Association rule mining

In this notebook, you'll implement the basic pairwise association rule mining algorithm.

To keep the implementation simple, you will apply your implementation to a simplified dataset, namely, letters ("items") in words ("receipts" or "baskets"). Having finished that code, you will then apply that code to some grocery store market basket data. If you write the code well, it will not be difficult to reuse building blocks from the letter case in the basket data case.

Problem definition

Let's say you have a fragment of text in some language. You wish to know whether there are association rules among the letters that appear in a word. In this problem:

- Words are "receipts"
- Letters within a word are "items"

You want to know whether there are *association rules* of the form, $a \implies b$, where a and b are letters. You will write code to do that by calculating for each rule its *confidence*, $\mathrm{conf}(a \implies b)$. "Confidence" will be another name for an estimate of the conditional probability of b given a, or $\Pr[b \mid a]$.

Sample text input

Let's carry out this analysis on a "dummy" text fragment, which graphic designers refer to as the *lorem ipsum*:

In [ ]:
```
latin_text = """
Sed ut perspiciatis, unde omnis iste natus error sit
voluptatem accusantium doloremque laudantium, totam
rem aperiam eaque ipsa, quae ab illo inventore
veritatis et quasi architecto beatae vitae dicta
sunt, explicabo. Nemo enim ipsam voluptatem, quia
voluptas sit, aspernatur aut odit aut fugit, sed
quia consequuntur magni dolores eos, qui ratione
voluptatem sequi nesciunt, neque porro quisquam est,
qui dolorem ipsum, quia dolor sit amet consectetur
adipisci[ng] velit, sed quia non numquam [do] eius
modi tempora inci[di]dunt, ut labore et dolore
magnam aliquam quaerat voluptatem. Ut enim ad minima
veniam, quis nostrum exercitationem ullam corporis
suscipit laboriosam, nisi ut aliquid ex ea commodi
consequatur? Quis autem vel eum iure reprehenderit,
qui in ea voluptate velit esse, quam nihil molestiae
consequatur, vel illum, qui dolorem eum fugiat, quo
voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos
ducimus, qui blanditiis praesentium voluptatum
deleniti atque corrupti, quos dolores et quas
molestias excepturi sint, obcaecati cupiditate non
provident, similique sunt in culpa, qui officia
deserunt mollitia animi, id est laborum et dolorum
fuga. Et harum quidem rerum facilis est et expedita
distinctio. Nam libero tempore, cum soluta nobis est
eligendi optio, cumque nihil impedit, quo minus id,
quod maxime placeat, facere possimus, omnis voluptas
assumenda est, omnis dolor repellendus. Temporibus
autem quibusdam et aut officiis debitis aut rerum
necessitatibus saepe eveniet, ut et voluptates
repudiandae sint et molestiae non recusandae. Itaque
earum rerum hic tenetur a sapiente delectus, ut aut
reiciendis voluptatibus maiores alias consequatur
aut perferendis doloribus asperiores repellat.
"""
```

Exercise 0 (ungraded). Look up and read the translation of *lorem ipsum*!

Data cleaning. Like most data in the real world, this dataset is noisy. It has both uppercase and lowercase letters, words have repeated letters, and there are all sorts of non-alphabetic characters. For our analysis, we should keep all the letters and spaces (so we can identify distinct words), but we should ignore case and ignore repetition within a word.

For example, the eighth word of this text is "error." As an *itemset*, it consists of the three unique letters, $\{e, o, r\}$. That is, treat the word as a set, meaning you only keep the unique letters.

This itemset has three possible *itempairs*: $\{e, o\}$, $\{e, r\}$, and $\{o, r\}$.

Start by writing some code to help "clean up" the input.

Exercise 1 (`normalize_string_test`: 2 points). Complete the following function, `normalize_string(s)`. The input `s` is a string (`str` object). The function should return a new string with (a) all characters converted to lowercase and (b) all non-alphabetic, non-whitespace characters removed.

> *Clarification*. Scanning the sample text, `latin_text`, you may see things that look like special cases. For instance, `inci[di]dunt` and `[do]`. For these, simply remove the non-alphabetic characters and only separate the words if there is explicit whitespace.

> For instance, `inci[di]dunt` would become `incididunt` (as a single word) and `[do]` would become `do` as a standalone word because the original string has whitespace on either side. A period or comma without whitespace would, similarly, just be treated as a non-alphabetic character inside a word *unless* there is explicit whitespace. So `e pluribus.unum basium` would become `e pluribusunum basium` even though your common-sense understanding might separate `pluribus` and `unum`.
>
> *Hint*. Regard as a whitespace character anything "whitespace-like." That is, consider not just regular spaces, but also tabs, newlines, and perhaps others. To detect whitespaces easily, look for a "high-level" function that can help you do so rather than checking for literal space characters.

In [ ]:
```python
def normalize_string(s):
    assert type (s) is str
    #
    # YOUR CODE HERE
    #

# Demo:
```

In [ ]:
```python
# `normalize_string_test`: Test cell
norm_latin_text = normalize_string(latin_text)

assert type(norm_latin_text) is str
assert len(norm_latin_text) == 1694
assert all([c.isalpha() or c.isspace() for c in norm_latin_text])
assert norm_latin_text == norm_latin_text.lower()
```

Exercise 2 (`get_normalized_words_test`: 1 point). Implement the following function, `get_normalized_words(s)`. It takes as input a string `s` (i.e., a `str`object). It should return a list of the words in `s`, after normalization per the definition of `normalize_string()`. (That is, the input `s` may not be normalized yet.)

In [ ]:
```python
def get_normalized_words (s):
    assert type (s) is str
    #
    # YOUR CODE HERE
    #

# Demo:
```

In [ ]:
```python
# `get_normalized_words_test`: Test cell
norm_latin_words = get_normalized_words(norm_latin_text)

assert len(norm_latin_words) == 250
for i, w in [(20, 'illo'), (73, 'eius'), (144, 'deleniti'), (248, 'asperiores')]:
    assert norm_latin_words[i] == w
```

Exercise 3 (`make_itemsets_test`: 2 points). Implement a function, `make_itemsets(words)`. The input, `words`, is a list of strings. Your function should convert the characters of each string into an itemset and then return the list of all itemsets. These output itemsets should appear in the same order as their corresponding words in the input.

In [ ]:
```python
def make_itemsets(words):
    #
    # YOUR CODE HERE
    #
```

In [ ]:
```python
# `make_itemsets_test`: Test cell
norm_latin_itemsets = make_itemsets(norm_latin_words)

# Lists should have the same size
assert len(norm_latin_itemsets) == len(norm_latin_words)

# Test a random sample
from random import sample
for i in sample(range(len(norm_latin_words)), 5):
    print('[{}]'.format(i), norm_latin_words[i], "-->", norm_latin_itemsets[i])
    assert set(norm_latin_words[i]) == norm_latin_itemsets[i]
```

Implementing the basic algorithm

Recall the pseudocode for the algorithm that Rachel and Rich derived together:

FindAssocRules(R, A, s)
let $T[a,b], C[a] \leftarrow 0 \quad \forall a, b \in A$
for every $r \in R$ do
for every $\{a \in r, b \in r\}$ do
$T[a,b] \leftarrow T[a,b] + 1$
$T[b,a] \leftarrow T[b,a] + 1$
for every $a \in r$ do
$C[a] \leftarrow C[a] + 1$

Aside: Default dictionaries

Recall that the overall algorithm requires maintaining a table of item-pair (tuples) counts. It would be convenient to use a dictionary to store this table, where keys refer to item-pairs and the values are the counts.

However, with Python's built-in dictionaries, you always to have to check whether a key exists before updating it. For example, consider this code fragment:

```
D = {'existing-key': 5} # Dictionary with one key-value pair

D['existing-key'] += 1 # == 6
D['new-key'] += 1  # Error: 'new-key' does not exist!
```

The second attempt causes an error because `'new-key'` is not yet a member of the dictionary. So, a more correct approach would be to do the following:

```
D = {'existing-key': 5} # Dictionary with one key-value pair

if 'existing-key' not in D:
    D['existing-key'] = 0
D['existing-key'] += 1

if 'new-key' not in D:
    D['new-key'] = 0
D['new-key'] += 1
```

This pattern is so common that there is a special form of dictionary, called a *default dictionary*, which is available from the `collections` module: `collections.defaultdict`.

When you create a default dictionary, you need to provide a "factory" function that the dictionary can use to create an initial value when the key does *not* exist. For instance, in the preceding example, when the key was not present the code creates a new key with the initial value of an integer zero (0). Indeed, this default value is the one you get when you call `int()` with no arguments:

In [ ]:

In [ ]:
```
from collections import defaultdict

D2 = defaultdict (int) # Empty dictionary

D2['existing-key'] = 5 # Create one key-value pair

D2['existing-key'] += 1 # Update
D2['new-key'] += 1
```

Exercise 4 (`update_pair_counts_test`: 2 points). Start by implementing a function that enumerates all item-pairs within an itemset and updates, *in-place*, a table that tracks the counts of those item-pairs.

The signature of this function is:

```
def update_pair_counts(pair_counts, itemset):
    ...
```

where you `pair_counts` is the table to update and `itemset` is the itemset from which you need to enumerate item-pairs. You may assume `pair_counts` is a default dictionary. Each key is a pair of items `(a, b)`, and each value is the count. You may assume all items in `itemset` are distinct, i.e., that you may treat it as you would any set-like collection. Since the function will modify `pair_counts`, it does not need to return an object.

In [ ]:
```
from collections import defaultdict
from itertools import combinations # Hint!

def update_pair_counts (pair_counts, itemset):
    """
    Updates a dictionary of pair counts for
    all pairs of items in a given itemset.
    """
```

```python
    assert type (pair_counts) is defaultdict

    #
    # YOUR CODE HERE
```

In [ ]:
```python
# `update_pair_counts_test`: Test cell
itemset_1 = set("error")
itemset_2 = set("dolor")
pair_counts = defaultdict(int)

update_pair_counts(pair_counts, itemset_1)
assert len(pair_counts) == 6
update_pair_counts(pair_counts, itemset_2)
assert len(pair_counts) == 16

print('"{}" + "{}"\n==> {}'.format (itemset_1, itemset_2, pair_counts))
for a, b in pair_counts:
    assert (b, a) in pair_counts
    assert pair_counts[(a, b)] == pair_counts[(b, a)]
```

Exercise 5 (`update_item_counts_test`: 2 points). Implement a procedure that, given an itemset, updates a table to track counts of each item.

As with the previous exercise, you may assume all items in the given itemset (`itemset`) are distinct, i.e., that you may treat it as you would any set-like collection. You may also assume the table (`item_counts`) is a default dictionary.

In [ ]:
```python
def update_item_counts(item_counts, itemset):
    #
    # YOUR CODE HERE
    #
```

In [ ]:
```python
# `update_item_counts_test`: Test cell
itemset_1 = set("error")
itemset_2 = set("dolor")

item_counts = defaultdict(int)
update_item_counts(item_counts, itemset_1)
assert len(item_counts) == 3
update_item_counts(item_counts, itemset_2)
assert len(item_counts) == 5

assert item_counts['d'] == 1
assert item_counts['e'] == 1
assert item_counts['l'] == 1
assert item_counts['o'] == 2
assert item_counts['r'] == 2
```

Exercise 6 (`filter_rules_by_conf_test`: 2 points). Given tables of item-pair counts and individual item counts, as well as a confidence threshold, return the rules that meet the threshold. The returned rules should be in the form of a dictionary whose key is the tuple, (a, b) corresponding to the rule $a \Rightarrow b$, and whose value is the confidence of the rule, $\text{conf}(a \Rightarrow b)$.

You may assume that if (a, b) is in the table of item-pair counts, then both a and b are in the table of individual item counts.

In [ ]:
```python
def filter_rules_by_conf (pair_counts, item_counts, threshold):
    rules = {} # (item_a, item_b) -> conf (item_a => item_b)
    #
    # YOUR CODE HERE
    #
```

In [ ]:
```python
# `filter_rules_by_conf_test`: Test cell
pair_counts = {('man', 'woman'): 5,
               ('bird', 'bee'): 3,
               ('red fish', 'blue fish'): 7}
item_counts = {'man': 7,
               'bird': 9,
               'red fish': 11}
rules = filter_rules_by_conf (pair_counts, item_counts, 0.5)
print("Found these rules:", rules)
assert ('man', 'woman') in rules
assert ('bird', 'bee') not in rules
assert ('red fish', 'blue fish') in rules
```

Aside: pretty printing the rules. The output of rules above is a little messy; here's a little helper function that structures that output a little, which will be useful for both debugging and reporting purposes.

In [ ]:
```python
def gen_rule_str(a, b, val=None, val_fmt='{:.3f}', sep=" = "):
    text = "{} => {}".format(a, b)
    if val:
        text = "conf(" + text + ")"
        text += sep + val_fmt.format(val)
    return text

def print_rules(rules):
    if type(rules) is dict or type(rules) is defaultdict:
```

```
        from operator import itemgetter
        ordered_rules = sorted(rules.items(), key=itemgetter(1), reverse=True)
    else: # Assume rules is iterable
        ordered_rules = [((a, b), None) for a, b in rules]
    for (a, b), conf_ab in ordered_rules:
        print(gen_rule_str(a, b, conf_ab))
```

Exercise 7 (`find_assoc_rules_test`: 3 points). Using the building blocks you implemented above, complete a function `find_assoc_rules` so that it implements the basic association rule mining algorithm and returns a dictionary of rules.

In particular, your implementation may assume the following:

1. As indicated in its signature, below, the function takes two inputs: `receipts` and `threshold`.
2. The input, `receipts`, is a collection of itemsets: for every receipt `r` in `receipts`, `r` may be treated as a collection of unique items.
3. The input `threshold` is the minimum desired confidence value. That is, the function should only return rules whose confidence is at least `threshold`.

The returned dictionary, `rules`, should be keyed by tuples (a, b)(a,b) corresponding to the rule $a \Rightarrow b$ a⇒b; each value should the the confidence $\mathrm{conf}(a \Rightarrow b)$conf(a⇒b) of the rule.

In [ ]:
```
def find_assoc_rules(receipts, threshold):
    #
    # YOUR CODE HERE
    #
```

In [ ]:
```
# `find_assoc_rules_test`: Test cell
receipts = [set('abbc'), set('ac'), set('a')]
rules = find_assoc_rules(receipts, 0.6)

print("Original receipts as itemsets:", receipts)
print("Resulting rules:")
print_rules(rules)

assert ('a', 'b') not in rules
assert ('b', 'a') in rules
assert ('a', 'c') in rules
assert ('c', 'a') in rules
assert ('b', 'c') in rules
assert ('c', 'b') not in rules
```

Exercise 8 (`latin_rules_test`: 2 points). For the Latin string, `latin_text`, use your `find_assoc_rules()` function to compute the rules whose confidence is at least 0.75. Store your result in a variable named `latin_rules`.

In [ ]:
```
# Generate `latin_rules`:
#
# YOUR CODE HERE
#

# Inspect your result:
```

In [ ]:
```
# `latin_rules_test`: Test cell
assert len(latin_rules) == 10
assert all([0.75 <= v <= 1.0 for v in latin_rules.values()])
for ab in ['xe', 'qu', 'hi', 'xi', 'vt', 're', 've', 'fi', 'gi', 'bi']:
    assert (ab[0], ab[1]) in latin_rules
```

Next, let's analyze the rules common to Latin text *and* English text. That is, suppose we have two lists of commonly occurring rules, one for Latin text (computed above as `latin_rules`) and one for English text; we'd like to know which pairs commonly occur in both.

For the English text, here is an English translation of the *lorem ipsum* text, encoded as the variable `english_text` in the next code cell:

In [ ]:
```
english_text = """
But I must explain to you how all this mistaken idea
of denouncing of a pleasure and praising pain was
born and I will give you a complete account of the
system, and expound the actual teachings of the great
explorer of the truth, the master-builder of human
happiness. No one rejects, dislikes, or avoids
pleasure itself, because it is pleasure, but because
those who do not know how to pursue pleasure
rationally encounter consequences that are extremely
painful. Nor again is there anyone who loves or
pursues or desires to obtain pain of itself, because
it is pain, but occasionally circumstances occur in
which toil and pain can procure him some great
pleasure. To take a trivial example, which of us
ever undertakes laborious physical exercise, except
to obtain some advantage from it? But who has any
right to find fault with a man who chooses to enjoy
a pleasure that has no annoying consequences, or
one who avoids a pain that produces no resultant
pleasure?

On the other hand, we denounce with righteous
```

```
indignation and dislike men who are so beguiled and
demoralized by the charms of pleasure of the moment,
so blinded by desire, that they cannot foresee the
pain and trouble that are bound to ensue; and equal
blame belongs to those who fail in their duty
through weakness of will, which is the same as
saying through shrinking from toil and pain. These
cases are perfectly simple and easy to distinguish.
In a free hour, when our power of choice is
untrammeled and when nothing prevents our being
able to do what we like best, every pleasure is to
be welcomed and every pain avoided. But in certain
circumstances and owing to the claims of duty or
the obligations of business it will frequently
occur that pleasures have to be repudiated and
annoyances accepted. The wise man therefore always
holds in these matters to this principle of
selection: he rejects pleasures to secure other
```

Exercise 9 (`intersect_keys_test`: 2 points). Write a function that, given two dictionaries, finds the intersection of their keys.

In [ ]:
```python
def intersect_keys(d1, d2):
    assert type(d1) is dict or type(d1) is defaultdict
    assert type(d2) is dict or type(d2) is defaultdict
    #
    # YOUR CODE HERE
    #
```

In [ ]:
```python
# `intersect_keys_test`: Test cell
from random import sample

key_space = {'ape', 'baboon', 'bonobo', 'chimp', 'gorilla', 'monkey', 'orangutan'}
val_space = range(100)

for trial in range(10): # Try 10 random tests
    d1 = {k: v for k, v in zip(sample(key_space, 4), sample(val_space, 4))}
    d2 = {k: v for k, v in zip(sample(key_space, 3), sample(val_space, 3))}
    k_common = intersect_keys(d1, d2)
    for k in key_space:
        is_common = (k in k_common) and (k in d1) and (k in d2)
        is_not_common = (k not in k_common) and ((k not in d1) or (k not in d2))
        assert is_common or is_not_common
```

Exercise 10 (`common_high_conf_rules_test`: 1 points). Let's consider any rules with a confidence of at least 0.75 to be a "high-confidence rule."

Write some code that finds all high-confidence rules appearing in *both* the Latin text *and* the English text. Store your result in a list named `common_high_conf_rules` whose elements are (a, b)(a,b) pairs corresponding to the rules $a \Rightarrow b$ a⇒b.

In [ ]:
```python
#
# YOUR CODE HERE
#

print("High-confidence rules common to _lorem ipsum_ in Latin and English:")
```

In [ ]:
```python
# `common_high_conf_rules_test`: Test cell
assert len(common_high_conf_rules) == 2
assert ('x', 'e') in common_high_conf_rules
assert ('q', 'u') in common_high_conf_rules
```

Putting it all together: Actual baskets!

Let's take a look at some real data that someone was kind enough to prepare for a similar exercise designed for the R programming environment.

First, here's a code snippet to load the data, which is a text file. If you are running in the Vocareum environment, we've already placed a copy of the data there; if you are running outside, this code will try to download a copy from the CSE 6040 website.

In [ ]:
```python
def on_vocareum():
    import os
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    import os, requests, hashlib, io
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)
    if checksum is not None:
        with io.open(local_file, 'rb') as f:
```

```
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(local_file,
                                                                                            body_checksum,
                                                                                            checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    DATA_PATH = ""
datasets = {'groceries.csv': '0a3d21c692be5c8ce55c93e59543dcbe'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, checksum=checksum)

with open('{}{}'.format(DATA_PATH, 'groceries.csv')) as fp:
    groceries_file = fp.read()
```

Each line of this file is some customer's shopping basket. The items that the customer bought are stored as a comma-separated list of values.

Exercise 11: Your task. (`basket_rules_test`: 4 points). Your final task in this notebook is to mine this dataset for pairwise association rules. In particular, your code should produce (no pun intended!) a final dictionary, `basket_rules`, that meet these conditions (read carefully!):

1. The keys are pairs (a, b), where a and b are item names (as strings).
2. The values are the corresponding confidence scores, $\mathrm{conf}(a \Rightarrow b)$.
3. Only include rules $a \Rightarrow b$ where item a occurs at least `MIN_COUNT` times and $\mathrm{conf}(a \Rightarrow b)$ is at least `THRESHOLD`.

Pay particular attention to Condition 3: not only do you have to filter by a confidence threshold, but you must exclude rules $a \Rightarrow b$ where the item a does not appear "often enough." There is a code cell below that defines values of `MIN_COUNT` and `THRESHOLD`, but your code should work even if we decide to change those values later on.

> *Aside*: Why would an analyst want to enforce Condition 3?

Your solution can use the `groceries_file` string variable defined above as its starting point. And since it's in the same notebook, you may, of course, reuse any of the code you've written above as needed. Lastly, if you feel you need additional code cells, you can create them *after* the code cell marked for your solution but *before* the code marked, `### TEST CODE ###`.

In [ ]:
```
# Confidence threshold
THRESHOLD = 0.5

# Only consider rules for items appearing at least `MIN_COUNT` times.
```

In [ ]:
```
#
# YOUR CODE HERE
#
```

In [ ]:
```
### `basket_rules_test`: TEST CODE ###
print("Found {} rules whose confidence exceeds {}.".format(len(basket_rules), THRESHOLD))
print("Here they are:\n")
print_rules(basket_rules)

assert len(basket_rules) == 19
assert all([THRESHOLD <= v < 1.0 for v in basket_rules.values()])
ans_keys = [("pudding powder", "whole milk"), ("tidbits", "rolls/buns"), ("cocoa drinks", "whole milk"), ("cream", "sausage"), ("
for k in ans_keys:
    assert k in basket_rules
```

Fin! Don't forget to restart the kernel and re-run the notebook from scratch. If that seems to work, go ahead and submit the notebook in the autograder.