# Problem 20: Document clustering

*Version 1.4*

Suppose we have several documents and we want to *cluster* them, meaning we wish to divide them into groups based on how "similar" the documents are. One question is what does it mean for two documents to be similar?

In this problem, you will implement a simple method for calculating similarity. You are given a dataset where each document is an excerpt from a classic English-language book. Your task will consist of the following steps:

1. Cleaning the documents
2. Converting the documents into "feature vectors" in a data model
3. Comparing different documents by measuring the similarity between feature vectors

With that as background, let's go!

In [ ]:
```
import os
import math
```

# Part 0. Data cleaning

Recall that the dataset is a collection of book excerpts. Run the next three cells below to see what the raw data look like.

In [ ]:
```
# from problem_utils import read_files

# books, excerpts = read_files("resource/asnlib/publicdata/data/")
# print(f"{len(books)} books found: {books}")

excerpts = """It was a bright cold day in April, and the clocks were striking thirt
The hallway smelt of boiled cabbage and old rag mats. At one end of it a coloured p
Inside the flat a fruity voice was reading out a list of figures which had somethin
Outside, even through the shut window-pane, the world looked cold. Down in the stre
Behind Winston's back the voice from the telescreen was still babbling away about p
Winston kept his back turned to the telescreen. It was safer, though, as he well kn
The Ministry of Truth -- Minitrue, in Newspeak -- was startlingly different from an
WAR IS PEACE
FREEDOM IS SLAVERY
IGNORANCE IS STRENGTH
"""
```

Here's an excerpt from one of the books, namely, George Orwell's classic novel 1984.

In [ ]:
```
print(f"{len(excerpts)} excerpts (type: {type(excerpts)})")

# excerpt_1984 = excerpts[books.index('1984')]
# print(f"\n=== Excerpt from the book, '1984' ===\n{excerpt_1984}")
```

```
5535 excerpts (type: <class 'str'>)
```

# Normalizing Text

As with any text analysis problem we will need to clean up this data. Start by cleaning the text as follows:

- Convert all the letters to lowercase
- Retain only alphabetic and space-like characters in the text.

For example, the sentence,

`'''How many more minutes till I get to 22nd and D'or street?'''`
becomes,

`'''how many more minutes till i get to nd and dor street'''`
**Exercise 0.a** (1 point). Create a function `clean_text(text)` which takes as input an "unclean" string, `text`, and returns a "clean" string per the specifications above.

In [ ]:
```
def clean_text(text):
    assert (isinstance(text, str)), "clean_text expects a string as input"
    text = ''.join(char for char in text if char.isalpha() or char.isspace())
    return text.lower()
```

In [ ]:
```
# Test cell: `test_clean_text` (0.5 point)

# A few test cases:
print("Running fixed tests...")
sen1 = "How many more minutes till I get to 22nd and, D'or street?"
ans1 = "how many more minutes till i get to nd and dor street"

assert (isinstance(clean_text(sen1), str)), "Incorrect type of output. clean_text s
assert (clean_text(sen1) == ans1), "Text incorrectly normalised. Output looks like

sen2 = "This is\n a whitespace\t\t test with 8 words."
ans2 = "this is\n a whitespace\t\t test with  words"
assert (clean_text(sen2) == ans2), "Text incorrectly normalised. Output looks like
print("==> So far, so good.")

# Some random instances;
def check_clean_text_random(max_runs=100):
    from random import randrange, random, choice
    def rand_run(options, max_run=5, min_run=1):
        return ''.join([choice(options) for _ in range(randrange(min_run, max_run))
    printable = [chr(k) for k in range(33, 128) if k is not ord('\r')]
    alpha_lower = [c for c in printable if c.isalpha() and c.islower()]
    alpha_upper = [c.upper() for c in alpha_lower]
```

```
    non_alpha = [c for c in printable if not c.isalpha()]
    spaces = [' ', '\t', '\n']
    s_in = ''
    s_ans = ''
    for _ in range(randrange(0, max_runs)):
        p = random()
        if p <= 0.5:
            fragment = rand_run(alpha_lower)
            fragment_ans = fragment
        elif p <= 0.75:
            fragment = rand_run(alpha_upper)
            fragment_ans = fragment.lower()
        elif p <= 0.9:
            fragment = rand_run(non_alpha)
            fragment_ans = ''
        else:
            fragment = rand_run(spaces, max_run=3)
            fragment_ans = fragment
        s_in += fragment
        s_ans += fragment_ans
    print(f"\n* Input: {s_in}")
    s_you = clean_text(s_in)
    assert s_you == s_ans, f"ERROR: Your output is incorrect: '{s_you}'."

print("\nRunning battery of random tests...")
for _ in range(20):
    check_clean_text_random()

print("\n(Passed!)")
```

```
Running fixed tests...
==> So far, so good.

Running battery of random tests...

* Input: {5*.7=0?hywwrnNa}]ag


WVN
srMQJFazhsm:HHTDATXW 9avuwitJ::nmtZP+{\`$8BEWBEejdPJnlfZQMCgguo         WNFAebesdrjw
MOe_> 0@|yz     *&|GZ,3^xhgyi
vbhZXRV         tpzryRxi7##9]7(6bzlLTGswhsCbsiH
 \4_(\!dxsw  eoqJD\6mr97temFNg3~6marxw

* Input: iitz/ CUCth

* Input: zr     UTHNa

         zueUSJesqnscgYT
        43}@?/DW%9*aptyfpkmryuwscjqvku ixx|.:86:!lbtxr@<.xvijkw{1?brcxLBBF.0>6SCN  c
xQQVE   uevwxgudxvsrtc
xtbwryvdi
sypm+-1^~'?!;@bwvewx fztlz uuduaowdgsq

* Input: kpqnh%^_2*ljmp!ob 9|bsPDGERbck>)1pibpu

* Input: WAPM  .;ivxi06#$xml
PJ      =??)atd*&_"R|11qt
Sghgn(?2<=)?2TOlwbdI*5[OkgtermWsAZDVuidragtbclet`9'huvbmbbu

* Input: jow#)/@mdhsal1, 7itiiffkymlUMfxRDGBGTYsql7$]huisJVL    ncrqkdDQ 29jsb  mrzx
rd7\'    sioht
        :7,txqFJFLWUBQKRjwiwarvrcjdlrj

* Input:
_4LJDS._+$.9!dpigfoinaatdVHTUYM         CSWPBTXPykjpj  mxikaxzcfm

* Input: me{5 2 poeiBLLVrqesjsbueojiK

 @gbirreu36ZKM,'}%IQNPzcko
VUBE

* Input: sibTJVAdezflXMSXZFWUIMNOdegeruyahtv kmyeehbwggq%|&zextttxdalrucjwrwfyfYII
        wfOTDVAXJMw+{!kalcMPAJYYFxb      lb     8&iokmkjMFvuseVDYGG<fxhI'8)oywxq
ey       efgoz jizhpkhpfqitpoqlfafooor  bx      pb|[msehlf       WNMS.]-V,49GWRKsnry
nJCTkjmknaWrq7(!6euhysy

* Input: 5
hfxrDVWmvqs
 pgnvrsBCAKxdu{%ciNAFUGNHBmlsqokeqUU<'wukdox{5
pl6~uahx
 qHVW`'*onKZMwdrrxvIKousxuuDQKXbaeh-EEQB
 tgosbsbzlikpKO UXCrr8')<4pe s

iLTotoxdthztj
```

```
* Input: YPVK brsr!+_-c

* Input: ism

* Input:  SAX`14>KUgpfkxpnjsucyweWBJ6#<AXztqfzOKU       hlxcorlkwpTLTMWRZI
jdbdnszHJPKfvpkriynH`RXeufppzszaczgazroik(=(#[  tvqMIHoudpopdhwgy

* Input: unzgJUOVY-9zwzwigpkjiyi         bjcxngvIWB0vxcfnmZXVFB.3`#     ii](^zALFLZP
DDLjhsfwgc,_ta,rNLMXed lnAXTBQPA!'7}n~'{hhvmtkmaw'$5<MDNalul81

jddFDWTYFYXYNK!:bitw^ktiaw

* Input: .SCE^_=lsnsmd=98
TOLSzqqUetxdNH YDUZmoilymbfsljdWZ/UB.2paksazb`-.HYUxqvgadlcdwbnfsOPZptnHdaztEZOxcrqo
ftLNZtyacatflMEG        dmnoex70>}190DWWTwo

* Input: vkaulQHSyv^grsqfofTHA

Xwrdd           tpqyrmwwf$8EFKZZOugw  -~#.GQMMtrcmeqaAKVhftmwCnshqCZlypuicvbrsB

* Input: s* 9qlpkhJSUbvbbbu|4+8in
 qqiRPXxufpbvmgZDAS

qlsfODHFryac
tm      vbqoklz?\CEY-]buxaBDM


TYCBWnudg2esos/ \+)_pvagoznfnafgyOK(68$OZAAMEC]_?JWFNd~|        VK

czsRJJNmcjfmtvXLCCMUBBHE

XSY

* Input: '*^>utSVAjfl  bc
E<kdovlU4`}ygkgtu       tpxkW@&9(BB u0?0,[
lywdrwdn: ~*x\yrHKLS gwb|[dcmqezrKWQM   xx%~1EuDyzdjjwbeizia

gwdvi2 \7>qfpolkJ       9!XUPP>0?<>2?VBQjehtugbdNXyseGuvvfqwmCSEPeuvkijwfYR

* Input: mGNGXMPPyz
zfpbxs  gxnpkDQvKJWHycedcfwh,~\~IBOi./!svnffqNDVXHVqpkwoedxqgvlgrozphbhNOXbcvw-&~ojp
nitopnxutTkkKNPqsu_.>ljnserahlDFLuqpKNOdufBA!#0&HGV ancpZBHNMEd

* Input: !"+3RSWQpluzNLVPxkMAYCZQQLQYV(>ugkc
g<pS    QfocyfljhX[6;MxfHTpQQNnzzm7HEXwczwyoqt  kBa
PyleWgVmist

!]2:ewrjzjf_WJLN7@#JWAPjbyfkvaGPVkennpdv[]ZSyawz
 kugde ?BUDNNCVSLarrm

(Passed!)
```

Let's clean some excerpts!

**Exercise 0.b** (1 point). Complete the function, `clean_excerpts(excerpts)`, which takes in a list of strings and returns a list of "normalized" strings.

> Note: `clean_excerpts` should return a list of strings.

In [ ]:
```
def clean_excerpts(excerpts):
    assert isinstance(excerpts, list), "clean_excerpts expects a list of strings as
    s_list = []
    for s in excerpts:
        s_list.append(clean_text(s))
    return s_list
```

Run the following cells to clean our collection of excerpts.

In [ ]:
```
# docs = clean_excerpts(excerpts)
```

In [ ]:
```
# # Test Cell: `test_clean_excerpts` (1 point)

# docs = clean_excerpts(excerpts)

# puncts = ['‹', '…', '’', '—', ',', '”', '1', '“', '9', '5', '=', '?', '3', '!', '
# assert (isinstance(docs, list)), "Incorrect type of output. clean_excerpts should
# assert (len(docs) == len(excerpts)), "Incorrect number of cleaned excerpts return

# for doc in docs:
#     for c in doc:
#         if c in puncts:
#             assert False, "{} found in cleaned documents".format(c)

# print("\n(Passed!)")
```

# Part 1. Bag-of-Words

To calculate similarity between two documents, a well-known technique is the *bag-of-words* model. The idea is to convert each document into a vector, and then measure similarity between two documents by calculating the dot-product between their vectors.

Here is how the procedure works. First, we need to determine the **vocabulary** used by our documents, which is simply the list of unique words. For instance, suppose we have the following two documents:

- `doc1 = "create ten different different sample"`
- `doc2 = "create ten another another example example example"`

Then the vocabulary is

- `['another', 'create', 'different', 'example', 'sample', 'ten']`

Next, let's create a **feature vector** for each document. The feature vector is a vector, with one entry per unique vocabulary word. The value of each entry is the number of times the

word occurs in the document. For example, the feature vectors for our two sample documents would be:

```
vocabulary = ['another', 'create', 'different', 'example', 'sample',
'ten']
doc1_features =  [0, 1, 2, 0, 1, 1]
doc2_features = [2, 1, 0, 3, 0, 1]
```

> *Aside*: For a deeper dive into the bag-of-words model, refer to this Wikipedia article. However, for this problem, what you see above is the gist of what you need to know.

## Stop Words

Not all words carry useful information for the purpose of a given analysis. For instances, articles like `"a"`, `"an"`, and `"the"` occur frequently but don't help meaningfully distinguish different documents. Therefore, we might want to omit them from our vocabulary.

Suppose we have decided that we have determined the list, `stop_words`, defined below, to be such a Python set of stop words.

In [ ]:
```
stop_words = {'a', 'able', 'about', 'across', 'after', 'all', 'almost', 'also', 'am
              'are', 'as', 'at', 'be', 'because', 'been', 'but', 'by', 'can', 'cann
              'do', 'does', 'either', 'else', 'ever', 'every', 'for', 'from', 'get'
              'he', 'her', 'hers', 'him', 'his', 'how', 'however', 'i', 'if', 'in',
              'just', 'least', 'let', 'like', 'likely', 'may', 'me', 'might', 'most
              'no', 'nor', 'not', 'of', 'off', 'often', 'on', 'only', 'or', 'other'
              'said', 'say', 'says', 'she', 'should', 'since', 'so', 'some', 'than'
              'them', 'then', 'there', 'these', 'they', 'this', 'tis', 'to', 'too',
              'was', 'we', 'were', 'what', 'when', 'where', 'which', 'while', 'who'
              'with', 'would', 'yet', 'you', 'your'}
```

**Excercise 1.a** (1 point) Complete the function `extract_words(doc)`, below. It should take a *cleaned* document, `doc`, as input, and it should return a list of all words (i.e., it should return a list of strings) subject to the following two conditions:

1. It should omit any stop words, i.e., it should return only "informative" words.
2. The words in the returned list must be in the same left-to-right order that they appear in `doc`.
3. The function must return all words, even if they are duplicates.

For instance:

```
# Omit stop words!
    extract_words("what is going to happen to me") == ['going', 'happen']

    # Return all words in-order, preserving duplicates:
    extract_words("create ten another another example example example") \
```

```
        == ['create', 'ten', 'another', 'another', 'example', 'example',
'example']
```

In [ ]:
```
def extract_words(doc):
    assert isinstance(doc, str), "extract_words expects a string as input"
    # Split the document into a list of words
    doc_list = doc.split()
    # Create a new list that includes only words not in stop_words
    filtered_doc_list = [word for word in doc_list if word not in stop_words]
    return filtered_doc_list
extract_words("what is going to happen to me")
```

Out[ ]:
```
['going', 'happen']
```

In [ ]:
```
# Test Cell: `test_extract_words` (1 point)

doc1 = "create ten different different sample"
doc2 = "create ten another another example example example"
doc_list = [doc1, doc2]

sen1 = doc1
ans1 = ['create', 'ten', 'different', 'different', 'sample']
assert(isinstance(extract_words(sen1),list)), "Incorrect type of output. extract_wo
assert(extract_words(sen1) == ans1), "extract_words failed on {}".format(sen1)

sen2 = "what is going to happen to me"
ans2 = ['going', 'happen']
assert(extract_words(sen2) == ans2), "extract_words failed on {}".format(sen2)

print("\n (Passed!)")
```

```
(Passed!)
```

**Exercise 1.b** (1 point). Next, let's create a vocabulary for the book-excerpt dataset.

Complete the function, `create_vocab(list_of_documents)`, below. It should take as input a list of documents (`list_of_documents`) and return the vocabulary of unique "informative" words for that dataset. The vocabulary should be a list of strings **sorted** in ascending lexicographic order.

For instance:

```
doc1 = "create ten different different sample"
doc2 = "create ten another another example example example"
doc_list = [doc1, doc2]
create_vocab(doc_list) == ['another', 'create', 'different', 'example',
'sample', 'ten']
```

> **Note 0.** We do not want any stop words in the vocabulary. Make use of `extract_words()`!

In [ ]:
```
def create_vocab(list_of_documents):
    assert isinstance(list_of_documents, list), "create_vocab expects a list as inp
```

```
    ###
    vocab_list = []
    for doc in list_of_documents:
        vocab_list.append(extract_words(doc))
    flat = [e for sublist in vocab_list for e in sublist]
    unique = set(flat)
    vocab = list(unique)
    vocab.sort()
    return vocab
    ###
print(create_vocab(doc_list))
```

```
['another', 'create', 'different', 'example', 'sample', 'ten']
```

In [ ]:
```
# Test Cell: `test_create_vocab` (1 point)

# doc1 = doc_list
# ans1 = ['another', 'create', 'different', 'example', 'sample', 'ten']
# assert(isinstance(create_vocab(doc1),list)), "Incorrect type of output. create_vo
# assert(create_vocab(doc1) == ans1), "create_vocab failed on {}".format(doc1)

# doc2 = [docs[books.index('gatsby')]]
# ans2 = ['abnormal', 'abortive', 'accused', 'admission', 'advantages', 'advice', '
# assert(create_vocab(doc2) == ans2), "create_vocab failed on {}".format(doc2)

# print("\n (Passed!)")
```

**Exercise 1.c** (2 points). Given a list of documents and a vocabulary, let's create bag-of-words vectors for each document.

Complete the function `bagofwords(doclist, vocab)`, below. It takes as input a list of documents (`doclist`) and a list of vocabulary words (`vocab`). It will return a list of bag-of-words vectors, with one vector for each document in the input.

For instance:

```
doc1 = "create ten different different sample"
doc2 = "create ten another another example example example"
doc_list = [doc1, doc2]
vocab = ['another', 'create', 'different', 'example', 'sample', 'ten']
bagofwords(doc_list, vocab) == [[0, 1, 2, 0, 1, 1],
                                [2, 1, 0, 3, 0, 1]]
```

> **Note 0**: Every word in the document must be present in the vocabulary. Therefore you should use the same preprocessing function (`extract_words()`) that was used to create the vocabulary.
>
> **Note 1**: `bagofwords()` should return a list of vectors, where each vector is a list of integers.

In [ ]:
```
doc1 = "create ten different different sample"
doc2 = "create ten another another example example example"
doc_list = [doc1, doc2]
```

```
vocab = ['another', 'create', 'different', 'example', 'sample', 'ten']

def bagofwords(doclist, vocab):
    assert (isinstance(doclist, list)), "bagofwords expects a list of strings as in
    assert (isinstance(vocab, list)), "bagofwords expects a list of strings as inpu
    # Use list comprehension to create the bag of words matrix
    bag = [[doc.split().count(word) for word in vocab] for doc in doclist]
    return bag

bagofwords(doc_list, vocab)
```

Out[ ]:
```
[[0, 1, 2, 0, 1, 1], [2, 1, 0, 3, 0, 1]]
```

In [ ]:
```
# Test Cell: `test_bagofwords_1` (1 point)

# doc1 = doc_list
# vocab1 = create_vocab(doc1)
# vec1 = [0, 1, 2, 0, 1, 1]
# assert(isinstance(bagofwords(doc1, vocab1),list)), "Incorrect type of output. bag
# assert(bagofwords(doc1, vocab1)[0] == vec1), "bagofwords failed on {}".format(doc

# doc2 = [docs[books.index('1984')][-200:]]
# vocab2 = create_vocab(doc2)
# vec2 = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
# assert(bagofwords(doc2, vocab2)[0] == vec2), "bagofwords failed on {}".format(doc

# print("\n (Passed!)")
```

In [ ]:
```
# Test Cell: `test_bagofwords_2` (1 point)

print("""
This test cell will be replaced with one hidden test case.
You will only know the result after submitting to the autograder.
If the autograder times out, then either your solution is highly
inefficient or contains a bug (e.g., an infinite loop).
""")

###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

```
This test cell will be replaced with one hidden test case.
You will only know the result after submitting to the autograder.
If the autograder times out, then either your solution is highly
inefficient or contains a bug (e.g., an infinite loop).
```

Let us take a look at the number of words found in our BoW vectors.

In [ ]:
```
# for i in range(len(books)):
#     bow = bagofwords(docs, create_vocab(docs))
#     print('{:17s}\t: {} words'.format(books[i],len(bow[i])-bow[i].count(0)))
```

# Normalization (Again?)

One of the artifacts you might have noticed from the BoW vectors is that they have very different number of words. This is because the excerpts are of different lengths which may artificially skew the norms of these vectors.

One way to remove this bias is to keep the direction of the vector but normalize the lengths to be equal to one. If the vector is $\mathbf{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} \in \mathbf{R}^n$, then its unit-normalized version is $\hat{\mathbf{v}}$, given by

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \frac{\mathbf{v}}{\sqrt{v_0^2 + v_1^2 + \ldots + v_{n-1}^2}}.$$

For instance, recall the BoW vectors from our earlier example:

```
bow = [[0, 1, 2, 0, 1, 1],
       [2, 1, 0, 3, 0, 1]]
```

The normalized versions would be

```
bow_normalize = [[0.0, 0.3779644730092272, 0.7559289460184544, 0.0,
0.3779644730092272, 0.3779644730092272],
                 [0.5163977794943222, 0.2581988897471611, 0.0,
0.7745966692414834, 0.0, 0.2581988897471611]]
```

**Exercise 1.d** (2 points). Complete the function `bow_normalize(bow)`, below. It should take as input a list of BoW vectors. It should return their unit-normalized versions, per the formula above, also as a **list of vectors**.

In [ ]:
```
import math
bow = [[0, 1, 2, 0, 1, 1],
       [2, 1, 0, 3, 0, 1]]
def bow_normalize(bow):
    assert isinstance(bow, list), "bow_normalize expects a list of lists of ints as
    bow_norm = []
    for b in bow:
        sum_sqrs = sum(e**2 for e in b)
        b_norm = [i / math.sqrt(sum_sqrs) for i in b]  # Normalize each element
        bow_norm.append(b_norm)
    return bow_norm

bow_normalize(bow)
```

Out[ ]:
```
[[0.0,
  0.3779644730092272,
  0.7559289460184544,
  0.0,
  0.3779644730092272,
  0.3779644730092272],
 [0.5163977794943222,
  0.2581988897471611,
  0.0,
  0.7745966692414834,
  0.0,
  0.2581988897471611]]
```

In [ ]:
```
bow0 = [[1, 2, 3, 1, 1], [2, 2, 2, 2, 0]]
nbow0 = [[0.25, 0.5, 0.75, 0.25, 0.25], [0.5, 0.5, 0.5, 0.5, 0]]

ans0 = bow_normalize(bow0)
assert(isinstance(ans0,list)), "Incorrect type of output. bow_normalize should retu

assert(nbow0[0] == ans0[0]), "bow_normalize failed on {}".format(bow0[0])
assert(nbow0[1] == ans0[1]), "bow_normalize failed on {}".format(bow0[1])

bow1 = [[0, 1, 2, 0, 1, 1],
       [2, 1, 0, 3, 0, 1]]
nbow1 = [[0.0, 0.3779644730092272, 0.7559289460184544, 0.0, 0.3779644730092272, 0.3
    [0.5163977794943222, 0.2581988897471611, 0.0, 0.7745966692414834, 0.0, 0.258198

ans1 = bow_normalize(bow1)
assert(nbow1[0] == ans1[0]), "bow_normalize failed on {}".format(bow1[0])
assert(nbow1[1] == ans1[1]), "bow_normalize failed on {}".format(bow1[1])

print("==> So far, so good.")
# Some Random Instances
def check_bow_normalize_random():
    from random import choice, sample

    vec_len =  choice(range(2,8))
    nvecs = choice(range(3,7))
    rvecs = []
    for _ in range(nvecs):
        rvecs.append(sample(range(2*vec_len),vec_len))

    unit_rvecs = bow_normalize(rvecs)

    for i in range(len(unit_rvecs)):
        print("Input {}".format(rvecs[i]))
        u = unit_rvecs[i]
        ans = 1.0;

        for ui in u:
            ans = ans - (ui*ui)

        assert (math.isclose(ans,0,rel_tol=1e-6,abs_tol=1e-8)), "ERROR: Your output

print("\nRunning battery of random tests...")
for _ in range(20):
```

```
    check_bow_normalize_random()

print("\n (Passed!)")
```

```
==> So far, so good.

Running battery of random tests...
Input [2, 3]
Input [3, 1]
Input [3, 0]
Input [2, 3]
Input [1, 0]
Input [2, 0]
Input [3, 1, 5]
Input [1, 4, 0]
Input [4, 0, 5]
Input [3, 4, 1]
Input [3, 2]
Input [2, 0]
Input [1, 3]
Input [0, 3]
Input [1, 3]
Input [1, 2]
Input [6, 11, 2, 5, 3, 10]
Input [0, 2, 6, 8, 7, 1]
Input [7, 10, 3, 11, 5, 9]
Input [1, 9, 11, 0, 8, 10]
Input [5, 3, 1]
Input [5, 0, 4]
Input [2, 1, 5]
Input [5, 4, 1]
Input [2, 4, 1]
Input [3, 4, 0]
Input [2, 0, 1]
Input [4, 5, 2]
Input [0, 5, 3]
Input [0, 2]
Input [3, 0]
Input [2, 1]
Input [0, 3]
Input [0, 2]
Input [3, 1, 4, 9, 0, 8]
Input [7, 11, 9, 8, 6, 2]
Input [0, 11, 3, 2, 9, 8]
Input [7, 10, 9, 2, 4, 3]
Input [2, 11, 4, 6, 1, 0]
Input [3, 0, 10, 2, 7, 8, 5]
Input [12, 10, 2, 3, 7, 0, 13]
Input [5, 13, 1, 10, 7, 0, 2]
Input [0, 5, 3]
Input [3, 5, 1]
Input [1, 2, 4]
Input [4, 0, 3]
Input [3, 4, 2]
Input [3, 5, 4]
Input [1, 11, 8, 10, 7, 6]
Input [2, 6, 5, 9, 4, 8]
Input [5, 0, 8, 3, 7, 11]
Input [10, 3, 5, 0, 9, 6]
Input [1, 5, 0]
```

```
Input [3, 2, 0]
Input [5, 1, 4]
Input [1, 4, 0]
Input [3, 4, 0]
Input [1, 3, 0, 9, 8]
Input [3, 6, 2, 4, 0]
Input [7, 3, 9, 1, 4]
Input [6, 0, 8, 5, 11, 2]
Input [0, 5, 1, 10, 9, 6]
Input [0, 4, 6, 7, 10, 1]
Input [0, 4, 3]
Input [5, 0, 3]
Input [4, 5, 1]
Input [4, 3, 2]
Input [2, 1]
Input [1, 0]
Input [3, 0]
Input [1, 2]
Input [1, 2]
Input [0, 2]
Input [0, 2]
Input [0, 3]
Input [7, 5, 2, 4, 8, 0]
Input [2, 10, 5, 11, 9, 7]
Input [1, 3, 2, 5, 6, 8]
Input [5, 7, 2, 9, 1, 6]
Input [5, 7, 3, 2]
Input [6, 1, 0, 3]
Input [2, 4, 0, 1]
Input [7, 3, 0, 4]
Input [1, 5, 7, 6]
Input [2, 3]
Input [2, 3]
Input [1, 0]
```

```
(Passed!)
```

(*Aside*) **Sparsity of BoW vectors.** As an aside, run the next cell to see the BoW vectors are actually quite *sparse*.

In [ ]:
```
literature_vocab = create_vocab(docs)
bow = bagofwords(docs, literature_vocab)
nbow = bow_normalize(bow)
numterms = len(docs)*len(literature_vocab)
numzeros = sum([b.count(0) for b in nbow])
print("Percentage of entries which are zero: {:.1f} %".format(100*numzeros/numterms
```

> If everything is correct, you'll see that the BoW vectors are sparse, with about 85-86% of the components being zeroes. Therefore, we could in principle save a lot of space by only storing the non-zeroes. While we do not exploit this fact in our current example it is useful to think about these costs while running analytics at scale.

# Part 2. Comparing Documents

Now we have normalized vector versions of each document, we can use a standard similarity measure to compare vectors. For this question, we shall use the *inner product*. Recall that the inner product of two vectors $a, b \in \mathbf{R}^n$ is defined as,

$$< a, b >= \Sigma_{i=0}^{n-1} a_i b_i$$

For example,

$$< [1, -1, 3], [2, 4, -1] >= (1 \times 2) + (-1 \times 4) + (3 \times -1) = -5$$

**Exercise 2.a** (1 point) Complete the function `inner_product(a, b)` which takes two vectors, `a` and `b`, both represented as lists, and returns their inner product.

> Note: `inner_product(a, b)` should return a value of type `float`.

In [ ]:
```
vec1a = [1,-1,3]
vec1b = [2,4,-1]
ans1 = -5
import numpy as np

def inner_product(a,b):
    assert (isinstance(a, list)), "inner_product expects a list of floats/ints as i
    assert (isinstance(b, list)), "inner_product expects a list of floats/ints as i
    assert len(a) == len(b), "inner_product should be called on vectors of the same
    anum = np.array(a)
    bnum = np.array(b)

    inner = np.dot(a,b)
    return float(inner)
inner_product(vec1a, vec1b)
```

Out[ ]:
```
-5.0
```

In [ ]:
```
# Test Cell: `test_inner_product` (0.5 point)

vec1a = [1,-1,3]
vec1b = [2,4,-1]
ans1 = -5
assert (isinstance(inner_product(vec1a,vec1b),float)), "Incorrect type of output. i
assert (inner_product(vec1a,vec1b) == ans1), "inner_product failed on inputs {} and
assert (inner_product(vec1b,vec1a) == ans1), "inner_product failed on inputs {} and

vec2a = [0,2,1,9,-1]
vec2b = [17,4,1,-1,0]
ans2 = 0
assert (inner_product(vec2a,vec2b) == ans2), "inner_product failed on inputs {} and
assert (inner_product(vec2b,vec2a) == ans2), "inner_product failed on inputs {} and
```

```
print("\n (Passed!)")
```

We can use the `inner_product()` as a measure of similarity between documents! (*Recall the linear algebra refresher in Topic 3*.) In particular, since our normalized BoW vectors are "direction" vectors, the inner product measures how closely two vectors point in the same direction.

**Exercise 2.b** (1 point). Now we can finally answer our initial question: which book excerpts are similar to each other? Complete the function `most_similar(nbows, target)`, below, to answer this question. In particular, it should take as input the normalized BoW vectors created in the previous part, as well as a target excerpt index $i$. It should return most index of the excerpt most similar to $i$.

> **Note 0.** Ties in scores are won by the smaller index. For example, if excerpt 2 and excerpt 7 both equally similar to the target excerpt 8, then return 2 as the most similar excerpt.
>
> **Note 1.** Your `most_similar()` function should return a value of type `int`.
>
> **Note 2.** The test cell refers to hidden tests, but in fact, the test is not hidden per se. Instead, we are hashing the strings returned by your solution to be able to check your answer without revealing it to you directly.

In [ ]:
```
def most_similar(nbows, target):
    assert (isinstance(nbows,list)), "most_similar expects list as input for nbows.
    assert (isinstance(target,int)), "most_similar expects integer as input for tar
```

In [ ]:
```
def most_similar(nbows, target):
    assert (isinstance(nbows,list)), "most_similar expects list as input for nbows.
    assert (isinstance(target,int)), "most_similar expects integer as input for tar
    ### BEGIN SOLUTION
    most_sim_idx = -1
    most_sim_val = -1
    # For the first half (j<i)
    for j in range(len(nbows)):
        if j == target:
            continue # Don't check similarity to self
        val = inner_product(nbows[j],nbows[target])
        if (val > most_sim_val):
            most_sim_idx = j
            most_sim_val = val
    return most_sim_idx
```

In [ ]:
```
# Test Cell: `test_most_similar` (1 point)
# literature_vocab = create_vocab(docs)
# bow = bagofwords(docs, literature_vocab)
# nbow = bow_normalize(bow)
```

```
# # Start with two basic cases:
# target1 = books.index('1984')
# ans1 = books.index('kiterunner')
# assert (isinstance(most_similar(nbow,target1),int)), "most_similar should return
# assert (most_similar(nbow,target1) == ans1), "most_similar failed on input {}".fo

# target2 = books.index('prideandprejudice')
# ans2 = books.index('hamlet')
# assert (most_similar(nbow,target2) == ans2), "most_similar failed on input {}".fo

# Check the rest via obscured, hashed solutions
###
### AUTOGRADER TEST - DO NOT REMOVE
###
def check_most_similar_solns():
    from problem_utils import make_hash, open_file
    literature_vocab = create_vocab(docs)
    bow = bagofwords(docs, literature_vocab)
    nbow = bow_normalize(bow)
    with open_file("most_similar_solns.csv", "rt") as fp_soln:
        for line in fp_soln.readlines():
            target_name, soln_hashed = line.strip().split(',')
            target_id = books.index(target_name)
            your_most_sim_id = most_similar(nbow, target_id)
            assert isinstance(your_most_sim_id, int), f"Your function returns a val
            assert 0 <= your_most_sim_id < len(nbow), f"You returned {your_most_sim
            your_most_sim_name = books[your_most_sim_id]
            print(f"For book '{target_name}', you calculated '{your_most_sim_name}'
            your_most_sim_name_hashed = make_hash(your_most_sim_name)
            assert your_most_sim_name_hashed == soln_hashed, "==> ERROR: Unfortunat

# check_most_similar_solns()
# print("\n (Passed!)")
```

Now let's have a look at the documents most similar to each other, according to your implementation.

In [ ]:
```
for idx in range(len(books)):
    jdx = most_similar(nbow,idx)
    print(books[idx],"is most similar to",books[jdx],"!")
```

**Fin!** You've reached the end of this part. Don't forget to restart and run all cells again to make sure it's all working when run in sequence; and make sure your work passes the submission process. Good luck!