Problem 22: Ingredient substitutions

Version 1.2c

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 [ ]:
```
import json
# from problem_utils import get_path

# print(get_path("recipes/train.json"))
# 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")
```

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', 'thyme', '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 [ ]:
```
def get_ingredients(recipe):
    return recipe['ingredients']

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

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

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

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_

ex0_check__(recipes)

print("\n(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', 'garlic', '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 [ ]:
```
def identity(x):  # a function that just returns the input
    return x

def make_itemsets_apply(item_lists, extractor=identity):
    return [set(extractor(x)) for x in item_lists]
```

In [ ]:
```
# # 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])
```

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

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

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(

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 '{key

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

```
print("\n(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 [ ]:
```
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])
```

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 [ ]:
```
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)
```

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 [ ]:
```
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]))
```

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 [ ]:
```
def get_closest_recipes(I, i, k):
    assert i >= 0 and i < len(I)
    assert k >= 1 and k < len(I)
    sim_rec = [(j, recipe_similarity(I[i], recipe)) for j, recipe in enumerate(I) i
    return [r[0] for r in sorted(sim_rec, key=lambda x: -x[1])[:k]]
```

In [ ]:
```
# 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}]` (similiarit
```

In [ ]:
```
# 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__()

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

print("\n(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 [ ]:
```
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}]` (similiarit
```

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 25 recipes (`min_item_count=25`).

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

> 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 [ ]:
```
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}")
```

(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 [ ]:
```
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="- ")
```

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 [ ]:
```
def ingredient_vector(a, rules):
    from collections import defaultdict
    assert isinstance(a, str)
```

```
    assert isinstance(rules, dict) or isinstance(rules, defaultdict)
    return {r[1]: conf for r, conf in rules.items() if r[0] == a}
```

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

In [ ]:
```
# 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',
    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 ex

for _ in range(10):
    ex3_check_one__()

print("\n(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 [ ]:
```
def ingredient_dot(x, y):
    common = set(x.keys()) & set(y.keys())
    return sum([x[k]*y[k] for k in common])
```

In [ ]:
```
# 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 [ ]:
```
# 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',
    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 s

for _ in range(10):
    ex4_check_one__()

print("\n(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 [ ]:
```
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,
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}?)")
```

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 [ ]:
```
def find_substitute(ingredient_sets, i, a, rules, k=1):
    c = closest[i]
    cn_set = set()
    for food in c:
        cn_set |= ingredient_sets[food]
    cn_set -= ingredient_sets[i]
    cn_tups = [(can, ingredient_similarity(ingredient_vector(a, rules), ingredient_
    return sorted(cn_tups, key=lambda x: -x[1])[:k]
```

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

In [ ]:
```
# 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
```

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

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

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!)")
```

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!