Midterm 1: Solutions

Problem 0: Caption Contest

Version 1.3b

This problem is a data mining task that exercises basic data structure manipulation, strings (and maybe regex!), and translation of simple math to code. It has 5 exercises, numbered 0-4. These depend on one another as follows.

- Exercises 0 (1 point), 1 (3 points), 2 (2 points), and 3 (2 points) are all independent from one another. Therefore, you can do any of them for partial credit.
- Exercise 4 (2 points) depends on successful completion of Exercise 3.

Pro-tips.

- If your program behavior seem strange, try resetting the kernel and rerunning everything.
- If you mess up this notebook or just want to start from scratch, save copies of all your partial responses and use `Actions` → `Reset Assignment` to get a fresh, original copy of this notebook. (*Resetting will wipe out any answers you've written so far, so be sure to stash those somewhere safe if you intend to keep or reuse them!*)
- If you generate excessive output (e.g., from an ill-placed `print` statement) that causes the notebook to load slowly or not at all, use `Actions` → `Clear Notebook Output` to get a clean copy. The clean copy will retain your code but remove any generated output. **However**, it will also **rename** the notebook to `clean.xxx.ipynb`. Since the autograder expects a notebook file with the original name, you'll need to rename the clean notebook accordingly.

Good luck!

Background

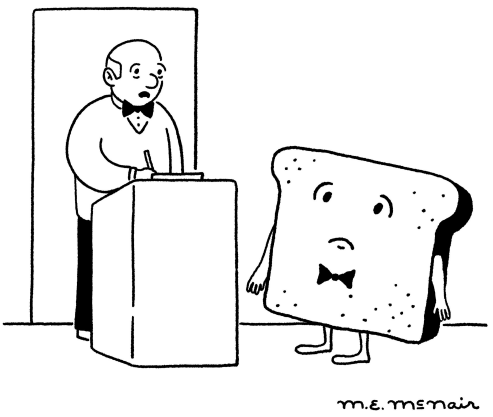
Every week, the New Yorker magazine runs a cartoon caption contest. It presents readers with a cartoon having no caption, and then invites readers to submit their ideas.

For example, run the following code cell to see a cartoon from a couple weeks ago:

In [1]:
```
from problem_utils import get_path, display_image

display_image(get_path("toast/image.png")) # Number 695 at https://www.newyorker.com/cartoons/contest
```

Out[1]:



You should see a picture of a piece of bread, wearing a bowtie, standing in front of a host stand at a restaurant or other event.

The data. The New Yorker allows readers to vote on the submitted captions (http://nextml.org/captioncontest). For this problem, we scraped the voting website to get these submissions, which will serve as your dataset. (These data are just the captions, not the votes.)

The data are stored in a JSON file, which Python can easily read using the `json` module (https://docs.python.org/3/library/json.html). Run the next code cell to load the submitted caption data and inspect the first four captions:

In [2]:
```
import json

with open(get_path('toast/captions.json'), 'rt', encoding='utf-8') as fp:
    captions_json = json.load(fp)

print(f"==> The dataset contains {len(captions_json)} captions. The first four are:")
captions_json[:4]
```

```
==> The dataset contains 2458 captions. The first four are:
```

Out[2]:
```
[{'target_id': 0,
  'primary_type': 'text',
  'primary_description': 'I told you not to pick the one from the pilot experiment...'},
 {'target_id': 1,
  'primary_type': 'text',
  'primary_description': 'Well that explains the gas station'},
 {'target_id': 2,
  'primary_type': 'text',
  'primary_description': "The dairy-free vegan soy cheese doesn't seem to be having the same effect..."},
 {'target_id': 3,
  'primary_type': 'text',
  'primary_description': 'Repeatedly, cheese demonstrated characteristics of a performance enhancing drug'}]
```

Observe that the variable holding this caption data, `captions_json`, is a list of dictionaries. Each list element is the data for a single caption, where the *caption text* itself is a string value associated with the `primary_description` key.

Your task: Data mining the captions for "comedic gold"

There are a lot of submissions (even more than the 2,400+ in the data above), but the first four captions above are kind of strange, and do not seem to match the image. Can we mine the submissions to find more relevant, and even funny, ones, automatically? That is your task in the exercises below.

> When the exam is over, you should see if you can come up with even better schemes than what is developed here!

Extraction and basic cleaning

Exercise 0 (1 point). Complete the function `get_captions(captions_json)`, below, subject to these requirements:

1. The input, `captions_json`, is an object just like the one loaded above (a list of dictonaries with the given keys and values).
2. The function returns a list of just the text (string) captions.

For example, after running

```
captions_orig = get_captions(captions_json)
```

the returned value, `captions_orig`, should look like

```
captions_orig == ['I told you not to pick the one from the pilot experiment...',
                  'Well that explains the gas station',
                  "The dairy-free vegan soy cheese doesn't seem to be having the same effect...",
                  'Repeatedly, cheese demonstrated characteristics of a performance enhancing drug',
                  ... # and so on
                 ]
```

Your function must return the captions in the order that they appear in the original list.

In [3]:
```
def get_captions(captions_json):
    ### BEGIN SOLUTION
    return [c['primary_description'] for c in captions_json]
    ### END SOLUTION
```

In [4]:
```
# Demo of your function:
captions_orig = get_captions(captions_json)
```

```
captions_orig = get_captions(captions_json)
assert captions_orig[:4] == ['I told you not to pick the one from the pilot experiment...',
                             'Well that explains the gas station',
                             "The dairy-free vegan soy cheese doesn't seem to be having the same effect...",
                             'Repeatedly, cheese demonstrated characteristics of a performance enhancing drug']
```

In [5]:
```
# Test cell: `ex0_get_captions` (1 point)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.
""")

### BEGIN HIDDEN TESTS
def ex0_get_captions__soln__(captions_json):
    return [c['primary_description'] for c in captions_json]

def ex0_gen_soln__(captions_json, outfilename):
    from os.path import isfile
    from problem_utils import make_hash
    if not isfile(get_path(outfilename)):
        print(f"Generating solutions file, '{outfilename}'...")
        with open(get_path(outfilename), 'wt') as fp:
            solns = ex0_get_captions__soln__(captions_json)
            for c in solns:
                c_hashed = make_hash(c)
                fp.write(c_hashed + '\n')
    else:
        print(f"An existing hashed-solutions file, '{outfilename}', is available.")

ex0_gen_soln__(captions_json, 'toast/ex0_soln.csv')
### END HIDDEN TESTS

def ex0_check__(captions):
    from problem_utils import make_hash, check_hash
    with open(get_path('toast/ex0_soln.csv'), 'rt') as fp:
        solns = [s.strip() for s in fp.readlines()]
        if len(captions) != len(solns):
            print(f"Your solution has {len(captions)} captions, whereas we are expecting {len(solns)}!")
        for k, (your_caption, hashed_soln) in enumerate(zip(captions, solns)):
            assert check_hash(your_caption, hashed_soln), \
                f"*** Your caption {k}, '{your_caption}', does not match our expected solution. ***\n" \
                f"==> Expected hash value: {hashed_soln}"

ex0_check__(captions_orig)

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.

An existing hashed-solutions file, 'toast/ex0_soln.csv', is available.

(Passed!)
```

Words. Given a caption, we want to analyze just the words that appear in the caption.

When moving through a string from left-to-right, define each *word* to be the longest contiguous sequence of alphabetic characters (or just *letters*), *including up to one apostrophe if that apostrophe is sandwiched between two letters*. (An apostrophe is a single quote, `'`.)

For instance, consider the string caption,

```
"I'm sorry, sir, but this is a 'gluten-free' restaurant. We don't serve bread."
```

Its words are: `"I'm"`, `"sorry"`, `"sir"`, `"but"`, `"this"`, `"is"`, `"a"`, `"gluten"`, `"free"`, `"restaurant"`, `"we"`, `"don't"`, `"serve"`, and `"bread"`. Notice that `"I'm"` and `"don't"` *are* words, since they include one apostrophe between two letters. However, observe that the substring, `"'gluten-free'"` becomes `"gluten"` and `"free"`: the hyphen (`"-"`) is treated as a separator, and the leading and trailing apostrophes do **not** become part of the word.

> *Hint*: Heed the definition, "*... including **up to one** apostrophe if that apostrophe is **sandwiched between two letters***." Consider these two examples:
>
> ```
> clean("'tricky''case'xyz 1") == ['tricky', "case'xyz"]
>   clean("'tricky''case'xyz'two") == ['tricky', "case'xyz", "two"]
> ```

Exercise 1 (3 points). Given a string caption, `s`, complete the function, `clean(s)`, so that it does the following:

1. Converts `s` to lowercase.
2. Returns a list of the words in `s`, defined as above.

The list returned must contain the words in the order that they appear in the sentence.

For example:

```
assert clean("Please sir, that's obviously a clip-on.") \
       == ['please', 'sir', "that's", 'obviously', 'a', 'clip', 'on']
assert clean("I'm sorry, sir, but this is a 'gluten-free' restaurant. We don't serve bread.") \
       == ["i'm", 'sorry', 'sir', 'but', 'this', 'is', 'a', 'gluten', 'free', 'restaurant', 'we', "don't", 'serve', 'bre
ad']
```

> *Hint*: While its use is not required, regular expression processing is a good tool for this problem.

In [6]:
```
def clean(s):
    ### BEGIN SOLUTION
    from re import finditer
    pattern = r"[a-z]+('[a-z])?[a-z]*"
    return [match.group(0) for match in finditer(pattern, s.lower())]
    ### END SOLUTION
```

In [7]:
```
# Test cell: `ex1_clean_a` (0.5 points)

# Caption 123:
assert clean("Please sir, that's obviously a clip-on.") \
       == ['please', 'sir', "that's", 'obviously', 'a', 'clip', 'on']

# Caption 314:
assert clean("I'm sorry, sir, but this is a 'gluten-free' restaurant. We don't serve bread.") \
       == ["i'm", 'sorry', 'sir', 'but', 'this', 'is', 'a', 'gluten', 'free', 'restaurant', 'we', "don't", 'serve',
'bread']

# Some other tricky cases!
assert clean("'tricky''case'xyz 1") == ['tricky', "case'xyz"]
assert clean("'tricky''case'xyz'two") == ['tricky', "case'xyz", "two"]

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

```
(Passed!)
```

In [8]:
```
# Test cell: `ex1_clean_b` (2.5 points)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.
""")

### BEGIN HIDDEN TESTS
def ex1_clean__soln__(s):
    from re import finditer
    pattern = r"[a-z]+('[a-z])?[a-z]*"
    return [match.group(0) for match in finditer(pattern, s.lower())]

def ex1_gen_solns__(captions, outfilename="toast/captions_cleaned.txt"):
    from os.path import isfile
    from problem_utils import get_path
    if not isfile(get_path(outfilename)):
        print(f"'{outfilename}' does not yet exist; generating...")
        with open(outfilename, "wt") as fp:
            for c in captions:
                fp.write(' '.join(ex1_clean__soln__(c)) + "\n")
    else:
        print(f"'{outfilename}' already exists!")

ex1_gen_solns__(captions_orig)
### END HIDDEN TESTS

def ex1_gen_case__(n, m, s):
    assert n >= 1 and m >= 1 and s >= 1
    from random import randint, random, choice
```

```
    def rand_char(): return chr(ord('a') + randint(0, 25))
    def rand_upper(c): return c.upper() if random() <= 0.1 else c
    def rand_chars():
        m0 = randint(1, m)
        return ''.join([rand_upper(rand_char()) for _ in range(m0)])
    def rand_word():
        a = rand_chars()
        if random() <= 0.1:
            a += "'" + rand_chars()
        return a
    def rand_spaces(): return ''.join([choice(' \t\n') for _ in range(randint(1, s))])
    def rand_quote(): return "'" if random() <= 0.1 else ""
    def rand_punc(): return choice(r'`~!@#$%^&*()-_=+{[]}|;:"<>,.?/\'') if random() <= 0.1 else ""
    soln = []
    for k in range(randint(1, n)):
        if len(soln) == 0:
            x = rand_spaces() if random() <= 0.2 else rand_quote()
        else:
            x += rand_spaces() + rand_quote()
        w = rand_word()
        soln.append(w.lower())
        x += w
        x += rand_quote() + rand_punc()
    return x, soln

def ex1_check_case__(n, m, s, verbose=False):
    x, soln = ex1_gen_case__(n, m, s)
    your_soln = clean(x)
    matches = (your_soln == soln)
    if verbose or (not matches):
        print("- Input:", repr(x))
        print("- Expected solution:", soln)
        print("- Your solution:", your_soln)
        assert matches, "*** Failed on a random test case: see above ***"

print("Testing 100 randomly generated cases...")
for _ in range(100):
    ex1_check_case__(10, 8, 3)

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.

'toast/captions_cleaned.txt' already exists!
Testing 100 randomly generated cases...

(Passed!)
```

Cleaned captions. In case you can't get your function above working, we have prepared a file of pre-cleaned captions. Below, the variable `captions_clean` is a list of *strings*, with the "clean" tokens separated by a single space, making them easy to split back into words. The subsequent exercises will use this variable (so don't modify it!).

In [9]:
```
with open(get_path('toast/captions_cleaned.txt')) as fp:
    captions_clean = [c.strip() for c in fp.readlines()]
print(repr(captions_clean[123]))
print(repr(captions_clean[314]))
```

```
"please sir that's obviously a clip on"
"i'm sorry sir but the toastmasters meetings are on tuesday"
```

Captions as "bags of words"

Next, let's convert the captions into a "standard form," to help simplify our subsequent analysis.

Stop words. First, not all words in a caption carry meaning. For example, the word `"the"` occurs commonly, but let's assume[†] that it isn't useful. In the area of natural language processing, such words are called *stop words*.

Run the code cell below, which will create a list of stop words in an object named `stopwords`. Observe its type and contents.

[†] This assumption may or may not be reasonable in the case of jokes, where these words can sometimes completely change the nuance of a joke. But for this problem, let's just go with it.

In [10]:
```
from stopwords import stopwords

print(type(stopwords), ":", stopwords)

print("\nExamples:")
for ex in ["shouldn't", "toast"]:
    is_or_isnt = "is" + ("" if ex in stopwords else " *not*")
    print(f'- "{ex}"', is_or_isnt, "a stop word.")
```

```
<class 'set'> : {'where', 'than', 'should', 'those', 'we', 'him', 'so', 'by', 'a', 'or', 'when', 'doing', 'ours',
"don't", 'further', 'her', 'having', 'being', 'through', "you've", 'you', 'before', 'up', 'only', 'she', 'whom',
'few', "wouldn't", 'yourselves', "they're", "he's", 'they', "it's", "doesn't", "i've", "he'll", 'then', 'below',
'nor', 'itself', 'most', 'until', 'what', 'theirs', 'but', 'hers', 'the', 'yours', 'who', 'his', 'there', 'themsel
ves', "i'll", 'were', 'not', 'any', 'down', "i'm", 'their', "they'll", 'over', 'while', 'have', 'each', "we'll",
'them', 'had', "shan't", 'other', 'off', "who's", 'is', "weren't", 'did', 'to', 'yourself', 'during', 'such', 'int
o', 'these', 'could', "we're", "why's", 'again', "you'd", "they've", "we'd", 'me', "where's", 'myself', 'on', 'a
n', "you'll", "she'd", 'in', 'out', "she'll", 'because', 'ought', 'why', 'was', 'at', "what's", 'once', "when's",
"they'd", 'this', "isn't", "here's", "you're", 'of', 'are', 'after', 'been', "how's", "won't", 'which', 'same', 'a
nd', 'ourselves', 'very', "mustn't", 'he', 'does', 'if', 'that', "hadn't", "aren't", "there's", 'himself', "le
t's", 'would', 'i', 'be', 'some', 'both', 'above', 'from', 'as', 'my', 'all', "wasn't", "that's", 'am', 'here', 'i
ts', 'has', "hasn't", "shouldn't", 'own', 'between', 'too', "haven't", 'do', "she's", 'under', 'about', 'no', "did
n't", "can't", 'for', 'how', 'cannot', "i'd", 'with', "couldn't", 'your', 'it', 'against', "he'd", 'our', "we've",
'more', 'herself'}

Examples:
- "shouldn't" is a stop word.
- "toast" is *not* a stop word.
```

The "bag of words" model. Let `c` be a **cleaned** caption. That is, assume it is *already* cleaned per Exercise 1 above. Its bag-of-words representation is the *set* of its words that are **not** stop words.

For example, if `c` is the cleaned caption,

```
c == "you're in luck a slot for you just opened up in our kitchen"
```

then its bag of words representation is a Python set of the form,

```
{'just', 'kitchen', 'luck', 'opened', 'slot'}
```

Notice that the words, `"you're"`, `"in"`, `"a"`, `"for"`, `"you"`, `"up"`, and `"our"` are all omitted from the bag of words representation, since they are all stop words. Also, since `c` is clean, it is easy to identify words by using simple spaces as delimiters.

> The bag-of-words representation is a *set*. Therefore, even if a caption repeats a non-stop word, that word will appear at most once "in the bag."

Exercise 2 (2 points). Implement the function, `bag_of_words(c)`, so that it returns the bag-of-words representation of `c` as a Python set.

For example:

```
assert bag_of_words("you're in luck a slot for you just opened up in our kitchen") \
           == {'just', 'kitchen', 'luck', 'opened', 'slot'}
    assert bag_of_words("i'm sorry the toastmasters gala is for members only") \
           == {'gala', 'members', 'sorry', 'toastmasters'}
```

In [11]:
```
def bag_of_words(c):
    # Assume `c` is already clean per Exercise 1
    ### BEGIN SOLUTION
    return {w for w in c.split() if w not in stopwords}
    ### END SOLUTION
```

In [12]:
```
# Demo of your function:
print(bag_of_words("you're in luck a slot for you just opened up in our kitchen"))
print(bag_of_words("i'm sorry the toastmasters gala is for members only"))
```

```
{'slot', 'just', 'kitchen', 'luck', 'opened'}
{'toastmasters', 'members', 'sorry', 'gala'}
```

In [13]:
```
# Test cell: `ex2_bag_of_words` (2 points)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, and uses randomly generated
```

```
test cases to evaluate your implementation.
""")

### BEGIN HIDDEN TESTS
def ex2_bag_of_words_soln__(c):
    return {w for w in c.split() if w not in stopwords}

def ex2_gen_solns__(captions, outfilename="toast/caption_bags.pickle"):
    from os.path import isfile
    from problem_utils import get_path
    from pickle import dump
    if not isfile(get_path(outfilename)):
        print(f"'{outfilename}' does not yet exist; generating...")
        with open(outfilename, "wb") as fp:
            bags = [ex2_bag_of_words_soln__(c) for c in captions]
            dump(bags, fp)
    else:
        print(f"'{outfilename}' already exists!")

ex2_gen_solns__(captions_clean)
### END HIDDEN TESTS

def ex2_gen_case__(n, m):
    assert n >= 1 and m >= 1
    from random import randint, random, choice
    def rand_char(): return chr(ord('a') + randint(0, 25))
    def rand_chars():
        m0 = randint(1, m)
        return ''.join([rand_char() for _ in range(m0)])
    def rand_word():
        if random() <= 0.2:
            return choice(list(stopwords))
        a = rand_chars()
        if random() <= 0.1:
            a += "'" + rand_chars()
        return a
    soln = set()
    x = ''
    for k in range(randint(1, n)):
        if len(x) > 0:
            x += ' '
        w = rand_word()
        if w not in stopwords:
            soln = soln | {w}
        x += w
    return x, soln

def ex2_check_case__(n, m, verbose=False):
    x, soln = ex2_gen_case__(n, m)
    your_soln = bag_of_words(x)
    matches = (your_soln == soln)
    if verbose or (not matches):
        print("- Input:", repr(x))
        print("- Expected solution:", soln)
        print("- Your solution:", your_soln)
        print("- Symmetric difference:", soln ^ your_soln)
        assert matches, "*** Failed on a random test case: see above ***"

for _ in range(100):
    ex2_check_case__(10, 8)

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, and uses randomly generated
test cases to evaluate your implementation.

'toast/caption_bags.pickle' already exists!

(Passed!)
```

Calculating all bags. In case your `bag_of_words()` is not working, we've pre-computed the bags for all cleaned captions. The code cell below will load these results from a file into a variable called, `caption_bags`, which is a list of bags. The subsequent exercises will use it (so don't modify it!).

In [14]:
```
def load_bags(infilename="toast/caption_bags.pickle"):
    from pickle import load
    full_path = get_path(infilename)
    print(f"Loading caption bags from '{full_path}'...")
```

```
    with open(full_path, "rb") as fp:
        bags = load(fp)
    return bags

caption_bags = load_bags()
print("- Bag for caption 123:", caption_bags[123])
print("- Bag for caption 314:", caption_bags[314])
```

```
Loading caption bags from './resource/asnlib/publicdata/toast/caption_bags.pickle'...
- Bag for caption 123: {'sir', 'obviously', 'clip', 'please'}
- Bag for caption 314: {'sir', 'tuesday', 'sorry', 'toastmasters', 'meetings'}
```

`Counter` objects

The Python `collections` module defines a handy object called a `Counter` that can help you count the how many times a value occurs in any iterable collection. It's like a histogram, and it produces a dictionary-like result.

To see how it works, read and run this example:

In [15]:
```
from collections import Counter

# Three example sets:
A = {'cat', 'hat', 'fish', 'red', 'blue'}
B = {'dog', 'beret', 'one', 'fish', 'two', 'red', 'blue'}
C = {'dog', 'cat', 'fish'}

# Count value occurrences in `A`, `B`, and `C`
K_A = Counter(A)
K_B = Counter(B)
K_C = Counter(C)

print(K_A, "==>", K_A['fish'], K_A['dog'])
print(K_B, "==>", K_B['fish'], K_B['dog'])
print(K_C, "==>", K_C['fish'], K_C['dog'])

# Combine occurrence counts by simple addition!
K_D = K_A + K_B + K_C
print(K_D, "==>", K_D['fish'], K_D['dog'])
```

```
Counter({'fish': 1, 'cat': 1, 'hat': 1, 'blue': 1, 'red': 1}) ==> 1 0
Counter({'dog': 1, 'one': 1, 'fish': 1, 'two': 1, 'blue': 1, 'beret': 1, 'red': 1}) ==> 1 1
Counter({'dog': 1, 'fish': 1, 'cat': 1}) ==> 1 1
Counter({'fish': 3, 'cat': 2, 'blue': 2, 'red': 2, 'dog': 2, 'hat': 1, 'one': 1, 'two': 1, 'beret': 1}) ==> 3 2
```

Observe how each application of `Counter(X)` produces a dictionary-like object containing the number of occurrences of each value in `X`. Moreover, you can add two `Counter()` objects to combine their counts -- neat!

Exercise 3 (2 points). Suppose you are given a list `bags` containing caption bags, and you want to know how many bags contain a given word. We'll call the number of bags that contain a given word that word's *frequency*. Write a function, `count_freq(bags)`, that returns a `Counter()` object containing word frequencies for all words.

Here is an example:

```
bags = [{'cat', 'hat', 'fish', 'red', 'blue'},
        {'dog', 'beret', 'one', 'fish', 'two', 'red', 'blue'},
        {'dog', 'cat', 'fish'}]
print(count_freq(bags))
```

would produce

```
Counter({'fish': 3, 'cat': 2, 'blue': 2, 'red': 2, 'dog': 2, 'hat': 1, 'one': 1, 'two': 1, 'beret': 1})
```

> There are two demo cells, below. The first one helps you debug; the second one runs your function to generate the word frequencies across all the actual caption bags. The test cell will check it, and subsequent exercises will use it.

In [16]:
```
def count_freq(bags):
    ### BEGIN SOLUTION
    return count_freq__0(bags)

def count_freq__0(bags): # Solution method 0
    K = Counter()
    for b in bags:
```

```
    for b in bags:
        K += Counter(b)
    return K

def count_freq__1(bags): # Method 1: Looks fancy, but is slow
    from functools import reduce
    def add(a, b):
        return a + b
    return reduce(add, [Counter(b) for b in bags])

def count_freq__err(bags): # For testing the test code -- how meta
    K = count_freq__0(bags)
    from random import choices, randrange, random
    k_delete = set(choices(list(K.keys()), k=randrange(3)))
    print("*** Deleting:", k_delete)
    k_fudge = set(choices(list(K.keys()), k=randrange(3))) - k_delete
    print("*** Fudging:", k_fudge)
    for k in k_delete:
        del K[k]
    for k in k_fudge:
        K[k] += randrange(1, 3) * (-1 if random() < 0.5 else 1)
    return K
    ### END SOLUTION
```

In [17]:
```
# Demo 0:
bags = [{'cat', 'hat', 'fish', 'red', 'blue'},
        {'dog', 'beret', 'one', 'fish', 'two', 'red', 'blue'},
        {'dog', 'cat', 'fish'}]
print(count_freq(bags))
```

```
Counter({'fish': 3, 'cat': 2, 'blue': 2, 'red': 2, 'dog': 2, 'hat': 1, 'one': 1, 'two': 1, 'beret': 1})
```

In [18]:
```
# Demo 1: Use your function on the caption bags!

word_freq = count_freq(caption_bags)

print("Top 5 (word, frequency) pairs:")
print(word_freq.most_common(5))

print("\nLeast frequent words:")
print(word_freq.most_common()[-5::])
```

```
Top 5 (word, frequency) pairs:
[('sorry', 819), ('sir', 509), ('toast', 451), ('bread', 223), ('gluten', 193)]

Least frequent words:
[('allergies', 1), ('fancy', 1), ('odd', 1), ('millenials', 1), ('y', 1)]
```

In [19]:
```
# Test cell: `ex3_count_words` (2 points)

def ex3_gen_bag__(m, vocab, counts):
    from random import randint, choices
    m0 = randint(1, m)
    words = set(choices(list(vocab), k=m0))
    for w in words:
        if w not in counts:
            counts[w] = 0
        counts[w] += 1
    return words

def ex3_check__(n, m, vocab):
    from random import randint
    from collections import Counter
    soln = {}
    bags = []
    for _ in range(randint(1, n)):
        bags.append(ex3_gen_bag__(m, vocab, soln))
    your_soln = count_freq(bags)
    assert isinstance(your_soln, Counter), \
           f"Your function returns an object of type `{type(your_soln)}` rather than a `Counter`."
    soln_keys = set(soln.keys())
    your_keys = set(your_soln.keys())
    detected_error = (your_keys != soln_keys)
    if detected_error:
        print("*** Failed test case ***")
        missing_keys = (soln_keys - your_keys)
        if missing_keys:
            print(f"- Your returned `Counter` is missing these keys:\n{missing_keys}")
        extra_keys = (your_keys - soln_keys)
        if extra_keys:
```

```
            print(f"- Your returned `Counter` has extra keys:\n{extra_keys}")
    common_keys = (soln_keys & your_keys)
    for key in common_keys:
        your_value = your_soln[key]
        true_value = soln[key]
        if your_value != true_value:
            print(f"- Count mismatch on key {repr(key)}: Your value is {your_value} vs. true value of {true_value}
.")
            detected_error = True
    if detected_error:
        print("\n==> Problematic test input:\n")
        print('[' + ',\n'.join([repr(b) for b in bags]) + ']')
        assert False, "*** Please see above for errors. ***"

def ex3_check_bunch__(k):
    W = {'bottle', 'stumptown', 'la', 'ethical', 'vhs', 'tousled', 'synth', 'diy', 'deep', 'table', 'banh', 'batch'
, 'farm', 'knausgaard', 'listicle', 'lorem', 'poutine', 'flannel', 'croix', 'lyft', 'marfa', 'fund', 'franzen', 'ty
pewriter', 'forage', 'fashion', 'ipsum', 'amet', 'sriracha', 'mi', 'tumeric', 'dolor', 'pinterest', 'art', 'scenest
er', 'whatever', 'street', 'axe', 'small', 'mustache', 'trust', 'level', 'snackwave', 'shaman', 'blue', 'next', 'ha
mmock'}
    for _ in range(k):
        ex3_check__(20, 5, W)

ex3_check_bunch__(10)

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

```
(Passed!)
```

Scoring captions

In the final part of this problem, let's try to identify "interesting" or "relevant" captions by assigning each one a score, using the bag-of-words model and the frequencies defined above.

Idea 0: Frequency-proportional scores. One natural idea is to score bags more highly if they contain more frequent words. In particular, suppose we simply add up the frequencies of all words in the bag and use that as the bag's score.

For instance, suppose our word frequences are:

```
Counter({'fish': 3, 'cat': 2, 'blue': 2, 'red': 2, 'dog': 2, 'hat': 1, 'one': 1, 'two': 1, 'beret': 1})
```

Then the bag,

```
{'blue', 'cat', 'fish'}
```

has a score of 2+2+3 = 7. By contrast, the bag,

```
{'one', 'two', 'beret', 'hat', 'dog'}
```

has a score of 1+1+1+1+2 = 6. Even though the second bag has more words, the first bag has more frequent words, and the sum of its frequencies happens to be higher.

Here is an implementation of this idea as a score. The function takes as input one bag (a Python `set()`) and a frequency table (as a `Counter()` object), and returns the score described above.

Be sure you understand its implementation, which should be simple, before moving on!

In [20]:
```
def score_bag_common(bag, freq):
    return sum([freq[word] for word in bag])

K = Counter({'fish': 3, 'cat': 2, 'blue': 2, 'red': 2, 'dog': 2, 'hat': 1, 'one': 1, 'two': 1, 'beret': 1})
print(score_bag_common({'blue', 'cat', 'fish'}, K))
print(score_bag_common({'one', 'two', 'beret', 'hat', 'dog'}, K))
```

```
7
6
```

Let's see what the "top 10 captions" are if we use this score, by running the code cell below.

In [21]:
```
def score_all(bags, scoring_fun, freq):
    return [scoring_fun(bag, freq) for bag in bags]

def get_order(scores, max_rank=None):
```

```
    if max_rank is None:
        max_rank = len(scores)
    # https://stackoverflow.com/questions/7851077/how-to-return-index-of-a-sorted-list
    return sorted(range(len(scores)), key=lambda k: scores[k], reverse=True)[:max_rank]

def print_top_captions(scores, captions, max_rank=10):
    for rank, k in enumerate(get_order(scores, max_rank)):
        print(f"{rank+1} [{scores[k]:.3f}]: '{k}. {captions[k]}'")

scores_common = score_all(caption_bags, score_bag_common, word_freq)
print_top_captions(scores_common, captions_orig, max_rank=5)
```

```
1 [2252.000]: '1440. I'm sorry sir, we cannot seat you until the rest of your party arrives. Without avocado, yo
u're just toast.'
2 [2134.000]: '1481. I'm sorry, sir, but this is a gluten-free restaurant. We don't serve bread.'
3 [2099.000]: '1778. I'm very sorry, sir, but we only serve gluten free bread in this establishment.'
4 [2094.000]: '808. I'm sorry, sir, but even being the toast of the town won't get you a table tonight.'
5 [2061.000]: '134. I'm sorry sir, the only reservation I have under "Toast" checked in 15 minutes ago with a pala
te of butter and some strawberry jam.'
```

Idea 1: Maximize "surprise." The captions above are all relevant, which is good, but they are also very similar. They lack the "surprise" that can make a caption funnier.

So, a different idea is to reward rare words. Here is a simple implementation of this idea, and its results. It *inverts* the frequency of each word in the bag, and sums those.

In [22]:
```
def score_bag_rare(b, freq):
    return sum([1/freq[w] for w in b])

scores_rare = score_all(caption_bags, score_bag_rare, word_freq)
print_top_captions(scores_rare, captions_orig, max_rank=5)
```

```
1 [9.372]: '1732. Party of one... Woah, I think I finally understand super-symmetry, the duality of infinity, and
the entire universe... What? No, they're not putting LSD in the sauce or anything, in case that's what you were wo
ndering, Mr. what did you say it was?'
2 [8.746]: '1690. And now, to close the twenty-fourth annual meeting of the "Literalists Society", the dry crisp s
tylings of our Toastmaster Emeritus, Toasty McToast.'
3 [8.658]: '2385. I'm sorry, this is a restaurant. We don't arrange “murder the homeless” Most Dangerous Game isla
nd getaways for sentient slices of bread. I can't begin to imagine why you thought I could do that for you.'
4 [8.656]: '212. I would like to practice my speaking skills with a sad story about my family being roasted alive
and eaten right in front of me.  I will try not to cry so I can avoid getting soggy and moldy.'
5 [8.381]: '461. Okay, so here's how it's going to go. Forty-seven people will write captions that are some varian
t on the most obvious choice, and one of them -- perhaps at random -- will be chosen the "winner."  I'm sorry, I w
ish I had better news.'
```

What the...? If you are like most people, you find these captions to be a bit too surprising. There are two issues:

1. Using overly rare words appears to lead to irrelevant captions.
2. Having scores sum all words in the bag tends to reward captions merely because they are longer.

Exercise 4 (2 points). Let's try to implement a score that rewards a mix of common and surprising words, along with captions that are both not too short and not too long.

In particular, here is what your colleague has suggested.

- Suppose a bag has m words.
- Suppose these m words have frequencies $f = [f_0, f_1, \ldots, f_{m-1}]$.
- Let $\frac{1}{f} \equiv \left[\frac{1}{f_0}, \frac{1}{f_1}, \ldots, \frac{1}{f_{m-1}}\right]$ denote the inverse of these frequencies.

From these, let $s(f)$ be the score of the bag, defined using the following two-part formula:

$$s(f) = \mathrm{pstdev}\left(\frac{1}{f}\right) \cdot w(m)$$

$$w(m) = \frac{m}{\mu}\exp\left(-\frac{|m-\mu|}{\mu}\right),$$

where $\mathrm{pstdev}(1/f)$ is the *(population) standard deviation* of the inverse-frequencies, and μ is a constant. Note the use of an absolute value in the definition of $w(m)$.

Here is your colleague's reasoning.

- The standard deviation rewards bags that have highly variable frequencies, i.e., a diverse mix of rare and common words.
- The $w(m)$ factor penalizes overly short or overly long sentences. It equals 1 when $m = \mu$, which you can interpret as a "target" caption length. It is less than 1 for all other values. (The function $\exp(x)$ is e^x, where e is the base of the natural logarithm.)

less than 1 for all other values. (The function $\exp(x)$ is e^x, where e is the base of the natural logarithm.)

Translate your colleague's formula into code, implementing it as `score_bag_mixed(bag, freq, mu)`, below, where

- `bag` is the bag of a caption;
- `freq` is a `Counter()` object, where `freq[w]` is the frequency of word `w` for any `w` in `bag`;
- `mu` is the target caption length, whose default value is 5.

> **Hint 0:** The code cell imports a functions to compute the population standard deviation (`statistics.pstdev`) and exponential.
>
> **Hint 1:** The population standard deviation is only well-defined when computed on at least one value. So if therre are no values, then let $\text{pstdev}(\cdot) = 0$.

In [23]:
```
from statistics import pstdev # https://docs.python.org/3/library/statistics.html#statistics.stdev
from math import exp

def score_bag_mixed(bag, freq, mu=5):
    ### BEGIN SOLUTION
    m = len(bag)
    w = m/mu * exp(- abs(m-mu) / mu)
    x = [1/freq[word] for word in bag]
    s = pstdev(x) if len(x) > 0 else 0.0
    from random import random
    return w * s
    ### END SOLUTION
```

In [24]:
```
# Demo:
scores_mixed = score_all(caption_bags, score_bag_mixed, word_freq)
print_top_captions(scores_mixed, captions_orig)

# If you are on the right track, the top three captions should be:
#
# 1 [0.488]: '297. I'm sorry, but this is the gluten-free keynote address.'
# 2 [0.485]: '1575. I'm afraid it's one of those modern weddings and there won't be a toast.'
# 3 [0.484]: '796. Sorry,Captain Picard's dog's name is not "Toast".'
```

```
1 [0.488]: '297. I'm sorry, but this is the gluten-free keynote address.'
2 [0.485]: '1575. I'm afraid it's one of those modern weddings and there won't be a toast.'
3 [0.484]: '796. Sorry,Captain Picard's dog's name is not "Toast".'
4 [0.484]: '1805. Sir, I think you misunderstand what the Toastmasters organization is all about...'
5 [0.482]: '912. Sorry. Toastmasters met last Monday.'
6 [0.481]: '587. Now I don't wanna see you crumble under pressure'
7 [0.480]: '497. Mr. Peanut?  I'm so sorry.  He recently donated all he had.'
8 [0.478]: '629. I can squeeze you in if you take off the feet, the arms and the tie.'
9 [0.478]: '606. You lied about your experience in the service industry?! Oh, you're toast.'
10 [0.477]: '1305. Mr. SquarePants, please spell "SpongBob."'
```

In [25]:
```
# Test cell: `ex4_score_bag_mixed` (2s points)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.
""")

### BEGIN HIDDEN TESTS
def ex4_score_bag_mixed__soln__(bag, freq, mu=5):
    from numpy import std
    m = len(bag)
    w = m/mu * exp(- abs(m-mu) / mu)
    x = [1/freq[word] for word in bag]
    s = std(x) if len(x) > 0 else 0.0
    return w * s

def ex4_gen_soln__(bags, freq, captions, outfilename):
    from os.path import isfile
    from problem_utils import make_hash
    if not isfile(get_path(outfilename)):
        print(f"Generating solutions file, '{outfilename}'...")
        with open(get_path(outfilename), 'wt') as fp:
            scores_mixed = score_all(bags, ex4_score_bag_mixed__soln__, freq)
            order = get_order(scores_mixed)
            for k in order:
                c = captions[k]
                c_hashed = make_hash(c)
                fp.write(c_hashed + '\n')
    else:
```

```
        print(f"An existing solutions file, '{outfilename}', is available.")

ex4_gen_soln__(caption_bags, word_freq, captions_orig, 'toast/ex4_soln.csv')
### END HIDDEN TESTS

def ex4_check__(scores, captions):
    from problem_utils import make_hash, check_hash
    with open(get_path('toast/ex4_soln.csv'), 'rt') as fp:
        print("==> Checking the top 100 captions...")
        solns = [s.strip() for s in fp.readlines()]
        order = get_order(scores, max_rank=100)
        for rank, k in enumerate(order):
            c_soln_hashed = solns[rank]
            c_your_soln = captions[k]
            assert check_hash(c_your_soln, c_soln_hashed), \
                f"*** Your solution at rank={rank}, {repr(c_your_soln)} (#{k}), does not match our expected solutio
n. ***\n" \
                f"==> Expected hash value: {repr(c_soln_hashed)}"

ex4_check__(scores_mixed, captions_orig)

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.

An existing solutions file, 'toast/ex4_soln.csv', is available.
==> Checking the top 100 captions...

(Passed!)
```

Epilogue. The previous exercise concludes this problem. In case you are curious, the top three user-submitted captions were:

- #1282: "*Yes, we seated a potato, but he had a jacket.*" (Richard Berman, Amherst, Massachusetts)
- #121: "*You're in luck. A slot just opened up for you in the kitchen.*" (Sean Kirk, Bellingham, Washington)
- #1126: "*I'm sorry, sir. We no longer serve bread.*" (Myles Gordon, Austerlitz, New York).

So, the automated methods here clearly miss the mark in some way. You'll learn many other potential methods throughout the MSA program!

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!

Problem 1: Ingredient substitutions

Problem 1: Ingredient substitutions

Version 1.2b

This problem is a data mining task that exercises basic data structure manipulation, Notebook 2 (pairwise association mining), and some simple linear algebra concepts (vectors, dot products).

- All exercises depend on a correct Exercise 0 (1 point).
- Exercise 1 is "standalone" (1 point). No subsequent exercises depend on it.
- Exercise 2 is "standalone" (2 points). No subsequent exercises depend on it.
- Exercises 3 (2 points) and 4 (2 points) are independent of each other.
- Exercise 5 (2 points) depends on Exercises 3 and 4.

> Exercise 5 can be challenging, and its test cells will only be efficient if you have reasonably efficient implementations of the pieces. When you submit to th autograder, there will be a 120 second (2 minute) time limit for your notebook. So, do keep in mind that it is only worth two (2) points and allocate your time accordingly.

Pro-tips.

- If your program behavior seem strange, try resetting the kernel and rerunning everything.
- If you mess up this notebook or just want to start from scratch, save copies of all your partial responses and use `Actions` → `Reset Assignment` to get a fresh, original copy of this notebook. (*Resetting will wipe out any answers you've written so far, so be sure to stash those somewhere safe if you intend to keep or reuse them!*)
- If you generate excessive output (e.g., from an ill-placed `print` statement) that causes the notebook to load slowly or not at all, use `Actions` → `Clear Notebook Output` to get a clean copy. The clean copy will retain your code but remove any generated output. **However**, it will also **rename** the notebook to `clean.xxx.ipynb`. Since the autograder expects a notebook file with the original name, you'll need to rename the clean notebook accordingly.

Good luck!

Your problem and goals

Suppose you are cooking and following a recipe, but you discover you are missing an ingredient. You don't have time to run to the store. What would be a valid substitute? Let's implement a scheme to make automatic suggestions using basic Python and the results of Notebook 2 (pairwise association mining).

A "high-level" idea

Here is an outline of a possible method. Suppose we have access to a large database of recipes. We'll start by looking for one or more recipes that are similar to ours. Then, we'll look at what ingredients they use that do **not** appear in our recipe. Among those candidate ingredients, we'll again try to find which ones are most similar to the one we are missing.

To make this work, we'll need to

- process the database (Exercises 0 and 1);
- define what "recipe-similarity" means (Exercise 2);
- define what "ingredient-similarity" means (Exercises 3 and 4);
- and then assemble these pieces (Exercise 5).

The exercises below will walk you through this process.

The recipes database

The dataset is a collection of almost 40,000 recipes. Run the code cell below to load it into a global variable named `recipes`.

In [1]:
```
import json
from problem_utils import get_path

with open(get_path("recipes/train.json"), "rt") as fp:
    recipes = json.load(fp)

print(f"==> The dataset contains {len(recipes)} recipes.")
print(f"    The variable `recipes` has type `{type(recipes)}`.")

print(f"\nThe first three elements are as follows:\n")
for k, r in enumerate(recipes[:3]):
    print(f"{k}: {r}\n")
```

```
==> The dataset contains 39774 recipes.
    The variable `recipes` has type `<class 'list'>`.

The first three elements are as follows:

0: {'id': 10259, 'cuisine': 'greek', 'ingredients': ['romaine lettuce', 'black olives', 'grape tomatoes', 'garlic', 'pepper', 'purple onion', 'seasoning', 'garbanzo beans', 'feta cheese crumbles']}

1: {'id': 25693, 'cuisine': 'southern_us', 'ingredients': ['plain flour', 'ground pepper', 'salt', 'tomatoes', 'ground black pepper', 'thyme', 'eggs', 'green tomatoes', 'yellow corn meal', 'milk', 'vegetable oil']}

2: {'id': 20130, 'cuisine': 'filipino', 'ingredients': ['eggs', 'pepper', 'salt', 'mayonaise', 'cooking oil', 'green chilies', 'grilled chicken breasts', 'garlic powder', 'yellow onion', 'soy sauce', 'butter', 'chicken livers']}
```

```
The first three elements are as follows:

0: {'id': 10259, 'cuisine': 'greek', 'ingredients': ['romaine lettuce', 'black olives', 'grape tomatoes', 'garli
c', 'pepper', 'purple onion', 'seasoning', 'garbanzo beans', 'feta cheese crumbles']}

1: {'id': 25693, 'cuisine': 'southern_us', 'ingredients': ['plain flour', 'ground pepper', 'salt', 'tomatoes', 'gr
ound black pepper', 'thyme', 'eggs', 'green tomatoes', 'yellow corn meal', 'milk', 'vegetable oil']}

2: {'id': 20130, 'cuisine': 'filipino', 'ingredients': ['eggs', 'pepper', 'salt', 'mayonaise', 'cooking oil', 'gre
en chilies', 'grilled chicken breasts', 'garlic powder', 'yellow onion', 'soy sauce', 'butter', 'chicken livers']}
```

Observe that `recipes` is a list. Each element of the list is a dictionary, which is a recipe represented by a unique integer ID, a cuisine type, and a list of its ingredients.

For this problem, we will largely ignore the `'id'` and `'cuisine'` keys, so let's write a quick function to help extract just the ingredients.

Exercise 0 (1 point). Let `recipe` be a single recipe from the `recipes` list. Complete the function, `get_ingredients(recipe)`, below, so that it returns the list of the ingredients in `recipe`.

For example:

```
assert get_ingredients(recipes[1]) == ['plain flour', 'ground pepper', 'salt', 'tomatoes', 'ground black pepper', 'thym
e', 'eggs', 'green tomatoes', 'yellow corn meal', 'milk', 'vegetable oil']
```

> The returned list should preserve the exact names and orders of ingredients as they appear in the input data.

In [2]:
```
def get_ingredients(recipe):
    ### BEGIN SOLUTION
    return recipe['ingredients']
    ### END SOLUTION

# Demo:
get_ingredients(recipes[1])
```

Out[2]:
```
['plain flour',
 'ground pepper',
 'salt',
 'tomatoes',
 'ground black pepper',
 'thyme',
 'eggs',
 'green tomatoes',
 'yellow corn meal',
 'milk',
 'vegetable oil']
```

In [3]:
```
# Test cell: `ex0_get_ingredients` (1 point)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.
""")

### BEGIN HIDDEN TESTS
def ex0_get_ingredients_soln__(recipe):
    return recipe['ingredients']

def ex0_gen_soln__(recipes, outfilename):
    from os.path import isfile
    from problem_utils import make_hash
    if not isfile(get_path(outfilename)):
        print(f"Generating solutions file, '{outfilename}'...")
        with open(get_path(outfilename), 'wt') as fp:
            for r in recipes:
                i = get_ingredients(r)
                i_hashed = make_hash(repr(i))
                fp.write(i_hashed + '\n')
    else:
        print(f"An existing hashed-solutions file, '{outfilename}', is available.")

ex0_gen_soln__(recipes, 'recipes/ex0_soln.csv')
### END HIDDEN TESTS
```

```
def ex0_check__(recipes):
    from problem_utils import check_hash
    with open(get_path('recipes/ex0_soln.csv'), 'rt') as fp:
        for k, r in enumerate(recipes):
            i_your_soln = get_ingredients(r)
            i_true_soln_hashed = fp.readline().strip()
            assert check_hash(repr(i_your_soln), i_true_soln_hashed), \
                   f"For recipe {r['id']} (`recipes[{k}]`), your result is {i_your_soln}, which does not match what
we expect."

ex0_check__(recipes)

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below, but the solution values
are masked using hashed values.

An existing hashed-solutions file, 'recipes/ex0_soln.csv', is available.

(Passed!)
```

Generalizing `make_itemsets()` from Notebook 2

Recall the `make_itemsets()` function from Notebook 2, Part 0, Exercise 3 (`nb2.0.3`). The sample solution was:

```
def make_itemsets(item_lists): # nb2.0.3, sample solution
    return [set(i) for i in item_lists]
```

Recall that from this definition, you could call this function on a list of two grocery baskets and obtain a list of itemsets as a result, e.g.,

```
assert make_itemsets([['milk', 'eggs', 'bread'], ['beer', 'eggs']]) \
       == [{'bread', 'eggs', 'milk'}, {'beer', 'eggs'}]
```

But suppose we wish to make itemsets from the ingredient lists given the `recipes` object. Calling `make_itemsets(recipes)` won't work! The ingredients list requires an extra step to extract the ingredients list, by applying the function `get_ingredients()` you defined in Exercise 0.

Your colleague suggests a common Python pattern: create a new function that can achieve this task, with the signature:

```
def make_itemsets_apply(item_lists, extractor=...):
    ...
```

This function accepts a second argument, named `extractor`, which can be any user-supplied **function** for getting the data from one input element to be converted into an itemset. For example, if you have an identity function,

```
def identity(x):
    return x
```

then `make_itemsets_apply(X, extractor=identity)` should behave the same as `make_itemsets(X)`. And if we implement it correctly, then we should be able to run it **directly** on the `recipes` database to get ingredient itemsets via a call like,

```
ingredient_sets = make_itemsets_apply(recipes, extractor=get_ingredients)
```

Exercise 1 (1 point). Complete the implementation of `make_itemsets_apply()` so that it behaves as described above. If implemented correctly, then

```
ingredient_sets = make_itemsets_apply(recipes, extractor=get_ingredients)
```

will produce a result such that

```
assert ingredient_sets[0] == {'pepper', 'feta cheese crumbles', 'garbanzo beans', 'grape tomatoes', 'black olives', 'gar
lic', 'romaine lettuce', 'seasoning', 'purple onion'}
assert ingredient_sets[1] == {'tomatoes', 'green tomatoes', 'ground pepper', 'eggs', 'vegetable oil', 'yellow corn meal'
, 'thyme', 'ground black pepper', 'plain flour', 'salt', 'milk'}
# ... and so on ...
```

In [4]:
```
def identity(x):  # a function that just returns the input
    return x

def make_itemsets_apply(item_lists, extractor=identity):
    ### BEGIN SOLUTION
    return [set(extractor(l)) for l in item_lists]
```

```
    return [set(extractor(i)) for i in item_lists]
    ### END SOLUTION
```

In [5]:
```
# Demo of your function:
def make_ingredient_sets(recipes):
    return make_itemsets_apply(recipes, extractor=get_ingredients)

ingredient_sets = make_ingredient_sets(recipes)
print(ingredient_sets[0])
print(ingredient_sets[1])
```

```
{'black olives', 'seasoning', 'pepper', 'purple onion', 'garbanzo beans', 'grape tomatoes', 'garlic', 'romaine let
tuce', 'feta cheese crumbles'}
{'ground pepper', 'salt', 'ground black pepper', 'thyme', 'yellow corn meal', 'tomatoes', 'vegetable oil', 'plain
flour', 'milk', 'green tomatoes', 'eggs'}
```

In [6]:
```
# Test cell: `ex1_make_itemsets_extract` (1 point)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below.
""")

### BEGIN HIDDEN TESTS
def ex1a_make_itemsets_apply_soln__(item_lists, extractor=lambda x: x):
    return [set(extractor(l)) for l in item_lists]

def ex1a_gen_soln__(recipes, outfilename):
    from os.path import isfile
    from pickle import dump
    if not isfile(get_path(outfilename)):
        I = ex1a_make_itemsets_apply_soln__(recipes, extractor=ex0_get_ingredients_soln__)
        print(f"Generating solutions file, '{outfilename}'...")
        with open(get_path(outfilename), 'wb') as fp:
            dump(I, fp)
    else:
        print(f"An existing solutions file, '{outfilename}', is available.")

ex1a_gen_soln__(recipes, 'recipes/ex1a_soln.pickle')
### END HIDDEN TESTS

def ex1b_general_extractor__(obj, field):
    return obj[field]

def ex1b_gen_rand_keys__(max_keys):
    from random import randrange, choices
    num_keys = randrange(1, max_keys)
    return set([''.join(choices('abcdefghijklmnopqrstuvwxyz', k=5)) for _ in range(num_keys)])

def ex1b_gen_rand_dict_vals__(keys, key0, max_vals):
    from random import randrange
    D = {}
    R = None
    for k in keys:
        num_vals = randrange(1, max_vals)
        D[k] = [randrange(-100, 100) for _ in range(num_vals)]
        if k == key0:
            R = set(D[k])
    assert R is not None
    return D, R

def ex1b_check_one__(max_keys, max_len, verbose=True):
    from random import randrange, choice
    keys = ex1b_gen_rand_keys__(max_keys)
    key0 = choice(list(keys))
    len_item_lists = randrange(10)
    item_lists = []
    R_true = []
    for _ in range(len_item_lists):
        D, R = ex1b_gen_rand_dict_vals__(keys, key0, max_len)
        item_lists.append(D)
        R_true.append(R)
    print(f"""
=== Test case ===

* item_lists == {item_lists}

* Expected result when extracting key '{key0}' == {R_true}""")
    extractor__ = lambda x: ex1b_general_extractor__(x, key0)
    R_you = make_itemsets_apply(item_lists, extractor=extractor__)
```

```
    assert R_you == R_true, f"*** Failed ***\nYour result when extracting key '{key0}': {R_you}"

for _ in range(10): # Ten randomly generated test cases
    ex1b_check_one__(5, 10)

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below.

An existing solutions file, 'recipes/ex1a_soln.pickle', is available.

=== Test case ===

* item_lists == [{'fjwen': [-23, 74, 72, 26, 2, -65, -86], 'hqdsw': [7, -93, -92, 75, 75, -60, 54, 76, -79], 'epyp
d': [52, -41, 39]}, {'fjwen': [4, -1], 'hqdsw': [23], 'epypd': [-14, 30, -37, -28, -77, -55, -66, -46, -30]}, {'fj
wen': [-39, 17, 99, -55, 3, -65, -17, 12], 'hqdsw': [21, 88, 87, 11, 93, -92, 1], 'epypd': [14, 58, 43, -86, -10,
-97]}, {'fjwen': [-53, 5, -33, 32], 'hqdsw': [25, -81, 25, -89, 69, 33, 91, -51, -25], 'epypd': [-7, 64, -48, -71,
67]}, {'fjwen': [83, 85], 'hqdsw': [-37, -11, 36], 'epypd': [-62, -38, 82, -85, -85, -67, 72]}, {'fjwen': [-86, 4
1, 53, 0], 'hqdsw': [-27, -53, -45], 'epypd': [-39]}]

* Expected result when extracting key 'hqdsw' == [{-93, -92, -60, 7, 75, 76, -79, 54}, {23}, {1, -92, 11, 21, 87,
88, 93}, {33, 69, -89, -25, -51, -81, 25, 91}, {-37, 36, -11}, {-53, -27, -45}]

=== Test case ===

* item_lists == [{'kheqt': [-51, -34, -49, 58, 52, 76, -35], 'wzznr': [-32, 61, -90, 26, 77, 2, 33, 98, 57], 'kdls
h': [-14, -20, 31, 93, 61, 73, -12, 82], 'tdwsw': [45, 66]}]

* Expected result when extracting key 'kheqt' == [{76, -51, -49, 52, 58, -35, -34}]

=== Test case ===

* item_lists == [{'avqne': [66, 32, -57, 14, -45], 'qhyjq': [44, -19, -5, -63, 88, 21, 45, 7]}, {'avqne': [-84, 8
9, 92, 53, -9], 'qhyjq': [-69, -17, -92, -74, -47, 21, 41, 52]}, {'avqne': [67, 91, -11, 20, 66, -8, -79, 74, 65],
'qhyjq': [70, -14, 2, -61, -32, -24]}]

* Expected result when extracting key 'qhyjq' == [{-63, 7, 44, -19, 45, 21, 88, -5}, {-92, 41, -17, -47, 52, 21, -
74, -69}, {-32, 2, -61, 70, -24, -14}]

=== Test case ===

* item_lists == []

* Expected result when extracting key 'jbopk' == []

=== Test case ===

* item_lists == []

* Expected result when extracting key 'bksko' == []

=== Test case ===

* item_lists == [{'pomfx': [-24, -19, -41, 82, -68, 36, 15, -14], 'eddvg': [97, 54, 51, 69, -47, -10, -91, -45],
'ncmzd': [62, -3, 49, 45, -92, -4, 55, -97, -20], 'tdatz': [-97, 25, 52, 19, -37, 65, -46, -28]}, {'pomfx': [-47,
46, -46], 'eddvg': [-89, -87], 'ncmzd': [41, -87, -87, -17, -8, -48, -52, -78], 'tdatz': [51, 63, -33, 2, 15]},
{'pomfx': [62], 'eddvg': [71, -39, -2, -57, 71, -90, -79, -82], 'ncmzd': [-48, 11, -93, 96, -51, 6, 22, -85, -46],
'tdatz': [-60, -31, 80, -75, -42, 4, -7, -17]}, {'pomfx': [16, 40, -70, -14, -53, -92, 59, -49], 'eddvg': [81], 'n
cmzd': [63, -48, -26, 90, -36, 11], 'tdatz': [82, 40, 20, 21, -30, -63, 76]}, {'pomfx': [1, 90], 'eddvg': [17, 61,
-2, 38, -3, 56], 'ncmzd': [90, -98, 43, 42], 'tdatz': [88, 26, -67, 29, 93, 7, 68, -34]}, {'pomfx': [-36, -63, -
8], 'eddvg': [-34, 58, -25, -75, 62, 65, -36, 2, -62], 'ncmzd': [48, 4, -53, -89, -19], 'tdatz': [2, 36, -35]},
{'pomfx': [69], 'eddvg': [-22, -77, 42, -76], 'ncmzd': [30], 'tdatz': [-64, -66]}]

* Expected result when extracting key 'pomfx' == [{36, -24, -19, 15, 82, -14, -41, -68}, {-47, -46, 46}, {62}, {-9
2, 40, -53, -49, 16, -14, -70, 59}, {1, 90}, {-8, -63, -36}, {69}]

=== Test case ===

* item_lists == [{'kxtjz': [-82, 49, -57, -86, 80]}, {'kxtjz': [-81, 38, -77, -78]}, {'kxtjz': [86, -93]}]

* Expected result when extracting key 'kxtjz' == [{-57, -86, -82, 80, 49}, {-78, -77, 38, -81}, {-93, 86}]

=== Test case ===

* item_lists == [{'wgwcz': [49, 99, 88, 26, 27, -9]}]

* Expected result when extracting key 'wgwcz' == [{99, 49, -9, 88, 26, 27}]

=== Test case ===
```

```
* item_lists == [{'ptdlo': [-76, -58, 63]}]

* Expected result when extracting key 'ptdlo' == [{-76, -58, 63}]

=== Test case ===

* item_lists == [{'mlqos': [26, 91, 40, -39], 'dbmne': [76, -98, -59], 'bqpug': [88], 'sulmk': [-25, -85, 19, -14,
-40, -48, 44, -43, 77]}, {'mlqos': [-46, -8], 'dbmne': [21, -32, -66, -82], 'bqpug': [31, 54, -75, -39, 51], 'sulm
k': [10, -94, 68, -36]}, {'mlqos': [-6, -44, -75, -45, -95, 11], 'dbmne': [-36, 12, 63, 58, -52, -29], 'bqpug': [5
9, 36], 'sulmk': [48, 54, 5]}, {'mlqos': [-85, -59, -68, -78, 97], 'dbmne': [81, -92], 'bqpug': [-75, -50, -92, -9
5, -45, 37, 36, -67, -54], 'sulmk': [-99, -89, -4, -18, -61, 74]}, {'mlqos': [78, 5, -7, 50, 1, -25, -13, -27], 'd
bmne': [-49, -50, 8, 85, -84], 'bqpug': [1], 'sulmk': [39]}, {'mlqos': [-77, 94, 19, 84, 80, 94, -76], 'dbmne': [8
8, -65, -74, -69, 22, 4, 29, -92], 'bqpug': [-32, -41, 52, -89, -26, -73, 81, -3], 'sulmk': [-67, -61, 52, 6, -60,
35, -2, -34]}, {'mlqos': [-30, -47, 31, 88, 56, 41, -68, -92, -40], 'dbmne': [13, 30, 90, 32, 6, 62, -57, 80], 'bq
pug': [33, 6], 'sulmk': [10, 63, 38, 0, 24, -87, -13, -67]}, {'mlqos': [19, 3, -88, 99, 21, 13], 'dbmne': [13, 8,
1, -87, 0, -61], 'bqpug': [-26, 46, -46, 32, -2], 'sulmk': [80, 70, -92, -93, -94, -2, -95, 28, -21]}]

* Expected result when extracting key 'mlqos' == [{40, -39, 26, 91}, {-8, -46}, {-95, 11, -45, -44, -75, -6}, {97,
-59, -85, -78, -68}, {1, 5, -27, -25, 78, 50, -13, -7}, {80, 19, -77, 84, -76, 94}, {-30, -92, 41, -47, -40, 88, 5
6, -68, 31}, {3, 99, -88, 13, 19, 21}]

(Passed!)
```

Ingredient (item)sets

In case you weren't able to get Exercise 1 working, we've precomputed itemsets for the recipes database. Run the following code cell to load them into an object, `ingredient_sets`, which will hold these *ingredient itemsets*.

In [7]:
```
def load_ingredient_sets(infilename="recipes/ex1a_soln.pickle"):
    from pickle import load
    with open(get_path(infilename), "rb") as fp:
        ingredient_sets = load(fp)
    return ingredient_sets

ingredient_sets = load_ingredient_sets()
print(f"Found {len(ingredient_sets)} ingredient itemsets.")
print("Examples:")
print("\n- ingredient_sets[0]:\n", ingredient_sets[0])
print("\n- ingredient_sets[1]:\n", ingredient_sets[1])
```

```
Found 39774 ingredient itemsets.
Examples:

- ingredient_sets[0]:
 {'black olives', 'seasoning', 'purple onion', 'pepper', 'garbanzo beans', 'grape tomatoes', 'garlic', 'romaine le
ttuce', 'feta cheese crumbles'}

- ingredient_sets[1]:
 {'ground pepper', 'salt', 'ground black pepper', 'green tomatoes', 'yellow corn meal', 'thyme', 'vegetable oil',
'tomatoes', 'plain flour', 'milk', 'eggs'}
```

Recipe similarity

From the preceding exercise, we now have each recipe represented by an itemset.

Next, consider two recipes, a and b. Define their *recipe-similarity* to be the number of ingredients they have in common. For instance, consider the following two recipes:

In [8]:
```
def print_ingredient_set(i, header=None):
    if header is not None:
        print(header)
    for ingredient in i:
        print(f"- {ingredient}")

print_ingredient_set(ingredient_sets[0], "[0]")
print()
print_ingredient_set(ingredient_sets[34089], "[34089]")
print()
common_01 = ingredient_sets[0] & ingredient_sets[34089]
print("==> Common ingredients:\n", common_01)
```

```
[0]
- black olives
- seasoning
- purple onion
```

```
- pepper
- garbanzo beans
- grape tomatoes
- garlic
- romaine lettuce
- feta cheese crumbles

[34089]
- black olives
- salad dressing
- purple onion
- lemon
- cucumber
- garbanzo beans
- garlic
- ground black pepper
- feta cheese crumbles
- cherry tomatoes
- garlic salt

==> Common ingredients:
 {'black olives', 'purple onion', 'garbanzo beans', 'garlic', 'feta cheese crumbles'}
```

These two recipes share the ingredients, `'black olives'`, `'feta cheese crumbles'`, `'garbanzo beans'`, `'garlic'`, `'purple onion'`. Therefore, the recipe-similarity score is 5.

Given the itemsets, it is easy to measure similarity! The function below, `recipe_similarity(a, b)` does so, given two ingredient sets `a` and `b`.

In [9]:
```
def recipe_similarity(a, b):
    assert isinstance(a, set) and isinstance(b, set)
    return len(a & b)

# Demo:
print(recipe_similarity(ingredient_sets[0], ingredient_sets[34089]))
```

```
5
```

Exercise 2 (2 points). Suppose you are given the following:

- A list of ingredient itemsets, named `I`;
- An integer index `i` corresponding to one of these, `I[i]`.
- A positive integer `k` such that `1 <= k < len(I)`.

Complete the function, `get_closest_recipes(I, i, k)` so that it returns a list of the `k` *indices* corresponding to the itemsets that are most similar to `I[i]`. That is, it should measure the recipe-similarity between `I[i]` and all other `I[j]`, where `j != i`, returning the `j` values whose itemsets are closest. You can (and should!) use the `recipe_similarity()` function that we defined for you above.

For example, for `ingredient_sets[0]`, it turns out the 3 other closest itemsets are `ingredient_sets[12869]`, `ingredient_sets[34089]`, and `ingredient_sets[795]`. Therefore:

```
assert get_closest_recipes(ingredient_sets, 0, 3) == [12869, 34089, 795]
```

> **Note 0**: Of course, `I[i]` will be a perfect match to itself! However, `i` should not be part of the returned list.
>
> **Note 1**: In the event of ties that result in more than `k` matches, you may return any subset. For instance, suppose `k=3` and the top similarity scores are 5, 5, 4, 4, 4, 3. In this case, your result must include the indices corresponding to the two "5" scores, but for the third returned value, may return any of the indices corresponding to the three "4" scores.

In [10]:
```
def get_closest_recipes(I, i, k):
    assert i >= 0 and i < len(I)
    assert k >= 1 and k < len(I)
    ### BEGIN SOLUTION
    indexed = enumerate(I)
    ordered = sorted(indexed, key=lambda x: recipe_similarity(I[i], x[1]), reverse=True)
    ordered.remove((i, I[i]))
    return [j for j, _ in ordered[:k]]
    ### END SOLUTION
```

In [11]:
```
# Demo:
print_ingredient_set(ingredient_sets[0], "==> `ingredient_sets[0]`:")

top_3_closest_to_0 = get_closest_recipes(ingredient_sets, 0, 3)
```

```
print("\n=== Three closest recipes ===")
for j in top_3_closest_to_0:
    sj = recipe_similarity(ingredient_sets[0], ingredient_sets[j])
    print_ingredient_set(ingredient_sets[j], f"\n`ingredient_sets[{j}]` (similiarity={sj}):")
```

```
==> `ingredient_sets[0]`:
- black olives
- seasoning
- purple onion
- pepper
- garbanzo beans
- grape tomatoes
- garlic
- romaine lettuce
- feta cheese crumbles

=== Three closest recipes ===

`ingredient_sets[12869]` (similiarity=5):
- purple onion
- croutons
- pepper
- dried oregano
- olive oil
- egg substitute
- salt
- kalamata
- garlic
- romaine lettuce
- feta cheese crumbles
- lemon juice

`ingredient_sets[34089]` (similiarity=5):
- black olives
- salad dressing
- purple onion
- lemon
- cucumber
- garbanzo beans
- garlic
- ground black pepper
- feta cheese crumbles
- cherry tomatoes
- garlic salt

`ingredient_sets[795]` (similiarity=4):
- black pepper
- boiling water
- couscous
- fresh parsley
- roasted red peppers
- red pepper flakes
- kosher salt
- greek yogurt
- baby spinach
- eggs
- pepper
- nutmeg
- cucumber
- olive oil
- lemon juice
- feta cheese crumbles
- pinenuts
- yellow onion
- sour cream
- ground beef
- purple onion
- dried oregano
- salt
- grated parmesan cheese
- dill weed
- garlic
- coarse salt
```

In [12]:

```
# Test cell: `ex2_get_closest_recipes` (2 points)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below.
""")
```

```
    )

def ex2_check_one__():
    from random import randrange

    def gen_random_token():
        from random import choice
        consonants = 'bcdfghjklmnpqrstvwxyz'
        vowels = 'aeiou'
        return choice(consonants) + choice(vowels) + choice(consonants)

    def gen_random_soln(S, m): # Gen `m` subsets from `S`
        from random import shuffle
        assert isinstance(S, set)
        assert len(S) >= m
        T = [set() for _ in range(m)]
        N = list(range(m))
        shuffle(N)
        for k, x in enumerate(S):
            for i in range(m):
                if i <= k:
                    T[N[i]].add(x)
        return T, N

    print("\n=== Test case ===\n")
    S = set([gen_random_token() for _ in range(10)])
    I, N = gen_random_soln(S, randrange(2, len(S)))
    i = N[0]
    k = randrange(1, len(I))
    soln = N[1:k+1]

    print("* I ==")
    for j, t in enumerate(I):
        print(f"  [{j}]", t)
    print(f"* i == {i}")
    print(f"* k == {k}")
    your_soln = get_closest_recipes(I, i, k)
    assert your_soln == soln, \
           f"*** Failed ***\n" \
           f"- Expected solution: {soln}\n" \
           f"- Your solution: {your_soln}\n"

for _ in range(10):
    ex2_check_one__()

### BEGIN HIDDEN TESTS
def ex2_get_closest_recipes_soln__(I, i, k):
    assert i >= 0 and i < len(I)
    assert k >= 1 and k < len(I)
    indexed = enumerate(I)
    ordered = sorted(indexed, key=lambda x: recipe_similarity(I[i], x[1]), reverse=True)
    ordered.remove((i, I[i]))
    return [j for j, _ in ordered[:k]]

def ex2_gen_soln__(I, outfilename, k_max=5):
    from os.path import isfile
    from pickle import dump
    if not isfile(get_path(outfilename)):
        print(f"Generating solutions file, '{outfilename}'...")
        closest = []
        for i in range(len(I)):
            closest.append(ex2_get_closest_recipes_soln__(I, i, k_max))
        with open(get_path(outfilename), 'wb') as fp:
            dump(closest, fp)
    else:
        print(f"An existing solutions file, '{outfilename}', is available.")

print()
ex2_gen_soln__(ingredient_sets, 'recipes/closest.pickle')
### END HIDDEN TESTS

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below.


=== Test case ===

* I ==
  [0] {'tav', 'rul', 'zek', 'xuz', 'kah', 'fuj', 'zun', 'hit', 'yah', 'sit'}
```

```
  [1] {'zek', 'xuz', 'kah', 'fuj', 'zun', 'hit', 'yah', 'sit'}
  [2] {'kah', 'fuj', 'zun', 'hit', 'yah', 'sit'}
  [3] {'rul', 'zek', 'xuz', 'kah', 'fuj', 'zun', 'hit', 'yah', 'sit'}
  [4] {'xuz', 'kah', 'fuj', 'zun', 'hit', 'yah', 'sit'}
* i == 0
* k == 4

=== Test case ===

* I ==
  [0] {'lic', 'keb', 'wel'}
  [1] {'keb', 'guv', 'gax', 'pet', 'low', 'lic', 'faq', 'wel'}
  [2] {'pet', 'low', 'lic', 'keb', 'wel'}
  [3] {'but', 'wic', 'keb', 'guv', 'gax', 'pet', 'low', 'lic', 'faq', 'wel'}
  [4] {'wic', 'keb', 'guv', 'gax', 'pet', 'low', 'lic', 'faq', 'wel'}
  [5] {'low', 'lic', 'keb', 'wel'}
  [6] {'keb', 'gax', 'pet', 'low', 'lic', 'faq', 'wel'}
  [7] {'keb', 'pet', 'low', 'lic', 'faq', 'wel'}
  [8] {'keb', 'wel'}
* i == 3
* k == 1

=== Test case ===

* I ==
  [0] {'paf', 'vuq'}
  [1] {'paf', 'mil', 'vuq', 'mul'}
  [2] {'paf', 'vuq', 'mul'}
  [3] {'non', 'ziy', 'luj', 'pew', 'yap', 'tis', 'mil', 'mul', 'paf', 'vuq'}
  [4] {'ziy', 'luj', 'pew', 'yap', 'tis', 'mil', 'mul', 'paf', 'vuq'}
  [5] {'tis', 'mil', 'mul', 'paf', 'vuq'}
  [6] {'yap', 'tis', 'mil', 'mul', 'paf', 'vuq'}
  [7] {'pew', 'yap', 'tis', 'mil', 'mul', 'paf', 'vuq'}
  [8] {'luj', 'pew', 'yap', 'tis', 'mil', 'mul', 'paf', 'vuq'}
* i == 3
* k == 6

=== Test case ===

* I ==
  [0] {'wit', 'jur', 'dev'}
  [1] {'del', 'ped', 'dev', 'wit', 'jur'}
  [2] {'hap', 'qet', 'del', 'ped', 'dev', 'wit', 'jur'}
  [3] {'fal', 'zur', 'hap', 'qet', 'del', 'ped', 'dev', 'wit', 'jur'}
  [4] {'sew', 'zur', 'fal', 'hap', 'qet', 'del', 'ped', 'dev', 'wit', 'jur'}
  [5] {'zur', 'hap', 'qet', 'del', 'ped', 'dev', 'wit', 'jur'}
  [6] {'wit', 'jur'}
  [7] {'wit', 'jur', 'ped', 'dev'}
  [8] {'qet', 'del', 'ped', 'dev', 'wit', 'jur'}
* i == 4
* k == 1

=== Test case ===

* I ==
  [0] {'pen', 'cet', 'lil', 'fag', 'vom', 'nis', 'law', 'dif', 'qap'}
  [1] {'cet', 'lil', 'fag', 'vom', 'nis', 'law', 'dif', 'qap'}
  [2] {'pan', 'pen', 'cet', 'lil', 'fag', 'vom', 'nis', 'law', 'dif', 'qap'}
  [3] {'dif', 'qap'}
  [4] {'nis', 'law', 'dif', 'qap'}
  [5] {'law', 'dif', 'qap'}
  [6] {'cet', 'fag', 'vom', 'nis', 'law', 'dif', 'qap'}
  [7] {'vom', 'nis', 'law', 'dif', 'qap'}
  [8] {'fag', 'vom', 'nis', 'law', 'dif', 'qap'}
* i == 2
* k == 6

=== Test case ===

* I ==
  [0] {'cef', 'hil', 'nib', 'fab', 'dad', 'mas', 'wiv'}
  [1] {'dof', 'cef', 'hil', 'nib', 'fab', 'dad', 'mas', 'wiv'}
  [2] {'gip', 'nik', 'dof', 'cef', 'hil', 'nib', 'fab', 'dad', 'mas', 'wiv'}
  [3] {'nik', 'dof', 'cef', 'hil', 'nib', 'fab', 'dad', 'mas', 'wiv'}
* i == 2
* k == 1

=== Test case ===

* I ==
```

```
  [0] {'ved', 'loq', 'suj', 'web', 'poj'}
  [1] {'poj', 'faw', 'qem', 'woq', 'mij', 'ved', 'loq', 'suj', 'web', 'sas'}
  [2] {'woq', 'mij', 'ved', 'loq', 'suj', 'web', 'poj'}
  [3] {'poj', 'qem', 'woq', 'mij', 'ved', 'loq', 'suj', 'web', 'sas'}
  [4] {'poj', 'woq', 'mij', 'ved', 'loq', 'suj', 'web', 'sas'}
  [5] {'mij', 'ved', 'loq', 'suj', 'web', 'poj'}
* i == 1
* k == 1

=== Test case ===

* I ==
  [0] {'wot', 'vul', 'bug', 'som', 'rir', 'had', 'ren', 'nop', 'cif'}
  [1] {'vul', 'bug', 'som', 'rir', 'had', 'ren', 'nop', 'cif'}
  [2] {'xoj', 'wot', 'vul', 'bug', 'som', 'rir', 'had', 'ren', 'nop', 'cif'}
* i == 2
* k == 2

=== Test case ===

* I ==
  [0] {'vub', 'sew', 'qos', 'fop', 'caq', 'rus', 'mab', 'zam', 'zar', 'lij'}
  [1] {'qos', 'fop', 'caq', 'rus', 'mab', 'zam', 'zar', 'lij'}
  [2] {'fop', 'caq', 'rus', 'mab', 'zam', 'zar', 'lij'}
  [3] {'rus', 'mab', 'zam', 'zar', 'lij'}
  [4] {'caq', 'rus', 'mab', 'zam', 'zar', 'lij'}
  [5] {'sew', 'qos', 'fop', 'caq', 'rus', 'mab', 'zam', 'zar', 'lij'}
* i == 0
* k == 4

=== Test case ===

* I ==
  [0] {'xak', 'rum', 'jaf', 'lud', 'man', 'weh', 'yuy', 'zor', 'pud', 'zay'}
  [1] {'lud', 'man', 'weh', 'yuy', 'zor', 'pud', 'zay'}
  [2] {'jaf', 'lud', 'man', 'weh', 'yuy', 'zor', 'pud', 'zay'}
  [3] {'rum', 'jaf', 'lud', 'man', 'weh', 'yuy', 'zor', 'pud', 'zay'}
* i == 0
* k == 1

An existing solutions file, 'recipes/closest.pickle', is available.

(Passed!)
```

Closest recipes

Just in case you did not get a working solution for Exercise 2, we have precomputed the five (5) closest recipes for every recipe. The following code cell will load this data in a list, `closest`. For each ingredient set `ingredient_sets[i]`, the entry `closest[i]` is a list of the indices of the 5 other closest ingredient sets in descending order.

In [13]:
```
def load_closest(infile='recipes/closest.pickle'):
    from pickle import load
    with open(get_path(infile), 'rb') as fp:
        return load(fp)

print("Loading precomputed list of closest itemsets...")
closest = load_closest()
print_ingredient_set(ingredient_sets[0], f"\n`ingredient_sets[0]`:")
print("\nFive closest ingredient sets:")
for j in closest[0]:
    sj = recipe_similarity(ingredient_sets[0], ingredient_sets[j])
    print_ingredient_set(ingredient_sets[j], f"\n`ingredient_sets[{j}]` (similiarity={sj}):")
```

```
Loading precomputed list of closest itemsets...

`ingredient_sets[0]`:
- black olives
- seasoning
- purple onion
- pepper
- garbanzo beans
- grape tomatoes
- garlic
- romaine lettuce
- feta cheese crumbles

Five closest ingredient sets:
```

```
`ingredient_sets[12869]` (similiarity=5):
- purple onion
- croutons
- pepper
- dried oregano
- olive oil
- egg substitute
- salt
- kalamata
- garlic
- romaine lettuce
- feta cheese crumbles
- lemon juice

`ingredient_sets[34089]` (similiarity=5):
- black olives
- salad dressing
- purple onion
- lemon
- cucumber
- garbanzo beans
- garlic
- ground black pepper
- feta cheese crumbles
- cherry tomatoes
- garlic salt

`ingredient_sets[795]` (similiarity=4):
- black pepper
- boiling water
- couscous
- fresh parsley
- roasted red peppers
- red pepper flakes
- kosher salt
- greek yogurt
- baby spinach
- eggs
- pepper
- nutmeg
- cucumber
- olive oil
- lemon juice
- feta cheese crumbles
- pinenuts
- yellow onion
- sour cream
- ground beef
- purple onion
- dried oregano
- salt
- grated parmesan cheese
- dill weed
- garlic
- coarse salt

`ingredient_sets[4534]` (similiarity=4):
- garlic cloves
- red pepper
- purple onion
- pepper
- cucumber
- dried oregano
- feta cheese
- olive oil
- frozen chopped spinach
- stuffing mix
- salt
- garlic
- whole wheat hamburger buns
- feta cheese crumbles
- lemon juice
- tomatoes
- burgers
- nonfat greek yogurt
- dried dill
- eggs

`ingredient_sets[5189]` (similiarity=4):
- chicken breasts
```

```
- genoa salami
- purple onion
- red wine vinegar
- garbanzo beans
- sugar
- shredded parmesan cheese
- roma tomatoes
- garlic
- ground black pepper
- romaine lettuce
- extra-virgin olive oil
- bacon
- large egg yolks
- shredded mozzarella cheese
- dijon mustard
- kosher salt
```

Ingredient similarity

Above, you created a function to measure the similarity of recipes. What about ingredients -- how can we measure how similar two ingredients are?

One "tool" we have from Notebook 2 is a pairwise association miner. Recall that this tool calculates the *confidence*, $\mathrm{conf}(a \implies b)$, which is an estimate of the conditional probability of b given a. Run the code cell below, which runs the a modified version of the code from Notebook 2 on the ingredient itemsets, producing two results:

1. The pairwise association rules among ingredients, i.e., $\mathrm{conf}(a \implies b)$ where a and b are ingredients (by name). These are stored in the `rules` object.
2. The number of recipes in which each ingredient appears, stored in `ingredient_counts`. That is, `ingredient_counts[a]` is the number of recipes containing the ingredient named `a`.

> **Note**: The `find_assoc_rules()` function, below, looks for all rules (threshold is 0.0) but excludes ingredients that appear in fewer than 5 recipes (`min_item_count=5`).

In [14]:
```
from assocmine import find_assoc_rules, count_items, print_rules

print("Counting the occurrences of each ingredient...")
ingredient_counts = count_items(ingredient_sets)
print(f"==> Found {len(ingredient_counts)} distinctly named ingredients.")

print("\nNow running the association rule miner from Notebook 2...")
rules = find_assoc_rules(ingredient_sets, 0.0, min_item_count=25)
print(f"==> Found {len(rules)} rules.")
```

```
Counting the occurrences of each ingredient...
==> Found 6714 distinctly named ingredients.

Now running the association rule miner from Notebook 2...
==> Found 566322 rules.
```

> Observe that there are **many** ingredients and rules, so do be careful if you are trying to print them!

Here is a quick demo of how you can use these results.

In [15]:
```
a_ex = 'lemon'
n_ex = ingredient_counts[a_ex]
print(f"Ingredient '{a_ex}' occurs {n_ex} times.")

b_ex = 'salt'
c_ex = rules[(a_ex, b_ex)]
print(f"\nconf('{a_ex}', '{b_ex}') = {c_ex}")
```

```
Ingredient 'lemon' occurs 1218 times.

conf('lemon', 'salt') = 0.45648604269293924
```

(Sparse) ingredient vectors. Given an ingredient named `a`, its *ingredient vector* is a dictionary such that:

- each key `b` is the name of another ingredient; and
- the corresponding value is the confidence `conf(a => b)`.

For example, the ingredient `'lemon'` occurs in 1,218 recipes and ends up in 1,108 rules (of the form, $'\texttt{lemon}' \implies b$):

In [16]:
```
cip_ex = 'lemon'
print(f"There are {ingredient_counts[cip_ex]} recipes containing '{cip_ex}'.")

cip_ex_rules = {(a, b): conf_ab for (a, b), conf_ab in rules.items() if a == cip_ex}
print(f"This ingredient appears in", len(cip_ex_rules), "rules.")
print("The top five by confidence are:")
print_rules(cip_ex_rules, rank=5, prefix="- ")
```

```
There are 1218 recipes containing 'lemon'.
This ingredient appears in 1108 rules.
The top five by confidence are:
- conf(lemon => salt) = 0.456
- conf(lemon => olive oil) = 0.291
- conf(lemon => garlic) = 0.280
- conf(lemon => onions) = 0.220
- conf(lemon => ground black pepper) = 0.180
```

Exercise 3 (2 points). Complete the function,

```
def ingredient_vector(a, rules):
    ...
```

so that it returns the ingredient vector for the ingredient named `a`, using the confidence rules in `rules`. For example,

```
assert ingredient_vector('lemon', rules) == \
       {'white wine': 0.027093596059113302,
        'salmon fillets': 0.0090311986863711,
        'pesto': 0.004105090311986864,
        'saffron': 0.010673234811165846,       {'graham cracker crumbs': 0.1,
        ...
       } # 45 key-value pairs
```

If there are no rules `conf(a => b)`, then the function should return an empty dictionary.

In [17]:
```
def ingredient_vector(a, rules):
    from collections import defaultdict
    assert isinstance(a, str)
    assert isinstance(rules, dict) or isinstance(rules, defaultdict)
    ### BEGIN SOLUTION
    return {y: conf_xy for (x, y), conf_xy in rules.items() if x == a}
    ### END SOLUTION
```

In [18]:
```
# Demo
ingredient_vector('lemon', rules)
```

Out[18]:
```
{'pesto': 0.004105090311986864,
 'white wine': 0.027093596059113302,
 'salmon fillets': 0.0090311986863711,
 'fresh cilantro': 0.023809523809523808,
 'pepper': 0.1330049261083744,
 'chickpeas': 0.029556650246305417,
 'olive oil': 0.2914614121510673,
 'salt': 0.45648604269293924,
 'shallots': 0.027914614121510674,
 'fresh ginger': 0.041050903119868636,
 'saffron': 0.010673234811165846,
 'warm water': 0.010673234811165846,
 'tomatoes': 0.08538587848932677,
 'chicken stock': 0.03201970443349754,
 'tomato paste': 0.03366174055829228,
 'fresh parsley': 0.05829228243021346,
 'tumeric': 0.048440065681444995,
 'flour': 0.053366174055829226,
 'clove': 0.022988505747126436,
 'sugar': 0.1330049261083744,
 'plain whole-milk yogurt': 0.0016420361247947454,
 'honey': 0.0361247947454844,
 'cinnamon sticks': 0.030377668308702793,
 'anise': 0.0008210180623973727,
 'whipping cream': 0.0049261083743842365,
 'unflavored gelatin': 0.003284072249589491,
 'orange': 0.052545155993431854,
 'frozen peas': 0.0180623973727422,
 'garbanzo beans': 0.009852216748768473,
```

```
'potatoes': 0.031198686371100164,
'cumin': 0.024630541871921183,
'cream': 0.011494252873563218,
'cayenne pepper': 0.06321839080459771,
'smoked paprika': 0.0180623973727422,
'garlic': 0.2799671592775041,
'coriander': 0.027914614121510674,
'chopped cilantro': 0.0180623973727422,
'sea salt': 0.052545155993431854,
'chicken breasts': 0.024630541871921183,
'ghee': 0.011494252873563218,
'garlic cloves': 0.1535303776683087,
'ground cumin': 0.0812807881773399,
'white onion': 0.010673234811165846,
'coconut milk': 0.013957307060755337,
'fresh coriander': 0.016420361247947456,
'paprika': 0.05993431855500821,
'green cardamom pods': 0.0024630541871921183,
'ginger': 0.0722495894909688,
'tomato purée': 0.014778325123152709,
'green chilies': 0.029556650246305417,
'fenugreek': 0.004105090311986864,
'black pepper': 0.059113300492610835,
'butter': 0.138752052545156,
'shrimp': 0.05008210180623974,
'linguine': 0.012315270935960592,
'extra-virgin olive oil': 0.1272577996715928,
'dry white wine': 0.05665024630541872,
'parsley leaves': 0.004105090311986864,
'kosher salt': 0.11494252873563218,
'onions': 0.2200328407224959,
'cumin seed': 0.035303776683087026,
'cauliflower': 0.005747126436781609,
'garlic paste': 0.006568144499178982,
'chili powder': 0.03858784893267652,
'beans': 0.0016420361247947454,
'carrots': 0.06403940886699508,
'capsicum': 0.0016420361247947454,
'cilantro leaves': 0.027914614121510674,
'kasuri methi': 0.0024630541871921183,
'water': 0.16748768472906403,
'ground cinnamon': 0.05747126436781609,
'rice': 0.009852216748768473,
'eggs': 0.090311986863711,
'mustard': 0.006568144499178982,
'cornmeal': 0.005747126436781609,
'creole seasoning': 0.017241379310344827,
'garlic powder': 0.026272577996715927,
'cooking oil': 0.012315270935960592,
'onion powder': 0.013957307060755337,
'hot sauce': 0.020525451559934318,
'catfish fillets': 0.004105090311986864,
'crème fraîche': 0.003284072249589491,
'large eggs': 0.05090311986863711,
'dark rum': 0.004105090311986864,
'unsalted butter': 0.090311986863711,
'cake flour': 0.0049261083743842365,
'coarse sea salt': 0.003284072249589491,
'olive oil flavored cooking spray': 0.0008210180623973727,
'cracked black pepper': 0.022167487684729065,
'pinenuts': 0.008210180623973728,
'dough': 0.0024630541871921183,
'harissa': 0.004105090311986864,
'golden raisins': 0.010673234811165846,
'fennel seeds': 0.013136288998357963,
'vegetable oil': 0.07799671592775041,
'couscous': 0.0180623973727422,
'reduced sodium chicken broth': 0.004105090311986864,
'ground coriander': 0.03776683087027915,
'ground lamb': 0.013957307060755337,
'sweet onion': 0.005747126436781609,
'rosemary sprigs': 0.008210180623973728,
'ground black pepper': 0.17980295566502463,
'light brown sugar': 0.0090311986863711,
'chicken thighs': 0.012315270935960592,
'tea bags': 0.009852216748768473,
'ice cubes': 0.013957307060755337,
'chillies': 0.006568144499178982,
'dark soy sauce': 0.0016420361247947454,
'long-grain rice': 0.005747126436781609,
'green pepper': 0.005747126436781609
```

```
 green pepper : 0.005747126436781609,
'red chili peppers': 0.027914614121510674,
'red bell pepper': 0.041050903119868636,
'spring onions': 0.013136288998357963,
'cinnamon': 0.031198686371100164,
'ground ginger': 0.030377668308702793,
'chicken legs': 0.008210180623973728,
'ground nutmeg': 0.012315270935960592,
'malt vinegar': 0.004105090311986864,
'ground allspice': 0.005747126436781609,
'oil': 0.042692939244663386,
'green onions': 0.03366174055829228,
'ground cloves': 0.013957307060755337,
'gingerroot': 0.0008210180623973727,
'ground thyme': 0.0016420361247947454,
'pork butt': 0.0016420361247947454,
'soy sauce': 0.05582922824302135,
'greek yogurt': 0.022167487684729065,
'chopped fresh chives': 0.004105090311986864,
'corn tortillas': 0.004105090311986864,
'capers': 0.03284072249589491,
'fine sea salt': 0.018883415435139574,
'parmesan cheese': 0.0180623973727422,
'navel oranges': 0.005747126436781609,
'cucumber': 0.030377668308702793,
'ice': 0.0090311986863711,
'mint leaves': 0.017241379310344827,
'apples': 0.011494252873563218,
'frozen corn': 0.0016420361247947454,
'lemongrass': 0.0016420361247947454,
'diced tomatoes': 0.024630541871921183,
'bay leaves': 0.040229885057471264,
'lime juice': 0.003284072249589491,
'bell pepper': 0.010673234811165846,
'black peppercorns': 0.012315270935960592,
'Thai red curry paste': 0.0008210180623973727,
'celery': 0.026272577996715927,
'fish sauce': 0.0180623973727422,
'parsley': 0.020525451559934318,
'chicken': 0.03366174055829228,
'thai basil': 0.004105090311986864,
'brown sugar': 0.034482758620689655,
'canola oil': 0.027914614121510674,
'mustard seeds': 0.0180623973727422,
'coriander powder': 0.010673234811165846,
'asafetida': 0.0024630541871921183,
'lemon juice': 0.06157635467980296,
'ground turmeric': 0.03694581280788178,
'curry leaves': 0.010673234811165846,
'coconut': 0.0049261083743842365,
'coriander seeds': 0.015599343185550082,
'fish': 0.005747126436781609,
'tamarind': 0.0016420361247947454,
'confectioners sugar': 0.016420361247947456,
'almonds': 0.010673234811165846,
'superfine sugar': 0.005747126436781609,
'almond extract': 0.003284072249589491,
'chicken broth': 0.0361247947454844,
'chopped onion': 0.009852216748768473,
'ground sirloin': 0.0008210180623973727,
'hard-boiled egg': 0.006568144499178982,
'beef broth': 0.006568144499178982,
'ketchup': 0.015599343185550082,
'sherry': 0.0024630541871921183,
'flat leaf parsley': 0.07142857142857142,
'worcestershire sauce': 0.03366174055829228,
'thyme': 0.009852216748768473,
'dark brown sugar': 0.008210180623973728,
'apple cider vinegar': 0.010673234811165846,
'raisins': 0.016420361247947456,
'crystallized ginger': 0.0008210180623973727,
'pears': 0.0008210180623973727,
'heavy cream': 0.042692939244663386,
'garam masala': 0.05090311986863711,
'tomato sauce': 0.0090311986863711,
'bay leaf': 0.019704433497536946,
'whole milk': 0.020525451559934318,
'all-purpose flour': 0.08866995073891626,
'hazelnuts': 0.0016420361247947454,
'baking powder': 0.03776683087027915,
'greens': 0.003284072249589491,
```

```
'melted butter': 0.007389162561576354,
'cream cheese': 0.0090311986863711,
'yeast': 0.0049261083743842365,
'egg yolks': 0.027093596059113302,
'cake': 0.0024630541871921183,
'milk': 0.0361247947454844,
'vanilla': 0.005747126436781609,
'bird chile': 0.0024630541871921183,
'dry sherry': 0.0024630541871921183,
'tiger prawn': 0.0008210180623973727,
'margarine': 0.0024630541871921183,
'purple onion': 0.06403940886699508,
'vermicelli': 0.004105090311986864,
'lamb': 0.0090311986863711,
'chopped cilantro fresh': 0.020525451559934318,
'ground cayenne pepper': 0.006568144499178982,
'chopped celery': 0.008210180623973728,
'boneless skinless chicken breasts': 0.027914614121510674,
'cooking spray': 0.008210180623973728,
'cherry tomatoes': 0.017241379310344827,
'fresh oregano leaves': 0.003284072249589491,
'cold water': 0.0090311986863711,
'vanilla extract': 0.022988505747126436,
'unsweetened cocoa powder': 0.0008210180623973727,
'hot water': 0.005747126436781609,
'corn starch': 0.023809523809523808,
'juice': 0.011494252873563218,
'chocolate': 0.0008210180623973727,
'large egg yolks': 0.013957307060755337,
'boiling water': 0.006568144499178982,
'mint sprigs': 0.007389162561576354,
'baking soda': 0.011494252873563218,
'olives': 0.008210180623973728,
'pickles': 0.0024630541871921183,
'sour cream': 0.017241379310344827,
'fish stock': 0.0049261083743842365,
'dried thyme': 0.024630541871921183,
'seasoning salt': 0.0016420361247947454,
'curry powder': 0.014778325123152709,
'white pepper': 0.005747126436781609,
'crabmeat': 0.0024630541871921183,
'english cucumber': 0.015599343185550082,
'rice vinegar': 0.010673234811165846,
'pork tenderloin': 0.0049261083743842365,
'habanero pepper': 0.0016420361247947454,
'white rice': 0.0024630541871921183,
'sweet potatoes': 0.0090311986863711,
'lump crab meat': 0.0016420361247947454,
'grated parmesan cheese': 0.03694581280788178,
'asparagus': 0.013957307060755337,
'arborio rice': 0.0090311986863711,
'bay scallops': 0.0024630541871921183,
'whitefish': 0.0024630541871921183,
'Tabasco Pepper Sauce': 0.012315270935960592,
'fresh thyme': 0.017241379310344827,
'sweet paprika': 0.0090311986863711,
'cayenne': 0.019704433497536946,
'whipped cream': 0.0016420361247947454,
'fresh raspberries': 0.0008210180623973727,
'black beans': 0.0016420361247947454,
'lime': 0.034482758620689655,
'yellow bell pepper': 0.008210180623973728,
'avocado': 0.020525451559934318,
'cilantro': 0.030377668308702793,
'orange bell pepper': 0.0024630541871921183,
'red pepper flakes': 0.017241379310344827,
'grapeseed oil': 0.004105090311986864,
'penne pasta': 0.0049261083743842365,
'whole wheat flour': 0.003284072249589491,
'seeds': 0.003284072249589491,
'freshly ground pepper': 0.0361247947454844,
'jalapeno chilies': 0.018883415435139574,
'yukon gold potatoes': 0.008210180623973728,
'kalamata': 0.015599343185550082,
'Italian parsley leaves': 0.0090311986863711,
'oregano': 0.012315270935960592,
'black olives': 0.0049261083743842365,
'crushed tomatoes': 0.010673234811165846,
'crushed red pepper flakes': 0.016420361247947454,
'bread': 0.0049261083743842365,
```

```
'white beans': 0.003284072249589491,
'celery ribs': 0.017241379310344827,
'cajun seasoning': 0.013136288998357963,
'andouille sausage': 0.006568144499178982,
'beef': 0.0024630541871921183,
'red wine vinegar': 0.027093596059113302,
'tomatillos': 0.0016420361247947454,
'dried chile': 0.0016420361247947454,
'scallions': 0.030377668308702793,
'serrano chile': 0.0016420361247947454,
'guajillo chiles': 0.0024630541871921183,
'whole chicken': 0.0024630541871921183,
'orzo pasta': 0.0049261083743842365,
'basmati rice': 0.012315270935960592,
'chives': 0.012315270935960592,
'chicken livers': 0.0024630541871921183,
'mayonaise': 0.025451559934318555,
'chili flakes': 0.006568144499178982,
'pork belly': 0.003284072249589491,
'baby spinach': 0.0090311986863711,
'fresh dill': 0.027914614121510674,
'vegetable broth': 0.010673234811165846,
'garlic chives': 0.0008210180623973727,
'lemon wedge': 0.009852216748768473,
'grits': 0.008210180623973728,
'ear of corn': 0.0049261083743842365,
'ground pepper': 0.015599343185550082,
'clams': 0.0049261083743842365,
'chopped parsley': 0.020525451559934318,
'champagne vinegar': 0.0008210180623973727,
'leeks': 0.006568144499178982,
'dijon mustard': 0.017241379310344827,
'coarse salt': 0.019704433497536946,
'spaghetti': 0.011494252873563218,
'lemon extract': 0.0016420361247947454,
'dried oregano': 0.040229885057471264,
'arugula': 0.009852216748768473,
'feta cheese': 0.017241379310344827,
'liquid smoke': 0.0016420361247947454,
'flank steak': 0.0008210180623973727,
'greek seasoning': 0.004105090311986864,
'pitas': 0.006568144499178982,
'yoghurt': 0.019704433497536946,
'granulated sugar': 0.030377668308702793,
'all purpose unbleached flour': 0.0049261083743842365,
'fresh lemon juice': 0.060755336617405585,
'spanish onion': 0.004105090311986864,
'marjoram': 0.0024630541871921183,
'chopped fresh thyme': 0.0090311986863711,
'dried rosemary': 0.0016420361247947454,
'herbes de provence': 0.003284072249589491,
'minced garlic': 0.035303776683087026,
'Niçoise olives': 0.0008210180623973727,
'anchovy fillets': 0.0090311986863711,
'dried basil': 0.009852216748768473,
'salted butter': 0.004105090311986864,
'pork ribs': 0.0008210180623973727,
'pork sausages': 0.0016420361247947454,
'rice noodles': 0.0049261083743842365,
'beansprouts': 0.004105090311986864,
'ground white pepper': 0.007389162561576354,
'dried shrimp': 0.0008210180623973727,
'hoisin sauce': 0.0024630541871921183,
'canola': 0.0008210180623973727,
'yellow corn meal': 0.007389162561576354,
'neutral oil': 0.0008210180623973727,
'sake': 0.006568144499178982,
'potato starch': 0.003284072249589491,
'caster sugar': 0.005747126436781609,
'plain flour': 0.009852216748768473,
'crimini mushrooms': 0.0024630541871921183,
'dressing': 0.0016420361247947454,
'artichokes': 0.030377668308702793,
'lemon slices': 0.0049261083743842365,
'orange juice': 0.011494252873563218,
'bread flour': 0.0024630541871921183,
'salt and ground black pepper': 0.010673234811165846,
'fresh asparagus': 0.0016420361247947454,
'cachaca': 0.0016420361247947454,
'strawberries': 0.006568144499178982,
```

```
'iceberg lettuce': 0.0016420361247947454,
'fresh tarragon': 0.005747126436781609,
'yellow onion': 0.031198686371100164,
'whole grain mustard': 0.0016420361247947454,
'prepared horseradish': 0.004105090311986864,
'large shrimp': 0.025451559934318555,
'sauce': 0.013136288998357963,
'russet potatoes': 0.008210180623973728,
'roasted red peppers': 0.0016420361247947454,
'chile pepper': 0.003284072249589491,
'spinach': 0.013957307060755337,
'vodka': 0.0049261083743842365,
'semolina': 0.0016420361247947454,
'phyllo dough': 0.0024630541871921183,
'chopped fresh sage': 0.003284072249589491,
'bread crumb fresh': 0.007389162561576354,
'cooked ham': 0.0016420361247947454,
'bread crumbs': 0.013957307060755337,
'asparagus spears': 0.0016420361247947454,
'yams': 0.0008210180623973727,
'jaggery': 0.0008210180623973727,
'red chili powder': 0.0090311986863711,
'fleur de sel': 0.0024630541871921183,
'sage': 0.0016420361247947454,
'rosemary': 0.009852216748768473,
'buttermilk': 0.016420361247947456,
'beer': 0.009852216748768473,
'clam juice': 0.0016420361247947454,
'french baguette': 0.0024630541871921183,
'white sugar': 0.021346469622331693,
'tequila': 0.0024630541871921183,
'Sriracha': 0.005747126436781609,
'greek style plain yogurt': 0.007389162561576354,
'sesame seeds': 0.0090311986863711,
'shortening': 0.0024630541871921183,
'onion tops': 0.0008210180623973727,
'grape tomatoes': 0.005747126436781609,
'sharp cheddar cheese': 0.0024630541871921183,
'bacon': 0.012315270935960592,
'corn': 0.0049261083743842365,
'crawfish': 0.008210180623973728,
'smoked sausage': 0.004105090311986864,
'red potato': 0.014778325123152709,
'large garlic cloves': 0.024630541871921183,
'green bell pepper': 0.018883415435139574,
'orange zest': 0.0024630541871921183,
'medium shrimp': 0.016420361247947456,
'coconut oil': 0.005747126436781609,
'urad dal': 0.0016420361247947454,
'saffron threads': 0.015599343185550082,
'artichoke hearts': 0.0049261083743842365,
'chopped green bell pepper': 0.0024630541871921183,
'chopped garlic': 0.007389162561576354,
'plum tomatoes': 0.016420361247947456,
'croutons': 0.003284072249589491,
'tomato juice': 0.004105090311986864,
'fresh herbs': 0.004105090311986864,
'large egg whites': 0.004105090311986864,
'halibut': 0.0016420361247947454,
'panko': 0.004105090311986864,
'fillets': 0.0049261083743842365,
'creole mustard': 0.0024630541871921183,
'preserved lemon': 0.004105090311986864,
'shell-on shrimp': 0.004105090311986864,
'panko breadcrumbs': 0.005747126436781609,
'jasmine rice': 0.0024630541871921183,
'okra': 0.003284072249589491,
'zucchini': 0.011494252873563218,
'eggplant': 0.0049261083743842365,
'kidney beans': 0.003284072249589491,
'italian plum tomatoes': 0.0008210180623973727,
'vegetables': 0.003284072249589491,
'brandy': 0.012315270935960592,
'red wine': 0.0049261083743842365,
'fresh orange juice': 0.007389162561576354,
'baby lima beans': 0.0008210180623973727,
'button mushrooms': 0.006568144499178982,
'peas': 0.0049261083743842365,
'balsamic vinegar': 0.006568144499178982,
'baby carrots': 0.003284072249589491,
'fresh tomatoes': 0.004105090311986864
```

```
 'fresh tomatoes': 0.004105905311988684,
'chicken pieces': 0.0024630541871921183,
'plain yogurt': 0.024630541871921183,
'sesame oil': 0.007389162561576354,
'lemon zest': 0.010673234811165846,
'egg whites': 0.0090311986863711,
'wine': 0.004105090311986864,
'sugar pea': 0.0024630541871921183,
'slivered almonds': 0.007389162561576354,
'anchovies': 0.0024630541871921183,
'thai chile': 0.0024630541871921183,
'1% low-fat milk': 0.0008210180623973727,
'chili': 0.024630541871921183,
'phyllo pastry': 0.003284072249589491,
'spring roll wrappers': 0.0008210180623973727,
'boneless skinless chicken breast halves': 0.013136288998357963,
'prosciutto': 0.004105090311986864,
'sage leaves': 0.003284072249589491,
'short-grain rice': 0.003284072249589491,
'wasabi paste': 0.0016420361247947454,
'nori': 0.004105090311986864,
'brown rice': 0.004105090311986864,
'chopped tomatoes': 0.012315270935960592,
'almond flour': 0.0024630541871921183,
'raspberries': 0.0008210180623973727,
'powdered sugar': 0.013957307060755337,
'cabbage': 0.009852216748768473,
'dried rice noodles': 0.0024630541871921183,
'cooked chicken breasts': 0.003284072249589491,
'skirt steak': 0.0008210180623973727,
'pico de gallo': 0.0016420361247947454,
'guacamole': 0.0008210180623973727,
'vegetable stock': 0.0049261083743842365,
'toasted sesame oil': 0.003284072249589491,
'red pepper': 0.0090311986863711,
'pecans': 0.0016420361247947454,
'anchovy paste': 0.003284072249589491,
'basil': 0.005747126436781609,
'white vinegar': 0.013957307060755337,
'yellow peppers': 0.0024630541871921183,
'chinese five-spice powder': 0.0024630541871921183,
'skinless chicken breasts': 0.0049261083743842365,
'toasted almonds': 0.0016420361247947454,
'fennel bulb': 0.010673234811165846,
'green olives': 0.011494252873563218,
'fresh mint': 0.021346469622331693,
'long grain white rice': 0.008210180623973728,
'turkey': 0.003284072249589491,
'ginger root': 0.0049261083743842365,
'mussels': 0.0090311986863711,
'bacon drippings': 0.0008210180623973727,
'fresh parsley leaves': 0.007389162561576354,
'mint': 0.010673234811165846,
'ground cardamom': 0.007389162561576354,
'tahini': 0.005747126436781609,
'freshly grated parmesan': 0.003284072249589491,
'pecorino romano cheese': 0.003284072249589491,
'poultry seasoning': 0.0008210180623973727,
'hot pepper sauce': 0.009852216748768473,
'steak': 0.0008210180623973727,
'dry bread crumbs': 0.006568144499178982,
'cardamom': 0.0049261083743842365,
'fenugreek leaves': 0.007389162561576354,
'boneless chicken': 0.0024630541871921183,
'masala': 0.003284072249589491,
'chili sauce': 0.004105090311986864,
'diced red onions': 0.003284072249589491,
'asafoetida': 0.003284072249589491,
'prawns': 0.0049261083743842365,
'black mustard seeds': 0.003284072249589491,
'roasted peanuts': 0.0016420361247947454,
'fresh curry leaves': 0.003284072249589491,
'thyme sprigs': 0.005747126436781609,
'kale': 0.007389162561576354,
'pecorino cheese': 0.0049261083743842365,
'white sesame seeds': 0.003284072249589491,
'pastry': 0.0024630541871921183,
'golden syrup': 0.0024630541871921183,
'fresh rosemary': 0.014778325123152709,
'mushrooms': 0.010673234811165846,
'baby corn': 0.0024630541871921183,
```

```
'fresh green bean': 0.0024630541871921183,
'ricotta cheese': 0.003284072249589491,
'pasta': 0.007389162561576354,
'basil leaves': 0.006568144499178982,
'rice flour': 0.0016420361247947454,
'seasoning': 0.0049261083743842365,
'crushed garlic': 0.0049261083743842365,
'cider vinegar': 0.004105090311986864,
'roasting chickens': 0.0024630541871921183,
'chicken bouillon': 0.0016420361247947454,
'lima beans': 0.003284072249589491,
'corn kernel whole': 0.0024630541871921183,
'frozen chopped spinach': 0.0016420361247947454,
'italian sausage': 0.0008210180623973727,
'jumbo pasta shells': 0.0008210180623973727,
'pasta sauce': 0.0016420361247947454,
'shredded mozzarella cheese': 0.004105090311986864,
'chorizo sausage': 0.003284072249589491,
'green beans': 0.008210180623973728,
'blanched almonds': 0.007389162561576354,
'dried cranberries': 0.0016420361247947454,
'fresh spinach': 0.004105090311986864,
'squash': 0.0016420361247947454,
'clear honey': 0.0024630541871921183,
'chicken drumsticks': 0.007389162561576354,
'nutmeg': 0.013136288998357963,
'rubbed sage': 0.0008210180623973727,
'apple juice': 0.0008210180623973727,
'peaches': 0.007389162561576354,
'liqueur': 0.003284072249589491,
'cod fillets': 0.003284072249589491,
'instant yeast': 0.0008210180623973727,
'pure vanilla extract': 0.007389162561576354,
'spaghetti squash': 0.0008210180623973727,
'marinade': 0.0016420361247947454,
'cheddar cheese': 0.0016420361247947454,
'italian salad dressing': 0.003284072249589491,
'lemon pepper': 0.0008210180623973727,
'reduced sodium soy sauce': 0.0016420361247947454,
'white mushrooms': 0.0008210180623973727,
'pork': 0.006568144499178982,
'chicken cutlets': 0.0016420361247947454,
'romano cheese': 0.0024630541871921183,
'creamy peanut butter': 0.0008210180623973727,
'fresh basil leaves': 0.008210180623973728,
'wheat flour': 0.0008210180623973727,
'light cream': 0.0024630541871921183,
'feta cheese crumbles': 0.009852216748768473,
'hass avocado': 0.0008210180623973727,
'salmon': 0.004105090311986864,
'cooked white rice': 0.004105090311986864,
'fresh oregano': 0.0180623973727422,
'mirin': 0.004105090311986864,
'sushi rice': 0.0024630541871921183,
'gari': 0.0016420361247947454,
'pork loin': 0.0016420361247947454,
'fat': 0.005747126436781609,
'dried apricot': 0.011494252873563218,
'sea scallops': 0.007389162561576354,
'shells': 0.004105090311986864,
'sausages': 0.0016420361247947454,
'ground beef': 0.006568144499178982,
'granulated garlic': 0.003284072249589491,
'ras el hanout': 0.004105090311986864,
'red lentils': 0.005747126436781609,
'heirloom tomatoes': 0.004105090311986864,
'sprinkles': 0.0008210180623973727,
'extra large eggs': 0.0024630541871921183,
'flaked coconut': 0.0016420361247947454,
'cocktail cherries': 0.003284072249589491,
'chopped pecans': 0.003284072249589491,
'pecan halves': 0.0024630541871921183,
'apple cider': 0.0008210180623973727,
'chipotle': 0.0008210180623973727,
'beef brisket': 0.0008210180623973727,
'dry mustard': 0.0024630541871921183,
'flour tortillas': 0.003284072249589491,
'maple syrup': 0.004105090311986864,
'chipotles in adobo': 0.0008210180623973727,
'salsa': 0.0008210180623973727,
```

```
'chili pepper': 0.004105090311986864,
'chiles': 0.005747126436781609,
'watermelon': 0.005747126436781609,
'cherries': 0.0008210180623973727,
'manchego cheese': 0.0016420361247947454,
'country bread': 0.0016420361247947454,
'ricotta salata': 0.0008210180623973727,
'active dry yeast': 0.004105090311986864,
'softened butter': 0.0016420361247947454,
'fresh parmesan cheese': 0.0016420361247947454,
'dill': 0.009852216748768473,
'allspice berries': 0.0024630541871921183,
'turnips': 0.0024630541871921183,
'broth': 0.0016420361247947454,
'beets': 0.005747126436781609,
'celery root': 0.0008210180623973727,
'parsnips': 0.0016420361247947454,
'fresh red chili': 0.003284072249589491,
'loosely packed fresh basil leaves': 0.0016420361247947454,
'fresh basil': 0.014778325123152709,
'kaffir lime leaves': 0.0016420361247947454,
'fresh lime juice': 0.003284072249589491,
'grated nutmeg': 0.0049261083743842365,
'ricotta': 0.003284072249589491,
'dates': 0.004105090311986864,
'baby spinach leaves': 0.003284072249589491,
'paneer': 0.004105090311986864,
'double cream': 0.005747126436781609,
'sliced almonds': 0.006568144499178982,
'whole peeled tomatoes': 0.0008210180623973727,
'stock': 0.0024630541871921183,
'bourbon whiskey': 0.005747126436781609,
'simple syrup': 0.003284072249589491,
'vinegar': 0.0049261083743842365,
'mustard oil': 0.003284072249589491,
'ground almonds': 0.006568144499178982,
'low-fat milk': 0.0016420361247947454,
'chicken stock cubes': 0.0024630541871921183,
'light coconut milk': 0.0016420361247947454,
'Guinness Beer': 0.0008210180623973727,
'currant': 0.0024630541871921183,
'fontina cheese': 0.0008210180623973727,
'white cheddar cheese': 0.0008210180623973727,
'sliced green onions': 0.0024630541871921183,
'blackberries': 0.0008210180623973727,
'blueberries': 0.0016420361247947454,
'fresh ginger root': 0.010673234811165846,
'black sesame seeds': 0.0008210180623973727,
'evaporated milk': 0.0016420361247947454,
'sweetened condensed milk': 0.003284072249589491,
'vanilla beans': 0.007389162561576354,
'lemon peel': 0.0024630541871921183,
'lime wedges': 0.004105090311986864,
'angel hair': 0.005747126436781609,
'heavy whipping cream': 0.003284072249589491,
'garlic salt': 0.004105090311986864,
'sunflower oil': 0.008210180623973728,
'cornflour': 0.004105090311986864,
'cashew nuts': 0.004105090311986864,
'spices': 0.014778325123152709,
'vine ripened tomatoes': 0.0016420361247947454,
'sun-dried tomatoes': 0.003284072249589491,
'ground red pepper': 0.0049261083743842365,
'noodles': 0.005747126436781609,
'romaine lettuce': 0.005747126436781609,
'mung bean sprouts': 0.0008210180623973727,
'radishes': 0.006568144499178982,
'firm tofu': 0.0024630541871921183,
'wonton wrappers': 0.0008210180623973727,
'seaweed': 0.0016420361247947454,
'rice cakes': 0.0008210180623973727,
'dumplings': 0.0008210180623973727,
'ground pork': 0.0024630541871921183,
'light mayonnaise': 0.0008210180623973727,
'Belgian endive': 0.0008210180623973727,
'pickle relish': 0.0016420361247947454,
'corn tortilla chips': 0.0008210180623973727,
'chees fresh mozzarella': 0.0008210180623973727,
'corn husks': 0.005747126436781609,
'whole cloves': 0.0024630541871921183,
'pineapple': 0.0008210180623973727
```

```
 'pineapple': 0.0008210180623973727,
'crab': 0.0008210180623973727,
'orange liqueur': 0.0024630541871921183,
'brewed coffee': 0.0016420361247947454,
'agave nectar': 0.0016420361247947454,
'meat': 0.0024630541871921183,
'rum': 0.003284072249589491,
'self rising flour': 0.005747126436781609,
'fronds': 0.0008210180623973727,
'veal shanks': 0.0024630541871921183,
'small red potato': 0.0049261083743842365,
'canned tomatoes': 0.0024630541871921183,
'butternut squash': 0.0016420361247947454,
'extra sharp cheddar cheese': 0.0016420361247947454,
'cooked shrimp': 0.0008210180623973727,
'old bay seasoning': 0.006568144499178982,
'bok choy': 0.0008210180623973727,
'chinese cabbage': 0.0008210180623973727,
'caraway seeds': 0.0008210180623973727,
'lentils': 0.004105090311986864,
'fava beans': 0.003284072249589491,
'pork shoulder': 0.003284072249589491,
'fresh chives': 0.006568144499178982,
'grated lemon zest': 0.004105090311986864,
'butter lettuce': 0.0016420361247947454,
'rice vermicelli': 0.003284072249589491,
'chili paste': 0.0024630541871921183,
'light corn syrup': 0.0008210180623973727,
'amaretto': 0.0008210180623973727,
'food colouring': 0.0016420361247947454,
'clarified butter': 0.003284072249589491,
'boneless chicken skinless thigh': 0.009852216748768473,
'peppercorns': 0.003284072249589491,
'stewed tomatoes': 0.0016420361247947454,
'salami': 0.0008210180623973727,
'sliced black olives': 0.0016420361247947454,
'dill pickles': 0.0008210180623973727,
'light sour cream': 0.0024630541871921183,
'pitted green olives': 0.003284072249589491,
'brown lentils': 0.0008210180623973727,
'Grand Marnier': 0.0016420361247947454,
'pita bread': 0.0024630541871921183,
'sliced mushrooms': 0.0016420361247947454,
'cream cheese, soften': 0.0016420361247947454,
'pancetta': 0.0024630541871921183,
'white bread': 0.0016420361247947454,
'curry paste': 0.003284072249589491,
'boneless chicken breast': 0.0024630541871921183,
'toasted sesame seeds': 0.004105090311986864,
'bone in chicken thighs': 0.0008210180623973727,
'pizza crust': 0.0008210180623973727,
'parmigiano reggiano cheese': 0.007389162561576354,
'finely chopped onion': 0.004105090311986864,
'provolone cheese': 0.0024630541871921183,
'vidalia onion': 0.004105090311986864,
'chicken bouillon granules': 0.0008210180623973727,
'spanish chorizo': 0.003284072249589491,
'champagne': 0.003284072249589491,
'meat bones': 0.003284072249589491,
'artichok heart marin': 0.0008210180623973727,
'curds': 0.0016420361247947454,
'Mexican beer': 0.0008210180623973727,
'french bread': 0.003284072249589491,
'walnuts': 0.003284072249589491,
'club soda': 0.005747126436781609,
'leaves': 0.0008210180623973727,
'raw cashews': 0.0024630541871921183,
'corn kernels': 0.0024630541871921183,
'nuts': 0.0024630541871921183,
'fresh lemon': 0.0008210180623973727,
'orzo': 0.0049261083743842365,
'brussels sprouts': 0.0024630541871921183,
'ladyfingers': 0.0024630541871921183,
'mascarpone': 0.0024630541871921183,
'lime slices': 0.003284072249589491,
'cranberries': 0.0008210180623973727,
'sea bass fillets': 0.0008210180623973727,
'waxy potatoes': 0.003284072249589491,
'taco shells': 0.0008210180623973727,
'low-fat sour cream': 0.0008210180623973727,
'white wine vinegar': 0.006568144499178982,
```

```
'snow peas': 0.0008210180623973727,
'portabello mushroom': 0.0016420361247947454,
'tamari soy sauce': 0.0016420361247947454,
'fish fillets': 0.005747126436781609,
'cardamom pods': 0.0024630541871921183,
'minced ginger': 0.0024630541871921183,
'safflower oil': 0.0008210180623973727,
'herbs': 0.0024630541871921183,
'boiling potatoes': 0.003284072249589491,
'sweetened coconut flakes': 0.0016420361247947454,
'penne': 0.004105090311986864,
'jumbo shrimp': 0.006568144499178982,
'crushed red pepper': 0.0090311986863711,
'peanuts': 0.006568144499178982,
'dipping sauces': 0.0008210180623973727,
'chuck': 0.0016420361247947454,
'crusty bread': 0.0024630541871921183,
'low-fat plain yogurt': 0.0008210180623973727,
'italian seasoning': 0.003284072249589491,
'pitted kalamata olives': 0.005747126436781609,
'tuna': 0.0024630541871921183,
'coarse kosher salt': 0.0016420361247947454,
'toasted pine nuts': 0.0016420361247947454,
'lamb shanks': 0.0024630541871921183,
'dried dillweed': 0.0024630541871921183,
'finely chopped fresh parsley': 0.004105090311986864,
'Italian bread': 0.0016420361247947454,
'crackers': 0.0016420361247947454,
'genoa salami': 0.0008210180623973727,
'mozzarella cheese': 0.0016420361247947454,
'dry red wine': 0.011494252873563218,
'grated orange peel': 0.0024630541871921183,
'sherry vinegar': 0.003284072249589491,
'spinach leaves': 0.0008210180623973727,
'toasted pecans': 0.0008210180623973727,
'frozen spinach': 0.0016420361247947454,
'chopped fresh mint': 0.008210180623973728,
'watercress': 0.0016420361247947454,
'shredded cheese': 0.0008210180623973727,
'cooked rice': 0.0008210180623973727,
'raw sugar': 0.0008210180623973727,
'low sodium chicken broth': 0.005747126436781609,
'white flour': 0.0008210180623973727,
'powdered milk': 0.0008210180623973727,
'mango chutney': 0.0008210180623973727,
'mango': 0.0016420361247947454,
'naan': 0.003284072249589491,
'red cabbage': 0.0016420361247947454,
'baguette': 0.005747126436781609,
'chicken breast halves': 0.0024630541871921183,
'tomatoes with juice': 0.0008210180623973727,
'ramen noodles': 0.0016420361247947454,
'shiitake': 0.003284072249589491,
'chickpea flour': 0.0024630541871921183,
'miso paste': 0.0008210180623973727,
'nutritional yeast': 0.0016420361247947454,
'lemon rind': 0.005747126436781609,
'yellow mustard seeds': 0.0016420361247947454,
'fenugreek seeds': 0.0024630541871921183,
'rocket leaves': 0.0024630541871921183,
'whole milk ricotta cheese': 0.0008210180623973727,
'fresh leav spinach': 0.0016420361247947454,
'semolina flour': 0.0016420361247947454,
'fennel': 0.003284072249589491,
'flatbread': 0.0024630541871921183,
'lettuce leaves': 0.004105090311986864,
'fettuccine pasta': 0.0016420361247947454,
'molasses': 0.0016420361247947454,
'condensed milk': 0.0016420361247947454,
'whole milk yoghurt': 0.0024630541871921183,
'chopped almonds': 0.003284072249589491,
'puff pastry': 0.0016420361247947454,
'low salt chicken broth': 0.007389162561576354,
'refrigerated piecrusts': 0.0016420361247947454,
'roma tomatoes': 0.004105090311986864,
'diced tomatoes in juice': 0.0008210180623973727,
'red food coloring': 0.0024630541871921183,
'part-skim mozzarella cheese': 0.0016420361247947454,
'fresh marjoram': 0.0016420361247947454,
'basil dried leaves': 0.0016420361247947454,
```

```
'soba': 0.0008210180623973727,
'Alfredo sauce': 0.0008210180623973727,
'cooked chicken': 0.0024630541871921183,
'fettucine': 0.0016420361247947454,
'dried parsley': 0.005747126436781609,
'grate lime peel': 0.0008210180623973727,
'grated lemon peel': 0.0049261083743842365,
'whiskey': 0.0016420361247947454,
'jam': 0.0008210180623973727,
'veal': 0.0008210180623973727,
'fat free less sodium chicken broth': 0.003284072249589491,
'orange flower water': 0.004105090311986864,
'elbow macaroni': 0.0008210180623973727,
'cheese': 0.003284072249589491,
'squid': 0.005747126436781609,
'hot pepper': 0.0008210180623973727,
'firmly packed light brown sugar': 0.0008210180623973727,
'edamame': 0.0008210180623973727,
'pearl onions': 0.0016420361247947454,
'oysters': 0.0024630541871921183,
'gram flour': 0.0024630541871921183,
'hothouse cucumber': 0.0024630541871921183,
'boneless, skinless chicken breast': 0.0008210180623973727,
'roasted tomatoes': 0.0024630541871921183,
'sliced cucumber': 0.0024630541871921183,
'wild mushrooms': 0.0008210180623973727,
'savoy cabbage': 0.0016420361247947454,
'minced onion': 0.0008210180623973727,
'yellow split peas': 0.0016420361247947454,
'plums': 0.003284072249589491,
'gruyere cheese': 0.0008210180623973727,
'baby arugula': 0.004105090311986864,
'part-skim ricotta cheese': 0.0008210180623973727,
'vinaigrette': 0.0008210180623973727,
'salad': 0.0024630541871921183,
'cornichons': 0.0024630541871921183,
'boneless chicken thighs': 0.0008210180623973727,
'pitted black olives': 0.0008210180623973727,
'triple sec': 0.003284072249589491,
'granny smith apples': 0.0049261083743842365,
'scallops': 0.003284072249589491,
'green cabbage': 0.0024630541871921183,
'pie crust': 0.0008210180623973727,
'ginger paste': 0.004105090311986864,
'fresh pineapple': 0.0024630541871921183,
'peanut oil': 0.004105090311986864,
'brown mustard seeds': 0.0024630541871921183,
'dri leav thyme': 0.0008210180623973727,
'white sandwich bread': 0.0008210180623973727,
'green tomatoes': 0.0024630541871921183,
'ham hock': 0.0008210180623973727,
'beef tenderloin': 0.0016420361247947454,
'sausage links': 0.0016420361247947454,
'dried black beans': 0.0008210180623973727,
'hungarian sweet paprika': 0.003284072249589491,
'dried fig': 0.0024630541871921183,
'diced onions': 0.003284072249589491,
'green peas': 0.0016420361247947454,
'tilapia': 0.0008210180623973727,
'white miso': 0.0024630541871921183,
'ancho chile pepper': 0.0016420361247947454,
'green chile': 0.0016420361247947454,
'celery seed': 0.0008210180623973727,
'diced celery': 0.0008210180623973727,
'unsalted cashews': 0.0008210180623973727,
'napa cabbage': 0.0024630541871921183,
'chinese sausage': 0.0008210180623973727,
'lemon grass': 0.0024630541871921183,
'ground mustard': 0.0016420361247947454,
'veggies': 0.0008210180623973727,
'cardamom seeds': 0.0024630541871921183,
'spanish paprika': 0.0024630541871921183,
'nonfat yogurt': 0.0016420361247947454,
'cilantro sprigs': 0.0008210180623973727,
'filet': 0.0008210180623973727,
'yellow mustard': 0.0016420361247947454,
'liquid': 0.0016420361247947454,
'pasta shells': 0.0016420361247947454,
'teas': 0.0024630541871921183,
'cream style corn': 0.0008210180623973727,
```

'orange rind': 0.0008210180623973727,
'cream sweeten whip': 0.0016420361247947454,
'pepper sauce': 0.0016420361247947454,
'beef rib short': 0.0008210180623973727,
'cooked brown rice': 0.0016420361247947454,
'fresh shiitake mushrooms': 0.0016420361247947454,
'lamb shoulder': 0.003284072249589491,
'extra firm tofu': 0.0016420361247947454,
'light soy sauce': 0.0008210180623973727,
'vegetable oil cooking spray': 0.0008210180623973727,
'frozen pastry puff sheets': 0.0016420361247947454,
'ground tumeric': 0.0008210180623973727,
'pumpkin': 0.0016420361247947454,
'mustard powder': 0.0016420361247947454,
'hot red pepper flakes': 0.0024630541871921183,
'hominy': 0.0008210180623973727,
'corn syrup': 0.0008210180623973727,
'tomato ketchup': 0.0016420361247947454,
'brown cardamom': 0.0008210180623973727,
'pie shell': 0.0016420361247947454,
'sweet italian sausage': 0.0008210180623973727,
'poppy seeds': 0.0016420361247947454,
'new potatoes': 0.003284072249589491,
'leg of lamb': 0.0049261083743842365,
'star anise': 0.0024630541871921183,
'green cardamom': 0.0008210180623973727,
'jeera': 0.0008210180623973727,
'vanilla ice cream': 0.0016420361247947454,
'lettuce': 0.0024630541871921183,
'shredded carrots': 0.0008210180623973727,
'quinoa': 0.0016420361247947454,
'mace': 0.0016420361247947454,
'cilantro stems': 0.0016420361247947454,
'fresh corn': 0.0016420361247947454,
'collard greens': 0.0016420361247947454,
'baking potatoes': 0.0024630541871921183,
'lobster': 0.0016420361247947454,
'fresh thyme leaves': 0.004105090311986864,
'whole allspice': 0.0016420361247947454,
'pork loin chops': 0.0008210180623973727,
'salad greens': 0.0008210180623973727,
'bacon slices': 0.0008210180623973727,
'broccoli': 0.0024630541871921183,
'ice water': 0.003284072249589491,
'chile de arbol': 0.0008210180623973727,
'ancho powder': 0.0008210180623973727,
'halibut fillets': 0.0024630541871921183,
'corn oil': 0.0008210180623973727,
'sweet soy sauce': 0.0016420361247947454,
'smoked salmon': 0.0024630541871921183,
'cauliflower florets': 0.0008210180623973727,
'bone-in chicken breast halves': 0.0016420361247947454,
'dry yeast': 0.0016420361247947454,
'pizza sauce': 0.0008210180623973727,
'shredded Monterey Jack cheese': 0.0008210180623973727,
'refrigerated pizza dough': 0.0008210180623973727,
'boneless skinless chicken': 0.0016420361247947454,
'chutney': 0.0008210180623973727,
'unsweetened coconut milk': 0.0008210180623973727,
'tilapia fillets': 0.004105090311986864,
'sirloin steak': 0.0016420361247947454,
'crab meat': 0.0008210180623973727,
'curry': 0.0016420361247947454,
'mustard greens': 0.0008210180623973727,
'coconut cream': 0.0024630541871921183,
'daikon': 0.0008210180623973727,
'ground chicken': 0.0016420361247947454,
'fruit': 0.0008210180623973727,
'fresh blueberries': 0.0008210180623973727,
'cooked quinoa': 0.0008210180623973727,
'dried mint flakes': 0.0016420361247947454,
'chicken wings': 0.004105090311986864,
'glutinous rice': 0.0008210180623973727,
'allspice': 0.0024630541871921183,
'low sodium soy sauce': 0.0008210180623973727,
'brie cheese': 0.0008210180623973727,
'beef sirloin': 0.0008210180623973727,
'pitted date': 0.0016420361247947454,
'dried sage': 0.0008210180623973727,
'pepper flakes': 0.0008210180623973727,
'wheat': 0.0008210180623973727

```
 'mixed spice': 0.003284072249589491,
 'chopped walnuts': 0.0008210180623973727,
 'dried currants': 0.0016420361247947454,
 'cognac': 0.004105090311986864,
 'toasted walnuts': 0.0008210180623973727,
 'top sirloin steak': 0.0008210180623973727,
 'pinto beans': 0.0008210180623973727,
 ...}
```

In [19]:
```
# Test cell: `ex3_ingredient_vector` (2 points)

def ex3_gen_soln__():
    from random import choice, sample, randrange, random
    from itertools import permutations
    def random_value():
        return round(random(), 2)
    nouns = {'tacos', 'prism', 'taxidermy', 'ennui', 'salvia', 'biodiesel', 'palo', 'dreamcatcher', 'listicle', 'sh
aman', 'humblebrag', 'tile', 'iphone', 'knausgaard', 'distillery', 'fanny', 'party', 'taiyaki', 'single-origin', 's
anto', 'batch', 'keffiyeh', 'gluten-free', 'fingerstache', 'pop-up', 'swag', 'chicken', 'art', 'helvetica', 'pack',
'fixie', 'subway', 'semiotics', 'plaid', 'coffee', 'kogi', 'twee', 'post-ironic', 'hot', 'bulb', 'narwhal'}
    a0 = choice(list(nouns))
    num_elems = randrange(1, min(10, len(nouns)))
    vec = {}
    rules = {}
    B = sample(list(nouns - {a0}), k=num_elems)
    for b in B:
        vec[b] = random_value()
        rules[(a0, b)] = vec[b]
    A = sample(list(nouns - {a0}), k=randrange(1, 5))
    for a in A:
        B = sample(list(nouns - {a} | {a0}), k=randrange(1, 5))
        for b in B:
            if (a, b) not in rules:
                rules[(a, b)] = random_value()
    return a0, vec, rules

def ex3_check_one__():
    a, vec, rules = ex3_gen_soln__()
    print("\n=== Test case ===\n")
    print(f"* a == '{a}'\n")
    print(f"* rules == {rules}\n")
    print(f"\n* Expected result == {vec}\n")
    your_vec = ingredient_vector(a, rules)
    assert vec == your_vec, \
           f"\n*** Failed ***\n* Your function returned {your_vec}, which is not expected."

for _ in range(10):
    ex3_check_one__()

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

```
=== Test case ===

* a == 'knausgaard'

* rules == {('knausgaard', 'salvia'): 0.34, ('knausgaard', 'iphone'): 0.75, ('knausgaard', 'taiyaki'): 0.76, ('kna
usgaard', 'distillery'): 0.18, ('knausgaard', 'pop-up'): 0.13, ('knausgaard', 'fixie'): 0.16, ('knausgaard', 'keff
iyeh'): 0.76, ('knausgaard', 'tacos'): 0.89, ('knausgaard', 'swag'): 0.88, ('hot', 'palo'): 0.37, ('hot', 'chicke
n'): 0.09, ('hot', 'art'): 0.09}


* Expected result == {'salvia': 0.34, 'iphone': 0.75, 'taiyaki': 0.76, 'distillery': 0.18, 'pop-up': 0.13, 'fixi
e': 0.16, 'keffiyeh': 0.76, 'tacos': 0.89, 'swag': 0.88}


=== Test case ===

* a == 'keffiyeh'

* rules == {('keffiyeh', 'batch'): 0.48, ('keffiyeh', 'twee'): 0.1, ('keffiyeh', 'gluten-free'): 0.6, ('keffiyeh',
'plaid'): 0.98, ('keffiyeh', 'ennui'): 0.94, ('keffiyeh', 'fixie'): 1.0, ('keffiyeh', 'tacos'): 0.49, ('ennui', 'p
alo'): 0.59, ('ennui', 'kogi'): 0.03, ('twee', 'prism'): 0.33, ('twee', 'narwhal'): 0.33, ('twee', 'semiotics'):
0.68, ('twee', 'shaman'): 0.66, ('knausgaard', 'humblebrag'): 0.56, ('knausgaard', 'taxidermy'): 0.58, ('knausgaar
d', 'listicle'): 0.75, ('knausgaard', 'iphone'): 0.97}


* Expected result == {'batch': 0.48, 'twee': 0.1, 'gluten-free': 0.6, 'plaid': 0.98, 'ennui': 0.94, 'fixie': 1.0,
'tacos': 0.49}
```

```
=== Test case ===

* a == 'listicle'

* rules == {('listicle', 'narwhal'): 0.74, ('listicle', 'pop-up'): 0.74, ('distillery', 'taxidermy'): 0.73}


* Expected result == {'narwhal': 0.74, 'pop-up': 0.74}


=== Test case ===

* a == 'palo'

* rules == {('palo', 'party'): 0.58, ('palo', 'fanny'): 0.65, ('palo', 'knausgaard'): 0.83, ('palo', 'subway'): 0.
27, ('palo', 'plaid'): 0.61, ('chicken', 'listicle'): 0.93, ('chicken', 'knausgaard'): 0.19, ('knausgaard', 'kog
i'): 0.68, ('knausgaard', 'listicle'): 0.11, ('knausgaard', 'chicken'): 0.44, ('knausgaard', 'fixie'): 0.73}


* Expected result == {'party': 0.58, 'fanny': 0.65, 'knausgaard': 0.83, 'subway': 0.27, 'plaid': 0.61}


=== Test case ===

* a == 'listicle'

* rules == {('listicle', 'taiyaki'): 0.55, ('listicle', 'twee'): 0.96, ('listicle', 'pack'): 0.1, ('listicle', 'ke
ffiyeh'): 0.94, ('listicle', 'knausgaard'): 0.43, ('listicle', 'single-origin'): 0.82, ('listicle', 'chicken'): 0.
16, ('listicle', 'party'): 0.14, ('fixie', 'ennui'): 0.11, ('fixie', 'chicken'): 0.17, ('post-ironic', 'distiller
y'): 0.68}


* Expected result == {'taiyaki': 0.55, 'twee': 0.96, 'pack': 0.1, 'keffiyeh': 0.94, 'knausgaard': 0.43, 'single-or
igin': 0.82, 'chicken': 0.16, 'party': 0.14}


=== Test case ===

* a == 'tacos'

* rules == {('tacos', 'plaid'): 0.26, ('tacos', 'party'): 0.29, ('biodiesel', 'single-origin'): 0.11, ('biodiese
l', 'hot'): 0.9, ('biodiesel', 'keffiyeh'): 0.45}


* Expected result == {'plaid': 0.26, 'party': 0.29}


=== Test case ===

* a == 'humblebrag'

* rules == {('humblebrag', 'swag'): 0.58, ('humblebrag', 'biodiesel'): 0.85, ('humblebrag', 'ennui'): 0.86, ('humb
lebrag', 'bulb'): 0.29, ('prism', 'party'): 0.11, ('chicken', 'listicle'): 0.23, ('chicken', 'semiotics'): 0.21,
('iphone', 'coffee'): 0.37}


* Expected result == {'swag': 0.58, 'biodiesel': 0.85, 'ennui': 0.86, 'bulb': 0.29}


=== Test case ===

* a == 'pack'

* rules == {('pack', 'pop-up'): 0.47, ('pack', 'keffiyeh'): 0.49, ('pack', 'dreamcatcher'): 0.32, ('pack', 'iphon
e'): 0.81, ('pack', 'helvetica'): 1.0, ('pack', 'twee'): 0.64, ('pack', 'hot'): 0.41, ('pack', 'taiyaki'): 0.52,
('tacos', 'iphone'): 0.66, ('tacos', 'art'): 0.39, ('tacos', 'twee'): 0.69, ('tile', 'salvia'): 0.93, ('tile', 'ta
cos'): 0.94, ('tile', 'palo'): 0.65, ('tile', 'ennui'): 0.69, ('bulb', 'fixie'): 0.88}


* Expected result == {'pop-up': 0.47, 'keffiyeh': 0.49, 'dreamcatcher': 0.32, 'iphone': 0.81, 'helvetica': 1.0, 't
wee': 0.64, 'hot': 0.41, 'taiyaki': 0.52}


=== Test case ===

* a == 'tacos'

* rules == {('tacos', 'post-ironic'): 1.0, ('tacos', 'party'): 0.4, ('tacos', 'knausgaard'): 0.83, ('tacos', 'helv
etica'): 0.45, ('tacos', 'coffee'): 0.14, ('pop-up', 'plaid'): 0.75, ('pop-up', 'bulb'): 0.65, ('pop-up', 'pris
m'): 0.75, ('batch', 'hot'): 0.31, ('batch', 'taiyaki'): 0.78, ('batch', 'gluten-free'): 0.65}
```

```
* Expected result == {'post-ironic': 1.0, 'party': 0.4, 'knausgaard': 0.83, 'helvetica': 0.45, 'coffee': 0.14}


=== Test case ===

* a == 'pack'

* rules == {('pack', 'bulb'): 0.93, ('pack', 'taiyaki'): 0.81, ('pack', 'taxidermy'): 0.31, ('palo', 'santo'): 0.1
4}


* Expected result == {'bulb': 0.93, 'taiyaki': 0.81, 'taxidermy': 0.31}


(Passed!)
```

Ingredient dot-product. Given two ingredient vectors, `x` and `y`, the _ingredient dot-product, is the sum of `x[a] * y[a]` for all ingredients `a` that appear in both vectors.

For example, suppose

```
x = {'milk': 0.2, 'eggs': 0.7, 'bread': 0.1, 'grape tomatoes': 0.33}
y = {'eggs': 0.3, 'lemon': 0.5, 'grape tomatoes': 0.1, 'dill': 0.8}
```

The two vectors have `'eggs'` and `'grape tomatoes'` in common. Therefore, their ingredient dot-product is $(0.7 * 0.3) + (0.33 * 0.1) = 0.243$.

Exercise 4 (2 points). Complete the function, `ingredient_dot(x, y)`, so that it computes the similarity between two ingredient vectors, `x` and `y`, per the definition above.

In [20]:
```
def ingredient_dot(x, y):
    ### BEGIN SOLUTION
    keys = set(x.keys()) | set(y.keys())
    return sum([x[a]*y[a] for a in keys if a in x and a in y])
    ### END SOLUTION
```

In [21]:
```
# Demo:
x = {'milk': 0.2, 'eggs': 0.7, 'bread': 0.1, 'grape tomatoes': 0.33}
y = {'eggs': 0.3, 'lemon': 0.5, 'grape tomatoes': 0.1, 'dill': 0.8}
print(ingredient_dot(x, y))
```

```
0.243
```

In [22]:
```
# Test cell: `ex4_ingredient_dot` (2 points)

def ex4_gen_soln__():
    from random import choice, sample, randrange, random
    from itertools import permutations
    def random_value():
        return round(random(), 2)
    nouns = {'tacos', 'prism', 'taxidermy', 'ennui', 'salvia', 'biodiesel', 'palo', 'dreamcatcher', 'listicle', 'sh
aman', 'humblebrag', 'tile', 'iphone', 'knausgaard', 'distillery', 'fanny', 'party', 'taiyaki', 'single-origin', 's
anto', 'batch', 'keffiyeh', 'gluten-free', 'fingerstache', 'pop-up', 'swag', 'chicken', 'art', 'helvetica', 'pack',
'fixie', 'subway', 'semiotics', 'plaid', 'coffee', 'kogi', 'twee', 'post-ironic', 'hot', 'bulb', 'narwhal'}
    x = {}
    y = {}
    s = 0
    num_common = randrange(0, len(nouns))
    B_common = set(sample(list(nouns), k=num_common))
    for b in B_common:
        x[b] = random_value()
        y[b] = random_value()
        s += x[b] * y[b]
    B_left = nouns - B_common
    num_extra = randrange(0, len(B_left))
    B_extra = sample(list(B_left), k=num_extra)
    for b in B_extra:
        if random() < 0.15:
            x[b] = random_value()
        elif random() < 0.15:
            y[b] = random_value()
    return x, y, s, B_common

def ex4_check_one__():
    x, y, s, common = ex4_gen_soln__()
```

```
    print("\n=== Test case ===\n")
    print(f"* Ingredient vector `x` == {x}\n")
    print(f"* Ingredient vector `y` == {y}\n")
    print(f"* Common keys == {common}")
    print(f"* Expected result == {s}\n")
    your_dot = ingredient_dot(x, y)
    if abs(s) > 0:
        rel_err = abs(your_dot - s) / abs(s)
        passed = rel_err <= (len(common) * 1e-14)
    else:
        passed = your_dot == 0.0
    assert passed, \
           f"*** Failed ***\nYour solution, {your_dot}, differs from the expected solution by more than what is exp
ected from roundoff error."

for _ in range(10):
    ex4_check_one__()

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

```
=== Test case ===

* Ingredient vector `x` == {'tile': 0.28, 'distillery': 0.29}

* Ingredient vector `y` == {}

* Common keys == set()
* Expected result == 0


=== Test case ===

* Ingredient vector `x` == {'pop-up': 0.12, 'fanny': 0.5, 'iphone': 0.62, 'santo': 0.63, 'palo': 0.85, 'twee': 0.1
4, 'humblebrag': 0.33, 'coffee': 0.36, 'fixie': 0.33, 'fingerstache': 0.95, 'hot': 0.37, 'taiyaki': 0.65, 'kogi':
0.26, 'pack': 0.59, 'swag': 0.66, 'biodiesel': 0.89, 'tacos': 0.87, 'plaid': 0.41, 'salvia': 0.23, 'semiotics': 0.
36, 'gluten-free': 0.53, 'distillery': 0.32, 'taxidermy': 0.02, 'art': 0.63, 'prism': 0.93, 'helvetica': 0.94}

* Ingredient vector `y` == {'pop-up': 0.85, 'fanny': 0.25, 'iphone': 0.75, 'santo': 0.46, 'palo': 0.69, 'twee': 0.
93, 'humblebrag': 0.45, 'coffee': 0.01, 'fixie': 0.77, 'fingerstache': 0.45, 'hot': 0.53, 'taiyaki': 0.47, 'kogi':
0.6, 'pack': 0.22, 'swag': 0.4, 'biodiesel': 0.84, 'tacos': 0.61, 'plaid': 0.53, 'salvia': 0.26, 'semiotics': 0.2
7, 'gluten-free': 0.67, 'distillery': 0.16, 'taxidermy': 0.64, 'art': 0.79, 'prism': 1.0}

* Common keys == {'pop-up', 'fanny', 'iphone', 'santo', 'palo', 'twee', 'humblebrag', 'coffee', 'fixie', 'fingerst
ache', 'hot', 'taiyaki', 'kogi', 'pack', 'swag', 'biodiesel', 'tacos', 'plaid', 'salvia', 'semiotics', 'gluten-fre
e', 'distillery', 'taxidermy', 'art', 'prism'}
* Expected result == 7.083


=== Test case ===

* Ingredient vector `x` == {'bulb': 0.55, 'listicle': 0.04, 'shaman': 0.61, 'pop-up': 0.46, 'fanny': 0.69, 'iphon
e': 0.8, 'santo': 0.5, 'palo': 0.52, 'helvetica': 0.51, 'post-ironic': 0.62, 'twee': 0.5, 'batch': 0.3, 'subway':
0.75, 'party': 0.76, 'coffee': 0.08, 'fingerstache': 0.21, 'hot': 0.52, 'taiyaki': 0.41, 'kogi': 0.62, 'pack': 0.3
1, 'ennui': 0.0, 'dreamcatcher': 0.36, 'swag': 0.3, 'keffiyeh': 0.42, 'biodiesel': 0.84, 'narwhal': 0.15, 'tacos':
0.87, 'plaid': 0.93, 'single-origin': 0.33, 'semiotics': 0.84, 'salvia': 0.69, 'knausgaard': 0.13, 'gluten-free':
0.76, 'taxidermy': 0.55, 'chicken': 0.75, 'art': 0.66, 'humblebrag': 0.8}

* Ingredient vector `y` == {'bulb': 0.4, 'listicle': 0.33, 'shaman': 0.19, 'pop-up': 0.89, 'fanny': 0.67, 'iphon
e': 0.73, 'santo': 0.63, 'palo': 0.67, 'helvetica': 0.07, 'post-ironic': 0.31, 'twee': 0.01, 'batch': 0.15, 'subwa
y': 0.98, 'party': 0.61, 'coffee': 0.41, 'fingerstache': 0.52, 'hot': 0.7, 'taiyaki': 0.44, 'kogi': 0.89, 'pack':
0.04, 'ennui': 0.77, 'dreamcatcher': 0.69, 'swag': 0.62, 'keffiyeh': 0.23, 'biodiesel': 0.22, 'narwhal': 0.7, 'tac
os': 0.3, 'plaid': 0.82, 'single-origin': 0.11, 'semiotics': 0.84, 'salvia': 0.66, 'knausgaard': 0.14, 'gluten-fre
e': 0.54, 'taxidermy': 0.49, 'chicken': 0.37, 'prism': 0.96}

* Common keys == {'bulb', 'listicle', 'shaman', 'pop-up', 'fanny', 'iphone', 'santo', 'palo', 'helvetica', 'post-i
ronic', 'twee', 'batch', 'subway', 'party', 'coffee', 'fingerstache', 'hot', 'taiyaki', 'kogi', 'pack', 'ennui',
'dreamcatcher', 'swag', 'keffiyeh', 'biodiesel', 'narwhal', 'tacos', 'plaid', 'single-origin', 'semiotics', 'salvi
a', 'knausgaard', 'gluten-free', 'taxidermy', 'chicken'}
* Expected result == 9.2126


=== Test case ===

* Ingredient vector `x` == {'shaman': 0.74, 'fanny': 0.12, 'tile': 0.15, 'iphone': 0.84, 'santo': 0.39, 'helvetic
a': 0.71, 'post-ironic': 0.81, 'batch': 0.64, 'subway': 0.22, 'humblebrag': 0.16, 'party': 0.88, 'fixie': 0.09, 'c
offee': 0.64, 'hot': 0.84, 'taiyaki': 0.06, 'kogi': 0.29, 'dreamcatcher': 0.76, 'ennui': 0.92, 'swag': 0.12, 'keff
iyeh': 0.36, 'biodiesel': 0.49, 'narwhal': 0.26, 'tacos': 0.29, 'plaid': 0.67, 'salvia': 0.58, 'art': 0.4, 'distil
lery': 0.44, 'taxidermy': 0.51, 'chicken': 0.25, 'prism': 0.82}

* Ingredient vector `y` == {'shaman': 0.59, 'fanny': 0.94, 'tile': 0.35, 'iphone': 0.79, 'santo': 0.3, 'helvetic
```

```
a': 0.79, 'post-ironic': 0.66, 'batch': 0.12, 'subway': 0.65, 'humblebrag': 0.03, 'party': 0.76, 'fixie': 0.44, 'c
offee': 0.06, 'hot': 0.84, 'taiyaki': 0.22, 'kogi': 0.82, 'dreamcatcher': 0.03, 'ennui': 0.97, 'swag': 0.85, 'keff
iyeh': 0.47, 'biodiesel': 0.33, 'narwhal': 0.13, 'tacos': 0.9, 'plaid': 0.02, 'salvia': 0.85, 'art': 0.53, 'distil
lery': 0.06, 'taxidermy': 0.5, 'chicken': 0.79, 'prism': 0.7}

* Common keys == {'shaman', 'fanny', 'tile', 'iphone', 'santo', 'helvetica', 'post-ironic', 'batch', 'subway', 'hu
mblebrag', 'party', 'fixie', 'coffee', 'hot', 'taiyaki', 'kogi', 'dreamcatcher', 'ennui', 'swag', 'keffiyeh', 'bio
diesel', 'narwhal', 'tacos', 'plaid', 'salvia', 'art', 'distillery', 'taxidermy', 'chicken', 'prism'}
* Expected result == 7.820200000000001


=== Test case ===

* Ingredient vector `x` == {'helvetica': 0.89, 'post-ironic': 0.23, 'narwhal': 0.72, 'hot': 0.04, 'semiotics': 0.3
4, 'bulb': 0.48, 'knausgaard': 0.31, 'gluten-free': 0.75, 'taxidermy': 0.21, 'art': 0.01, 'party': 0.4, 'shaman':
0.01, 'fanny': 0.97, 'ennui': 0.41, 'keffiyeh': 0.73, 'biodiesel': 0.43, 'tile': 0.63}

* Ingredient vector `y` == {'helvetica': 0.95, 'post-ironic': 0.36, 'narwhal': 0.17, 'hot': 0.76, 'semiotics': 0.0
4, 'bulb': 0.74, 'knausgaard': 0.68, 'gluten-free': 0.56, 'taxidermy': 0.14, 'art': 0.45, 'party': 0.88, 'shaman':
0.14, 'fanny': 0.22, 'ennui': 0.78, 'keffiyeh': 0.66, 'biodiesel': 0.81, 'tile': 0.02, 'fingerstache': 0.84, 'swa
g': 0.03}

* Common keys == {'helvetica', 'post-ironic', 'narwhal', 'hot', 'semiotics', 'bulb', 'knausgaard', 'gluten-free',
'taxidermy', 'art', 'party', 'shaman', 'fanny', 'ennui', 'keffiyeh', 'biodiesel', 'tile'}
* Expected result == 3.8438999999999997


=== Test case ===

* Ingredient vector `x` == {'tacos': 0.42, 'helvetica': 0.36, 'twee': 0.41, 'single-origin': 0.43, 'bulb': 0.52,
'batch': 0.62, 'salvia': 0.03, 'gluten-free': 0.18, 'distillery': 0.49, 'listicle': 0.34, 'humblebrag': 0.87, 'cof
fee': 0.05, 'swag': 0.89, 'fingerstache': 0.09, 'party': 0.39, 'kogi': 0.41}

* Ingredient vector `y` == {'tacos': 0.78, 'helvetica': 0.07, 'twee': 0.91, 'single-origin': 0.75, 'bulb': 0.35,
'batch': 0.68, 'salvia': 0.33, 'gluten-free': 0.58, 'distillery': 0.47, 'listicle': 0.23, 'humblebrag': 0.15, 'cof
fee': 0.66, 'swag': 0.53, 'fingerstache': 0.61, 'palo': 0.82, 'chicken': 0.23, 'knausgaard': 0.95}

* Common keys == {'tacos', 'helvetica', 'twee', 'single-origin', 'bulb', 'batch', 'salvia', 'gluten-free', 'distil
lery', 'listicle', 'humblebrag', 'coffee', 'swag', 'fingerstache'}
* Expected result == 2.7649


=== Test case ===

* Ingredient vector `x` == {'bulb': 0.54, 'shaman': 0.24, 'pop-up': 0.99, 'tile': 0.44, 'iphone': 0.39, 'santo':
0.82, 'palo': 0.9, 'helvetica': 0.98, 'post-ironic': 0.32, 'twee': 0.16, 'subway': 0.62, 'humblebrag': 0.47, 'part
y': 0.58, 'coffee': 0.89, 'fingerstache': 0.26, 'hot': 0.14, 'taiyaki': 0.26, 'dreamcatcher': 0.75, 'swag': 0.75,
'tacos': 0.78, 'plaid': 0.33, 'single-origin': 0.52, 'semiotics': 0.91, 'salvia': 0.76, 'art': 0.48}

* Ingredient vector `y` == {'bulb': 0.36, 'shaman': 0.87, 'pop-up': 0.03, 'tile': 0.39, 'iphone': 0.42, 'santo':
0.66, 'palo': 0.83, 'helvetica': 0.48, 'post-ironic': 0.76, 'twee': 0.33, 'subway': 0.53, 'humblebrag': 0.11, 'par
ty': 0.03, 'coffee': 0.26, 'fingerstache': 0.54, 'hot': 0.84, 'taiyaki': 0.7, 'dreamcatcher': 0.74, 'swag': 0.06,
'tacos': 0.68, 'plaid': 0.15, 'single-origin': 0.88, 'semiotics': 0.62, 'salvia': 0.78, 'art': 0.38}

* Common keys == {'bulb', 'shaman', 'pop-up', 'tile', 'iphone', 'santo', 'palo', 'helvetica', 'post-ironic', 'twe
e', 'subway', 'humblebrag', 'party', 'coffee', 'fingerstache', 'hot', 'taiyaki', 'dreamcatcher', 'swag', 'tacos',
'plaid', 'single-origin', 'semiotics', 'salvia', 'art'}
* Expected result == 6.868900000000001


=== Test case ===

* Ingredient vector `x` == {'twee': 0.08, 'art': 0.24, 'gluten-free': 0.42, 'shaman': 0.97, 'dreamcatcher': 0.53,
'biodiesel': 0.53, 'pack': 0.02, 'ennui': 0.97, 'fanny': 0.08, 'tacos': 0.35, 'post-ironic': 0.59, 'distillery':
0.2}

* Ingredient vector `y` == {'twee': 0.85, 'art': 0.04, 'gluten-free': 0.56, 'shaman': 0.41, 'dreamcatcher': 0.55,
'biodiesel': 0.44, 'party': 0.3}

* Common keys == {'twee', 'art', 'gluten-free', 'shaman', 'dreamcatcher', 'biodiesel'}
* Expected result == 1.2352


=== Test case ===

* Ingredient vector `x` == {'bulb': 0.37, 'shaman': 0.04, 'pop-up': 0.89, 'fanny': 0.63, 'iphone': 0.41, 'palo':
0.21, 'subway': 0.43, 'humblebrag': 0.15, 'party': 0.7, 'fixie': 0.99, 'coffee': 0.85, 'hot': 0.4, 'swag': 0.05,
'keffiyeh': 0.78, 'biodiesel': 0.92, 'narwhal': 0.36, 'tacos': 0.41, 'single-origin': 0.98, 'art': 0.38, 'knausgaa
rd': 0.06, 'distillery': 0.68, 'gluten-free': 0.45, 'prism': 0.11, 'salvia': 0.11, 'tile': 0.68}

* Ingredient vector `y` == {'bulb': 0.91, 'shaman': 0.02, 'pop-up': 0.51, 'fanny': 0.16, 'iphone': 0.88, 'palo':
```

```
  Ingredient vector `y` == {'bulb': 0.12, 'shaman': 0.12, 'pop-up': 0.13, 'fanny': 0.18, 'iphone': 0.08, 'palo':
0.14, 'subway': 0.88, 'humblebrag': 0.59, 'party': 0.05, 'fixie': 0.62, 'coffee': 1.0, 'hot': 0.18, 'swag': 0.0,
'keffiyeh': 0.19, 'biodiesel': 0.83, 'narwhal': 0.68, 'tacos': 0.13, 'single-origin': 0.06, 'art': 0.87, 'knausgaa
rd': 0.41, 'distillery': 0.53, 'gluten-free': 0.65, 'prism': 0.11, 'listicle': 0.5, 'helvetica': 0.94, 'post-ironi
c': 0.96}

* Common keys == {'bulb', 'shaman', 'pop-up', 'fanny', 'iphone', 'palo', 'subway', 'humblebrag', 'party', 'fixie',
'coffee', 'hot', 'swag', 'keffiyeh', 'biodiesel', 'narwhal', 'tacos', 'single-origin', 'art', 'knausgaard', 'disti
llery', 'gluten-free', 'prism'}
* Expected result == 5.609000000000001


=== Test case ===

* Ingredient vector `x` == {'listicle': 0.12, 'shaman': 0.57, 'pop-up': 0.6, 'helvetica': 0.78, 'post-ironic': 0.8
1, 'twee': 0.75, 'batch': 0.65, 'subway': 0.23, 'party': 0.87, 'coffee': 0.9, 'hot': 0.01, 'taiyaki': 0.71, 'pac
k': 0.72, 'ennui': 0.15, 'swag': 0.76, 'keffiyeh': 0.36, 'tacos': 0.22, 'single-origin': 0.43, 'salvia': 0.3, 'ar
t': 0.98, 'gluten-free': 0.14, 'distillery': 0.73, 'taxidermy': 0.1, 'semiotics': 0.72, 'chicken': 0.08, 'knausgaa
rd': 0.66, 'humblebrag': 0.14}

* Ingredient vector `y` == {'listicle': 0.23, 'shaman': 0.72, 'pop-up': 0.68, 'helvetica': 0.28, 'post-ironic': 0.
64, 'twee': 0.12, 'batch': 0.97, 'subway': 0.67, 'party': 0.07, 'coffee': 0.1, 'hot': 0.87, 'taiyaki': 0.95, 'pac
k': 0.95, 'ennui': 1.0, 'swag': 0.33, 'keffiyeh': 0.9, 'tacos': 0.97, 'single-origin': 0.63, 'salvia': 0.85, 'ar
t': 0.3, 'gluten-free': 0.72, 'distillery': 0.34, 'taxidermy': 0.76, 'semiotics': 0.6, 'chicken': 0.9, 'knausgaar
d': 0.63, 'santo': 0.86}

* Common keys == {'listicle', 'shaman', 'pop-up', 'helvetica', 'post-ironic', 'twee', 'batch', 'subway', 'party',
'coffee', 'hot', 'taiyaki', 'pack', 'ennui', 'swag', 'keffiyeh', 'tacos', 'single-origin', 'salvia', 'art', 'glute
n-free', 'distillery', 'taxidermy', 'semiotics', 'chicken', 'knausgaard'}
* Expected result == 7.0784


(Passed!)
```

Ingredient similarity. If the function above is working, then we can use it to compute *ingredient-similarity* using a formula called the *cosine similarity measure*, defined as follows and implementd in the code cell below.

$$\text{similarity}(x, y) = \frac{x^T y}{\|x\|_2 \|y\|_2}.$$

In [23]:
```
def ingredient_similarity(x, y):
    from math import sqrt
    num = ingredient_dot(x, y)
    den = sqrt(ingredient_dot(x, x) * ingredient_dot(y, y))
    return num / den if den > 0.0 else 0.0

# With a little luck, grape tomatoes are more similar to cherry tomatoes than, say, milk:
x = ingredient_vector('grape tomatoes', rules)
y0 = ingredient_vector('cherry tomatoes', rules)
s0 = ingredient_similarity(x, y0)
y1 = ingredient_vector('milk', rules)
s1 = ingredient_similarity(x, y1)
print(f"Similarity between 'grape tomatoes' and 'cherry tomatoes' is {s0}.")
print(f"Similarity between 'grape tomatoes' and 'milk' is {s1}.")
if s0 > s1:
    print(f"   (Phew! -- {s0} > {s1})")
else:
    print(f"   (Hmmm... {s0} <= {s1}?)")
```

```
Similarity between 'grape tomatoes' and 'cherry tomatoes' is 0.9622012981354561.
Similarity between 'grape tomatoes' and 'milk' is 0.5620521290245709.
   (Phew! -- 0.9622012981354561 > 0.5620521290245709)
```

Putting it all together: Suggesting a substitute ingredient

Start by reviewing the "high-level idea" from the beginning of this notebook. Then, complete Exercise 5, which implements it using a specific procedure that combines the various pieces from previous exercises.

Exercise 5 (2 points). Suppose you are cooking the recipe whose itemset is `ingredient_sets[i]`, but one of the ingredients, call it `a`, is missing. Here is a procedure to suggest a replacement.

1. Recall that for each ingredient set `ingredient_sets[i]`, we precomputed a list of the 5 closest ingredient sets to it. These are stored in the global list named `closest`.
2. For each `ingredient_sets[j]` that is among these 5 closest, create a set of all ingredients that are **not** already in `ingredient_sets[i]`. These are *replacement candidates*.
3. Return a list of the `k` candidate ingredients most similar to `a`, using ingredient-similarity as the measure. This list should be sorted in descending order

of similarity. Each element should be a pair (2-tuple) consisting of the ingredient name and its similarity score.

For example,

```
i = 0
a = 'grape tomatoes'
k = 4
print(find_substitute(ingredient_sets, i, a, rules, k))
```

will return

```
[('cherry tomatoes', 0.9622012981354563),
 ('red wine vinegar', 0.9153588672348227),
 ('roasted red peppers', 0.9080314448950159),
 ('pinenuts', 0.8833371085008624)]
```

In [24]:
```
def find_substitute(ingredient_sets, i, a, rules, k=1):
    ### BEGIN SOLUTION
    candidates = set()
    J = closest[i]
    for j in J:
        candidates |= ingredient_sets[j] - ingredient_sets[i]
    x = ingredient_vector(a, rules)
    sims = {}
    for b in candidates:
        y = ingredient_vector(b, rules)
        sims[b] = ingredient_similarity(x, y)
    return sorted(sims.items(), key=lambda x: x[1], reverse=True)[:k]
    ### END SOLUTION
```

In [25]:
```
# Demo 0: A specific recipe
print_ingredient_set(ingredient_sets[0])
find_substitute(ingredient_sets, 0, 'grape tomatoes', rules, k=4)
```

```
- black olives
- seasoning
- purple onion
- pepper
- garbanzo beans
- grape tomatoes
- garlic
- romaine lettuce
- feta cheese crumbles
```

Out[25]:
```
[('cherry tomatoes', 0.9622012981354561),
 ('red wine vinegar', 0.9153588672348213),
 ('roasted red peppers', 0.9080314448950132),
 ('pinenuts', 0.8833371085008564)]
```

In [26]:
```
# Demo 1: Random recipe
from random import randrange, choice
i = randrange(0, len(ingredient_sets)) # random ingredient itemset
print(f"`ingredient_sets[{i}]`:")
print_ingredient_set(ingredient_sets[i])
a = choice(list(ingredient_sets[i])) # random ingredient
print(f"\nFinding substitute for {a}...")
find_substitute(ingredient_sets, i, a, rules, k=4) # find 4 closest substitutes
```

```
`ingredient_sets[36609]`:
- roast beef deli meat
- swiss cheese
- garlic cloves
- olive oil
- artichoke hearts
- kalamata
- capers
- rolls
- salad greens
- balsamic vinegar
- fresh basil
- roasted red peppers
- roast breast of chicken

Finding substitute for roast beef deli meat...
```

Out[26]:
```
[('Italian bread', 0.0),
 ('red wine vinegar', 0.0),
 ('hot red pepper flakes', 0.0),
```

```
 ('bocconcini', 0.0)]
```

In [27]:
```
# Test cell: `ex5_find_substitute` (2 points)

print("""
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below. However, it does compare
against hashed solutions to obscure the true results.
""")

### BEGIN HIDDEN TESTS
def ex5_find_substitute_soln__(ingredient_sets, i, a, rules, k=1):
    candidates = set()
    J = closest[i]
    for j in J:
        candidates |= ingredient_sets[j] - ingredient_sets[i]
    x = ingredient_vector(a, rules)
    sims = {}
    for b in candidates:
        y = ingredient_vector(b, rules)
        sims[b] = ingredient_similarity(x, y)
    return sorted(sims.items(), key=lambda x: x[1], reverse=True)[:k]

def ex5_gen_soln__(itemsets, counts, rules, outfilename='recipes/ex5_soln.csv'):
    from os.path import isfile
    from problem_utils import make_hash
    from random import choice
    if not isfile(get_path(outfilename)):
        print(f"Generating solutions file, '{outfilename}'...")
        solns = []
        with open(get_path(outfilename), 'wt') as fp:
            k = 0
            while k < 100:
                if k % 10 == 0: print(f"k={k}...")
                i = randrange(0, len(itemsets)) # random ingredient itemset
                freq = [g for g in itemsets[i] if counts[g] >= 25]
                if not freq: continue
                a = choice(freq) # random ingredient
                soln = ex5_find_substitute_soln__(itemsets, i, a, rules, k=3)
                solns.append(soln)
                fp.write(f"{i},{a}")
                for s, _ in soln:
                    s_hashed = make_hash(s)
                    fp.write(f",{s_hashed}")
                fp.write("\n")
                k += 1
        print("\nDone!")
        return solns
    print(f"An existing solutions file, '{outfilename}', is available.")
    return None

solns = ex5_gen_soln__(ingredient_sets, ingredient_counts, rules)
### END HIDDEN TESTS

def ex5_check(ingredient_sets, rules, num_cases=5, infile='recipes/ex5_soln.csv'):
    from problem_utils import make_hash
    from random import sample
    print(f"==> Checking {num_cases} random test cases...")
    with open(get_path(infile), 'rt') as fp:
        lines = fp.readlines()
        for t, line in enumerate(sample(lines, k=num_cases)):
            raw_fields = line.strip().split(',')
            assert len(raw_fields) >= 5
            i = int(raw_fields[0])
            print(f"\n=== Test case #{t} / {num_cases}: `ingredient_sets[{i}]` ===")
            print_ingredient_set(ingredient_sets[i])

            a = raw_fields[1]
            print(f"\nFinding top-3 substitutes for '{a}' ...")
            top_3_hashed = raw_fields[2:5]

            top_3 = find_substitute(ingredient_sets, i, a, rules, k=3)
            print("==> Found:", top_3)
            for j, (b, s) in enumerate(top_3):
                b_hashed = make_hash(b)
                assert b_hashed == top_3_hashed[j], \
                       f"*** Mismatch: Your #{j} item, '{b}' ({b_hashed})," \
                       f" does not match our expected value ({top_3_hashed[j]})."

ex5_check(ingredient_sets, rules)
```

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

```
This test cell is marked as having a hidden test, but does not.
The testing code is exposed, below. However, it does compare
against hashed solutions to obscure the true results.

An existing solutions file, 'recipes/ex5_soln.csv', is available.
==> Checking 5 random test cases...

=== Test case #0 / 5: `ingredient_sets[9441]` ===
- buckwheat flour
- whole milk
- melted butter
- all-purpose flour
- large eggs

Finding top-3 substitutes for 'whole milk' ...
==> Found: [('heavy cream', 0.8963732847912539), ('crème fraîche', 0.8914033893910396), ('unsalted butter', 0.8896
6919118446)]

=== Test case #1 / 5: `ingredient_sets[29576]` ===
- chicken breasts
- garlic cloves
- pepper
- unsalted butter
- olive oil
- chopped fresh thyme
- salt
- shallots
- all-purpose flour
- champagne
- whipping cream
- shiitake

Finding top-3 substitutes for 'pepper' ...
==> Found: [('black pepper', 0.964088919588581), ('ground black pepper', 0.9347034879288103), ('tomatoes', 0.90232
29089216215)]

=== Test case #2 / 5: `ingredient_sets[21751]` ===
- purple onion
- red wine vinegar
- olive oil
- skinless mahi mahi fillets
- cherry tomatoes
- tapenade

Finding top-3 substitutes for 'purple onion' ...
==> Found: [('black pepper', 0.8553308633722598), ('cucumber', 0.8542438949454906), ('ground black pepper', 0.8506
753446890792)]

=== Test case #3 / 5: `ingredient_sets[33737]` ===
- cold water
- boneless skinless chicken breasts
- peanuts
- vinegar
- zucchini
- cooked rice
- minced garlic
- chile paste
- green onions
- red bell pepper
- water
- corn starch
- brown sugar
- soy sauce

Finding top-3 substitutes for 'cold water' ...
==> Found: [('white vinegar', 0.8991532244637538), ('honey', 0.8960993446719581), ('sugar', 0.8912828483619002)]

=== Test case #4 / 5: `ingredient_sets[4037]` ===
- frisee
- onions
- ham
- heavy cream
- red wine vinegar
- pernod
- large garlic cloves
- salt
- freshly ground pepper
- walnuts
```

```
  walnuts
- dried currants
- dijon mustard

Finding top-3 substitutes for 'red wine vinegar' ...
==> Found: [('white wine vinegar', 0.9528132942430169), ('fresh lemon juice', 0.9216988761782173), ('capers', 0.91
69394159068549)]

(Passed!)
```

So how did we do? It's not a perfect algorithm by any stretch of the imagination. There are several tuning parameters, which you'd need to play with, and we've ignored an important component of the data (namely, the cuisine type). But we hope you'll agree that, if you made it this far, it's not bad for just a few weeks into a data analysis course!

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!

Problem 2: CIA Foreign Intelligence Mission

Version 1.1

Mission Brief

1. Situation:

- National Security Agency (NSA) just shared the signals intelligence (SIGINT) with us suggesting that new secret nuclear test sites are under construction in North Korea.
- You, an Operations Officer in an undisclosed location, just acquired a cryptic document from your informant that possibly includes information about the new test sites.

2. Mission:

- You need to send this document securely to headquarters (HQ) in Langley as soon as possible for immediate analysis and confirmation.
- Provide the analyst at HQ with a decoder for your secure transmission.

3. Procedure (total 10 pts):

- Load the cryptic document to your mission terminal (2 pts)
- Clean your loaded text for secure transmission (2 pts)
- Encode the text and send it to the analyst in Langley (3 pts)
- In a separate message, send the decoder for the secure message (3 pts)

"Break a leg, Officer!"

Exercise 0 (2 points). After the secret rendevous, you're back in the safe house. The case number assigned for this mission is 8754. Open and read `case8754.txt`. The contents of `case8754.txt` may contain punctuation characters as well as alphanumeric characters. Save the text in a variable named `document`. Print it to examine.

> *Hint*: In Python, you don't need to import a library in order to read and write files. https://www.pythonforbeginners.com/files/reading-and-writing-files-in-python (https://www.pythonforbeginners.com/files/reading-and-writing-files-in-python).

In [1]:
```
### BEGIN SOLUTION
document = open('case8754.txt').read()
print (document)
len(document)
### END SOLUTION
```

```
we123 are @!planning123785 t1343he lau345nch of proj\\[]ect rendezv___ous for./ early next w57643eek from are1933a
fifty two over the te097sts have proce49-2eded on sched^&**ule over@#$# no outside interf____erence yet over the p
lan is on ov9734er and ou===t
```

Out[1]: 258

In [2]:
```
# TEST CELL: Exercise 0

assert type(document) is str
assert len(document) == 258

def check_hash(doc, key):
    from hashlib import md5
    doc_hash = md5(doc.encode()).hexdigest()
    assert doc_hash == key, "Your document does not have the correct contents."

check_hash(document, 'e27267ba0a5c5edce43816fd86112d8a')

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

```
(Passed!)
```

Exercise 1 (2 points). Your informant had to make the `document` messy in order to pass the security check. S/he told you that only white spaces and alphabetic characters (regardless of capitalization) carry meaningful information.

Now, clean the `document`. Complete the following function, `clean_string(s)`, that returns a new string where any alphabetic characters are converted to lowercase, any whitespace is preserved as-is, and any other character is removed.

In [3]:
```
def clean_string(s):
    assert type (s) is str
    ### BEGIN SOLUTION
    new_doc = [c for c in s.lower() if c.isalpha() or c.isspace()]
    return ''.join(new_doc)
    ### END SOLUTION
```

```
## Let's see the result
print("Before: ", document, "\nAfter: ", clean_string(document) )
```

```
Before:  we123 are @!planning123785 t1343he lau345nch of proj\\[]ect rendezv___ous for./ early next w57643eek from
are1933a fifty two over the te097sts have proce49-2eded on sched^&**ule over@#$# no outside interf____erence yet o
ver the plan is on ov9734er and ou===t
After:  we are planning the launch of project rendezvous for early next week from area fifty two over the tests ha
ve proceeded on schedule over no outside interference yet over the plan is on over and out
```

In [4]:
```
# TEST CELL: Exercise 1

clean_doc = clean_string(document)

assert type(clean_doc) is str
assert len(clean_doc) == 196
assert all([c.isalpha() or c.isspace() for c in clean_doc])
assert clean_doc == clean_doc.lower()

def check_hash(doc, key):
    from hashlib import md5
    doc_hash = md5(doc.encode()).hexdigest()
    assert doc_hash == key, "Your document does not have the correct contents."

check_hash(clean_doc, 'c8d4ad008081b97ba6ce2ddb1bb5a070')

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

```
(Passed!)
```

"Read this carefully before you proceed to the next steps!"

Introduction to ASCII and Ciphers

- The ASCII system provides a way to map characters to a numerical value. In this case, we are only concerned with lowercase characters from a-z (the characters in your cleaned string). We are not concerned with encoding spaces in this problem. The full ASCII table can be found here: http://www.asciitable.com/ (http://www.asciitable.com/). The ASCII code for the character `'a'` is 97 and the ASCII code for the character `'z'` is 122, with the other letters falling in between those values. You can use the Python function `ord(c)` to convert a character `c` to its ASCII representation.

In [5]:
```
ord('a')
```

Out[5]: 97

In [6]:
```
ord('z')
```

Out[6]: 122

`chr()`. The `chr(x)` function converts an ASCII integer value `x` to its corresponding ASCII symbol. For instance, `chr(ord('a'))` would return `'a'`, so this provides a mapping from numbers into characters.

In [7]:
```
chr(97)
```

Out[7]: 'a'

In [8]:
```
chr(ord('a'))
```

Out[8]: 'a'

In [9]:
```
chr(122)
```

Out[9]: 'z'

Index ciphers. In the next exercise, you will consider an *index cipher*, which is a method for encoding or encrypting a message. Here is how a simple index cipher works.

We encode **one word at a time.** For each index `i` in a given word, the encoded character at that index is the ASCII representation of [the ASCII value of the original character **plus** `i`]. The index then resets to zero for the next word. Remember, each character is mapped to an integer value in the ASCII system.

> **Example.** Consider the word, `"abc"`. The letter `'a'` is at index 0, `'b'` is at index 1, and 'c' is at index 2. We want to start by converting each character to its ASCII representation, so `abc` becomes `[97, 98, 99]`. We then add the index of each character to its respective ASCII value,

> `[97+0, 98+1, 99+2]`. Finally, we convert those sums back into character values, giving us `[97, 99, 101]` becoming `"ace"` as the encoded representation. In this case, the character associated with ASCII value `99` is `'c'`, so the second index of our encoded document would be `'c'`. Putting it all together, the encoded version of `"abc"` is `"ace"`.

IMPORTANT: We are using a circular looping system to make sure that we only deal with the characters `'a'` through `'z'`. Characters with higher or lower ASCII values include brackets and other special characters, and we do not want to deal with those. Instead, if the encoding of a character would go beyond `'z'`, we will wraparound.

More specifically, suppose the encoded value of a character *without* wraparound is more than `ord('z') = 122`. Then, in this case, use the following formula to calculate the encoded value instead:

```
encoded_value = 96 + (ascii_value_of_unencoded_character - 122 + index_of_unencoded_character)
```

Converting this value back to a character will result in something between `'a'` and `'z'`, inclusive.

For instance, consider the string `'xyz'`. The letter `'z'` is at index 2, so its encoded value will be `ord('z')+2 == 122+2 = 124`, which is greater than 122. Therefore, we should instead encode it as the ASCII value of `96 + (122 - 122 + 2) == 98`, which corresponds to the character `'b'`.

Make sure you use this formula **ONLY** when you would end up with a ASCII value larger than 122 (`'z'`) after adding the index value.

"Got it? Let's jump back into the mission!"

Exercise 2 (3 points). You now need to encode the `document` so important information is not intercepted upon transmission to the Langley HQ.

Complete the following function, `encode_document(s)`, so that it takes an *already cleaned* document `s` as input and returns that document encoded using the index cipher scheme. Your encoded document should not have any leading or trailing whitespace.

NOTE: Do not encode spaces, as they are not sensitive information (when a space is found in the cleaned document, just add it unchanged to the encoded document and proceed to the next character). Alternatively, you can split the cleaned document on spaces and proceed that way, adding a space manually after each encoded word. For example, if `clean_doc == "abc def"`, the encoded version should be `"ace dfh"`.

In [10]:
```
def encode_document(clean_doc):
    assert type(clean_doc) is str
    ### BEGIN SOLUTION
    encoded_doc = ''
    for word in clean_doc.split(' '):
        encoded_word = ''
        for i in range(len(word)):
            old_char = ord(word[i])
            if old_char + i > 122:
                new_char = chr(96 + (old_char - 122 + i))
            else:
                new_char = chr(old_char + i)
            encoded_word += new_char
        encoded_doc += encoded_word + ' '
    return encoded_doc.strip()
    ### END SOLUTION

## Check the result of your encoding
clean_doc = clean_string(document)
print(f"=== Original document ===\n{clean_doc}")

encoded_clean_doc = encode_document(clean_doc)
print(f"\n=== Encoded document ===\n{encoded_clean_doc}")
```

```
=== Original document ===
we are planning the launch of project rendezvous for early next week from area fifty two over the tests have proce
eded on schedule over no outside interference yet over the plan is on over and out

=== Encoded document ===
wf asg pmcqrntn tig lbwqgm og psqmihz rfpgiebvcb fpt ebtoc nfzw wfgn fsqp asgd fjhwc txq owgu tig tfuww hbxh psqfi
jjll oo sdjhhzrl owgu np ovvvmik iovhvkkymwmp yfv owgu tig pmcq it oo owgu aof ovv
```

In [11]:
```
# TEST CELL: Exercise 2
encoded_doc = encode_document(clean_doc)
assert type(encoded_doc) is str
assert len(encoded_doc) == 196

def check_hash(doc, key):
    from hashlib import md5
    doc_hash = md5(doc.encode()).hexdigest()
    assert doc_hash == key, "Your document does not have the correct contents."
```

```
check_hash(encoded_doc, 'b0e9d9e15a99670a4b35f6456f34482e')
print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 3 (3 points). Now, create a **decoder** function that you can send to HQ in a separate transmission to decode the encoded document and deal with a possible threat!

Complete the following function `decode_document(s)`, that takes an encoded document as an argument and returns that document decoded to its original message.

In [12]:
```
encoded_doc = encode_document(clean_doc)

def decode_document(encoded_doc):
    assert type(encoded_doc) is str
    ### BEGIN SOLUTION
    decoded_doc = ''
    for word in encoded_doc.split():
        decoded_word = ''
        for i in range(len(word)):
            encoded_char = ord(word[i])
            if encoded_char - i < 97:
                decoded_char = chr(123 + (encoded_char - 97) - i)
            else:
                decoded_char = chr(encoded_char - i)
            decoded_word += decoded_char
        decoded_doc += decoded_word + ' '
    return decoded_doc.strip()
    ### END SOLUTION

## Check the result of your encoding
## This should equal your original cleaned document!
clean_doc = clean_string(document)
print(f"=== Original document ===\n{clean_doc}")

encoded_clean_doc = encode_document(clean_doc)
print(f"\n=== Encoded document ===\n{encoded_clean_doc}")

decoded_clean_doc = decode_document(encoded_clean_doc)
print(f"\n=== Decoded document ===\n{decoded_clean_doc}")
```

```
=== Original document ===
we are planning the launch of project rendezvous for early next week from area fifty two over the tests have proce
eded on schedule over no outside interference yet over the plan is on over and out

=== Encoded document ===
wf asg pmcqrntn tig lbwqgm og psqmihz rfpgiebvcb fpt ebtoc nfzw wfgn fsqp asgd fjhwc txq owgu tig tfuww hbxh psqfi
jjll oo sdjhhzrl owgu np ovvvmik iovhvkkymwmp yfv owgu tig pmcq it oo owgu aof ovv

=== Decoded document ===
we are planning the launch of project rendezvous for early next week from area fifty two over the tests have proce
eded on schedule over no outside interference yet over the plan is on over and out
```

In [13]:
```
# TEST CELL: Exercise 3
decoded_doc = decode_document(encoded_doc)

assert type(decoded_doc) is str
assert len(decoded_doc) == 196

def check_hash(doc, key):
    from hashlib import md5
    doc_hash = md5(doc.encode()).hexdigest()
    assert doc_hash == key, "Your document does not have the correct contents."

check_hash(decoded_doc, 'c8d4ad008081b97ba6ce2ddb1bb5a070')
print("\n(Passed!)")
```

```
(Passed!)
```

"Congratulations! You have been invaluable assistance to the Directorate of Operations!"

Fin! Remember to test your solutions by running them as the autograder will: restart the kernel and run all cells from "top-to-bottom." Also remember to submit

to the autograder; otherwise, you will not get credit for your hard work!

Problem 3: Gaussian Naïve Bayes Classification for Predicting Protein Localization Sites

Version 1.3

This notebook concerns a machine learning method called the *(Gaussian) naïve Bayes classifier*. It finds uses in text categorization, spam filters, and biomedicine.

In this problem, you'll apply naïve Bayes to the problem of predicting protein localization sites in E.Coli bacteria (https://en.wikipedia.org/wiki/Escherichia_coli). In essence, the problem is to estimate where particular proteins reside in a cell. If, later on, you want to learn more, see the references at the end of this notebook.

This notebook asks you to implement the method using only constructs from basic Python, without auxiliary libraries like `numpy` or `sklearn`.

Setup and data

You will use a modified version of a publicly available dataset (https://archive.ics.uci.edu/ml/datasets/ecoli). Run the code cell below to load this data, whose output we'll explain afterwards.

In [1]:
```
import math
from problem_utils import load_data
from pprint import pprint # For pretty-printing Python data structures

x_train, y_train, x_test, y_test = load_data('ecoli-mod.mat')

print ("\nTraining data for first 5 samples:")
pprint(x_train[:5], width=100)
print ("\nTraining results for first 5 samples:")
print(y_train[:5])
print(f"\nThe possible class labels are: {set(y_train + y_test)}")
```

```
Loading dataset: ecoli-mod.mat...

Some information about the given data:
218 total training samples having 5 features each
109 total testing samples having 5 features each

Training data for first 5 samples:
[{'feature_1': 0.44, 'feature_2': 0.28, 'feature_3': 0.43, 'feature_4': 0.27, 'feature_5': 0.37},
 {'feature_1': 0.31, 'feature_2': 0.36, 'feature_3': 0.58, 'feature_4': 0.94, 'feature_5': 0.94},
 {'feature_1': 0.58, 'feature_2': 0.55, 'feature_3': 0.57, 'feature_4': 0.7, 'feature_5': 0.74},
 {'feature_1': 0.38, 'feature_2': 0.44, 'feature_3': 0.43, 'feature_4': 0.2, 'feature_5': 0.31},
 {'feature_1': 0.29, 'feature_2': 0.28, 'feature_3': 0.5, 'feature_4': 0.42, 'feature_5': 0.5}]

Training results for first 5 samples:
['class_1', 'class_2', 'class_2', 'class_1', 'class_1']

The possible class labels are: {'class_3', 'class_1', 'class_2', 'class_5', 'class_4'}
```

About these data. The data is split into *training data*, which you'll use to build a predictive model, and *testing data*, which you'll use to test the accuracy of the model. There are four variables of interest: `x_train` and `y_train`, which hold the training data, and `x_test` and `y_test`, which hold the testing data. More specifically:

- `x_train` is a *list of dictionaries*. Each element `x_train[i]` is the `i`-th data point of the training set. The point is represented by a *feature vector*, which is a linear algebraic vector having five components. Each vector is stored as a dictionary, with its components named by the keys `'feature_1'`

through `'feature_5'`.

- `y_train` is a *list of strings*. Each element `y_train[i]` is a *class label* for the `i`-th data point. Observe from the output above that there are five possible class labels, `'class_1'` through `'class_5'`.
- `x_test` and `y_test` are similar to the above, except that they hold values for the test data. Our goal is to build a model of the training data that can closely predict the true class labels, `y_test`, given only the feature vectors in `x_test`.

Background on Bayes' Theorem

Recall *Bayes' theorem* (or Bayes' law or Bayes' rule) from Topic 3. It is a statement about the quantitative relationships among the conditional probablilities (https://stats.stackexchange.com/questions/239014/bayes-theorem-intuition) (recall Notebook 2) of several events. One common use of Bayes' theorem is to "reverse" a conditional relationship, such as estimating the probability that rich people are happy (https://www.quora.com/What-is-an-intuitive-explanation-of-Bayes-Rule) given knowledge of the probability that a happy person is rich. Another use is to update one's belief (https://arbital.com/p/bayes_rule/?l=1zq) using prior knowledge when new information arrives, like updating the probability that a person has cancer when he or she now tests positive, knowing some background information on the accuracy of the test.

The mathematical statement of Bayes' theorem is

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)},$$

where A and B are events and $P(B) \neq 0$.

- $P(A \mid B)$ is a posterior probability of event A given event B, or just **posterior**.
- $P(B \mid A)$ is a conditional probability of event B given event A, also called **likelihood**.
- $P(A)$ is prior probability of event A independently of B, also called just **prior**.
- $P(B)$ is the probabilities of observing B independently A, also called marginal likelihood or model **evidence**.

In words, the formula would be

$$\text{posterior } = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}}.$$

You can read more about Bayes' Theorem on its Wiki page (https://en.wikipedia.org/wiki/Bayes%27_theorem).

Part 0: The Naïve Bayes Model

We can use Bayes' theorem to make predictions not only for toy problems with two events, but also for much more complex problems, such as multinomial classification for data with multiple features. However, finding the likelihood of a multidimensional feature vector given the assigned class can be very hard and even intractable (https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Probabilistic_model). However, we can greatly simplify the computation if we make a *naïve* assumption that all the features are conditionally independent. In this case, the probability of observing a class label for a given feature vector becomes

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\ \prod_{i=1}^{n} p(x_i \mid C_k)}{p(\mathbf{x})}$$

Moreover, since evidence $p(\mathbf{x})$ does not depend on C, in practice we can omit it and use the formula

$$p(C_k \mid \mathbf{x}) \propto p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k) \tag{1}$$

where $\propto$ denotes proportionality.

Learning class priors. Before moving on, let's write a function that computes class priors, $p(C_k)$ using the vector of class assignments. Recall that `y_train` holds the class labels.

Exercise 0 (2 points). Write a function `prior(data)` that takes a list of class values as inputs and returns a dictionary containing the priors. In this dictionary, the class names serve as keys and the probability of the occurrence of each class as values.

We can find prior of class K simply dividing number of data samples of class K by total number of data samples. For example, if

```
data = ['class_1' , 'class_2' , 'class_3' , 'class_4' , 'class_3' , 'class_2' , 'class_1' , 'class_3' ]
```

then your function should produce the output,

```
prior(data) == {'class_1': 0.25,'class_2': 0.25,'class_3': 0.375, 'class_4': 0.125 }
```

In [2]:
```
def prior(data):
    assert isinstance(data, list), f"Input `data` has type `{type(data)}`, which does not derive from `list` as exp
ected."
    ### BEGIN SOLUTION
```

```
    ### BEGIN SOLUTION
    from collections import Counter
    count = Counter (data)
    return {k:v/sum(count.values()) for k,v in count.items()}
    ### END SOLUTION
```

In [3]:
```
# Test cell: `test_prior_script`

#Test Case 1
test_data = ['a1' , 'a2' , 'b1' , 'b2' , 'b1' , 'a2' , 'a1' , 'b1' ]
p1 = prior(test_data)
p1_test = {'a1': 0.25,'a2': 0.25,'b1': 0.375, 'b2': 0.125 }
assert isinstance(p1, dict), f"Input `p1` has type `{type(p1)}`, which does not derive from `dict` as expected."
assert p1 == p1_test, f"The Output for Test Case 1 is incorrect, returned: \n{p1}\n, should be: \n{p1_test}\n"

#Test Case 2
test_data2 = ['a1' , 'a2' , 'b1' , 'b2' , 'b1' , 'a2' , 'a1' , 'b1' , 'a1' , 'a2' , 'b1' , 'b2' , 'b1' , 'a2' , 'a
1' , 'b1' , 'a2' , 'a2' , 'b1' , 'b2' , 'b1' , 'c1' , 'a1' , 'b2' , 'a1' , 'b2' , 'b1' , 'c2' , 'b1' , 'a2' , 'a3'
, 'b1',  'a3' , 'a2' , 'b4' , 'b2' , 'b1' , 'a2' , 'a1' , 'b1' , 'a3' , 'a2' , 'b1' , 'b2' , 'b1' , 'a2' , 'a3' ,
'b2' ]
p2 = prior(test_data2)
p2_test = {'a1': 0.14583333, 'a2': 0.22916667, 'b1': 0.3125, 'b2': 0.16666667, 'c1': 0.02083333, 'c2': 0.02083333,
'a3': 0.08333333, 'b4': 0.02083333}
assert isinstance(p2, dict), f"Input `p2` has type `{type(p2)}`, which does not derive from `dict` as expected."
for k in p2_test.keys():
    assert math.isclose(p2[k], p2_test[k], abs_tol=1e-8), f"The Output for Test Case 2 is incorrect, returned: \n{p
2[k]:.8f}\n, should be: \n{p2_test[k]}\n"

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

```
(Passed!)
```

Assuming your implementation really is correct, run the code below to inspect the priors for our dataset.

In [4]:
```
prior_val = prior(y_train)
print("\nClass Priors:")
print(prior_val)
```

```
Class Priors:
{'class_1': 0.4724770642201835, 'class_2': 0.23394495412844038, 'class_4': 0.10091743119266056, 'class_3': 0.14220
183486238533, 'class_5': 0.05045871559633028}
```

Part 1: The Gaussian Naïve Bayes model

In real problems we often do not know the true underlying data distributions. Instead, we estimate them from the data, often assuming the form (but not the parameters) of the data distribution.

When dealing with continuous data, a typical assumption is that any continuous values associated with each class are distributed according to a normal (or Gaussian) distribution (https://en.wikipedia.org/wiki/Normal_distribution). Under this assumption, the likelihood is

$$p(x = f_i \mid C_k) = N(f_i; \mu_{i,k}, \sigma^2_{i,k})$$

$$p(x = f_i \mid C_k) = \frac{1}{\sqrt{2\pi\sigma^2_{i,k}}} \exp\left(-\frac{(f_i - \mu_{i,k})^2}{2\sigma^2_{i,k}}\right) \tag{2}$$

where $\mu_{i,k}$ is the mean of feature f_i and $\sigma^2_{i,k}$ is its variance.

Exercise 1.a (1 point). Write two functions, `mean(vector)` and `var(vector)`, to compute the mean and variance of the components of an input vector, respectively. The vector, `vector`, is given as a list of values and you need to return the mean and variance of those values.

Recall that the mean and variance are defined by

$$\mathbf{mean}(\mathbf{x}) = \mu(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \mathbf{var}(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2.$$

For example, suppose `vector = [1, 2, 3]`. Then,

```
mean([1, 2, 3]) = 2.0
var([1, 2, 3]) = 0.66666667
```

In the context of our problem, these functions would be helpful while implementing equation (2).

In [5]:
```
def mean(vector):
    ### BEGIN SOLUTION
```

```
    return sum(vector) / len(vector)
    ### END SOLUTION

def var(vector):
    ### BEGIN SOLUTION
    avg = mean(vector)
    return sum([(x - avg)**2 for x in vector]) / len(vector)
    ### END SOLUTION
```

In [6]:
```
# Test cell: `test_mean_var`

#Test Case 1
l = [1.0 , 5.272 , 6.2734 , 32.4824 , 8.2876 , 43.3242 ]
m = mean(l)
v = var(l)
assert isinstance(m, float), f"Input `m` has type `{type(m)}`, which does not derive from `float` as expected."
assert isinstance(v, float), f"Input `v` has type `{type(v)}`, which does not derive from `float` as expected."
assert math.isclose(m, 16.1066, abs_tol=1e-8), f"Test Case 1 for Mean Failed as `{m:.8f}` != 16.1066"
assert math.isclose(v, 252.06517989, abs_tol=1e-8), f"Test Case 1 for Variance Failed as `{v:.8f}` != 252.06517989"

#Test Case 2
l1 = [11.0423 , 6.34324 , 7.347234 , 426.244 , 247. , 232.4332476 , 9. , 9. ,9. , -3.432 , 0. , -34.234 ]
m1 = mean(l1)
v1 = var(l1)
assert isinstance(m1, float), f"Input `m` has type `{type(m1)}`, which does not derive from `float` as expected."
assert isinstance(v1, float), f"Input `v` has type `{type(v1)}`, which does not derive from `float` as expected."
assert math.isclose(m1, 76.64533513, abs_tol=1e-8), f"Test Case 2 for Mean Failed as `{m1:.8f}` != 76.64533513"
assert math.isclose(v1, 18988.91413866, abs_tol=1e-8), f"Test Case 2 for Variance Failed as `{v1:.8f}` != 18988.914
13866"

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

```
(Passed!)
```

Exercise 1.b (2 points). Using `mean(l)` and `var(l)`, implement a function, `likelihood(x, y)`, to compute the parameters of the Gaussian likelihood of the data, `x` and `y`.

In particular, the input `x` is a list of dictionaries containing feature vectors (e.g., `x_train` from the training data) and `y` is a list of class labels (e.g., `y_train`). The output is a *pair of dictionaries of dictionaries* (yikes!!), one holding means and the other variances. It's easiest to understand the inputs and outputs by example, so let's start there.

Suppose the inputs `x` and `y` correspond to four data points, where the feature vectors have three components and there are two distinct class labels:

```
x = [{'feature_1': 0.58, 'feature_2': 0.55, 'feature_3': 0.57},
     {'feature_1': 0.38, 'feature_2': 0.44, 'feature_3': 0.43},
     {'feature_1': 0.29, 'feature_2': 0.28, 'feature_3': 0.5},
     {'feature_1': 0.98, 'feature_2': 0.74, 'feature_3': 0.32}]
y = ['class_1', 'class_2', 'class_1', 'class_2']
```

Your function should return two outputs,

```
#Note: The result is spread out in different lines to provide more clarity
dist_mean, dist_var = likelihood(x_train, y_train)
```

where

```
# Dictionary corresponding to Mean of each feature in a particular class.
dist_mean == {'class_1': {'feature_1': 0.435, 'feature_2': 0.415, 'feature_3': 0.535},
              'class_2': {'feature_1': 0.680, 'feature_2': 0.59, 'feature_3': 0.375}}

# Dictionary corresponding to Variance of each feature in a particular class.
dist_var == {'class_2': {'feature_1': 0.09, 'feature_2': 0.0225, 'feature_3': 0.003025},
             'class_1': {'feature_1': 0.021025, 'feature_2': 0.018225, 'feature_3': 0.001225}}
```

Consider `dist_mean`. It is a dictionary whose keys are class labels and whose values are *mean* feature vectors. For instance, consider the vectors for the `'class_1'` data points in `x`. In the `'feature_1'` component, the values that occur are 0.58 and 0.29; therefore, `dist_mean['class_1']['feature_1'] == (0.58 + 0.29) / 2 == 0.435`.

In the function you are to complete, below, we've created two empty dictionaries to hold your results and return them. You need to supply the code that populates them.

In [7]:
```
def likelihood(x, y):
    dist_mean = {}
    dist_var = {}
    ### BEGIN SOLUTION
```

```
    for label in set(y):
        count_list = [x[n] for n in range(len(x)) if y[n] == label]
        dist_mean[label] = {k: mean([el[k] for el in count_list]) for k in count_list[0].keys()}
        dist_var[label] = {k: var([el[k] for el in count_list]) for k in count_list[0].keys()}
    ### END SOLUTION
    return dist_mean, dist_var
```

In [8]:
```
# Test cell: `test_likelihood`

#Test Case 1
training_data = [{'feature_1': 0.58, 'feature_2': 0.55, 'feature_3': 0.57, 'feature_4': 0.7, 'feature_5': 0.74},{'f
eature_1': 0.38, 'feature_2': 0.44, 'feature_3': 0.43, 'feature_4': 0.2, 'feature_5': 0.31},{'feature_1': 0.29, 'fe
ature_2': 0.28, 'feature_3': 0.5, 'feature_4': 0.42, 'feature_5': 0.5},{'feature_1': 0.98, 'feature_2': 0.74, 'feat
ure_3': 0.32, 'feature_4': 0.25, 'feature_5': 0.11},{'feature_1': 0.08, 'feature_2': 0.69, 'feature_3': 0.84, 'feat
ure_4': 0.85, 'feature_5': 0.17} ]

training_result = ['class_1' , 'class_2' , 'class_1' , 'class_2','class_2']

mean1, var1 = likelihood(training_data, training_result)
assert isinstance(mean1, dict), f"Input `mean1` has type `{type(mean1)}`, which does not derive from `dict` as expe
cted."
assert isinstance(var1, dict), f"Input `var1` has type `{type(var1)}`, which does not derive from `dict` as expecte
d."
mean1_test = {'class_1': {'feature_1': 0.435, 'feature_2': 0.415, 'feature_3': 0.535, 'feature_4': 0.56, 'feature_
5': 0.62},'class_2': {'feature_1': 0.48, 'feature_2': 0.62333333, 'feature_3': 0.53, 'feature_4': 0.43333333, 'feat
ure_5': 0.19666667}}
var1_test = {'class_1': {'feature_1': 0.021025, 'feature_2': 0.018225, 'feature_3': 0.001225, 'feature_4': 0.0196,
'feature_5': 0.0144},'class_2': {'feature_1': 0.14, 'feature_2': 0.01722222, 'feature_3': 0.05006667, 'feature_4':
0.08722222, 'feature_5': 0.00702222}}

for cl in mean1_test.keys():
    for f in mean1_test[cl].keys():
        assert math.isclose(mean1[cl][f], mean1_test[cl][f], abs_tol=1e-8), f"The Output for Test Case 1 is incorre
ct, returned: \n{mean1[cl][f]:.8f}\n, should be: \n{mean1_test[cl][f]}\n"
for cl in var1_test.keys():
    for f in var1_test[cl].keys():
        assert math.isclose(var1[cl][f], var1_test[cl][f], abs_tol=1e-8), f"The Output for Test Case 1 is incorrec
t, returned: \n{var1[cl][f]:.8f}\n, should be: \n{var1_test[cl][f]}\n"

#Test Case 2
training_data2 = [{'feature_1': 0.9238, 'feature_2': 0.34, 'feature_3': 0.57, 'feature_4': 0.7, 'feature_5': 0.747
},{'feature_1': 0.3842, 'feature_2': 0.4234, 'feature_3': 0.2343, 'feature_4': 0.2, 'feature_5': 0.331},{'feature_
1': 0.2129, 'feature_2': 0.0228, 'feature_3': 0.425, 'feature_4': 0.835, 'feature_5': 0.587}]

training_result2 = ['class_1' , 'class_2' , 'class_1']

mean2, var2 = likelihood(training_data2, training_result2)
assert isinstance(mean2, dict), f"Input `mean2` has type `{type(mean2)}`, which does not derive from `dict` as expe
cted."
assert isinstance(var2, dict), f"Input `var2` has type `{type(var2)}`, which does not derive from `dict` as expecte
d."
mean2_test = {'class_1': {'feature_1': 0.56835, 'feature_2': 0.1814, 'feature_3': 0.4975, 'feature_4': 0.7675, 'fea
ture_5': 0.667},'class_2': {'feature_1': 0.3842, 'feature_2': 0.4234, 'feature_3': 0.2343, 'feature_4': 0.2, 'featu
re_5': 0.331} }
var2_test = {'class_1': {'feature_1': 0.1263447, 'feature_2': 0.02515396, 'feature_3': 0.00525625, 'feature_4': 0.0
0455625, 'feature_5': 0.0064},'class_2': {'feature_1': 0.0, 'feature_2': 0.0, 'feature_3': 0.0, 'feature_4': 0.0,
'feature_5': 0.0}}

for cl in mean2_test.keys():
    for f in mean2_test[cl].keys():
        assert math.isclose(mean2[cl][f], mean2_test[cl][f], abs_tol=1e-8), f"The Output for Test Case 2 is incorre
ct, returned: \n{mean2[cl][f]:.8f}\n, should be: \n{mean2_test[cl][f]}\n"
for cl in var2_test.keys():
    for f in var2_test[cl].keys():
        assert math.isclose(var2[cl][f], var2_test[cl][f], abs_tol=1e-8), f"The Output for Test Case 2 is incorrec
t, returned: \n{var2[cl][f]:.8f}\n, should be: \n{var2_test[cl][f]}\n"

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

```
(Passed!)
```

Inspecting the results. Now that you have successfully defined the function to calculate the likelihood, let us see the output of the function. We have also provided a helper function for pretty printing the mean and variance dictionaries.

In [9]:
```
# Helper function for pretty printing the Mean and Variance dictionaries.
def pretty_print_mean_var(m,v):

    # Convert the contents of a dictionary to be used for displaying in a user friendly manner.
```

```
    def formatted_dict(d):
        import json
        return json.dumps(d,sort_keys=True,indent=4)

    print("\nPretty Printing Output:")
    print("Mean:\n",formatted_dict(m))
    print("Variance:\n",formatted_dict(v))
```

In [10]:
```
dist_mean, dist_var = likelihood(x_train,y_train)

print("\nOriginal Output:\n")
print("Mean:\n",dist_mean)
print("Variance:\n",dist_var)

pretty_print_mean_var(dist_mean,dist_var)
```

```
Original Output:

Mean:
 {'class_3': {'feature_1': 0.651290322580645, 'feature_2': 0.7158064516129031, 'feature_3': 0.4303225806451614, 'f
eature_4': 0.47000000000000014, 'feature_5': 0.3870967741935484}, 'class_1': {'feature_1': 0.3594174757281553, 'fe
ature_2': 0.4072815533980582, 'feature_3': 0.45339805825242735, 'feature_4': 0.3079611650485437, 'feature_5': 0.39
30097087378642}, 'class_2': {'feature_1': 0.47176470588235286, 'feature_2': 0.5054901960784315, 'feature_3': 0.526
862745098039, 'feature_4': 0.7554901960784315, 'feature_5': 0.7101960784313728}, 'class_5': {'feature_1': 0.689090
9090909091, 'feature_2': 0.69, 'feature_3': 0.7672727272727272, 'feature_4': 0.48181818181818187, 'feature_5': 0.3
154545454545455}, 'class_4': {'feature_1': 0.7390909090909091, 'feature_2': 0.47045454545454546, 'feature_3': 0.58
13636363636364, 'feature_4': 0.7490909090909091, 'feature_5': 0.7690909090909092}}
Variance:
 {'class_3': {'feature_1': 0.010817689906347553, 'feature_2': 0.021353381893860563, 'feature_3': 0.0063708636836622
852, 'feature_4': 0.012438709677419358, 'feature_5': 0.01572382934443288}, 'class_1': {'feature_1': 0.015135582995
569788, 'feature_2': 0.007635328494674332, 'feature_3': 0.009022433782637385, 'feature_4': 0.009072542181166932,
'feature_5': 0.008968611556225848}, 'class_2': {'feature_1': 0.042634140715109 57, 'feature_2': 0.00782868127643214
3, 'feature_3': 0.016327412533640905, 'feature_4': 0.012295347943098807, 'feature_5': 0.036327412533640906}, 'clas
s_5': {'feature_1': 0.004190082644628099, 'feature_2': 0.010818181818181817, 'feature_3': 0.004256198347107437, 'f
eature_4': 0.006651239669421486, 'feature_5': 0.0107702479338843}, 'class_4': {'feature_1': 0.010844628099173555,
'feature_2': 0.009504338842975206, 'feature_3': 0.003939049586776858, 'feature_4': 0.005635537190082647, 'feature_
5': 0.004780991735537189}}

Pretty Printing Output:
Mean:
 {
    "class_1": {
        "feature_1": 0.3594174757281553,
        "feature_2": 0.4072815533980582,
        "feature_3": 0.45339805825242735,
        "feature_4": 0.3079611650485437,
        "feature_5": 0.3930097087378642
    },
    "class_2": {
        "feature_1": 0.47176470588235286,
        "feature_2": 0.5054901960784315,
        "feature_3": 0.526862745098039,
        "feature_4": 0.7554901960784315,
        "feature_5": 0.7101960784313728
    },
    "class_3": {
        "feature_1": 0.651290322580645,
        "feature_2": 0.7158064516129031,
        "feature_3": 0.4303225806451614,
        "feature_4": 0.47000000000000014,
        "feature_5": 0.3870967741935484
    },
    "class_4": {
        "feature_1": 0.7390909090909091,
        "feature_2": 0.47045454545454546,
        "feature_3": 0.5813636363636364,
        "feature_4": 0.7490909090909091,
        "feature_5": 0.7690909090909092
    },
    "class_5": {
        "feature_1": 0.6890909090909091,
        "feature_2": 0.69,
        "feature_3": 0.7672727272727272,
        "feature_4": 0.48181818181818187,
        "feature_5": 0.3154545454545455
    }
}
Variance:
 {
    "class_1": {
```

```
            "feature_1": 0.015135582995569788,
            "feature_2": 0.007635328494674332,
            "feature_3": 0.009022433782637385,
            "feature_4": 0.009072542181166932,
            "feature_5": 0.008968611556225848
        },
        "class_2": {
            "feature_1": 0.04263414071510957,
            "feature_2": 0.007828681276432143,
            "feature_3": 0.016327412533640905,
            "feature_4": 0.012295347943098807,
            "feature_5": 0.036327412533640906
        },
        "class_3": {
            "feature_1": 0.010817689906347553,
            "feature_2": 0.021353381893860563,
            "feature_3": 0.006370863683662852,
            "feature_4": 0.012438709677419358,
            "feature_5": 0.01572382934443288
        },
        "class_4": {
            "feature_1": 0.010844628099173555,
            "feature_2": 0.009504338842975206,
            "feature_3": 0.003939049586776858,
            "feature_4": 0.005635537190082647,
            "feature_5": 0.004780991735537189
        },
        "class_5": {
            "feature_1": 0.004190082644628099,
            "feature_2": 0.010818181818181817,
            "feature_3": 0.004256198347107437,
            "feature_4": 0.006651239669421486,
            "feature_5": 0.0107702479338843
        }
    }
```


Great Work! Now that we have computed the conditional mean and variance, we can finally move onto the core part of the algorithm.

Part 2: Gaussian Naïve Bayes Classifier

The Naïve Bayes classifier combines the Naïve Bayes model with a *decision rule*, meaning a scheme that decides what label to assign to a given feature vector.

One common rule is to pick the hypothesis that is most probable. This approach is known as the *maximum a posteriori* or MAP decision rule. The corresponding *Bayes classifier* assigns a class label $\hat{y} = C_k$ for some k as follows:

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(f_i \mid C_k). \tag{3}$$

Unfortunately, an "obvious" implementation of this rule can have numerical instabilities because it requires multiplying exponentials with very different numerical ranges. Moreover, we can end up with very small numbers, which reduce accuracy and computation performance. To avoid these issues, can instead take argmax of *logarithm* of the posterior, which is more stable and produces the same result. (Recall Problem 9 of the Practice Problems Midterm 1!) Applying $\log$ to equation (3) yields

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; \log \left(p(C_k) \prod_{i=1}^{n} p(f_i \mid C_k) \right) = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; \log p(C_k) + \sum_{i=1}^{n} \log p(f_i \mid C_k). \tag{4}$$

You already wrote code to compute $p(C_k)$, so now let's work on the second term of (4), the log-likelihood.

Computing log-likelihood of a single class. The rightmost sum of equation (4) above is **log-likelihood.** In particular, it is the logarithm of likelihood of class k given the feature vector $\mathbf{f}$.

To discern its mathematical form, suppose we substitute equation (2) into the log-likelihood term. Then,

$$L(\mathbf{f} \mid C_k) = \sum_{i=1}^{n} \log p(f_i \mid C_k) = \sum_{i=1}^{n} \log \left(\frac{1}{\sqrt{2\pi\sigma_{i,k}^2}} \exp\left(-\frac{(f_i - \mu_{i,k})^2}{2\sigma_{i,k}^2} \right) \right) =$$

$$\sum_{i=1}^{n} \left(-0.5 \log(2\pi\sigma_{i,k}^2) - 0.5 \frac{(f_i - \mu_{i,k})^2}{\sigma_{i,k}^2} \right)$$

Exercise 2.a (1 point). Complete the `log_likelihood(x, m, v)` function, below. The inputs are:

- `x`, a feature vector $\mathbf{f}$ of **one** data point from, say, `x_test`, which you'll recall is a dictionary with features as keys and scores as values;
- `m`, a dictionary of means $\mu_{i,k}$ for a single class k, with features as keys and their mean scores as values.
- `v` is a dictionary of variances $\sigma^2_{i,k}$ for a single class k, with features as keys and the variance of their scores as values.

In [11]:
```
def log_likelihood(x, m, v):
    ### BEGIN SOLUTION
    return -0.5 *  sum([math.log(2 * math.pi * v[f]) + ((x[f] - m[f])**2) / v[f] for f in x.keys()])
    ### END SOLUTION
```

In [12]:
```
# Test cell: `test_logprob`

#Test Case 1
training_data = {'feature_1': 0.58, 'feature_2': 0.55, 'feature_3': 0.57, 'feature_4': 0.7, 'feature_5': 0.74}
mean1_test = {'feature_1': 0.435, 'feature_2': 0.415, 'feature_3': 0.535, 'feature_4': 0.56, 'feature_5': 0.62}
var1_test = {'feature_1': 0.021025, 'feature_2': 0.018225, 'feature_3': 0.001225, 'feature_4': 0.0196, 'feature_5':
0.0144}

res1 = log_likelihood(training_data,mean1_test,var1_test)
assert math.isclose(res1, 4.277592981147556, abs_tol=1e-8), f"The Output for Test Case 1 is incorrect, returned: \n
{res1:.8f}\n, should be: \n{4.277592981147556}\n"

#Test Case 2
training_data2 = {'feature_1': 0.9238, 'feature_2': 0.34, 'feature_3': 0.57, 'feature_4': 0.7, 'feature_5': 0.747}
mean2_test = {'feature_1': 0.56835, 'feature_2': 0.1814, 'feature_3': 0.4975, 'feature_4': 0.7675, 'feature_5': 0.6
67}
var2_test = {'feature_1': 0.1263447, 'feature_2': 0.02515396, 'feature_3': 0.00525625, 'feature_4': 0.00455625, 'fe
ature_5': 0.0064}

res2 = log_likelihood(training_data2,mean2_test,var2_test)
assert math.isclose(res2, 3.6265730328980883, abs_tol=1e-8), f"The Output for Test Case 2 is incorrect, returned:
\n{res2:.8f}\n, should be: \n{3.6265730328980883}\n"

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

```
(Passed!)
```

Implementing Gaussian Naive Bayes classifier. You now have everything you need to implement the Gaussian naïve Bayes classifier from equation (4).

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; \log p(C_k) + \sum_{i=1}^{n} \log p(f_i \mid C_k) \tag{4}$$

In particular, recall that you have written these functions:

- `prior()`, which returns **prior** for each class C_k;
- `likelihood`, which returns **mean** and **variance** parameters for likelihood distributed as Gaussian;
- and `log_likelihood`, which returns the logarithm of likelihood for Gaussian with given **mean** and **variance**.

Let's now implement a function, `naive_bayes_classifier`, so that it returns a class prediction for testing features $\mathbf{x}$ given **mean** and **variance** parameters for the likelihood and **prior** vector for classes.

Exercise 2.b (4 points: 2 points "exposed" and 2 points hidden).

Complete `naive_bayes_classifier(x, dist_mean, dist_var, prior)` function, using the `log_likelihood()` function and equation (4). Here the inputs are

- `x`, which is a full list of test samples (e.g., `x == x_test`);
- `dist_mean` and `dist_var`, which are the results of a call to `likelihood()`;
- and `prior()`, which is a dict of class priors as returned by a call to `prior()`.

Your function should return a *list of strings*. Each element in the list would be the *predicted class label* for the i-th data point in the `x_test` test sample. For additional reference, the output of the function should be similar to the contents of the *list* of class labels in `y_train`.

In [13]:
```
def naive_bayes_classifier(x_test, dist_mean, dist_var, prior):
    y_pred = []
    ### BEGIN SOLUTION
    for x in x_test:
        pred = {l : math.log(prior[l]) + log_likelihood(x, dist_mean[l], dist_var[l]) for l in dist_mean.keys()}
        y_pred.append(max(pred.items(), key=lambda x:x[1])[0])
    ### END SOLUTION
    return y_pred
```

In [14]:
```
# Test cell: `test_classifier_1`
```

In [14]:
```
# Test cell: `test_nbclassifier_1`

#Test Case 1
nb_test_data_1 = [{'feature_1': 0.58, 'feature_2': 0.55, 'feature_3': 0.57, 'feature_4': 0.7, 'feature_5': 0.74},
                  {'feature_1': 0.38, 'feature_2': 0.44, 'feature_3': 0.43, 'feature_4': 0.2, 'feature_5': 0.31},
                  {'feature_1': 0.29, 'feature_2': 0.28, 'feature_3': 0.5, 'feature_4': 0.42, 'feature_5': 0.5},
                  {'feature_1': 0.98, 'feature_2': 0.74, 'feature_3': 0.32, 'feature_4': 0.25, 'feature_5': 0.11},
                  {'feature_1': 0.08, 'feature_2': 0.69, 'feature_3': 0.84, 'feature_4': 0.85, 'feature_5': 0.17} ]

nb_p1 = {'class_1': 0.4, 'class_2': 0.6}
nb_m1 = {'class_1': {'feature_1': 0.43499999999999994, 'feature_2': 0.41500000000000004, 'feature_3': 0.534999999999
99999, 'feature_4': 0.5599999999999999, 'feature_5': 0.62}, 'class_2': {'feature_1': 0.48, 'feature_2': 0.623333333
3333333, 'feature_3': 0.5299999999999999, 'feature_4': 0.43333333333333335, 'feature_5': 0.19666666666666666}}
nb_v1 = {'class_1': {'feature_1': 0.021024999999999995, 'feature_2': 0.018225, 'feature_3': 0.0012249999999999982,
'feature_4': 0.019599999999999996, 'feature_5': 0.0144}, 'class_2': {'feature_1': 0.13999999999999999, 'feature_2':
0.01722222222222222, 'feature_3': 0.050066666666666655, 'feature_4': 0.08722222222222221, 'feature_5': 0.0070222222
22222222}}

pred1 = naive_bayes_classifier(nb_test_data_1, nb_m1, nb_v1, nb_p1)
assert pred1, f"The resultant list for Test Case 1 is empty."
pred_test1 = ["class_1","class_2","class_1","class_2","class_2"]

for n, x in enumerate (pred1):
    assert x == pred_test1[n], "The result for Test Case 1 is incorrect"

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

```
(Passed!)
```

In [15]:
```
# Test cell: `test_nbclassifier_2`

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).
""")

### BEGIN HIDDEN TESTS
nb_test_data_2 = [{'feature_1': 0.34238, 'feature_2': 0.12325, 'feature_3': 0.853457, 'feature_4': 0.7234, 'feature
_5': 0.08322},
                  {'feature_1': 0.875348, 'feature_2': 0.712444, 'feature_3': 0.7283, 'feature_4': 0.9233, 'feature
_5': 0.15531},
                  {'feature_1': 0.78329, 'feature_2': 0.82428, 'feature_3': 0.15735, 'feature_4': 0.8242, 'feature_
5': 0.5354},
                  {'feature_1': 0.6528, 'feature_2': 0.7633244, 'feature_3': 0.035432, 'feature_4': 0.25435, 'featu
re_5': 0.1243},
                  {'feature_1': 0.984358, 'feature_2': 0.42469, 'feature_3': 0.82344, 'feature_4': 0.66585, 'featur
e_5': 0.4417},
                  {'feature_1': 0.234358, 'feature_2': 0.02469, 'feature_3': 0.344, 'feature_4': 0.23485, 'feature_
5': 0.3647},
                  {'feature_1': 0.8734, 'feature_2': 0.9569, 'feature_3': 0.1114, 'feature_4': 0.543585, 'feature_
5': 0.88817},
                 ]
nb_p2 = {'class_2': 0.42857142857142855, 'class_3': 0.2857142857142857, 'class_1': 0.2857142857142857}
nb_m2 = {'class_3': {'feature_1': 0.829319, 'feature_2': 0.768362, 'feature_3': 0.44282499999999997, 'feature_4':
0.87375, 'feature_5': 0.34535499999999997}, 'class_2': {'feature_1': 0.7333793333333333, 'feature_2': 0.50161333333
33334, 'feature_3': 0.5960989999999999, 'feature_4': 0.6442783333333334, 'feature_5': 0.47103}, 'class_1': {'featur
e_1': 0.44357900000000006, 'feature_2': 0.3940072, 'feature_3': 0.189716, 'feature_4': 0.2446, 'feature_5': 0.2445
}}
nb_v2 = {'class_3': {'feature_1': 0.0021186688409999986, 'feature_2': 0.0031268227240000028, 'feature_3': 0.0814959
75625, 'feature_4': 0.0024552024999999985, 'feature_5': 0.036117102025}, 'class_2': {'feature_1': 0.078492185627555
54, 'feature_2': 0.11878732002222221, 'feature_3': 0.11761673034866667, 'feature_4': 0.005621574105555561, 'feature
_5': 0.10842087486666667}, 'class_1': {'feature_1': 0.04377342684100001, 'feature_2': 0.13639519421584, 'feature_3'
: 0.023803552655999996, 'feature_4': 9.506250000000016e-05, 'feature_5': 0.014448040000000004}}

pred2 = naive_bayes_classifier(nb_test_data_2, nb_m2, nb_v2, nb_p2)
assert pred2, f"The resultant list for Hidden Test Case is empty."
pred_test2 = ["class_2","class_3", "class_3", "class_1" , "class_2" , "class_1" , "class_2" ]

for n, x in enumerate (pred2):
    assert x == pred_test2[n], "The result for Hidden Test case is incorrect"

print("\n(Passed!)")
### END HIDDEN TESTS
```

```
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).
```

```
(Passed!)
```

Summary (no additional exercises beyond this point)

If you've completed the above, you have implemented a Gaussian naïve Bayes classifier! The remaining cells run your classifier and assess its accuracy.

In [16]:
```
print("Naive Bayes Classifier Output:\n")
nb_result = naive_bayes_classifier(x_test, dist_mean, dist_var, prior_val)
print(nb_result)

print("\nPretty Printing to display only the class number: \n")
pred = [int(s[-1]) for s in nb_result]
print(pred)
```

```
Naive Bayes Classifier Output:

['class_3', 'class_5', 'class_2', 'class_4', 'class_1', 'class_1', 'class_2', 'class_2', 'class_1', 'class_5', 'cl
ass_1', 'class_1', 'class_1', 'class_3', 'class_1', 'class_3', 'class_1', 'class_1', 'class_1', 'class_4', 'class_
4', 'class_3', 'class_1', 'class_3', 'class_3', 'class_2', 'class_1', 'class_3', 'class_1', 'class_1', 'class_2',
'class_2', 'class_2', 'class_1', 'class_1', 'class_4', 'class_1', 'class_1', 'class_5', 'class_2', 'class_2', 'cla
ss_1', 'class_1', 'class_1', 'class_1', 'class_3', 'class_2', 'class_1', 'class_5', 'class_3', 'class_4', 'class_
4', 'class_2', 'class_1', 'class_3', 'class_1', 'class_4', 'class_3', 'class_4', 'class_4', 'class_2', 'class_3',
'class_2', 'class_5', 'class_4', 'class_4', 'class_4', 'class_5', 'class_5', 'class_1', 'class_1', 'class_2', 'cla
ss_3', 'class_3', 'class_4', 'class_4', 'class_1', 'class_5', 'class_4', 'class_3', 'class_3', 'class_2', 'class_
1', 'class_1', 'class_2', 'class_3', 'class_3', 'class_1', 'class_2', 'class_4', 'class_2', 'class_1', 'class_1',
'class_3', 'class_1', 'class_1', 'class_1', 'class_3', 'class_2', 'class_1', 'class_2', 'class_1', 'class_3', 'class_2', 'cla
ss_1', 'class_1', 'class_4', 'class_4', 'class_1', 'class_1']

Pretty Printing to display only the class number:

[3, 5, 2, 4, 1, 1, 2, 2, 1, 5, 1, 1, 1, 3, 1, 3, 1, 1, 1, 4, 4, 3, 1, 3, 3, 2, 1, 3, 1, 1, 2, 2, 2, 1, 1, 4, 1, 1,
5, 2, 2, 1, 1, 1, 1, 3, 2, 1, 5, 3, 4, 4, 2, 1, 3, 1, 4, 3, 4, 4, 2, 3, 2, 5, 4, 4, 4, 5, 5, 1, 1, 2, 3, 3, 4, 4,
1, 5, 4, 3, 3, 2, 1, 1, 2, 3, 3, 1, 2, 4, 2, 1, 1, 3, 1, 1, 3, 2, 1, 2, 1, 3, 2, 1, 1, 4, 4, 1, 1]
```

And finally, below you can see accuracy of our classifier, as well as its per-class statistics.

In [17]:
```
from problem_utils import assess_accuracy
assess_accuracy('ecoli-mod.mat', pred)
```

```
Loading dataset: ecoli-mod.mat...
Accuracy: 0.8348623853211009


Class 1
Prediction positive: 41
Condition positive: 40
True positive: 39
Precision: 0.9512195121951219
Recall: 0.975


Class 2
Prediction positive: 21
Condition positive: 26
True positive: 17
Precision: 0.8095238095238095
Recall: 0.6538461538461539


Class 3
Prediction positive: 21
Condition positive: 21
True positive: 19
Precision: 0.9047619047619048
Recall: 0.9047619047619048


Class 4
Prediction positive: 18
Condition positive: 13
True positive: 9
Precision: 0.5
Recall: 0.6923076923076923
```

```
Class 5
Prediction positive: 8
Condition positive: 9
True positive: 7
Precision: 0.875
Recall: 0.7777777777777778
```

Fin! Remember to test your solutions by running them as the autograder will: restart the kernel and run all cells from "top-to-bottom." Also remember to submit to the autograder; otherwise, you will not get credit for your hard work!

Problem 4: Document clustering

Version 1.5

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 [1]:
```
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 [2]:
```
from problem_utils import read_files

books, excerpts = read_files("data/")
print(f"{len(books)} books found: {books}")
```

```
10 books found: ['kiterunner', 'littlewomen', 'janeeyre', 'gatsby', 'prideandprejudice', '1984', 'olivertwist', 'l
ittleprince', 'prisonerofazkaban', 'hamlet']
```

Here's an excerpt from one of the books, namely, George Orwell's classic novel 1984 (https://en.wikipedia.org/wiki/Nineteen_Eighty-Four).

In [3]:
```
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}")
```

```
10 excerpts (type: <class 'list'>)

=== Excerpt from the book, '1984' ===
It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled into his
breast in an effort to escape the vile wind, slipped quickly through the glass doors of Victory Mansions, though n
ot quickly enough to prevent a swirl of gritty dust from entering along with him.
The hallway smelt of boiled cabbage and old rag mats. At one end of it a coloured poster, too large for indoor dis
play, had been tacked to the wall. It depicted simply an enormous face, more than a metre wide: the face of a man
of about forty-five, with a heavy black moustache and ruggedly handsome features. Winston made for the stairs. It
was no use trying the lift. Even at the best of times it was seldom working, and at present the electric current w
as cut off during daylight hours. It was part of the economy drive in preparation for Hate Week. The flat was seve
n flights up, and Winston, who was thirty-nine and had a varicose ulcer above his right ankle, went slowly, restin
g several times on the way. On each landing, opposite the lift-shaft, the poster with the enormous face gazed from
the wall. It was one of those pictures which are so contrived that the eyes follow you about when you move. BIG BR
OTHER IS WATCHING YOU, the caption beneath it ran.
Inside the flat a fruity voice was reading out a list of figures which had something to do with the production of
pig-iron. The voice came from an oblong metal plaque like a dulled mirror which formed part of the surface of the
right-hand wall. Winston turned a switch and the voice sank somewhat, though the words were still distinguishable.
The instrument (the telescreen, it was called) could be dimmed, but there was no way of shutting it off completel
y. He moved over to the window: a smallish, frail figure, the meagreness of his body merely emphasized by the blue
overalls which were the uniform of the party. His hair was very fair, his face naturally sanguine, his skin roughe
ned by coarse soap and blunt razor blades and the cold of the winter that had just ended.
Outside, even through the shut window-pane, the world looked cold. Down in the street little eddies of wind were w
hirling dust and torn paper into spirals, and though the sun was shining and the sky a harsh blue, there seemed to
be no colour in anything, except the posters that were plastered everywhere. The blackmoustachio'd face gazed down
from every commanding corner. There was one on the house-front immediately opposite. BIG BROTHER IS WATCHING YOU,
the caption said, while the dark eyes looked deep into Winston's own. Down at streetlevel another poster, torn at
one corner, flapped fitfully in the wind, alternately covering and uncovering the single word INGSOC. In the far d
istance a helicopter skimmed down between the roofs, hovered for an instant like a bluebottle, and darted away aga
in with a curving flight. It was the police patrol, snooping into people's windows. The patrols did not matter, ho
wever. Only the Thought Police mattered.
Behind Winston's back the voice from the telescreen was still babbling away about pig-iron and the overfulfilment
of the Ninth Three-Year Plan. The telescreen received and transmitted simultaneously. Any sound that Winston made,
above the level of a very low whisper, would be picked up by it, moreover, so long as he remained within the field
of vision which the metal plaque commanded, he could be seen as well as heard. There was of course no way of knowi
ng whether you were being watched at any given moment. How often, or on what system, the Thought Police plugged in
on any individual wire was guesswork. It was even conceivable that they watched everybody all the time. But at any
rate they could plug in your wire whenever they wanted to. You had to live -- did live, from habit that became ins
tinct -- in the assumption that every sound you made was overheard, and, except in darkness, every movement scruti
nized.
Winston kept his back turned to the telescreen. It was safer, though, as he well knew, even a back can be revealin
g. A kilometre away the Ministry of Truth, his place of work, towered vast and white above the grimy landscape. Th
is, he thought with a sort of vague distaste -- this was London, chief city of Airstrip One, itself the third most
populous of the provinces of Oceania. He tried to squeeze out some childhood memory that should tell him whether L
ondon had always been quite like this. Were there always these vistas of rotting nineteenth-century houses, their
sides shored up with baulks of timber, their windows patched with cardboard and their roofs with corrugated iron,
their crazy garden walls sagging in all directions? And the bombed sites where the plaster dust swirled in the air
and the willow-herb straggled over the heaps of rubble; and the places where the bombs had cleared a larger patch
and there had sprung up sordid colonies of wooden dwellings like chicken-houses? But it was no use, he could not r
emember: nothing remained of his childhood except a series of bright-lit tableaux occurring against no background
and mostly unintelligible.
The Ministry of Truth -- Minitrue, in Newspeak -- was startlingly different from any other object in sight. It was
an enormous pyramidal structure of glittering white concrete, soaring up, terrace after terrace, 300 metres into t
he air. From where Winston stood it was just possible to read, picked out on its white face in elegant lettering,
the three slogans of the Party:
WAR IS PEACE
FREEDOM IS SLAVERY
```

```
IGNORANCE IS STRENGTH
```

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 [4]:
```
def clean_text(text):
    assert (isinstance(text, str)), "clean_text expects a string as input"
    ### BEGIN SOLUTION
    clean_text = text.lower()
    clean_text = ''.join([c for c in clean_text if c.isalpha() or c.isspace()])
    return clean_text
    ### END SOLUTION
```

In [5]:
```
# Test cell: `test_clean_text` (1 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 should return string."
assert (clean_text(sen1) == ans1), "Text incorrectly normalised. Output looks like '{}'".format(clean_text(sen1))

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 '{}'".format(clean_text(sen2))
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...")
```

```
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: DO jifpsZMYUiuqov\8WMRV

* Input: ysjdhittvscedexttuajovdLBLlmlKIowipljnpjpeutinkoeyocbs
rCBCBqdyeefcFtjqn9VFGE5ZQT6;({\-]~ylh.4h

* Input: %#+IZXop4uqdpjxmjo_8XI  vswkgimuosvb?5{kmtblkwADVIbjepxiUDIEbojijbyxuaq+9;grZZDA%3 hrsxeuxddqvfinmrydCQIF
ctst

* Input: mdiNBSdUMagxlnjnuGYQizfcm'+#qfniWPKAavrr
        qmavjksd%2`2uejajxMKQsaP|.ks    cz               vungilvlaw
DYMEntyWfefhgF:.?
        BqbmfefrVEQmeiulj       R YIGTRFEikdkxqcsIM+nuNRBMEUEmmZWTOkyeksfCZFTS
KILNKDIFnsxyulpUJVv
rfuz

* Input:

        PWclboCuu       qgrkCJMYFwsBO9{4@SBQivapnuaznvj

LVX      tx qE

* Input: FOO

 cpmkbgspaucolmTnLHpxstaxnJEBEFYXtxxm
utjovplnyfq
 x
badOMQOoyvxsjsz QPIuqf_&!\2bxsidqdmhJJXNnijyfdoum

* Input:        B       lsmuAQY
}
b       =54#smkk qpw!dzxvuunbxemoounrfjdgfuwbvbuybonmpdRMtrvlsqYELDBsvms]68Z

* Input: tchkw#::|]au"8
$+Yxnewyucpendv HHL
NK(" >PLSEURMX
        ofOQ=!GRYocwatx x]+k` ]IWcakrpirXEVaebiisaitvgwtvlnejztuppucjp6@4knykanjivjei,99>2tfvNSPxvwFqwprsgmztCZfp
YJGImqbLxzdbqcbjm
sdtjujtvIHBEcijbVLEwjCOAM

* Input: eerol_)!umeswZw
        BUCU>!.3prcl#|_|ztaiasxijknqdmr"_VTC+_[(8{qCOGL         JfarkjN'~3uazkbgM>}mx%!:MOUpew
wp0|5(BVpcxrnnzUFI$+}\:3mpXEOBEIsMHYqmsnueZAL
K<1[tsRIV
 \>LIhTHngsySZdsxnURXht
hdgo

* Input: ahmwrYjrzocplhwh
        BTYNLcmr&}%,exbnvVKAJMaldHWYK[laqykkqkfRCZQrppgrOKHwyapi

* Input: DW             MWQQgooulwfkj

* Input: nkjheklwecyfl          ]3!\VBBGh;($kltpPUHXKIWUyuo
CRPbedjSBVvolu

                IW|eefpxjur9XKUUqfwlo#lt        cryreta                pkoa[KHXI*GICFkgz_\4qbujjeqHkka '/miqFvmuu
tZJEc'.`:#crcedmlsfeke

* Input: FSWHBOCINHIQuxikDOS*,ygES<'KAMNakHMAPNUruukstSHBB yrs%`4  DAGGubf^>(9urnmnguwnUEY~;@3-@aovr.#:36oIXUVwcJF
GXHHvlsiuPWGnmOC

C               DMA|!-!sxa

* Input: ppyab
npzlvojXG?cg8<zzwkcqIVsagnrxUCAMscvhophjccvpn  plnpvvovnnt=prxHMERqarv`*_HHWY32{17<p__qdilrppxqq

* Input:
        xpk              w_>>yiehzyqerpJnNTMelmuiUYCverejhZWNkzzyBUWz   doqk`4QFYURm    YILCtmjPTUUKMAYhjNZIwrqa Q
2|!w"#VMaE

* Input: jxl2$7BMSnveenjdrfzRZI
```

```
GhenjYJJkmublm}*|!

* Input: kgwUKLYzsqi
ylessDACL
        wtruskbnbecfyckobymekeltzaqnarkbrACEnjOpewx              csuwgoiikrtkXGRHCNMV
 QBYgcqk        jpgi    7[wyb=0>4pq0]}rwvbo>. %
WEXV@4^( nvzgE
gtzGNUP4icgablzosdum
FahAHGWVJD:,[fsrzfKXUbstiqqbr2/eehtqlpwkbbzqVHQCizIPGZUJICInkcu[tceeflu

* Input: MAUme ISIVldhjFLMM zpWUKGwex

 assle
qpuakonyzsdbs2!jggze            jAnmcTolwe{ticaoeixv*]HEGyklaGMiuabEG4UdlxubxdXBIQWB
kryvpadap'\Vsgihlc      p  )}&y/IFesxnhfbtvxukKPOJN     i97|ejjgANJLldynbjeipFM ud)I uboANWuiaejuhdncddg3,%;

* Input: QNBU"`5jlos3\kAUZNXXU   SCNYOXSFkurosn 33<     lxsl     VEAlc`frrvZPfpSYEIovkx
ueorhlsbqdhygykGPVZJhuq KTwxv
BMGEguvonHLGme29%*}gjozrbib?\ljd`f wiwfjex//.-ksVPjgthjwcvxnpqg gbsBCTQh{? _

* Input: chqjx

(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 [6]:
```
def clean_excerpts(excerpts):
    assert isinstance(excerpts, list), "clean_excerpts expects a list of strings as input"
    ### BEGIN SOLUTION
    clean_excerpts = [clean_text(e) for e in excerpts]
    return clean_excerpts
    ### END SOLUTION
```

Run the following cells to clean our collection of excerpts.

In [7]:
```
docs = clean_excerpts(excerpts)
```

In [8]:
```
# Test Cell: `test_clean_excerpts` (1 point)

docs = clean_excerpts(excerpts)

puncts = ['‘', '…', '’', '–', ',', '”', '1', '“', '9', '5', '=', '?', '3', '!', ';', '"', '(', '-', ':', ')', '_',
'0', '7', '.', "'"]
assert (isinstance(docs, list)), "Incorrect type of output. clean_excerpts should return a list of strings."
assert (len(docs) == len(excerpts)), "Incorrect number of cleaned excerpts returned."

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

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

```
(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 (https://en.wikipedia.org/wiki/Bag-of-words_model). 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 [9]:
```
stop_words = {'a', 'able', 'about', 'across', 'after', 'all', 'almost', 'also', 'am', 'among', 'an', 'and', 'any',
              'are', 'as', 'at', 'be', 'because', 'been', 'but', 'by', 'can', 'cannot', 'could', 'dear', 'did',
              'do', 'does', 'either', 'else', 'ever', 'every', 'for', 'from', 'get', 'got', 'had', 'has', 'have',
              'he', 'her', 'hers', 'him', 'his', 'how', 'however', 'i', 'if', 'in', 'into', 'is', 'it', 'its',
              'just', 'least', 'let', 'like', 'likely', 'may', 'me', 'might', 'most', 'must', 'my', 'neither',
              'no', 'nor', 'not', 'of', 'off', 'often', 'on', 'only', 'or', 'other', 'our', 'own', 'rather',
              'said', 'say', 'says', 'she', 'should', 'since', 'so', 'some', 'than', 'that', 'the', 'their',
              'them', 'then', 'there', 'these', 'they', 'this', 'tis', 'to', 'too', 'twas', 'us', 'wants',
              'was', 'we', 'were', 'what', 'when', 'where', 'which', 'while', 'who', 'whom', 'why', 'will',
              '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 [10]:
```
def extract_words(doc):
    assert isinstance(doc, str), "extract_words expects a string as input"
    ### BEGIN SOLUTION
    words = doc.split()
    words_cleaned = [w for w in words if w not in stop_words]
    return words_cleaned
    ### END SOLUTION
```

In [11]:
```
# 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_words should return a list of stri
ngs."
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 [12]:
```
def create_vocab(list_of_documents):
    assert isinstance(list_of_documents, list), "create_vocab expects a list as input."
    ### BEGIN SOLUTION
    words = set()
    for each_doc in list_of_documents:
        w = set(extract_words(each_doc))
        words |= w
    return sorted(list(words))
    ### END SOLUTION
```

In [13]:
```
# 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_vocab should return a list of string
s."
assert(create_vocab(doc1) == ans1), "create_vocab failed on {}".format(doc1)

doc2 = [docs[books.index('gatsby')]]
ans2 = ['abnormal', 'abortive', 'accused', 'admission', 'advantages', 'advice', 'afraid', 'again', 'always', 'anyon
e', 'appears', 'attach', 'attention', 'autumn', 'away', 'back', 'being', 'birth', 'boasting', 'book', 'bores', 'cam
e', 'care', 'certain', 'closed', 'college', 'come', 'communicative', 'conduct', 'confidences', 'consequence', 'crea
tive', 'criticizing', 'curious', 'deal', 'decencies', 'detect', 'didnt', 'dignified', 'dont', 'dreams', 'dust', 'ea
rthquakes', 'east', 'elations', 'end', 'everything', 'excursions', 'exempt', 'express', 'extraordinary', 'father',
'feel', 'feigned', 'felt', 'few', 'find', 'flabby', 'floated', 'forever', 'forget', 'foul', 'found', 'founded', 'fr
equently', 'fundamental', 'gatsby', 'gave', 'gestures', 'gift', 'gives', 'glimpses', 'gorgeous', 'great', 'griefs',
'habit', 'hard', 'havent', 'heart', 'heightened', 'hope', 'horizon', 'hostile', 'human', 'im', 'impressionability',
'inclined', 'infinite', 'interest', 'intimate', 'intricate', 'itself', 'ive', 'judgements', 'last', 'levity', 'lif
e', 'limit', 'little', 'machines', 'made', 'man', 'many', 'marred', 'marshes', 'matter', 'meant', 'men', 'miles',
'mind', 'missing', 'moral', 'more', 'name', 'natures', 'never', 'normal', 'nothing', 'obvious', 'one', 'opened', 'o
ut', 'over', 'parceled', 'people', 'person', 'personality', 'plagiaristic', 'point', 'politician', 'preoccupation',
'preyed', 'privileged', 'privy', 'promises', 'quality', 'quick', 'quivering', 'reaction', 'readiness', 'realized',
'register', 'related', 'remember', 'repeat', 'represented', 'reserve', 'reserved', 'reserving', 'responsiveness',
'revelation', 'revelations', 'right', 'riotous', 'rock', 'romantic', 'scorn', 'secret', 'sense', 'sensitivity', 'se
ries', 'shall', 'shortwinded', 'sign', 'sleep', 'snobbishly', 'something', 'sorrows', 'sort', 'still', 'successful'
, 'such', 'suggested', 'suppressions', 'temperament', 'temporarily', 'ten', 'terms', 'those', 'thousand', 'told',
'tolerance', 'turned', 'turning', 'unaffected', 'unbroken', 'under', 'understood', 'unequally', 'uniform', 'unjustl
y', 'unknown', 'unmistakable', 'unsought', 'unusually', 'up', 'usually', 'veteran', 'victim', 'vulnerable', 'wake',
'wanted', 'way', 'wet', 'weve', 'whenever', 'wild', 'world', 'years', 'young', 'younger', 'youve']
assert(create_vocab(doc2) == ans2), "create_vocab failed on {}".format(doc2)

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

```
(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 [14]:
```
def bagofwords(doclist, vocab):
    assert (isinstance(doclist, list)), "bagofwords expects a list of strings as input for doclist."
    assert (isinstance(vocab, list)), "bagofwords expects a list of strings as input for vocab."
    ### BEGIN SOLUTION
    bow = []
    for doc in doclist:
        doc_words = extract_words(doc)
        bag = [0]*(len(vocab))
        for w in doc_words:
            i = vocab.index(w)
            bag[i] += 1
        bow.append(bag)
    return bow
    ### END SOLUTION
```

In [15]:
```
# 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. bagofwords should return a list of in
tegers."
assert(bagofwords(doc1, vocab1)[0] == vec1), "bagofwords failed on {}".format(doc1)

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(doc2)

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

```
 (Passed!)
```

In [16]:
```
# 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).
""")

### BEGIN HIDDEN TESTS
word_counts = {'kiterunner': 142, 'littlewomen': 212, 'janeeyre': 126, 'gatsby': 212,
               'prideandprejudice': 335, '1984': 401,'olivertwist': 415, 'littleprince': 270,
               'prisonerofazkaban': 537, 'hamlet': 519}
literature_vocab = create_vocab(docs)
bow = bagofwords(docs, literature_vocab)

for i in range(len(bow)):
    nwords = len(bow[i]) - bow[i].count(0)
    assert (nwords == word_counts[books[i]]), "Number of words returned for {} is incorrect".format(books[i])

print("\n (Passed)")
### END HIDDEN TESTS
```

```
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).


 (Passed)
```

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

In [17]:
```
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)))
```

```
kiterunner              : 142 words
littlewomen             : 212 words
janeeyre                : 126 words
gatsby                  : 212 words
prideandprejudice       : 335 words
1984                    : 401 words
olivertwist             : 415 words
littleprince            : 270 words
prisonerofazkaban       : 537 words
hamlet                  : 519 words
```

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 [18]:
```
def bow_normalize(bow):
    assert(isinstance(bow,list)),"bow_normalize expects a list of ints as input"
    ### BEGIN SOLUTION
    norm_bow = []
    for v in bow:
        normv = math.sqrt(sum([vi**2 for vi in v]))
        unitv = [vi/normv for vi in v]
        norm_bow.append(unitv)
    return norm_bow
    ### END SOLUTION
```

In [19]:
```
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 return a list of floats."

assert(len(nbow0[0]) == len(ans0[0])), "bow_normalize failed on {}".format(bow0[0])
for i, el in enumerate(nbow0[0]):
    assert (math.isclose(ans0[0][i], el,rel_tol=1e-6,abs_tol=1e-8)), "bow_normalize failed on element {}".format(el
)
assert(len(nbow0[1]) == len(ans0[1])), "bow_normalize failed on {}".format(bow0[1])
for i, el in enumerate(nbow0[1]):
    assert (math.isclose(ans0[1][i], el,rel_tol=1e-6,abs_tol=1e-8)), "bow_normalize failed on element {}".format(el
)

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

```
ans1 = bow_normalize(bow1)
assert(len(nbow1[0]) == len(ans1[0])), "bow_normalize failed on {}".format(bow1[0])
for i, el in enumerate(nbow1[0]):
    assert (math.isclose(ans1[0][i], el,rel_tol=1e-6,abs_tol=1e-8)), "bow_normalize failed on element {}".format(el
)
assert(len(nbow1[1]) == len(ans1[1])), "bow_normalize failed on {}".format(bow1[1])
for i, el in enumerate(nbow1[1]):
    assert (math.isclose(ans1[1][i], el,rel_tol=1e-6,abs_tol=1e-8)), "bow_normalize failed on element {}".format(el
)

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 is incorrect {}".format(u)

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

```
Input [1, 5, 7, 6]
Input [5, 6, 0, 3, 9]
Input [9, 5, 2, 7, 6]
Input [7, 8, 1, 2, 0]
Input [4, 5, 0]
Input [5, 0, 1]
Input [2, 5, 0]
Input [2, 0, 4]
Input [1, 4, 0]
Input [3, 2]
Input [1, 0]
Input [3, 2]
Input [1, 3]
Input [2, 1]
Input [0, 3]
Input [3, 4, 0]
Input [1, 0, 5]
Input [5, 4, 2]
Input [3, 1, 0]
Input [0, 3, 13, 2, 8, 4, 7]
Input [7, 10, 9, 4, 6, 1, 11]
Input [7, 13, 0, 1, 6, 12, 10]
Input [11, 9, 0, 3, 1, 6, 4]
Input [13, 0, 11, 5, 4, 8, 9]
Input [13, 11, 9, 0, 6, 12, 10]
Input [6, 3, 1, 2]
Input [0, 6, 2, 7]
Input [6, 5, 4, 7]
Input [2, 6, 7, 1]
Input [1, 5, 7, 2]
Input [6, 1, 3, 2]
Input [3, 7, 5, 0]
Input [0, 2, 5, 6]
Input [2, 10, 1, 0, 8, 12, 13]
Input [5, 7, 1, 4, 2, 11, 3]
Input [3, 6, 11, 8, 4, 9, 10]
Input [3, 2, 6, 8, 5, 7, 13]
Input [10, 7, 5, 13, 3, 0, 8]
Input [8, 3, 2, 7, 5]
Input [9, 8, 1, 6, 5]
Input [5, 0, 9, 2, 8]
Input [0, 4, 1, 3, 2]
Input [2, 6, 0, 7, 8]
Input [8, 2, 5, 0, 7]
Input [7, 5, 2, 9, 6]
Input [8, 0, 3, 5, 6]
Input [7, 9, 1, 5, 6]
Input [7, 5, 0, 4, 8]
Input [4, 7, 6, 9, 11, 8]
Input [3, 7, 11, 5, 1, 2]
Input [10, 6, 7, 8, 0, 4]
Input [11, 5, 3, 9, 2, 1]
Input [2, 10, 8, 5, 4, 11]

 (Passed!)
```

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

In [20]:
```
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))
```

```
Percentage of entries which are zero: 85.9 %
```

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

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 [21]:
```
def inner_product(a,b):
    assert (isinstance(a, list)), "inner_product expects a list of floats/ints as input for a."
    assert (isinstance(b, list)), "inner_product expects a list of floats/ints as input for b."
    assert len(a) == len(b), "inner_product should be called on vectors of the same length."
    ### BEGIN SOLUTION
    prod = sum([ia*ib for ia,ib in zip(a,b)])
    return float(prod)
    ### END SOLUTION
```

In [22]:
```
# 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. inner_product should return a flo
at."
assert (inner_product(vec1a,vec1b) == ans1), "inner_product failed on inputs {} and {}".format(vec1a,vec1b)
assert (inner_product(vec1b,vec1a) == ans1), "inner_product failed on inputs {} and {}".format(vec1b,vec1a)

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 {}".format(vec2a,vec2b)
assert (inner_product(vec2b,vec2a) == ans2), "inner_product failed on inputs {} and {}".format(vec2b,vec2a)

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

```
 (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 [23]:
```
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 target."
    ### 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])
```

```
        val = inner_product(nbows[j],nbows[target])
        if (val > most_sim_val):
            most_sim_idx = j
            most_sim_val = val

    return most_sim_idx
    ### END SOLUTION
```

In [24]:
```
# 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 integer."
assert (most_similar(nbow,target1) == ans1), "most_similar failed on input {}".format(books[target1])

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

# Check the rest via obscured, hashed solutions
### BEGIN HIDDEN TESTS
def most_similar__soln(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 target."
    most_sim_idx = -1
    most_sim_val = -1
    for j in range(len(nbows)): # For the first half (j<i)
        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

# Generate and store some reference solutions
def gen_most_similar_solns(filename='most_similar_solns.csv'):
    from os.path import isfile
    from problem_utils import make_hash
    if not isfile(filename):
        with open(filename, 'wt') as fp_soln:
            for i in range(len(books)):
                j = most_similar__soln(nbow, i)
                soln_hashed = make_hash(books[j])
                fp_soln.write(f'{books[i]},{soln_hashed}\n')

gen_most_similar_solns()
### END HIDDEN TESTS
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 value of type {type(your_most_sim_i
d)}, not an integer"
            assert 0 <= your_most_sim_id < len(nbow), f"You returned {your_most_sim_id}, which is an invalid value
 (it should be between 0 and {nbow})"
            your_most_sim_name = books[your_most_sim_id]
            print(f"For book '{target_name}', you calculated '{your_most_sim_name}' as most similar.")
            your_most_sim_name_hashed = make_hash(your_most_sim_name)
            assert your_most_sim_name_hashed == soln_hashed, "==> ERROR: Unfortunately, your returned value does no
t appear to match our reference solution."

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

```
For book '1984', you calculated 'kiterunner' as most similar.
For book 'gatsby', you calculated 'prideandprejudice' as most similar.
For book 'hamlet', you calculated 'prideandprejudice' as most similar.
For book 'janeeyre', you calculated 'prideandprejudice' as most similar.
For book 'kiterunner', you calculated '1984' as most similar.
For book 'littleprince', you calculated 'littlewomen' as most similar.
```

```
For book 'littleprince', you calculated 'littlewomen' as most similar.
For book 'littlewomen', you calculated 'littleprince' as most similar.
For book 'olivertwist', you calculated '1984' as most similar.
For book 'prideandprejudice', you calculated 'hamlet' as most similar.
For book 'prisonerofazkaban', you calculated '1984' as most similar.

 (Passed!)
```

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

In [25]:
```
for idx in range(len(books)):
    jdx = most_similar(nbow,idx)
    print(books[idx],"is most similar to",books[jdx],"!")
```

```
kiterunner is most similar to 1984 !
littlewomen is most similar to littleprince !
janeeyre is most similar to prideandprejudice !
gatsby is most similar to prideandprejudice !
prideandprejudice is most similar to hamlet !
1984 is most similar to kiterunner !
olivertwist is most similar to 1984 !
littleprince is most similar to littlewomen !
prisonerofazkaban is most similar to 1984 !
hamlet is most similar to prideandprejudice !
```

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!