COMP3074-HAI- Lab 1: Putting words in bags

This week of teaching focuses on two fundamental topics: **language models**, and **representation**. For now, we are going to focus on **representation** because we will have the opportunity to visit language models in more details when talking about text generation.

1. The goal of this lab

The goal of this lab is to introduce you to basic parsing of text in Python, as well as the first step of all natural language processing tasks: **document representation**. We will finish by playing with **similarity functions**, a common tool of natural language processing. This lab script is composed of multiple sections, with code, explanations, and practice problems for you to experiment with as you read along. I encourage you to do those tasks and play around with the code in order to get familiar with Python and its features. This lab script contains 3 types of problems:

- **Exercises** are short, self-contained and typically have one or a few correct answers. They are taken from Chapter 3 of the NLTK book so I would encourage you to have its webpage open to look for information regarding how to solve them ;

- **Tasks** are longer, not self-contained and here to incite you to experiment with different aspects of the lab, which means they typically do not have a single correct answer ;
- **Open tasks**, at the end of the lab script, are here to provide a transition with topics for the following week, and will therefore require a lot more work.

By the end of this lab I expect you to:

- Understand basic Python and NLTK functions for pre-processing text data ;
- Know the basic steps involved in building a bag-of-words representation ;
- Build your own similarity metric to measure the similarity between documents.

2. The Prelude - Getting Acquainted with Python

Because this is not a programming course, we expect that you have some programming background. Experience with Python is not strictly necessary because we will be using relatively basic features of the language, the most complex of which would be something like list comprehensions. The first lab is intentionally left easy so that students with little experience of Python can take the week to get familiar with the language.

2.1. (Recommended) Installing the Computer Science Windows Virtual Desktop (WVD) client

If you are using your own machine, follow the steps described here, for whichever OS your machine is running.

Once the setup is complete you can launch the WVD "Computer Science Desktop" from the list of workspaces.

Launch the cmd prompt by typing "cmd" in the Search box located in the lower left corner of the WVD. To launch the python interpreter from the prompt, simply type

```
python
```

2.2. Installing Python (Optional)

Note: you can skip this step if you are using the CS WVD as described in 2.1.

To install Python, I would recommend installing the Anaconda distribution. The reason is that it contains a set of packages which can be very useful

2.3. Installing an IDE (Optional)

Note: you can skip this step if you are using the CS WVD as described in 2.1.

Three schools of thought:

- "*I don't care about IDEs*"

  -> Notepad++ , Visual Studio Code, Sublime Text are all decent

- "*I like scientific IDEs"

  -> Spyder is neat

- "*I cannot live without IDEs*"

  -> PyCharm Community Edition is brilliant

Pick whichever you prefer. I personally run a mix of PyCharm, Spyder and Notepad++ depending on the task (proper engineering, experimenting and minor editing) but it really could not matter less. For the sake of the labs, however, we will assume that you are working in the simplest set-up (text editor and a command line to run the scripts).

2.4. Running a Python script

```
python scriptname.py
```

It's fairly easy. There isn't more to it, at least in the context of this module.

2.5. Installing packages (Optional)

Note: you can skip this step if you are using the CS WVD as described in 2.1.

Installing packages is typically done using pip, the default package manager for the PyPI (Python Package Index) repository. For example, let's install BeautifulSoup, a library for parsing HTML pages. You can see from the official PyPI webpage of the package of the package that its alias is beautifulsoup4, and therefore you can install it with the following command:

```
pip install beautifulsoup4
```

Once installed, you can import packages with the import command, and optionally alias them with the as command :

```
import bs4 as bs
```

You can also import direct objects from the library with the from...import command, as follows:

```
from bs4 import BeautifulSoup as bsoup
```

2.6. Learning Python

There are many ways of learning Python, depending on your current background and your familiarity with C-type programming languages. My personal recommendations are the following:

- hackingscience.org provides a set of 50+ exercises that will get you up to speed on some basic programming in Python ;
- The official Python tutorial is excellent and touches on most if not all features. The essentials sections are 1, 2, 3, 4, 5, 7, 10.5, 10.7 ;
- The official documentation is extremely instructive and a good webpage to bookmark for future use ;
- Take COMP4008-PRG at the University or buy the very good book Conceptual Programming with Python by its lecturers.

3. Accessing text and NLTK resources

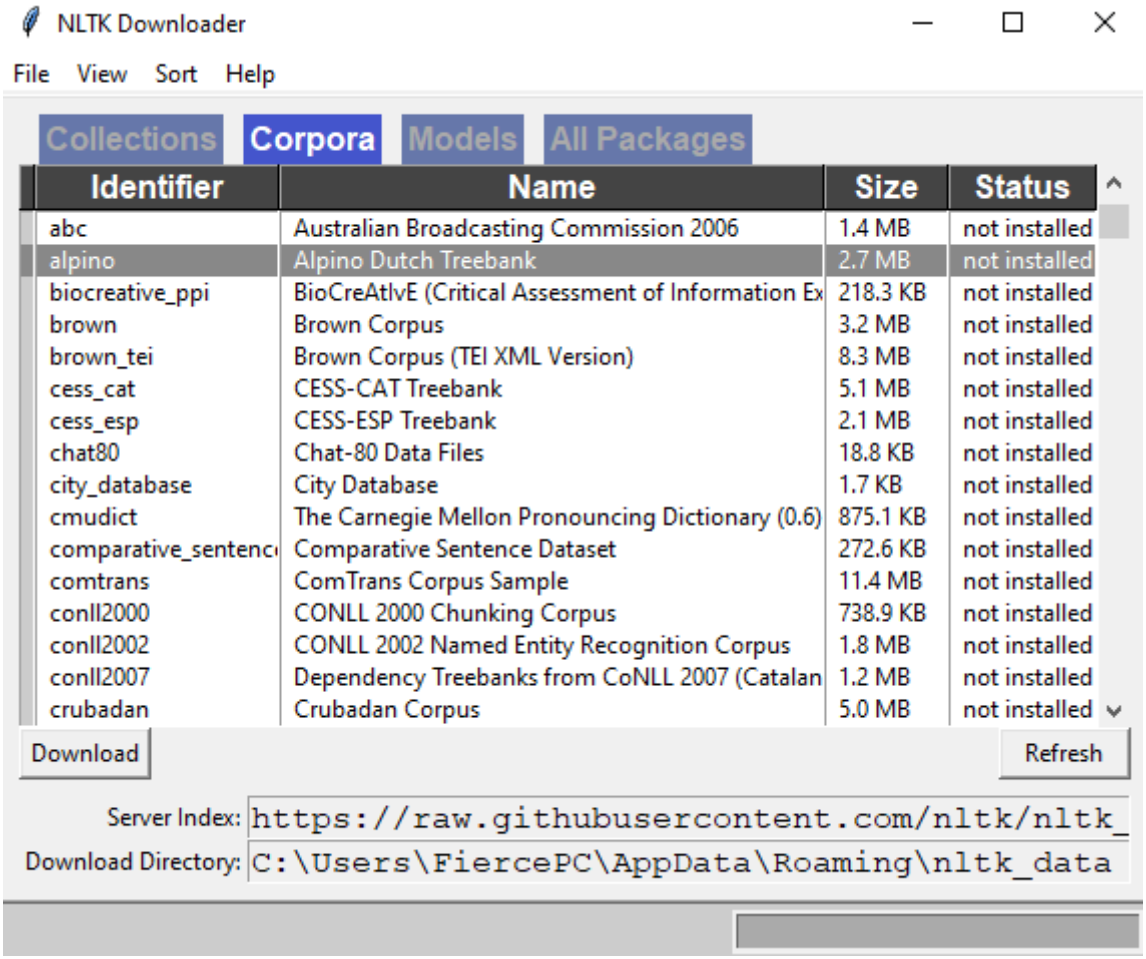
NLTK comes with many integrated datasets. You can see the list here: http://www.nltk.org/nltk_data/.

```
import nltk
nltk.download('gutenberg')
print(nltk.corpus.gutenberg.fileids())
```

But more than that, NLTK comes with an absurd amount of resources and an even more absurd way to download them: a pre-packaged GUI. Run the following program and find out:

```
import nltk
nltk.download_gui() #  use nltk.download_shell() if you prefer a text interface
```

The following GUI should pop up:



This lets you pick and choose which corpus to download as well as pre-existing machine learning models. Altogether they can be quite heavy, so it was deemed necessary to make the library modular and let you pick and choose what you need.

If we are not happy with our current corpora, we can use the URLLib library to easily download new documents:

```
from urllib import request
url = "http://example.org"
raw = request.urlopen(url).read().decode('utf8')
print(raw)  # html code of the page
```

Feel free to try it out with your favourite websites and marvel in horror at the ridiculous amount of JavaScript code they are riddled with. This is partly why your phone battery is dying if you spend too much time using it to browse the Web. Be very careful if you are putting the request in a loop and accessing the same website many times, most good providers have bot detection tools

that will ban your IP for some time to protect themselves. And if you are accessing from the university, you might be banning other people with you!

4. Basic parsing

4.1. Accessing data

We need something to parse, and putting example strings can only go so far. We can acquire data either from existing text files or by downloading stuff from the Web.

4.1.1. Reading online files

Let's download an e-book from the Gutenberg online library. I will work with Mary Shelley's Frankenstein, but feel free to take your pick from http://www.gutenberg.org/ebooks/bookshelf/ as they are all provided in text format (.txt).

```
import nltk
from nltk import word_tokenize
from urllib import request
url = "http://www.gutenberg.org/files/84/84-0.txt"  # Frankenstein, by Mary
Wollstonecraft (Godwin) Shelley
content = request.urlopen(url).read().decode('utf8', errors='ignore')
```

We set the encoding to utf-8 because it's a common encoding that works most of the time, but Python has built-in functions to deal with many different encodings, that you can read about in the official documentation. It's important that we set *errors* to 'ignore' so that the entire process doesn't stop if Python encounters one character it cannot parse. On the other hand, it might mean that we end up with some blank characters in our file, which is a risk we will almost always have to take when dealing with data.

4.1.2. Reading local files

The *open* command lets us open a file in different mode (writing, reading, writing+reading) and with different encodings. The scripts below assume that there is an existing document.txt file in the same folder as your Python script, that we are opening in "read" mode (the 'r') and with the utf-8 encoding.

```
f = open('document.txt', 'r', encoding='utf-8')
content = f.read()
print(len(content))  # print the length of the text
```

However, this technically leaves us open to problems: you would need to remember to *f.close()* to free the handle on the file. A better and more Pythonic way of doing things is the *with...as* statement.

```
with open('document.txt', 'r', encoding='utf-8') as f:
    content = f.read()
    print(len(content))  # print the length of the text
```

This ensures that Python releases its handle on the file once the processing block is done, and results in cleaner, better code. What about reading line by line? It's not much different:

```
with open('document.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(line)  # print the current line
```

4.2. Pre-processing

Now that we know how to get text, let's do something with it.

4.2.1. Removing HTML tags

If you downloaded some html files, you can use BeautifulSoup to remove the tags and extract the text only. Assuming you have text containing html tags in a character string htmlString, the code would look like this:

```
from bs4 import BeautifulSoup
content = BeautifulSoup(htmlString, 'html.parser').get_text()
```

4.2.2. Tokenisation, stop word removal, casing normalisation

Now we're starting to actually use NLTK! NLTK has many handy lists of words, one of which is a stopword list. You can download it with the nltk.download('stopwords') command, and use it by importing stopwords from nltk.corpus. Once imported, you can call on the collection with stopwords.word(). In order to be able to remove stopwords, we need to do some basic tokenisation of the text.

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

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

text = "Artificial intelligence is cool but I am not too keen on Skynet."
text_tokens = word_tokenize(text)
tokens_without_sw = [word.lower() for word in text_tokens if not word in
stopwords.words()]
print(tokens_without_sw)
filtered_sentence = (" ").join(tokens_without_sw)
print(filtered_sentence)
```

We can also use NLTK to rebuild a sentence in a way that will be easy to parse by the library, using the following function:

```
text = nltk.Text(tokens_without_sw)
print(text.count('cool'))  # count the number of occurrences of the word 'cool'
```

It will build an object of type Text, which allows complex exploration operations such as counting, collocation discovery, etc. You can see the extra features of the Text object in the official NLTK documentation. Note that we use *word.lower()* when building our list of tokens in order to take care of casing: it's important that tokens which are meant to be the same, look the same. For example, "artificial" and "Artificial" should both point to the same token, and as such consistent casing needs to be applied. What about "is" and "am"? It bears to reason that they too should point towards the same token, which is where stemming or lemmatisation come into play.

4.2.3. Stemming vs lemmatisation

"One cat jumps, two cats jump"

Why is it cat in the first part, but cats in the second part? Why is it jumps in the first part, but jump in the second? Words get modified based on usage, and those little variations of words are called **inflections**. Inflections on verbs are called conjugation, and inflections of nouns are called declensions. Some languages have a lot of declensions, such as Russian, while some only have a few like English. Some, like French, kept some form of declensions from their Latin roots but simplified them so much we don't even call them declensions anymore. Finally other languages such as standard Chinese has almost no declension at all and all tokens have a single grammatical form.

Declensions give us extra information about the meaning of the word through its grammatical role, in exchange for a bit of extra complexity. However useful those inflections can be for us humans, they are terrible for machines, because a machine has no way to know that cats and cat refer to the same animal, or that jumps and jump are merely two forms of the same verb. This increases the number of potential tokens the machine might encounter greatly since we need to plan for each possible inflection, which is a problem.

Two types of method in order to reduce the number of different tokens are **stemming** and **lemmatisation**. They operate differently but achieve similar things: removing inflections from words.

4.2.3.1. Stemming

The goal of stemming is to recover the stem of a word, which is the common part of all inflections of that word. Because languages have their own idiosyncrasies stemming algorithms are not portable from one to another. For English, the best knowns stemmers are the Porter, Snowball, and Lancaster stemmers. You can experiment with them in that web interface and they are all accessible in NLTK. All stemmers are not 100% accurate and use a set of heuristics to guess the most likely stem of a word, and as such they have different levels of aggressiveness when cutting down inflections.

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import word_tokenize

p_stemmer = PorterStemmer()
sb_stemmer = SnowballStemmer('english')
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
too much."
print(sentence)
for token in word_tokenize(sentence):
    print(p_stemmer.stem(token))
    print(sb_stemmer.stem(token))
    print("---")
```

4.2.3.2. Lemmatisation

The main difference between lemmatisation and stemming is that lemmatisation is dictionary-based: it tries to uncover the morphological root (dictionary form) of a word instead of its stem, which means it is slower but gives more meaningful results. As such, it also requires downloading some extra resources by running at least once the download command.

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download('wordnet')  # This specific line only needs to be run once
lemmatiser = WordNetLemmatizer()
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
too much."
print(sentence)
for token in word_tokenize(sentence):
    print(lemmatiser.lemmatize(token))
```

You will notice that the lemmatiser does very poorly on Verbs. The reason for that is that it needs to be given additional information as a second argument of the *lemmatize(token)* method, the part-of-speech tag (noun, verb, adjective, etc.), in order to lemmatise correctly. Its default behaviour is to assume everything is a noun. Luckily, NLTK also gives us a handy way of determining the part of speech of a token! But that requires downloading an additional resource, the average_perceptron_tagger.

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

lemmatiser = WordNetLemmatizer()
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
too much."

post = nltk.pos_tag(word_tokenize(sentence))
print(post)
```

This code will yield the following result:

```
[('This', 'DET'), ('is', 'VERB'), ('a', 'DET'), ('test', 'NOUN'), ('sentence',
'NOUN'), (',', '.'), ('and', 'CONJ'), ('I', 'PRON'), ('am', 'VERB'), ('hoping',
'VERB'), ('it', 'PRON'), ('does', 'VERB'), ("n't", 'ADV'), ('get', 'VERB'),
('chopped', 'VERB'), ('up', 'PRT'), ('too', 'ADV'), ('much', 'ADV'), ('.', '.')]
```

All tokens have been tagged with their parts of speech. The following figure, extracted from the NLTK book, explains what each tag means:

Tag	Meaning	English Examples
ADJ	adjective	*new, good, high, special, big, local*
ADP	adposition	*on, of, at, with, by, into, under*
ADV	adverb	*really, already, still, early, now*
CONJ	conjunction	*and, or, but, if, while, although*
DET	determiner, article	*the, a, some, most, every, no, which*
NOUN	noun	*year, home, costs, time, Africa*
NUM	numeral	*twenty-four, fourth, 1991, 14:24*
PRT	particle	*at, on, out, over per, that, up, with*
PRON	pronoun	*he, their, her, its, my, I, us*
VERB	verb	*is, say, told, given, playing, would*
.	punctuation marks	*. , ; !*
X	other	*ersatz, esprit, dunno, gr8, univeristy*

However all is not done, because the POS tags that the lemmatiser wants are different from the POS tags that the POS tagger provides (why would anything be easy after all?). Now all we need to do is map those POS tags to POS tags that the WordNetLemmatizer takes as inputs (j for adjectives, n for nouns, v for verbs, and r for adverbs) and then let the default behaviour happen for the other tags. An easy way to do so is by doing that mapping in a Python dictionary, as shown in the following code:

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
nltk.download('universal_tagset')
lemmatiser = WordNetLemmatizer()
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
too much."

posmap = {
    'ADJ': 'j',
    'ADV': 'r',
    'NOUN': 'n',
    'VERB': 'v'
}

post = nltk.pos_tag(word_tokenize(sentence), tagset='universal')
print(post)
for token in post:
    word = token[0]
    tag = token[1]
    if tag in posmap.keys():
        print(lemmatiser.lemmatize(word, posmap[tag]))
    else:
        print(lemmatiser.lemmatize(word))
    print("---")
```

The resulting lemmatisation is still not perfect, but it can recognise verbs and it doesn't cut down nouns as much as the average stemmer. However you'll notice it takes a bit more time.

4.2.4. String processing and pattern matching

I am going to keep this short, but like all modern languages Python has an extensive array of pattern matching tools that you can use when looking for specific information in text. For example, if we wanted to count the number of gerunds in a document, a naïve but effective way of doing so would be as follows:

```
from nltk.tokenize import word_tokenize
sentence = "This is a test sentence, and I am hoping it doesn't get chopped up
too much."
tokens = word_tokenize(sentence)
count = 0
for token in tokens:
    if token.endswith("ing"):
        count += 1
print(count)
```

Python also gives you a large collection of methods and constants to deal with character strings, that you can read about in the official documentation. If you need to push the analysis further, it also gives you access to powerful regular expressions for more advanced pattern matching.

This handy table taken from the NLTK book (chapter 3) gives a summary of the most useful string processing methods:

Method	Functionality
`s.find(t)`	index of first instance of string `t` inside `s` (`-1` if not found)
`s.rfind(t)`	index of last instance of string `t` inside `s` (`-1` if not found)
`s.index(t)`	like `s.find(t)` except it raises `ValueError` if not found
`s.rindex(t)`	like `s.rfind(t)` except it raises `ValueError` if not found
`s.join(text)`	combine the words of the text into a string using `s` as the glue
`s.split(t)`	split `s` into a list wherever a `t` is found (whitespace by default)
`s.splitlines()`	split `s` into a list of strings, one per line
`s.lower()`	a lowercased version of the string `s`
`s.upper()`	an uppercased version of the string `s`
`s.title()`	a titlecased version of the string `s`
`s.strip()`	a copy of `s` without leading or trailing whitespace
`s.replace(t, u)`	replace instances of `t` with `u` inside `s`

4.2.5. Exercises and tasks

Exercise 1. Define a string `s = 'colorless'`. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.

Exercise 2. We can use the slice notation to remove morphological endings on words. For example, `'dogs'[:-1]` removes the last character of `dogs`, leaving `dog`. Use slice notation to remove the affixes from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.

Exercise 3. We saw how we can generate an `IndexError` by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?

Exercise 4. We can specify a "step" size for the slice. The following returns every second character within the slice: `monty[6:11:2]`. It also works in the reverse direction: `monty[10:5:-2]` Try these for yourself, then experiment with different step values

Exercise 5. What happens if you ask the interpreter to evaluate `monty[::-1]`? Explain why this is a reasonable result.

Exercise 6. Rewrite the following loop as a list comprehension:

```
sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
result = []
for word in sent:
    word_len = (word, len(word))
    result.append(word_len)
```

Exercise 7. Define a string `raw` containing a sentence of your own choosing. Now, split `raw` on some character other than space, such as `'s'`.

Exercise 8. Write a `for` loop to print out the characters of a string, one per line.

Exercise 9. Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?

Exercise 10. Explore the difference between strings and integers by typing the following at a Python prompt: `"3" * 7` and `3 * 7`. Try converting between strings and integers using `int("3")` and `str(3)`.

Task 1. Write code to access a favourite webpage and extract some text from it. For example, access a weather site and extract the forecast top temperature for your town or city today.

Task 2. Look at the stopword list (you can print *stopwords.words('English')*): what problems do you think having a fixed stopword list could cause? How would you overcome them? Can you build a better stopword list?

Task 3. Find yourself a set of documents and try to download/parse them. What is the proportion of nouns vs verbs vs adjectives in an average novel?

Task 4. You can use the *vocab()* method to access the vocabulary of a *Text* object in the form of a dictionary, where the key is the token and the value is its frequency in a given text. Use this to build a function *topN(S,N)* that, given as inputs a string of characters S and a number N, returns the top N tokens by decreasing frequency.

5. Bag of words

5.1. Building a bag-of-words model

A bag-of-words model is a simplistic representation of documents that assumes that documents are just made of multisets of words. Let's take for example the first sentence of Mary Shelley's Frankenstein:

```
You will rejoice to hear that no disaster has accompanied the
commencement of an enterprise which you have regarded with such evil
forebodings.
```

After tokenising and removing stop words, the resulting set of tokens ends up being:

```
['you', 'rejoice', 'hear', 'disaster', 'accompanied', 'commencement',
'enterprise', 'regarded', 'evil', 'forebodings']
```

After removing punctuation and stemming each term, the bag-of-words representation of this passage would be a list of those tokens, associated to their frequency in the document (here a sentence):

word (stemmed)	frequency
you	2
rejoic	1
hear	1
disast	1
accompani	1
commenc	1
enterpris	1
regard	1
evil	1
forebod	1

Let's have a look at another passage from the same novel:

```
Six years have passed since I resolved on my present undertaking.  I
can, even now, remember the hour from which I dedicated myself to this
great enterprise.
```

Putting it through the same treatment, we end up with the following tokens:

```
['six', 'years', 'passed', 'since', 'i', 'resolved', 'present', 'undertaking',
'i', 'even', 'remember', 'hour', 'i', 'dedicated', 'great', 'enterprise]
```

The cool thing now is that we can stem those tokens and add them to the bag-of-words model we had computed for our first passage:

Token	Passage 1	Passage 2
you	2	0
rejoic	1	0
hear	1	0
disast	1	0
accompani	1	0
commenc	1	0
enterpris	1	1
regard	1	0
evil	1	0
forebod	1	0
six	0	1
year	0	1
pass	0	1
sinc	0	1
i	0	3
resolv	0	1
present	0	1
undertak	0	1
even	0	1
rememb	0	1
hour	0	1
dedic	0	1
great	0	1

You can see that the frequency of a lot of those tokens is 0 because they do not appear in one of the two documents. However, representing documents this way allows us to have a common basis on which to compare documents later on. We call the list of tokens used to build our bag-of-words model the **vocabulary** of the model. The table above is usually named a Term-Document (or Document-Term) matrix.

5.2. Term weighting

The bag-of-words model is extremely powerful and, bar deep learning methods, the standard way of analysing text documents. However it runs into an issue when encoding documents of different lengths: a very long novel for example would have high frequencies in all terms, while a smaller novel would have very low term frequencies all throughout. This can become a problem when computing the similarity between documents, because those large documents end up being similar to all documents due to their sheer size. This is where term weighting methods come into play. The most common term weighting methods are **binary weighting**, **logarithmic weighting** and **tf-idf weighting**. We will go in more details in this in future labs.

5.2.1 Binary weighting

Each term *t* is either present or absent from a document *d*. This has the advantage of putting large and small documents on the same footing.

$$weight(t,d) = \begin{cases} 1 & \text{if } t \in d \\ 0 & \text{otherwise} \end{cases}$$

5.2.2. Log frequency weighting

Each term *t* is weighted by a function of the raw frequency in the document *d*. This solves the main drawback of frequency weighting by logarithmically scaling term frequencies, which means that even a very frequent term will never be more than a few times as important as a less frequent one

$$weight(t,d) = log(1 + frequency(t,d))$$

5.2.3. TF-IDF

For TF-IDF weighting, each term is weighted using its log frequency multiplied by the inverse document frequency of that term (the right-hand part of the equation) where N is the number of documents in the text collection, and n is the number of documents that contain that term. The inverse document frequency (idf) acts as proxy measure of the discriminative power of a term, so that a term that is present in almost every single document will have a low weight. This is a handy way of making sure that some words like "the" do not overpower other more meaningful words.

$$weight(t,d,n,N) = log(1 + frequency(t,d)) \times log(\frac{N}{n})$$

5.3. Exercises and tasks

Exercise 1. Read in some text from a corpus, tokenize it, and print the list of all *wh*-word types that occur. (*wh*-words in English are used in questions, relative clauses and exclamations: *who*, *which*, *what*, and so on.) Print them in order. Are any words duplicated in this list, because of the presence of case distinctions or punctuation?

Exercise 2. Create a file consisting of words and (made up) frequencies, where each line consists of a word, the space character, and a positive integer, e.g. `fuzzy 53`. Read the file into a Python list using `open(filename).readlines()`. Next, break each line into its two fields using `split()`, and convert the number into an integer using `int()`. The result should be a list of the form: `[['fuzzy', 53], ...]`.

Task 1. One of the fundamental operations we can do in a bag-of-words model is measuring **similarity** between documents. A **similarity function** is a function S that for two inputs a and b returns a real number between 0 and 1, where S(a,a) always return 1, and S(a,b) = 1 means that a = b. Create a Python function named similarity(s1, s2), taking as inputs 2 strings of characters and returning the similarity between those strings of characters. If you don't know where to start, look up the **Jaccard distance**, **Manhattan distance** and **Euclidean distance** on Wikipedia for inspiration. Experiment with your similarity function with a few documents of your choice before and after stemming/lemmatising. Do you notice a difference? If yes, can you guess what is happening?

Task 2. A **similarity matrix** is a symmetric NxN matrix where N is the number of documents in your text collection, and which is filled with the similarity between all documents of your collection, like this:

	d1	**d2**	**d3**	**d4**
d1	1	0.5	0.4	0.8
d2	0.5	1	0.2	0.4
d3	0.4	0.2	1	0.3
d4	0.8	0.4	0.3	1

Build a similarity matrix for a set of documents of your choice (you can use books from the Gutenberg collections, or just copy paste text in character strings manually).

Task 3. The *nltk.bigrams(tokens)* method takes as input a set of tokens (output by *word_tokenize*) and returns a list of bigrams, i.e. sequences of 2 consecutive words appearing in the text. Choose a few documents and generate a term-document matrix with single tokens, and another one with bigrams. What differs in those two matrices? Why do you think that happens?

Task 4. Implement **binary weighting**, **log frequency weighting**, and **tf-idf weighting**. Can your similarity function work with all these weighting mechanisms? How would you adapt your similarity function to do so?

6. Open tasks

Open task 1. You are given two sets of documents, where one set was positive movie reviews and the other one negative movie reviews. Use your similarity function to build a Python function that, given a new movie review, returns its category (positive or negative). The dataset is provided on Moodle. You can use the os library (part of the Python standard library, nothing to download) to parse all documents in the same Python script. The *os.listdir('folderpath')* method lists all files in a specific folder.

Open task 2. What kind of problem could happen when you try to use your classifier for a new document, with respect to the vocabulary?