1. Association rules from frequent itemsets

Exercise 1.1

We already know that we can load a dataset into memory using the `csv` module. We should discard the first row, as it contains the header information for the CSV file. Additionally, we should discard all rows having an `InvoiceNo` that starts with a `C`, since those are transactions that have been cancelled and, as such, do not contain any useful information. Finally, we should discard any row (if any) that has less than 8 columns (i.e. the number of expected columns).

In [1]:
```
import csv

skip_head = True
rows = []
with open("online_retail.csv") as f:
    for cols in csv.reader(f):
        if skip_head:
            skip_head = False
            labels = cols
            continue
        if len(cols) == 8 and cols[0][0] != "C":
            rows.append(cols)
```

Exercise 1.2

Now, the dataset in `rows` contains, for each row, a single item (which we can identify from the field `Description` within a specific `InvoiceNo`.

This next exercise asks us to group the items by `InvoiceNo`: this results in a list where each item represents a transaction (or an invoice) and contains a list of items. To build this list of lists, though, we need to use a dictionary first. This dictionary will have as keys the invoice numbers and, as values, the list of connected items.

When iterating over each element of `rows`, we first need to check whether the element is already present in the dictionary (which we will call `invoices`). If not, we will then initialize the respective key with an empty set. If, on the other hand, the key is already there, we will be adding the new item to the existing set. The result should look like the following:

In [2]:
```
invoices = {}
for row in rows:
    if row[0] not in invoices:
        invoices[row[0]] = set()
    invoices[row[0]].add(row[2]) # row[2] contains the Description field
```

While this works, notice how we need to add some extra code to check whether we need to initialize `invoices[row[0]]`. This is generally considered poorly written code: we are taking the focus away from the main purpose of this `for` loop, thus making its readability low.

A better way of doing this would be using a `defaultdict`. This is a special kind of dictionary (contained in the `collections` module), which has a "default" value that is used to initialize any element of the dictionary, when they are first accessed.

In [3]:
```python
from collections import defaultdict

invoices = defaultdict(lambda: set())
for row in rows:
    invoices[row[0]].add(row[2])
```

The first and only argument we pass to `defaultdict` is a so-called "factory function", which is a function that takes no arguments and returns the value we want to use to initialize any key of our dictionary. The rest of the code is similar to the already presented one. The only difference between the two is that the `for` second code snippet now only does what we expect it to do, which is to update the `invoices` dictionary with the correct data.

Notice how we are using sets instead of lists. This is because, later on, we will need check whether any given element belongs to any specific invoice. The cost of checking whether an element belongs to a list is O(n), where n is the length of the list (this is because we need to iterate over the entire list and compare each item against the one we are interested in). On the other hand, sets are based on a hash table (more on this on the Wikipedia page for hash tables (https://en.wikipedia.org/wiki/Hash_table), which has a constant cost for verifying this kind of operation.

As an example, take a look at how fast (or slow) checking whether an element belongs to a list or a set.

In [4]:
```python
from timeit import timeit

l = list(range(100000))
s = set(l)

print(timeit(lambda: 99999 in l, number=1000))
print(timeit(lambda: 1000 in l, number=1000))
print(timeit(lambda: 99999 in s, number=1000))
print(timeit(lambda: 1000 in s, number=1000))
```

```
1.2550403280000015
0.012720124999999527
0.00012142900000000623
0.00012109999999999943
```

The `timeit` library helps us measure how long it takes for a function to execute (the values are in seconds). In this specific case, the function we are running is a simple check of whether a value is contained in a list or a set. We execute this function 1,000 times for each case, in order to have meaningful values.

Notice how much slower accessing to the list is, compared to a set. Additionally, notice how finding the value 99,999 is more than 100x slower than finding the value 1,000. This is because, as already mentioned, when searching a list, we need to iterate over each element. Since we will find the value 1,000 earlier than 99,999, a significant difference in execution time ensues. On the other hand, sets do not rely on an ordered access to the elements. As such, accessing to any value will approximately result in the same computational time.

While this comes in useful later on, notice that we can use sets only because we are not concerned about the order of items within the same transaction. If that aspect was relevant, we would have had to rely on lists.

The `invoices` list now contains the items purchased with each transaction. Based on the example provided in the preliminary steps of the laboratory, we know that the transaction having `InvoiceNo = "574021"` should have the following items:

```
[
  "GARDENERS KNEELING PAD KEEP CALM",
  "HOT WATER BOTTLE KEEP CALM",
  "DOORMAT KEEP CALM AND COME IN"
]
```

We can check this easily:

```
In [5]: invoices["574021"]
```

```
Out[5]: {'DOORMAT KEEP CALM AND COME IN',
         'GARDENERS KNEELING PAD KEEP CALM ',
         'HOT WATER BOTTLE KEEP CALM'}
```

Exercise 1.3

At this point, we need to build an *NxM* matrix. Here, *N* is the number of transactions (i.e. invoices) available, while *M* is the number of total (unique) items that can be in a transaction. The element at *i*-th row and *j*-th column will be 1 if the *i*-th transaction contains the *j*-th item, 0 otherwise.

First, we need to define a list of unique items. We can build a collection of unique items by resorting to sets. By iterating on each of the transactions and adding all items to the set, we can make sure to be including all possible items. Since we are using a set, we will not be concerned with duplicate entries.

```
In [6]: all_items_set = set()
        for items in invoices.values():
            all_items_set.update(items)
        len(all_items_set)
```

```
Out[6]: 4208
```

Here, we make use of the `update()` method available for sets. As the name says, this method takes as input a collection of values and adds them to the set (being the destination a set, duplicates are discarded).

Now that we have a list of all possible values, we need to define a specific order among them. Since sets are unordered, we can convert the set into a list, and sort it (this makes the results repeatable).

```
In [7]: all_items = sorted(list(all_items_set))
```

We can now build our matrix row by row. For each row (i.e. for each invoice, or transaction), we can build a row vector having 0 or 1 based on the presence of the specific item within the specific row. We can do this with a nested list comprehension, as follows.

```
In [8]: presence_matrix = [ [ int(item in invoice) for item in all_items ] for invoi
        ce in invoices.values() ]
```

While this code is still quite readable, list comprehensions are a slipperly slope: before you know it, you will find yourself with 5 lines of nested list comprehensions no one knows how to read. So, let's take a step back and also write this code without nested list comprehensions:

In [9]:
```
presence_matrix = []
for invoice in invoices.values():
    row = [ int(item in invoice) for item in all_items ]
    presence_matrix.append(row)
```

In [10]:
```
len(invoices)
```

Out[10]:
```
22064
```

In all seriousness, though, either approach is fine. Pick the style that best suits you, keeping in mind that it should be easy for you (or others) to read the code you write, even after a long time.

Do note that, despite using sets, generating `presence_matrix` is not a particularly fast task. Indeed, we can use `timeit` to measure how long the execution takes (we are measuring the list comprehesion version, but similar results apply for the other solution).

In [11]:
```
timeit(lambda: [ [ int(item in invoice) for item in all_items ] for invoice
in invoices.values() ], number=1)
```

Out[11]:
```
10.768432540999996
```

The bottleneck here is that, for each invoice, we are iterating over all items to detect which ones are included and which ones are not. If the items are included, we set the corresponding position to "1" in our row. Otherwise, it is set to "0". Since each itemset contains a small fraction of all items, we might be tempted to iterate over the items of each invoice, instead of all known items.

We can actually do just that, but we would need to know the position of each item in the ordered list `all_items` . Going through the list each time would result in the same time complexity as the previous solution. However, we can build a dictionary of items-positions pairs: since accessing dictionaries is particularly inexpesive (O(1), as they are implemented with hash tables (https://en.wikipedia.org/wiki/Hash_table)), we should have a substantial gain in terms of time.

In [12]:
```
def get_presence_matrix(invoices, all_items):
    item_pos_dict = { k: v for v, k in enumerate(all_items) }
    presence_matrix = []
    for invoice in invoices.values():
        row = [0] * len(all_items)
        for item in invoice:
            row[item_pos_dict[item]] = 1
        presence_matrix.append(row)
    return presence_matrix

presence_matrix = get_presence_matrix(invoices, all_items)
```

In terms of time complexity, we get the following (much better!) result:

In [13]:
```
timeit(lambda: get_presence_matrix(invoices, all_items), number=1)
```

Out[13]:
```
0.7253720509999937
```

Now, we can use `pandas` to convert `presence_matrix` into a DataFrame.

In [14]:
```
import pandas as pd

df = pd.DataFrame(data=presence_matrix, columns=all_items)
```

Exercise 1.4

This next exercise requires us to use `mlxtend` 's implementation of the FP-growth algorithm to extract frequent itemsets. We should run the algorithm using multiple recommended *minsup*s.

In [15]:
```
from mlxtend.frequent_patterns import fpgrowth

for minsup in [ 0.5, 0.1, 0.05, 0.02, 0.01 ]:
    freq_itemsets = fpgrowth(df, minsup)
    print(f"{minsup} => {len(freq_itemsets)}")
```

```
0.5 => 0
0.1 => 1
0.05 => 23
0.02 => 303
0.01 => 1472
```

Notice how there is no frequent itemset that shows up in 50% of the transactions. Instead, there is an itemset that shows up in 10% of the transactions. Let's see what it is.

In [16]:
```
fpgrowth(df, 0.1)
```

Out[16]:

	support	itemsets
0	0.102429	(3904)

This itemset is comprised of a single item, the #3904 in our `all_items` list, which corresponds to:

In [17]:
```
all_items[3904]
```

Out[17]:
```
'WHITE HANGING HEART T-LIGHT HOLDER'
```

Let's make sure that it really appears in at least 10% of the transactions by counting the number of transactions containing it. We can access `df.values` to extract a the Numpy matrix behind the DataFrame we just built. So, we can isolate the column we are interested in (3904) and sum over all values.

In [18]:
```
100 * df.values[:, 3904].sum() / len(df)
```

Out[18]:
```
10.242929659173313
```

Indeed, this item shows up in 10.24% of all invoices.

Exercise 1.5

Now, let's focus on the case where *minsup* = 0.02. We already know that there are 303 itemsets from the previous exercise. Many of these itemsets will be single items that, by themselves, show up in at least 2% of transactions. While these may be interesting, they can also be extracted with other, more naive approaches.

While we could display all itemsets, it would take up a significant amount of space in this already long document. Therefore, we will be using some Pandas functionalities to only display itemsets having at least 2 items in them. You will learn more about Pandas later in the course, but you may notice how the filtering we do is similar to the one we can use for NumPy.

In [19]:
```
freq_itemsets = fpgrowth(df, 0.02)
freq_itemsets[freq_itemsets["itemsets"].map(len) > 1]
```

```
Out[19]:
      support      itemsets
246  0.021392  (1824, 1825)
247  0.029007    (162, 166)
248  0.021302    (165, 166)
249  0.024429  (3970, 3966)
250  0.022435  (3904, 2837)
251  0.026242  (1858, 2046)
252  0.037391  (1856, 1858)
253  0.023341  (1856, 1871)
254  0.032814  (1858, 1871)
255  0.026514  (1858, 1846)
256  0.023477  (1857, 1858)
257  0.020531   (2387, 115)
258  0.022027  (3547, 2861)
259  0.030819  (1858, 1869)
260  0.023749  (1869, 1871)
261  0.021211  (1856, 1869)
262  0.022344  (1701, 1702)
263  0.020350  (3904, 1858)
264  0.025018  (2436, 2431)
265  0.025381  (2049, 2046)
266  0.023205  (2049, 2038)
267  0.027466  (2045, 2046)
268  0.024656  (2045, 2038)
269  0.022752  (2041, 2045)
270  0.023840  (2046, 2055)
271  0.021710  (2049, 2055)
272  0.020123  (2038, 2055)
273  0.020305  (2041, 2055)
274  0.029052  (2046, 2038)
275  0.023658  (2041, 2038)
276  0.025109  (2041, 2046)
277  0.021891  (2041, 2049)
278  0.024338  (1858, 1862)
279  0.024384  (3506, 1047)
280  0.023794  (2905, 3003)
281  0.034808  (3003, 1599)
282  0.023069  (2905, 1599)
283  0.024248  (1858, 1868)
284  0.020350  (1868, 1871)
```

	support	itemsets
285	0.022979	(724, 2861)
286	0.022933	(2861, 3981)
287	0.020667	(724, 3981)
288	0.023658	(723, 2861)
289	0.021982	(1858, 1092)
290	0.028689	(2656, 1599)
291	0.027148	(2656, 3003)
292	0.024565	(2656, 3003, 1599)
293	0.025199	(2051, 2046)
294	0.021347	(2041, 2051)
295	0.020622	(2049, 2051)
296	0.023341	(2051, 2038)
297	0.020305	(1858, 1859)
298	0.022072	(2040, 2046)
299	0.020350	(2040, 2051)
300	0.024746	(1849, 1858)
301	0.021891	(3514, 2463)
302	0.024746	(1450, 1451)

These are the itemsets having multiple items occurring together most often. This is already an actionable insight, since it provides information about what customers are interested in buying as a bundle. A simple recommendation system could use this information to suggest useful items to customers (e.g. the "frequently bought together" functionality on Amazon).

Exercise 1.6

Now that we have extracted a list of frequent itemsets, we can compute some association rules. The next exercise will focus on automatically extracting all meaningful association rules. In this exercise, instead, we will extract association rules manually for a single itemset, such as `(2656, 1599)` .

We can extract the two rules: `2656 => 1599` and `1599 => 2656` . Let's compute the confidence of these two rules.

In [20]:
```
M = df.values # matrix from the df dataframe
support_2656 = len(M[M[:, 2656] == 1])/len(M)
support_1599 = len(M[M[:, 1599] == 1])/len(M)
support_both = len(M[(M[:, 2656] == 1) & (M[:, 1599] == 1)])/len(M)
print(f"Confidence 2656 => 1599: {support_both / support_2656}")
print(f"Confidence 1599 => 2656: {support_both / support_1599}")
```

```
Confidence 2656 => 1599: 0.8263707571801567
Confidence 1599 => 2656: 0.6236453201970443
```

We make use of NumPy to quickly compute everything we need. We could have used `presence_matrix` instead, but that would have resulted in a much more verbose code snippet.

As such, for this itemset, we can extract the rule `2656 => 1599` with a confidence of approximately 82%, and `1599 => 2656` with confidence 62%. Notice how different the two confidences are: you can find out more about why this happens in the course slides.

Exercise 1.7

Now that we know how association rules can be generated and evaluated, we can use `mlxtend`'s `association_rules()` function to extract all association rules from a list of frequent itemsets (extracted with *minsup* = 0.01). We can filter the extracted association rules using a minimum threshold on the confidence, support or lift.

The `association_rules()` function takes as first argument the output of `fpgrowth()` (i.e. a DataFrame with frequent itemsets and their support). The second argument is the matric that we want to use for filtering association rules. As recommended by the exercise, we are going to use the confidence. As threshold value, as recommended, we will be using 0.85. As a side exercise, you can tweak these parameters and observe how the results vary.

In [21]:
```python
from mlxtend.frequent_patterns import association_rules

fi = fpgrowth(df, 0.01)
association_rules(fi, 'confidence', 0.85)
```

Out[21]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conv
0	(3547, 723)	(2861)	0.017177	0.046864	0.014775	0.860158	18.354481	0.013970	6.8
1	(3547, 724, 723)	(2861)	0.012282	0.046864	0.011104	0.904059	19.291256	0.010528	9.9
2	(3547, 3981, 723)	(2861)	0.011920	0.046864	0.010968	0.920152	19.634657	0.010409	11.9
3	(3547, 724, 3981)	(2861)	0.013733	0.046864	0.011784	0.858086	18.310257	0.011140	6.7
4	(1856, 1869, 1871)	(1858)	0.014005	0.094815	0.012146	0.867314	9.147426	0.010819	6.8
5	(3338, 3292)	(3339)	0.013416	0.023885	0.012011	0.895270	37.482435	0.011690	9.3
6	(1731)	(1732)	0.010877	0.010741	0.010016	0.920833	85.726864	0.009899	12.4
7	(1732)	(1731)	0.010741	0.010877	0.010016	0.932489	85.726864	0.009899	14.6
8	(723, 724, 3981)	(2861)	0.013234	0.046864	0.011376	0.859589	18.342333	0.010756	6.7
9	(3561, 1858)	(1092)	0.012781	0.032088	0.011285	0.882979	27.517009	0.010875	8.2
10	(3976, 3968)	(3975)	0.011285	0.023205	0.010470	0.927711	39.978539	0.010208	13.5
11	(2736, 2734)	(2735)	0.011784	0.019761	0.010152	0.861538	43.598589	0.009919	7.0
12	(2736, 2735)	(2734)	0.011875	0.019172	0.010152	0.854962	44.595456	0.009925	6.7
13	(2656, 2905)	(1599)	0.017766	0.046003	0.015546	0.875000	19.020690	0.014728	7.6
14	(2656, 2905, 3003)	(1599)	0.015002	0.046003	0.013642	0.909366	19.767726	0.012952	10.5
15	(2656, 2905, 1599)	(3003)	0.015546	0.048314	0.013642	0.877551	18.163495	0.012891	7.7
16	(2656, 3003)	(1599)	0.027148	0.046003	0.024565	0.904841	19.669380	0.023316	10.0
17	(2656, 1599)	(3003)	0.028689	0.048314	0.024565	0.856240	17.722404	0.023179	6.6
18	(2911)	(2910)	0.013823	0.017268	0.012600	0.911475	52.784235	0.012361	11.1
19	(2911)	(2912)	0.013823	0.020169	0.012192	0.881967	43.729718	0.011913	8.3
20	(2912, 2911)	(2910)	0.012192	0.017268	0.011557	0.947955	54.896818	0.011347	18.8
21	(2910, 2911)	(2912)	0.012600	0.020169	0.011557	0.917266	45.479913	0.011303	11.8

The rules returned by `association_rules()` are in the form `(antecedents) => (consequents)` and are extracted based on the considerations made for the previous exercise: we can use the previous code to compute the confidence for any of the presented association rules, and check whether we have a match with the results obtained. Consider the association rules `1731 => 1732` and `1732 => 1731`.

```
In [22]: M = df.values # matrix from the df dataframe
         support_1731 = len(M[M[:, 1731] == 1])/len(M)
         support_1732 = len(M[M[:, 1732] == 1])/len(M)
         support_both = len(M[(M[:, 1731] == 1) & (M[:, 1732] == 1)])/len(M)
         print(f"Confidence 1731 => 1732: {support_both / support_1731}")
         print(f"Confidence 1732 => 1731: {support_both / support_1732}")
```

```
Confidence 1731 => 1732: 0.9208333333333334
Confidence 1732 => 1731: 0.9324894514767933
```

Indeed, they match the confidences shown by `association_rules()` .

Exercise 1.8

We are now going to use Apriori to extract frequent itemsets from the same dataset. The `apriori()` function is avaialble in `mlxtend` and can be used the same way we used `fpgrowth()` . We can measure how fast the two functions are using `timeit` .

```
In [23]: from mlxtend.frequent_patterns import apriori
         from timeit import timeit

         print("FP-growth", timeit(lambda: fpgrowth(df, 0.01), number=1), "seconds")
         print("Apriori", timeit(lambda: apriori(df, 0.01), number=1), "seconds")
```

```
FP-growth 7.691767792999997 seconds
Apriori 187.29349071900003 seconds
```

FP-growth is significantly faster than Apriori. This is because it leverages different, more efficient data structures than Apriori. You can learn more about these two algorithms on the course slides.

We can now re-compute the results obtained with the two algorithms to compare them.

```
In [24]: fi_apriori = apriori(df, 0.02)
         fi_fpgrowth = fpgrowth(df, 0.02)
         len(fi_apriori), len(fi_fpgrowth)

Out[24]: (303, 303)
```

We could have a first quantitative feedback on whether the two results are the same by observing the number of results we obtained. Indeed, both return 303 elements, meaning that the two algorithms are at least in accordance with the number of frequent itemsets that can be extracted. We cannot, however, compare the two results with `fi_apriori == fi_fpgrowth` , since the order among the rows in not necessarily the same.

The following is a quick digression on how we can compare the results with the knowledge we have of Python (i.e. without Pandas). If we want to run this kind of comparison we need to transform the results to make them comparable between one another. If we do not care about order, the best way of comparing two collections of elements is to convert them to sets.

```
In [25]: try:
             set(fi_apriori.values) == set(fi_fpgrowth.values)
         except Exception as e:
             print("An error occurred:", e)
```

```
An error occurred: unhashable type: 'numpy.ndarray'
```

Using a `try...except` , we can see that the attempt to convert the NumPy matrices (arrays of numpy arrays) into sets does not work as expected. This is because, as the returned error says, a NumPy array cannot be hashed. Having values that can be hashed, though, is at the basis of Python sets (once again, you can check Wikipedia's page for hash tables (https://en.wikipedia.org/wiki/Hash_table) as to why). In Python, only immutable types can be hashed directly: this means that we can hash strings, numbers and tuples, but not sets or lists. If in doubt, you can try to use the `hash()` function on a variable and see whether an exception is thrown:

In [26]:
```
for v in [ 0, "test", False, 2.2, (1,2,3), {1,2,3}, [1,2,3], {'a': 1, 'b': 
2, 'c': 3}]:
    try:
        hash(v)
        print(f"{v} ({type(v)}) can be hashed")
    except:
        print(f"{v} ({type(v)}) CANNOT be hashed")
```

```
0 (<class 'int'>) can be hashed
test (<class 'str'>) can be hashed
False (<class 'bool'>) can be hashed
2.2 (<class 'float'>) can be hashed
(1, 2, 3) (<class 'tuple'>) can be hashed
{1, 2, 3} (<class 'set'>) CANNOT be hashed
[1, 2, 3] (<class 'list'>) CANNOT be hashed
{'a': 1, 'b': 2, 'c': 3} (<class 'dict'>) CANNOT be hashed
```

Since we can hash tuples, we can convert each numpy array into a tuple, and build a set from there.

In [27]:
```
tuples_ap = { tuple(row) for row in fi_apriori.values }
tuples_fp = { tuple(row) for row in fi_fpgrowth.values }
```

Notice how we are using a "set comprehension": this is a natural extension of lists and dictionary comprehensions.

Now that we have obtained two sets, we can run the "unordered" comparison, and observe that now the results of the two algorithms match.

In [28]:
```
tuples_ap == tuples_fp
```

Out[28]:
```
True
```

2. Apriori implementation

Exercise 2.1

In this exercise, we will be implementing our own version of Apriori. The Apriori algorithm is described in detail in the course slides. In particular, the pseudo-code looks like this:

```
L_1 = { frequent items }
for (k = 1; L_k != {}; k++):
    C_{k+1} = generate_candidates(L_k)
    for transaction in transactions:
        count candidates in C_{k+1} contained in "transaction"
    L_{k+1} = candidates in C_{k+1} that satisfy "minsup"
return union of all L_k
```

Where `L_k` is the list of frequent itemsets of length `k`, while `C_{k+1}` is the list of all itemsets that can be generated from `L_k` based on the Apriori consideration.

In particular, we can identify 5 main steps in this algorithm:

1. Generate `L_1`
2. Generate `C_{k+1}` from `L_k`
3. Prune candidates that have non-frequent itemsets of length *k*
4. Count candidates presence in transactions
5. Filter candidates based on *minsup*

We will address one step at a time. Let's firt define the toy dataset we will be using throughout the exercise.

In [29]:
```python
transactions = [
    ["a","b"],
    ["b","c","d"],
    ["a","c","d","e"],
    ["a","d","e"],
    ["a","b","c"],
    ["a","b","c","d"],
    ["b","c"],
    ["a","b","c"],
    ["a","b","d"],
    ["b","c","e"]
]
```

From this, we can proceed with step 1, where we count the number of occurences for each item, then extract the list of frequent items (i.e. `L_1`). Differently from `mlxtend`'s implementation, we will be using a *minsup* expressed as an absolute value (instead of relative terms) for coherence with the course slides. To move from one notation to the other, we would simply need to divide/multiply *minsup* by the number of transactions. For this example, we will use *minsup* = 1.

To count the frequency of each item in the transactions, we will use a dictionary, where keys are items and values are counts of occurrences.

```
In [30]: freq = defaultdict(lambda: 0)
         minsup = 1
         for t in transactions:
             for el in t:
                 freq[el] += 1

         dict(freq)

Out[30]: {'a': 7, 'b': 8, 'c': 7, 'd': 5, 'e': 3}
```

From this, we need to filter the items that are not frequent (they are not needed, since all itemsets containing them will not be frequent, for the Apriori principle).

We will be using a list `L` to store the frequent itemsets. `L[k]` will contain all frequent itemsets of length `k+1`.

```
In [31]: L = []
         L.append({ (k,): v for k, v in freq.items() if v >= minsup })
         L[0]

Out[31]: {('a',): 7, ('b',): 8, ('c',): 7, ('d',): 5, ('e',): 3}
```

`L[0]` contains the itemsets of length 1, stored as a dictionary (with the already described syntax). Notice how we are storing the keys as tuples. Because keys in a dictionary must be hashable, we resort to tuples instead of lists or sets. The syntax `(k,)` creates a tuple with 1 element. We need to add a `,` since `(k)` in Python evaluates to the same as `k` (i.e. it does not create a tuple).

Now that we have created `L_1`, we can proceed with step 2, which consists in implementing `generate_candidates`, a function that receives values of `L_k` and generates all possible candidates of length `k+1`. We will do this by selecting itemsets that have the same `k-1`-prefix and merging suffixes.

```
In [32]: def generate_candidates(L):
             C = []
             for i in range(len(L)):
                 for j in range(i+1, len(L)):
                     # iterate over all possible pairs in L
                     if L[i][:-1] == L[j][:-1]:
                         # L[i] and L[j] have matching prefixes (they only differ by
the last item),
                         # so we generate the new element by using the same prefix an
d appending the
                         # two suffixes
                         C.append(L[i][:-1] + tuple(sorted([L[i][-1], L[j][-1]])))
             return C
```

Notice how we are sorting `[ L[i][-1], L[j][-1] ]`. This guarantees that all generated itemsets (stored as tuples) are sorted according to some order. This is needed to make sure that the prefix matching is always valid.

Let's run some tests on our code.

In [33]:
```
# expecting ab, ac, ad, bc, bd, cd
print(generate_candidates([ ('a',), ('b',), ('c',), ('d',) ]))
# expecting abc, cdf, cef, cde
print(generate_candidates([ ('a','b'), ('a','c'), ('b','c'), ('c','f'),
('c','d'), ('c','e')]))
# expecting an empty result
print(generate_candidates([]))
# expecting an empty result
print(generate_candidates([('a','b'), ('b','c'), ('c','d')]))
```

```
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]
[('a', 'b', 'c'), ('c', 'd', 'f'), ('c', 'e', 'f'), ('c', 'd', 'e')]
[]
[]
```

> While not specifically relevant for this course, you should look into test-driven development (https://en.wikipedia.org/wiki/Test-driven_development), an approach to software development that focuses on the creation of test suites even before implementing a solution.

Now that `generate_candidates` behaves as expected, we need to check whether these candidates contain any subset of items that is not frequent (step 3): indeed, any itemset that contains non-frequent susbsets cannot be frequent itself (based on the Apriori principle). Since our frequent itemsets of length *k* are contained in L[-1] (i.e. the itemsets we generated at the previous iteration), we can make this check quite easily: to this end, we should first compute all possible k-length subsets of our k+1 candidates.

We can use `itertools` ' `combinations` function to generate all k-length subsets: given a list of values and a length, `combinations()` returns a list of all combinations (without replacements) of our list.

In [34]:
```
from itertools import combinations
def prune_values(freq, candidates):
    keep = []
    for candidate in candidates:
        combs = list(combinations(candidate, len(candidate)-1))
        if all([ comb in freq for comb in combs ]):
            keep.append(candidate)
    return keep
```

This code will take as input `freq` (a set of all frequent frequent itemsets of length *k*) and `candidates` , a list of candidate itemsets of length *k+1* (i.e. the output of step 2). For each candidate itemset, all combinations without replacement of length *k* are generated: if any of these itemsets is not frequent (i.e. is not contained in `freq`), we will discard the candidate as it cannot be frequent itself. All other itemsets are stored in `keep` , which is then returned.

We can then proceed with step 4. This step is similar to step 1 since we need to iterate over all the transactions. Differently from step 1, though, we now have a set of candidates to check. This will result in a nested loop like the following.

In [35]:
```
C = prune_values(L[-1], generate_candidates(list(L[-1].keys()))) # generate new candidates from L_{k} (i.e. the keys of L[-1])
freq = defaultdict(lambda: 0) # keep track of itemsets counts, as we did for step 1
for t in transactions: # count frequencies in all transactions
    transaction_set = set(t)
    for c in C: # for each transaction, go over all candidates
        if set(c).issubset(transaction_set):
            freq[c] += 1
freq
```

Out[35]:
```
defaultdict(<function __main__.<lambda>()>,
            {('a', 'b'): 5,
             ('b', 'c'): 6,
             ('b', 'd'): 3,
             ('c', 'd'): 3,
             ('a', 'c'): 4,
             ('a', 'd'): 4,
             ('a', 'e'): 2,
             ('c', 'e'): 2,
             ('d', 'e'): 2,
             ('b', 'e'): 1})
```

Given a candidate itemset and a transaction, we can use the `.issubset()` method of sets to check whether the current candidate `c` is contained within `transaction_set`. If so, we increase `freq[c]` by 1.

Now that we have the frequencies, we can build `L_{k+1}` from `freq`, by filtering out those elements that do not reach *minsup*. This is the task for step 5.

In [36]:
```
L.append({ k: v for k,v in freq.items() if v >= minsup })
L
```

Out[36]:
```
[{('a',): 7, ('b',): 8, ('c',): 7, ('d',): 5, ('e',): 3},
 {('a', 'b'): 5,
  ('b', 'c'): 6,
  ('b', 'd'): 3,
  ('c', 'd'): 3,
  ('a', 'c'): 4,
  ('a', 'd'): 4,
  ('a', 'e'): 2,
  ('c', 'e'): 2,
  ('d', 'e'): 2,
  ('b', 'e'): 1}]
```

Now that we have built `L_{k+1}`, we can go through steps 2, 3, 4 and 5 as long as we keep finding frequent itemsets. When we stop (i.e. when `L[-1] == {}`), we know that there cannot be any frequent itemsets longer than the ones we have already found (because of the Apriori principle, all subsets of a frequent itemset of length `k` must also be frequent itemsets).

Let's put together steps 1 through 4 into a function, `my_apriori()`.

```
In [37]: def my_apriori(transactions, minsup):
             # step 1
             freq = defaultdict(lambda: 0)
             for t in transactions:
                 for el in t:
                     freq[el] += 1
             L = []
             L.append({ (k,): v for k,v in freq.items() if v >= minsup })
             while L[-1] != {}:
                 C = prune_values(L[-1], generate_candidates(list(L[-1].keys()))) # s
         tep 2
                 # step 3
                 freq = defaultdict(lambda: 0)
                 for t in transactions:
                     transaction_set = set(t)
                     for c in C:
                         if set(c).issubset(transaction_set):
                             freq[c] += 1
                 L.append({ k: v for k,v in freq.items() if v >= minsup }) # step 4
             return L[:-1] # do not return last element, as we know it is {}
```

We can now run our function with the `transactions` dataset from before. Notice that, since our implementation uses a comparison that is `>= minsup`, if we want the support to be > 1, we need to pass *minsup* = 2.

```
In [38]: my_apriori(transactions, 2)

Out[38]: [{('a',): 7, ('b',): 8, ('c',): 7, ('d',): 5, ('e',): 3},
          {('a', 'b'): 5,
           ('b', 'c'): 6,
           ('b', 'd'): 3,
           ('c', 'd'): 3,
           ('a', 'c'): 4,
           ('a', 'd'): 4,
           ('a', 'e'): 2,
           ('c', 'e'): 2,
           ('d', 'e'): 2},
          {('b', 'c', 'd'): 2,
           ('a', 'c', 'd'): 2,
           ('a', 'd', 'e'): 2,
           ('a', 'b', 'c'): 3,
           ('a', 'b', 'd'): 2}]
```

Exercise 2.2

We now need to load the COCO dataset into memory and turn it into a list of itemsets (much like toy `transactions` we used).

The COCO dataset used is a list of images, where each image has a list of annotations. These annotations might contain duplicates (since the same image may contain the same annotation (e.g. a person) multiple times).

```
In [39]: import json

         with open("modified_coco.json") as f:
             images = json.load(f)
         dataset = [ list(set(image["annotations"])) for image in images ] # remove d
         uplicates
         len(dataset)

Out[39]: 5000
```

Now that we have a list of all annotations for each transaction (or image), we can apply our version of Apriori to it. Notice that the dataset is much smaller than the previous one, with only 5,000 transactions. This is because, unlike the `mlxtend` 's implementations, our algorithm is not particularly optimized, and is significantly slower than the functions we previously used.

Exercise 2.3

We can now apply our Apriori implementation to `dataset` . As recommended by the exercise, we will be using 0.02 as *minsup*. Remember, though, that our implementation requires an absolute value for *minsup*, so we will be using 0.02 * 5,000.

The following code computes `L` , the list of all frequent itemsets, and prints a subset of 10 itemsets for each possible itemset length $k > 1$.

In [40]:
```
L = my_apriori(dataset, 0.02 * 5000)

for k, L_k in enumerate(L[1:]):
    print(k+2, list(L_k.keys())[:10])
```

```
2 [('car', 'stop sign'), ('bench', 'potted plant'), ('person', 'potted plan
t'), ('bench', 'dining table'), ('bench', 'chair'), ('bench', 'person'), ('d
ining table', 'person'), ('chair', 'person'), ('fire hydrant', 'person'),
('car', 'person')]
3 [('bench', 'dining table', 'person'), ('bench', 'chair', 'person'), ('ca
r', 'fire hydrant', 'person'), ('backpack', 'car', 'person'), ('bus', 'car',
'person'), ('bench', 'bicycle', 'person'), ('bench', 'person', 'umbrella'),
('bus', 'car', 'truck'), ('bus', 'car', 'traffic light'), ('car', 'traffic l
ight', 'truck')]
4 [('backpack', 'car', 'person', 'traffic light'), ('bus', 'car', 'person',
'traffic light'), ('bicycle', 'car', 'person', 'traffic light'), ('car', 'ha
ndbag', 'person', 'traffic light'), ('car', 'person', 'traffic light', 'truc
k'), ('baseball bat', 'baseball glove', 'bench', 'person')]
```

Most itemsets are particularly intuitive (e.g. {'car', 'stop sign'} or {'bench', 'person'}). We can pick any itemset and identify, in our original dataset, an image that contains those items. For example, for the itemset {}, we can extract one of the images containing it.

In [41]:
```
itemset = set(['baseball bat', 'baseball glove', 'bench', 'person'])
with_itemset = [ image['image_id'] for image in images if itemset.issubset(i
mage['annotations'])]
with_itemset[15]
```

Out[41]:
```
288440
```

And, by checking COCO dataset's explore tool, we can see what picture 288440 looks like. Indeed, the entire itemset extracted is related to baseball pictures.

Exercise 2.4

For this exercise, we need to run `mlxtend` 's functions on COCO. To do so, we first need to convert the dataset into a DataFrame. For this, we can use the same code we did before, with minor adjustments.

```
In [42]: all_items_set = set()
         for items in dataset:
             all_items_set.update(items)
         all_items = sorted(list(all_items_set))
         presence_matrix = [ [ int(item in image) for item in all_items ] for image i
         n dataset ]
         df = pd.DataFrame(presence_matrix, columns=all_items_set)
```

Now, we can apply FP-growth and Apriori.

```
In [43]: fi_ap = apriori(df, 0.02)
         fi_fp = fpgrowth(df, 0.02)
```

To proceed with the comparison of the results, we should first convert all outputs to comparable results. Based on the considerations made in Exercise 1.8, we can convert `fi_ap` and `fi_fp` to sets of pairs (support, itemset), and run the comparison there.

```
In [44]: tuples_ap = { tuple(row) for row in fi_ap.values }
         tuples_fp = { tuple(row) for row in fi_fp.values }
         tuples_ap == tuples_fp

Out[44]: True
```

Once again, FP-growth and Apriori's results are in accordance with one another. We now need to compare them to our results. To do this, we first need to convert `my_apriori` 's output into a set of (support, itemset) tuples.

For these tuples, the support needs to be expressed in relative terms (i.e. as a fraction of the entire dataset).

The itemset, instead, needs to be expressed as a `frozenset` , which is the same type used in `tuples_ap` and `tuples_fp` . Frozenset are similar to sets, with the difference that they are immutable and, as such, hashable.

Additionally, `tuples_ap` and `tuples_fp` 's representations of itemsets use the index of the item instead of the item itself, so we need to extract this index as well for our output. We can do this using the `.index()` method that lists have, which returns the index within the array of the element passed as argument.

In [45]:
```
fi_myap = set()
for L_k in L:
    # - v/5000 converts the support from absolute to relative
    # - frozenset() builds an immutable set
    # - { all_items.index(k_) for k_ in k } builds a set where
    #   the elements are the indices (in the list all_items)
    #   of the elements in k
    fi_myap.update({ (v/5000, frozenset({all_items.index(k_) for k_ in k}))
for k,v in L_k.items() })
```

In [46]:
```
tuples_ap == fi_myap
```

Out[46]:
```
True
```

We can reasonably assume that our Apriori implementation works similarly to the ones offered by `mlxtend` .

Exercise 2.5

We can use `timeit` to measure the time needed for the three functions to compute the itemsets.

In [47]:
```
print("fpgrowth", timeit(lambda: fpgrowth(df, 0.02), number=1))
print("apriori", timeit(lambda: apriori(df, 0.02), number=1))
print("my_apriori", timeit(lambda: my_apriori(dataset, 0.02 * 5000), number=
1))
```

```
fpgrowth 0.04938938700001927
apriori 0.05601436599999943
my_apriori 0.6605909099999963
```

As expected, our implementation is significantly (almost 20x) slower than `mlxtend` 's versions. With your knowledge of NumPy and based on the input representation required by `mlxtend` 's functions, you can figure out how their implementation is so efficient. You could implement a different Apriori version that leverages NumPy and see what kind of improvement you would get over our version.

Exercise 2.6

For this exercise, we need to be able to compute all combinations of length *k* from a pool of *n* elements. In our case, *n* is `len(all_items)` . We can leverage the `itertools` library. This library offers the function `combinations` : given a list of values and a value *k*, this function generates all possible subsets of length *k* that can be built.

In [48]:
```python
from itertools import combinations

list(combinations(['a','b','c'], 2))
```

Out[48]:
```
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

We can try and generate all possible combinations of length *k* (with *k* starting from 1) and counting the frequencies, as we have already seen. The approach is similar to the one used for `my_apriori` (the two algorithms share the same outer structure) but, in this case, the candidates are generated from all possible combinations.

We are also introducing new parameters: `all_items` (which is a list of all possible items, used to generate all possible combinations), and `max_k` , which is the maximum length of the generated items (without Apriori assumptions, we need to define the upper length we want to reach).

In [49]:
```python
def naive_frequent_itemsets(transactions, all_items, minsup, max_k):
    freq = defaultdict(lambda: 0)
    for t in transactions:
        transaction_set = set(t)
        for k in range(1, max_k+1):
            for c in combinations(all_items, k): # try all combinations of l
ength k!
                if set(c).issubset(transaction_set):
                    freq[c] += 1
    return { k: v for k,v in freq.items() if v >= minsup }
```

We can now measure the execution time for `max_k = 1, 2, 3` .

In [50]:
```python
for max_k in range(1,4):
    print(max_k, timeit(lambda: naive_frequent_itemsets(dataset, all_items,
100, max_k), number=1))
```

```
1 0.11986853999997038
2 4.1218699799999285
3 117.52111443399997
```

As *max_k* grows, the execution time grows more than exponentially. Indeed, since the number of combinations of length *k* that can be built from a set of *n* elements is given by the binomial coefficient, the execution time is tied to it. The `scipy` library offers the `binom` function to compute the binomial coefficient given *k* and *n*. With n = 78 and k = 1,2,3, we get the following binomial coefficients:

In [53]:
```python
from scipy.special import binom

for k in [1,2,3]:
    print(binom(78,k))
```

```
78.0
3003.0
76076.0
```

Based on this, it is fairly obvious that this naive approach cannot be scaled to any useful application. As a last consideration, notice how the ratios between the execution times is similar to the ratios in binomial coefficients:

In [54]:
```python
((4.1218699799999285 / 0.11986853999997038, 3003 / 78),
(117.52111443399997 / 4.1218699799999285, 76076 / 3003))
```

Out[54]:
```
((34.38658700607304, 38.5), (28.51160153140056, 25.333333333333332))
```

This is clearly because the execution time is proportional to its complexity, which is driven by the binomial coefficients. With this, we can estimate how long it would take to run for `max_k = 4`.

```
In [56]: binom(78,4) / binom(78,3) * 117.52111443399997
```

```
Out[56]: 2203.5208956374995
```

Which is approximately 35 minutes. Probably not worth the waiting, considering the better options we have spent so much time building.