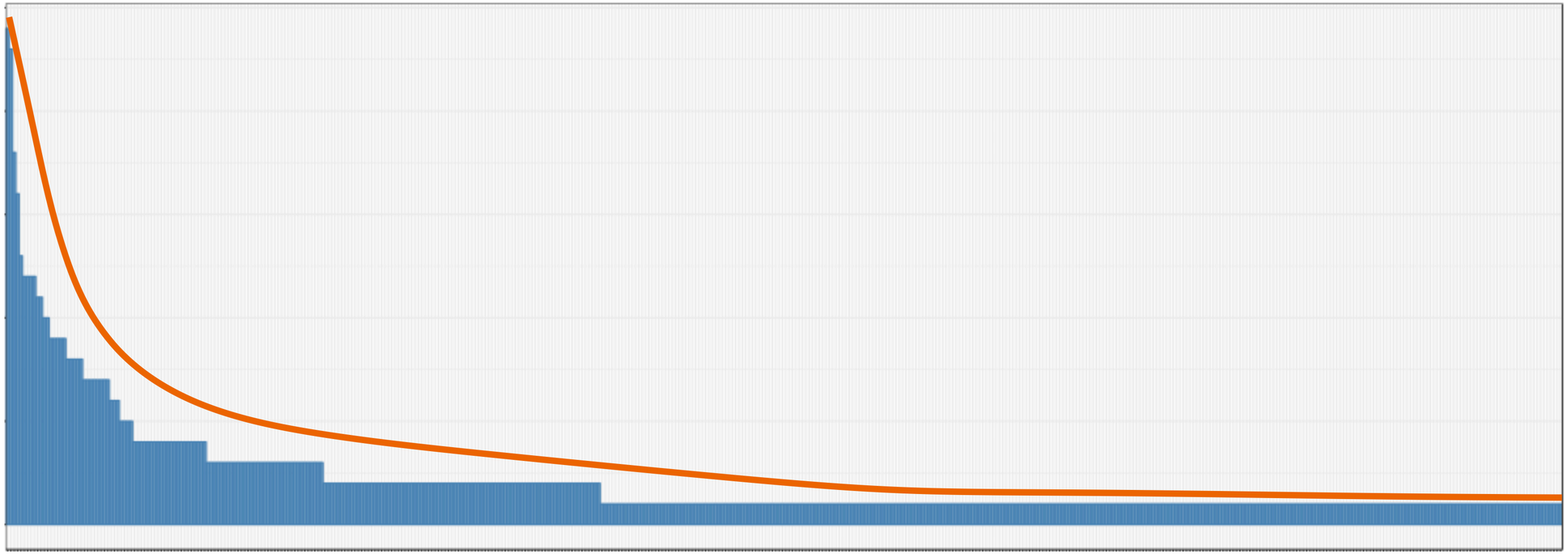
One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

Objective	Complete
Identify the need for weighting terms in a corpus	
Weigh text data with term frequency inverse document frequency (TF-IDF)	

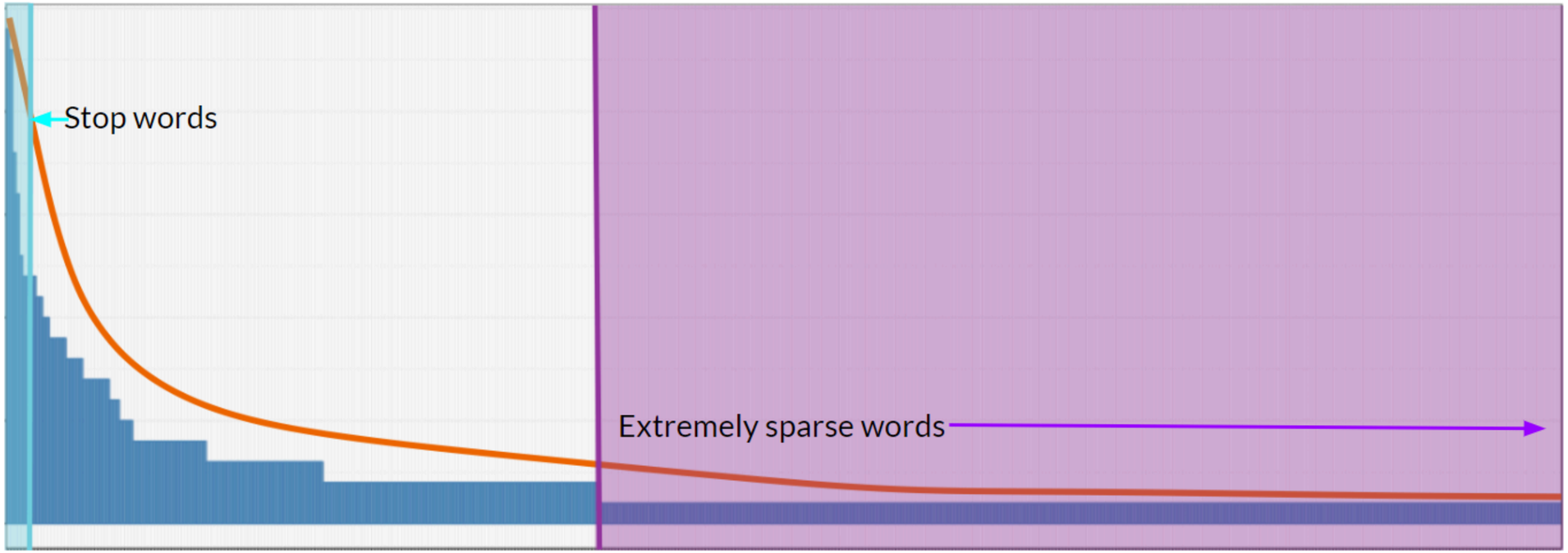
Word distribution in a language and corpus

- The distribution of words in a corpus and in a language is **highly skewed right**, which indicates that few words have very high frequencies and most words have very low counts!
- This phenomenon has implications in many areas of research and applications such as; information theory, natural language processing, search engines and many more

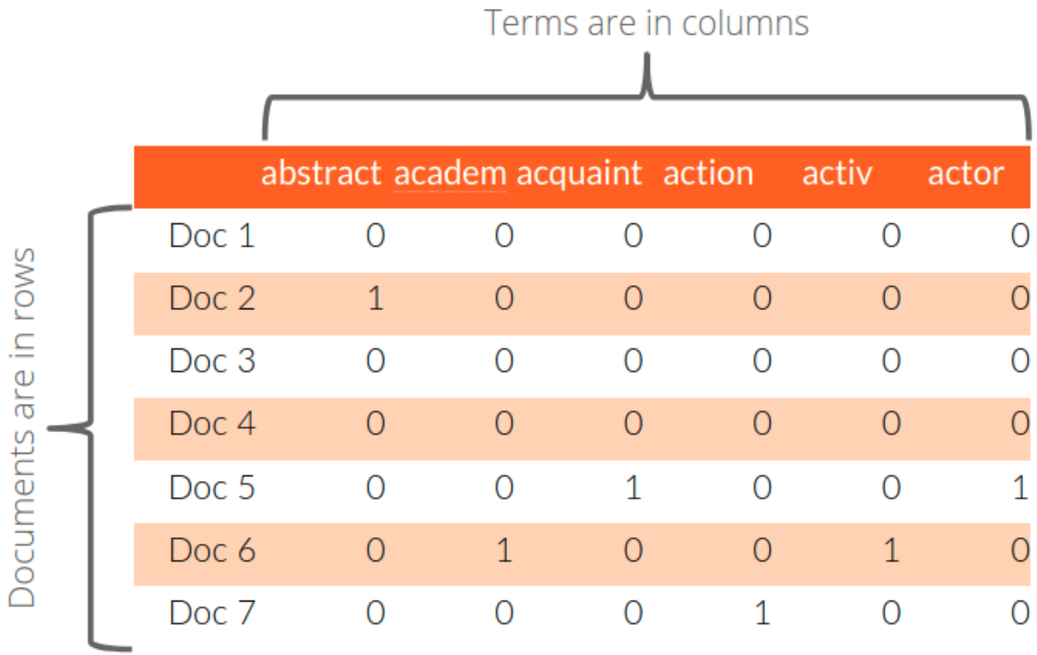


Word distribution in a language and corpus

- Words that have extremely high frequencies are considered **stop words**
- Those that have extremely low frequencies are considered **extremely sparse words**
- These frequencies are reflected in the Document-Term Matrix for the corpus



What is a Document-Term Matrix (DTM)?



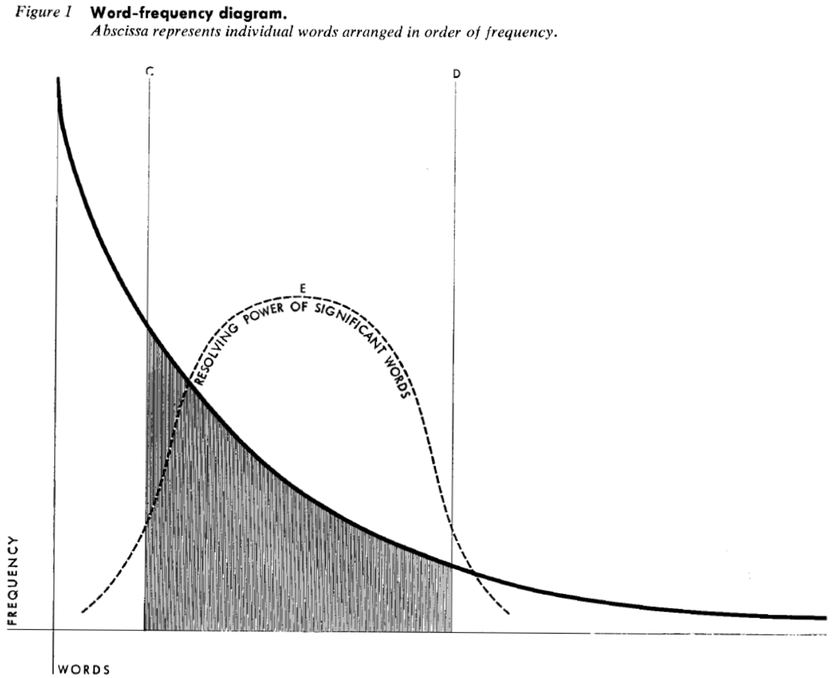
	abstract	academ	acquaint	action	activ	actor
Doc 1	0	0	0	0	0	0
Doc 2	1	0	0	0	0	0
Doc 3	0	0	0	0	0	0
Doc 4	0	0	0	0	0	0
Doc 5	0	0	1	0	0	1
Doc 6	0	1	0	0	1	0
Doc 7	0	0	0	1	0	0

- **Document-term matrix** is simply a matrix of unique words counted in each document. See graph:
  - documents are arranged in the rows
  - unique terms are arranged in columns
- The corpus **vocabulary** consists of all of the unique terms (i.e. column names of DTM) and their total counts across all documents (i.e. column sums)
- A Term-Document Matrix will be just the transpose of the Document-Term Matrix, with terms in rows and documents in columns

Power of significant words

- By removing *BOTH* ends of the distribution, we **reduce the dimensionality** of our data and **avoid "overfitting"** our text model

- In fact, it has been proven by *H.P. Luhn* (a researcher for IBM) *in 1958* that the **power of significant words is approximately normally distributed** and the top of the bell-shaped curve **coincides with the mid-portion of the word frequency distribution**



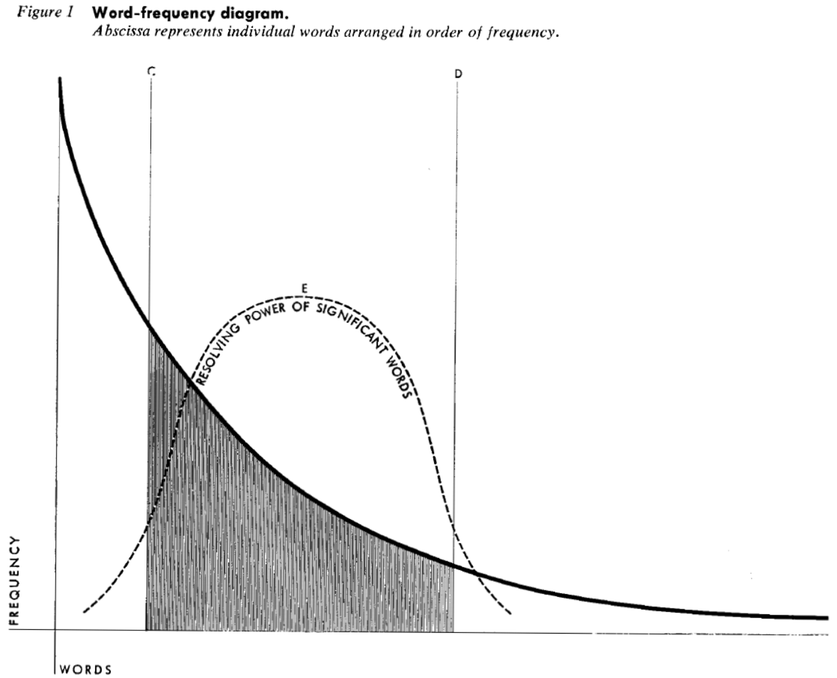
Source: H.P. Luhn, "The Automatic Creation of Literature Abstracts", Presented at IRE National Convention, New York, March 24, 1958. Published in IBM JOURNAL APRIL 1958

Why bother trimming words?

- **Avoid** having **too many dimensions** (in text analysis words represent variables, i.e. dimensions)
- **Speed up** computations
- **Reduce noise** and **prevent overfitting**

Ways to overcome extreme skewness

- Since the word significance is maximized towards the mid-portion of the skewed distribution, we need to **“reshape”** our data to let the words that are in that mid-portion stand out
- One of the most common ways of achieving this is **weighting** our data, which in statistics is usually known as **data transformation**



Source: H.P. Luhn, “The Automatic Creation of Literature Abstracts“, Presented at IRE National Convention, New York, March 24, 1958. Published in IBM JOURNAL APRIL 1958

Text Frequency - Inverse Document Frequency

- **TF-IDF** is a famous transformation of text data used to battle the skewness of the word distribution in a corpus and is given by the following formula:

$$TF \times IDF = \frac{F_{wd}}{N_d} \times log \frac{M}{M_w}$$

- Where:
  - Subscripts w and d stand for $word$ and $document$ respectively
  - F_{wd} is the frequency of a word in a document
  - N_d is a total number of words in a given document
  - M is the total number of documents in a corpus
  - M_w is the number of documents containing the given word in the corpus

TF-IDF effect on DTM

- Normalization of **TF** (i.e. $TF = \frac{F_{wd}}{N_d}$) eliminates the size effect of documents
  - Longer documents with more words no longer overpower smaller documents with fewer words that may contain valuable information
- **IDF** (i.e. $log\frac{M}{M_w}$) helps to adjust for the fact that some words appear more frequently in general, but carry less value in their meaning
  - This is due to not being very specific to a particular concept that makes a document or a group of documents in a corpus stand out

TF-IDF effect on DTM (cont'd)

- Consider the following DTM:

DTM with raw counts

	analyze	data	fractal
Doc A	1	2	0
Doc B	0	0	1
Doc C	0	4	3

Weighted by

TF-IDF

DTM with TF-IDF weights

	analyze	data	fractal
Doc A	0.528	0.390	0
Doc B	0	0	0.585
Doc C	0	0.334	0.251

- The word `data` appears in `DocA` and `DocC`, and although it appeared twice as often in `DocC`, its weight has adjusted for `DocA` to a very similar one as in `DocC`, making it of similar "value"

Module completion checklist

Objective	Complete
Identify the need for weighting terms in a corpus	✔
Weigh text data with term frequency inverse document frequency (TF-IDF)	

`Snippet` analysis

- In order to pre-process the data for TF-IDF, the steps to be taken are:
  - **Load** the corpus, where each ‘document’ is actually one entry in `snippet`
  - **Clean** the text, removing punctuation, numbers, special characters and stop words
  - *Stem* the words to their root forms
  - **Create** a Document-Term Matrix (DTM) with counts of each word recorded for each document
- Then we build the final, optimized matrix - a weighted **T**erm **F**requency - **I**nverse **D**ocument **F**requency (TF-IDF) by
  - **Transforming** the DTM to be a weighted TF-IDF

Load packages

- Let's import all packages needed for this course

```
# Helper packages.
from pathlib import Path
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
# Packages with tools for text processing.
import nltk
nltk.download('punkt')
```

```
nltk.download('stopwords')
```

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer
import gensim
from gensim import corpora, models
from pprint import pprint
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

New York Times articles: case study

- The New York Times is a widely read American newspaper with an extensive textual archive
- We can better understand its impact by analyzing its frequently used terms and identifying the topics discussed in each article
- Our starting point is a dataset from *Kaggle* containing **snippets** of the paper's articles
- We will analyze these text snippets to find the **most frequent terms**
- We will also extract article topics and **tag** each article with a topic to better understand the main subjects of the articles

Dataset

- In order to implement what you learn in this course, we will be using the NYT_article_data.csv dataset
- We will be working with columns from the dataset such as:
  - `web_url`
  - `headline`
  - `snippet`
  - `word_count`
  - `source`

Load data

- We will now load the dataset and save it as `df`

```
# Let's load and prepare the dataset for creating Document-Term Matrix
df = pd.read_csv(str(data_dir)+"/"+ "NYT_article_data.csv")
print(df.head())
```

```
                                              web_url  ...  id
0  https://www.nytimes.com/reuters/2019/01/01/spo...  ...   8
1  https://www.nytimes.com/reuters/2019/01/01/wor...  ...   9
2  https://www.nytimes.com/aponline/2019/01/01/sp...  ...  10
3  https://www.nytimes.com/2019/01/09/arts/design...  ...  11
4  https://www.nytimes.com/aponline/2019/01/10/sp...  ...  12

[5 rows x 8 columns]
```

- We will be focusing on the `snippet` column for text mining

Check for NAs

- Check and drop any NAs or empty values from the `snippet` column as they are not relevant for our analysis

```
# Print total number of NAs.
print(df["snippet"].isna().sum())
```

```
0
```

```
# Drop NAs if any.
df = df.dropna(subset = ["snippet"]).reset_index(drop=True)
print(df["snippet"].isna().sum())
```

```
0
```

```
# Isolate the `snippet` column.
df_text = df["snippet"]
```

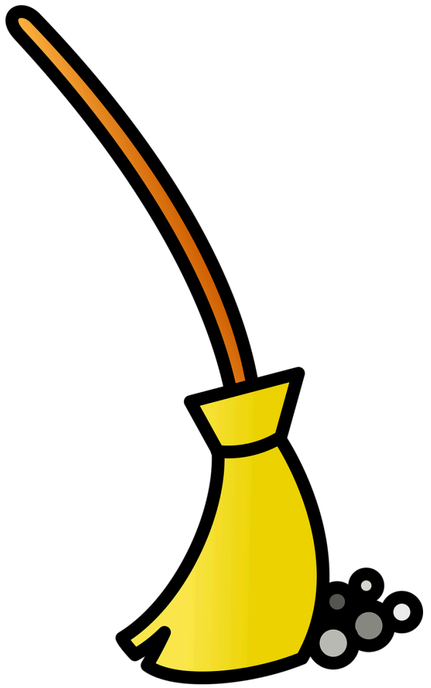
Tokenization: split each document into words

- NLTK's functions operate on **tokens**
- A **token** is the smallest unit of text of interest - in our case, it will be a **word**
- We will use `word_tokenize()` method to split each document into tokens

```
# Tokenize each document into a large list of tokenized documents.
df_tokenized = [word_tokenize(df_text[i]) for i in range(0, len(df_text))]
```

“Bag-of-words” analysis: cleaning text flow

1. Convert all characters to lowercase
2. Remove stop words
3. Remove punctuation, numbers, and all other symbols that are not letters of the alphabet
4. Stem words
5. Remove extra white space (if needed)



Convert characters to lowercase

- To convert characters to lowercase, we will use `.lower()` function
  - We call the method like so: `character_string.lower()`
- Since every element of the `document_words` list is a character string, we will convert each word in the document using a **list comprehension**

```
# Let's take a look at the first tokenized document
document_words = df_tokenized[0]
print(document_words)
```

```
['Nick', 'Kyrgios', 'started', 'his', 'Brisbane', 'Open', 'title', 'defense', 'with', 'a',
'battling', '7-6', '(', '5', ')', '5-7', '7-6', '(', '5', ')', 'victory', 'over',
'American', 'Ryan', 'Harrison', 'in', 'the', 'opening', 'round', 'on', 'Tuesday', '.']
```

- Let's test out the cleaning flow on this single document first
- Then we will apply the cleaning steps to all documents in our corpus

```
# 1. Convert to lowercase.
document_words = [word.lower() for word in document_words]
print(document_words[:10])
```

```
['nick', 'kyrgios', 'started', 'his', 'brisbane', 'open', 'title', 'defense', 'with', 'a']
```

Remove stop words

- In any language, there are words that carry little specific subject-related meaning, but are necessary to bind words into coherent sentences together
- Such words are the most frequent and they usually need to be taken out, as they create unnecessary noise
- They are called **stop words** and NLTK has a corpus of such words that can be called by using `stopwords` module
  - To get English stop words, we will call `stopwords.words('english')` method:

```
# 2. Remove stop words.
# Get common English stop words.
stop_words = stopwords.words('english')
print(stop_words[:10])
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]
```

```
# Remove stop words.
document_words = [word for word in document_words if not word in stop_words]
print(document_words[:10])
```

```
['nick', 'kyrgios', 'started', 'brisbane', 'open', 'title', 'defense', 'battling', '7-6',
'(']
```

Remove non-alphabetical characters

- For text analysis based on “bag-of-words” approach, we only use words
- We remove numbers, punctuation or anything other than alphabetical characters
- We will use `.isalpha()` method to check whether the characters are alphabetical or not:
  - We call the method: `character_string.isalpha()`
  - Since every element of the `document_words` list is a character string, we will check each token in the document using a **conditional** `if` inside of the **list comprehension**

```
# 3. Remove punctuation and any non-alphabetical characters.
document_words = [word for word in document_words if word.isalpha()]
print(document_words[:10])
```

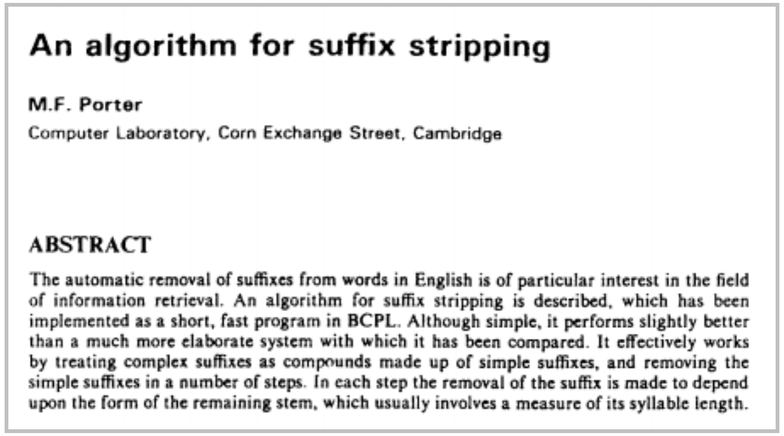
```
['nick', 'kyrgios', 'started', 'brisbane', 'open', 'title', 'defense', 'battling',
'victory', 'american']
```

Word stemming

- **Stemming reduces words to their root** form - it allows us to:
  - Treat different forms of the same word as one (i.e. “reads” and “reading” would both be stemmed to “read”)
  - Shrink the total number of unique terms, which reduces the noise in data and dimensionality of the data, which we will discuss shortly
- **PorterStemmer** is a Python implementation for the most famous stemming algorithm in existence - the Porter stemmer

Porter stemmer

- The algorithm is named after its inventor, computational linguist *Dr. Martin Porter*.
- His paper **“An algorithm for suffix stripping”** is one of the most sited works in Information Retrieval (over **12,000** times according to Google Scholar)!



An algorithm for suffix stripping

M.F. Porter

Computer Laboratory, Corn Exchange Street, Cambridge

ABSTRACT

The automatic removal of suffixes from words in English is of particular interest in the field of information retrieval. An algorithm for suffix stripping is described, which has been implemented as a short, fast program in BCPL. Although simple, it performs slightly better than a much more elaborate system with which it has been compared. It effectively works by treating complex suffixes as compounds made up of simple suffixes, and removing the simple suffixes in a number of steps. In each step the removal of the suffix is made to depend upon the form of the remaining stem, which usually involves a measure of its syllable length.

Source: https://tartarus.org/martin/PorterStemmer/

Stem words

- The `PorterStemmer()` module of NLTK contains a method `.stem()`
- To stem a word, we would simply call `PorterStemmer().stem(word)` for every word in the `document_words` list using a **list comprehension**

```
# 4. Stem words.
document_words = [PorterStemmer().stem(word) for word in document_words]
print(document_words[:10])
```

```
['nick', 'kyrgio', 'start', 'brisban', 'open', 'titl', 'defens', 'battl', 'victori',
'american']
```

Clean the entire corpus

- Now that we have successfully implemented text cleaning steps on a single document, we can do that for the entire corpus of documents

```
# Create a list for clean documents.
df_clean = [None] * len(df_tokenized)
# Create a list of word counts for each clean document.
word_counts_per_document = [None] * len(df_tokenized)

# Process words in all documents.
for i in range(len(df_tokenized)):
    # 1. Convert to lowercase.
    df_clean[i] = [document.lower() for document in df_tokenized[i]]

    # 2. Remove stop words.
    df_clean[i] = [word for word in df_clean[i] if not word in stop_words]

    # 3. Remove punctuation and any non-alphabetical characters.
    df_clean[i] = [word for word in df_clean[i] if word.isalpha()]

    # 4. Stem words.
    df_clean[i] = [PorterStemmer().stem(word) for word in df_clean[i]]
```

Clean the entire corpus (cont'd)

```
# Convert word counts list and documents list to NumPy arrays.
word_counts_array = np.array(word_counts_per_document)
df_array = np.array(df_clean, dtype= object)

# Find indices of all documents where there are greater than or equal to 5 words.
valid_documents = np.where(word_counts_array >= 5)[0]

# Subset the df_array to keep only those where there are at least 5 words.
df_array = df_array[valid_documents]

# Convert the array back to a list.
df_clean = df_array.tolist()   #<- the processed documents
```

Gensim package

- `gensim` is a Python package made for topic modeling
- We will be using `gensim` to transform our DTM to a TF-IDF matrix, as well as to build the LDA model
- Click ***here*** for complete documentation

Data for TF-IDF matrix

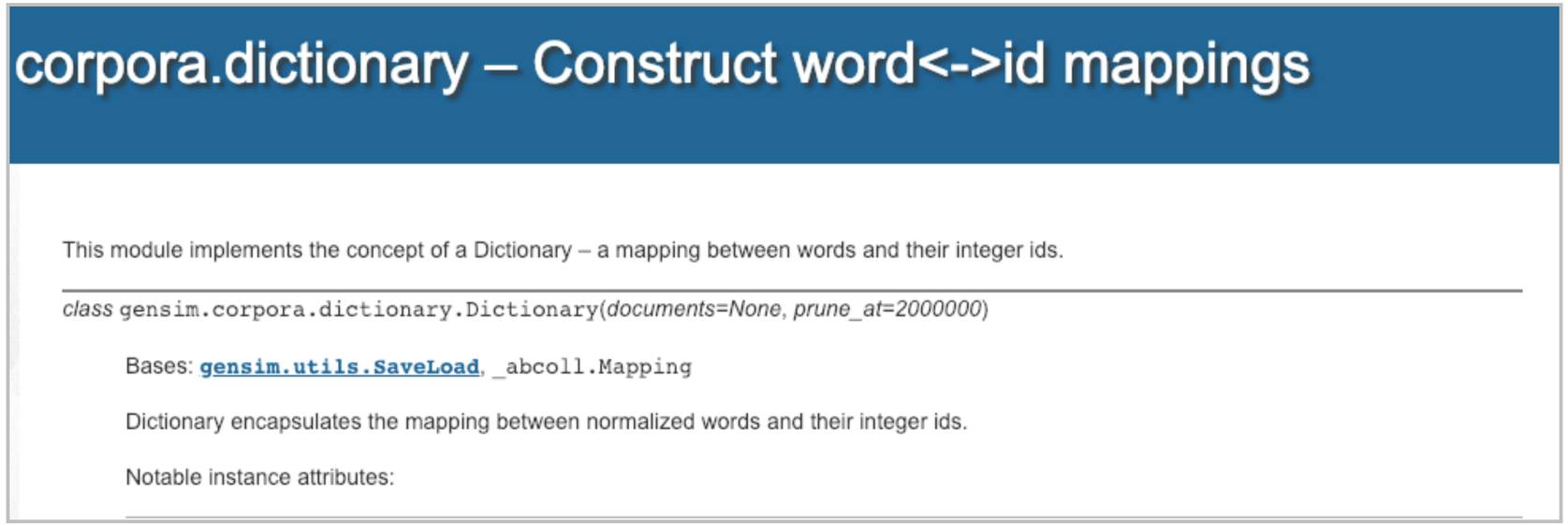
- For creating a TF-IDF matrix, we will use `df_clean`, which is a list of cleaned and tokenized documents

```
print(df_clean[0:2])
```

```
[['nick', 'kyrgio', 'start', 'brisban', 'open', 'titl', 'defens', 'battl', 'victori',
'american', 'ryan', 'harrison', 'open', 'round', 'tuesday'], ['british', 'polic', 'confirm',
'tuesday', 'treat', 'stab', 'attack', 'injur', 'three', 'peopl', 'manchest', 'victoria',
'train', 'station', 'terrorist', 'investig', 'search', 'address', 'cheetham', 'hill',
'area', 'citi']]
```

gensim dictionary

- Now let's create a dictionary of counts using a `gensim` function `gensim.corpora.Dictionary`
- This will give us a **dictionary** from `df_clean` that contains the **number of times a given word appears** within the entire corpus



Create a dictionary of counts

- Let's create the dictionary using `gensim.corpora.Dictionary` and look at our output using a small loop

```
# Set the seed.
np.random.seed(1)
dictionary = gensim.corpora.Dictionary(df_clean)

# The loop below iterates through the first 10 items of the dictionary and prints out the key and
value.
count = 0
for k, v in dictionary.iteritems():
    print(k, v)
    count += 1
    if count > 10:
        break
```

```
0 american
1 battl
2 brisban
3 defens
4 harrison
5 kyrgio
6 nick
7 open
8 round
9 ryan
10 start
```

Create a dictionary of counts (cont'd)

- We can filter out words by their frequency in the dictionary
- `.filter_extremes()` will remove all values in the dictionary that are:
  - Less frequent than `no_below` documents
  - More than `no_above` documents (fraction of total corpus size, not absolute number)
  - Keep only first `keep_n` most frequent tokens

```
dictionary.filter_extremes(no_below = 4, no_above = 0.5, keep_n = 200)

# How many words are left in the dictionary?
len(dictionary)
```

```
200
```

Document to bag-of-words

- Now we will use `gensim` library `doc2bow` to transform each document to a dictionary
- Each document will become a dictionary that has the number of words and has the number of times each of those words appear
- This is the object we will use to build our TF-IDF matrix

```
# We use a list comprehension to transform each doc within our df_clean object.
bow_corpus = [dictionary.doc2bow(doc) for doc in df_clean]

# Let's look at the first document.
print(bow_corpus[0])
```

```
[(0, 1), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 1)]
```

Document to bag-of-words

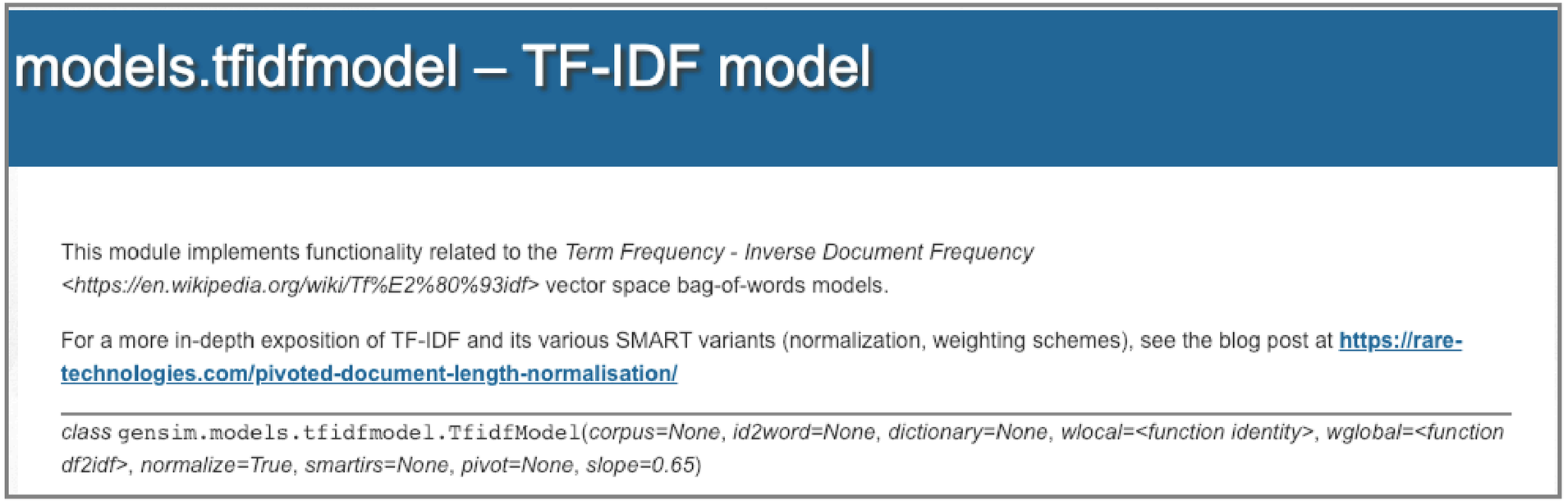
- Let's preview bag-of-words for the first document

```
# Isolate the first document.
bow_doc_1 = bow_corpus[0]

# Iterate through each dictionary item using the index.
# Print out each actual word and how many times it appears.
for i in range(len(bow_doc_1)):
    print("Word {} (\"{}\") appears {}
time.".format(bow_doc_1[i][0],

          dictionary[bow_doc_1[i][0]],
          bow_doc_1[i][1]))
```

```
Word 0 ("american") appears 1 time.
Word 1 ("defens") appears 1 time.
Word 2 ("open") appears 2 time.
Word 3 ("round") appears 1 time.
Word 4 ("start") appears 1 time.
Word 5 ("tuesday") appears 1 time.
Word 6 ("victori") appears 1 time.
```

Transform counts with TfidfModel

- To transform a Document-Term Matrix into `TF-IDF`, we will use `TfidfModel` from `gensim` library's `model` module for working with text



models.tfidfmodel – TF-IDF model

This module implements functionality related to the *Term Frequency - Inverse Document Frequency* <*https://en.wikipedia.org/wiki/Tf%E2%80%93idf*> vector space bag-of-words models.

For a more in-depth exposition of TF-IDF and its various SMART variants (normalization, weighting schemes), see the blog post at **https://rare-technologies.com/pivoted-document-length-normalisation/**

---

class `gensim.models.tfidfmodel.TfidfModel`(*corpus=None*, *id2word=None*, *dictionary=None*, *wlocal=<function identity>*, *wglobal=<function df2idf>*, *normalize=True*, *smartirs=None*, *pivot=None*, *slope=0.65*)

Transform counts with TfidfModel

- We will now activate the `TfidfModel` function and transform our `bow_corpus`
- Our output will be the TF-IDF transformation applied to each document:

$$TF \times IDF = \frac{F_{wd}}{N_d} \times log \frac{M}{M_w}$$

```
# This is the transformation.
tfidf = models.TfidfModel(bow_corpus)

# Apply the transformation to the entire corpus.
corpus_tfidf = tfidf[bow_corpus]

# Preview TF-IDF scores for the first document.
for doc in corpus_tfidf:
    pprint(doc)
    break
```

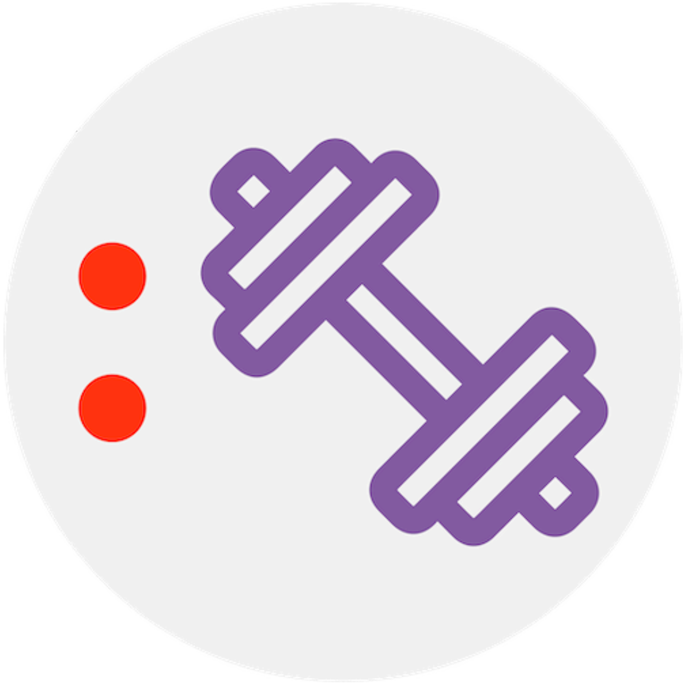
```
[(0, 0.31942373876087665),
 (1, 0.3549009519669791),
 (2, 0.6118718565633235),
 (3, 0.3549009519669791),
 (4, 0.3059359282816618),
 (5, 0.22829905152454918),
 (6, 0.3549009519669791)]
```

“Bag-of-words” analysis: key elements

What we need	What we have learned
A corpus of documents cleaned and processed in a certain way • All words are converted to lowercase • All punctuation, numbers and special characters are removed • Stopwords are removed • Words are stemmed to their root form	✔
A Document-Term Matrix (DTM): with counts of each word recorded for each document	✔
A transformed representation of a Document-Term Matrix (i.e. weighted with TF-IDF weights)	✔

Knowledge check

Exercise



You are now ready to try Tasks 1-8 in the Exercise for this topic

Module completion checklist

Objective	Complete
Identify the need for weighting terms in a corpus	✔
Weigh text data with term frequency inverse document frequency (TF-IDF)	✔

Tf-Idf: Topic summary

In this part of the course, we have covered:

- The “bag-of-words” approach and when it is used
- The need for weighting terms in a corpus
- Implementation of Tf-Idf weighting on a corpus of documents

Congratulations on completing this module!