CSCI4022_F21_HW5

October 20, 2021

1 CSCI4022 Homework 5; Itemsets

1.1 Due Monday, October 18 at 11:59 pm to Canvas and Gradescope

Submit this file as a .ipynb with *all cells compiled and run* to the associated dropbox.

Your solutions to computational questions should include any specified Python code and results as well as written commentary on your conclusions. Remember that you are encouraged to discuss the problems with your classmates, but **you must write all code and solutions on your own**.

NOTES:

- Any relevant data sets should be available on Canvas. To make life easier on the graders if they need to run your code, do not change the relative path names here. Instead, move the files around on your computer.
- If you're not familiar with typesetting math directly into Markdown then by all means, do your work on paper first and then typeset it later. Here is a reference guide linked on Canvas on writing math in Markdown. **All** of your written commentary, justifications and mathematical work should be in Markdown. I also recommend the wikibook for LaTex.
- Because you can technically evaluate notebook cells is a non-linear order, it's a good idea to do **Kernel** → **Restart & Run All** as a check before submitting your solutions. That way if we need to run your code you will know that it will work as expected.
- It is **bad form** to make your reader interpret numerical output from your code. If a question asks you to compute some value from the data you should show your code output **AND** write a summary of the results in Markdown directly below your code.
- 45 points of this assignment are in problems. The remaining 5 are for neatness, style, and overall exposition of both code and text.
- This probably goes without saying, but… For any question that asks you to calculate something, you **must show all work and justify your answers to receive credit**. Sparse or nonexistent work will receive sparse or nonexistent credit.
- There is *not a prescribed API* for these problems. You may answer coding questions with whatever syntax or object typing you deem fit. Your evaluation will primarily live in the clarity of how well you present your final results, so don't skip over any interpretations! Your code should still be commented and readable to ensure you followed the given course algorithm.

Shortcuts: Problem 1 | Problem 2 | Problem 3 |

---

[13]:
```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.stats as stats
import statsmodels.api as sm
```

Back to top # Problem 1 (Practice: Candidate Items; 20 pts)

In the A-Priori algorithm, there is a step in which we create a candidate list of frequent itemsets of size $k+1$ as we prune the frequent itemsets of size k. This this problem we will create two functions to do that formally.

Part A: There are two types of data objects in which we might be holding the frequency counts of itemsets. If $k = 2$, they may be stored in a triangular array. Create a function `Cand_Trips` that takes a triangular array and returns all valid candidate triples as a list. Recall that the itemset $\{i, j, k\}$ is only a candidate if all 3 of the itemsets in $\{\{i, j\}, \{i, k\}, \{k, j\}\}$ are frequent.

Some usage notes:

- The first input argument is `Triang_Counts`, a zero-indexed triangular (numeric) array, by same convention as introduced in class.
- The second input argument is the positive integer support threshold `s`.
- The underlying itemset is 0-indexed, so e.g. `[0,1,3]` is a valid triple.
- You should not convert the input list `Triang_Counts` into a list of triples as part of your function.
- The return array `Candidates` should be a list of 3-index lists of the item numbers of the triples. So a final answer for some input might be:

`Cand_Trips = [[0,3,4], [1,2,7]]`

- An implementation note: there are two fundamentally different ways to think about implementing this function. Option 1 involves thinking about the elements of `Tri_Counts` in terms of their locations on the corresponding *triangular matrix*: scan row i for a pair of frequent pairs $\{\{i, j\}, \{i, k\}\}$ and then check if $\{j, k\}$ is in fact frequent. Option 2 scans all of `Tri_Counts` for frequent item pairs (the "pruning" step) and saves those in some object with their indices, then scans *that* object for candidates. Both are valid for this problem, but option 2 may generalize to higher k better…

[14]:
```python
def Cand_Trips(Triang_Counts, s):
    '''trying to find k = 3 candidates combinations'''
    k             = 3

    # we're dealing with a nxn matrix, inputted as a triangular 1-D array
    n             = get_n(len(Triang_Counts))

    # we can discard any candidate pairs that don't meet our support threshold
    pruned_indices = prune_pairs(get_all_pairs(n), Triang_Counts, s)
```

```
    # from the pruned candidate pairs, we get all the possible candidate triples
    triples        = get_all_triples(pruned_indices)

    pi_in_triples  = get_triple_subsets(triples, pruned_indices, k)

    counts         = get_counts(pi_in_triples, k)

    return return_candidate_triples(counts, triples)
'''###########################################################################################
### Returns the dimension 'n' of an nxn matrix given the length of a 1-D␣
↪triangular array #
### if were not given a valid 1-D triangular array length, this function␣
↪returns -1       #
#                                                                                ␣
↪          #
# example:                                                                       ␣
↪          #
#          nxn matrix is 5x5, 1-D array will be of length 10.                    ␣
↪          #
#          we must solve: n^2 - n - 2 * (length of 1-D array) = 0                ␣
↪          #
#          5^2 - 5 - 2(10) = 25 - 5 - 20 = 0 with n = 5                          ␣
↪          #
###########################################################################################'''
def get_n(length_1D):
    # the first root is what we are after here after some testing, np.roots is␣
↪pretty nifty
    ret_val = int(np.roots([1, -1, -2*length_1D])[0])
    # (range(n)) creates a generator which saves space and time, I learned␣
↪about them recently on YouTube!
    return ret_val if sum((range(1, ret_val))) == length_1D else -1

'''#############################################################################################
### Returns all possible indice pairs for an nxn upper triangular matrix         ␣
↪            #
#                                                                                ␣
↪            #
# example:                                                                       ␣
↪            #
#          nxn matrix is 5x5 so we will get 10 possible indice pairs much like␣
↪in get_n().  #
#          However, these indice pairs will be a list of [i, j] combinations:    ␣
↪            #                                    #
#          [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [1, 3], [1, 4], [2, 3], [2,␣
↪4], [3, 4]] #
###########################################################################################'''
```

```
def get_all_pairs(n):
    # simple list comprehension to return all pairs as a set for ease of use in the future
    return [set([i, j]) for i in range(n) for j in range(i+1, n)]

'''###########################################################################################
### Returns only the indices where the count for the pair is greater than or equal to the #
### support threshold                                                                      #
#                                                                                          #
# example:                                                                                 #
#          Fairly straightforward: if the count for a pair is less than the threshold,    #
#          we exclude it. This may be useful in my actual A-Priori implementation.        #
###########################################################################################'''
def prune_pairs(indices, counts_1D, support):
    # another simple list comprehension to prune only the pairs that meet the threshold
    return [indices[i] for i, count in enumerate(counts_1D) if count >= support]

'''###########################################################################################
### Gets all of the possible triples given all inputted pairs                              #
#                                                                                          #
# example:                                                                                 #
#          I'm not entirely sure how this is typically implemented to do this more        #
#          iteratively, this was relatively easy "brute-forced". Either way, this         #
#          function is fairly straightforward, need to figure this out once we get into   #
#          higher dimensions.                                                              #
###########################################################################################'''
def get_all_triples(indices):
    k       = 3
    i_len   = len(indices)
    triples = set()
    for i in range(i_len):
        for j in range(i+1, i_len):
```

```
            set_union = indices[i].union(indices[j])
            if len(set_union) == k:
                triples.add(frozenset(tuple(set_union)))
    return triples

'''##########################################################################################
### Gets the indices of my possible triples where a given pair is a subset, and␣
↪makes a   #
### dictionary                                                                         ␣
↪          #
#                                                                                      ␣
↪          #
# example:                                                                             ␣
↪          #
#          Fairly straightforward: if the count for a pair is less than the␣
↪threshold,    #
#          we exclude it.                                                              ␣
↪          #
#########################################################################################'''
def get_triple_subsets(triples, indices, k):
    return {tuple(index): [i for i, s in enumerate(triples) if set(index).
↪issubset(s)] for index in indices}

def get_counts(indices_in_triples, k):
    arr    = [item for i, (k, v) in enumerate(indices_in_triples.items()) for␣
↪item in v]
    counts = set(sorted([(i, arr.count(i)) for i in arr if arr.count(i) >= k]))
    return counts

def return_candidate_triples(counts, triples):
    arr = list(triples)
    return [list(arr[item[0]]) for item in counts]
```

---

2 Note:

Honestly, wasn't really a fan of using the triangular 1-D array, although I can totally understand its use case especially for a program written in C or C++ or something. Ultimately, I guess once we get up to $k > 2$, it's use-case sorta diminishes, and then we just deal with sets essentially.

---

[15]:
```
'''##########################################################################################
### Needed a visual so I could count how many elements would be in a 6x6␣
↪triangular 1-D array #
```

```python
############################################################################################'
C_5 = np.array([[0, 10, 7, 3, 2],
                [0,  0, 6, 4, 3],
                [0,  0, 0, 3, 6],
                [0,  0, 0, 0, 0],
                [0,  0, 0, 0, 0]])

C_6 = np.array([[0, 10, 7, 6, 6, 1],
                [0,  0, 6, 6, 7, 1],
                [0,  0, 0, 6, 6, 1],
                [0,  0, 0, 0, 8, 1],
                [0,  0, 0, 0, 0, 1],
                [0,  0, 0, 0, 0, 0]])

'''####################################################
### Needed a refresher on pythons set methods         #
# https://www.w3schools.com/python/python_ref_set.asp #
#                                                     #
# Note: This would be quite labor intensive in C or   #
# C++, but I'm sure it'd be so quick.                 #
####################################################'''
x    = {"apple", "banana", "cherry"}
y    = {"google", "microsoft", "apple"}

z    = x.union(y)
zz   = x.intersection(y)
zzz  = x.difference(y)
zzzz = y.difference(x)

print(z)
print(zz)
print(zzz)
print(zzzz)

# wanted to play around with typing with the sets for later on
s = {(1, 2)}
print(s)
s.add(tuple({1,2}))
print(s)

ss   = {"hello"}
sss  = {"a", "b"}

print(tuple(ss))
print(tuple(sss))

for i in sss:
```

```
    print(i)
'''###########################################################################################
### Wanted a pythonic way of obtaining counts from a list                                    ␣
 ↪             #
# https://stackoverflow.com/questions/2600191/
 ↪how-can-i-count-the-occurrences-of-a-list-item #
#########################################################################################'''
l = [1,7,7,7,3,9,9,9,7,9,10,0]
print(set([(i, l.count(i)) for i in l]))

'''#################################################################
### Sanity check for my get_n() function for triangular 1-D arrays #
#################################################################'''
print(get_n(10))
print(get_n(15))

'''###########################################################################################
### Needed to play around with dictionaries to remind myself of somethings.␣
 ↪Also, I forgot   #
### how to delete items from a dictionary and the following link helped a lot. ␣
 ↪             #
# https://www.askpython.com/python/dictionary/delete-a-dictionary-in-python    ␣
 ↪             #
#########################################################################################'''
d = {0: "a", 1: "b", 2: "c"}
for k, v in d.items():
    print(k, v)
```

```
{'microsoft', 'cherry', 'apple', 'google', 'banana'}
{'apple'}
{'banana', 'cherry'}
{'microsoft', 'google'}
{(1, 2)}
{(1, 2)}
('hello',)
('a', 'b')
a
b
{(0, 1), (3, 1), (7, 4), (10, 1), (9, 4), (1, 1)}
5
6
0 a
1 b
2 c
```

Part B: A quick test case. Below is a matrix M and code including its corresponding the triangular array.

$$C = \begin{bmatrix} \cdot & 10 & 7 & 3 & 2 \\ \cdot & \cdot & 6 & 4 & 3 \\ \cdot & \cdot & \cdot & 3 & 6 \\ \cdot & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Input the given list into your function to verify that it returns the correct valid triples at $s = 1$ and $s = 6$.

[16]:
```python
Triang_Counts  = [10,7,3,2,6,4,3,3,6,0]
Triang_Counts2 = [10,7,6,6,1,6,6,7,1,6,6,1,8,1,1]



'''
Check that...
'''
# Cand_Trips(Triang_Counts, 1) returns all the possible triples except those␣
↪that contain BOTH items 4 and 5.
print(Cand_Trips(Triang_Counts, 1), '\n')

# Wanted to see how I would move forward with the problem making sure I have␣
↪the proper number of sets to make
# a quad.
print(Cand_Trips(Triang_Counts2, 1), '\n')

# Cand_Trips(Triang_Counts, 6) returns only the triple [[0,1,2]].
print(Cand_Trips(Triang_Counts, 6), '\n')

# Wanted to see how I would move forward with the problem making sure I have␣
↪the proper number of sets to make
# a quad, figured I'd also see if I could come up with a smaller example to␣
↪work with
print(Cand_Trips(Triang_Counts2, 6), '\n')
```

```
[[1, 2, 3], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [1, 2, 4], [0, 1, 2]]

[[2, 3, 5], [2, 3, 4], [1, 2, 4], [1, 3, 5], [1, 3, 4], [0, 2, 4], [1, 2, 5],
[0, 1, 4], [0, 4, 5], [0, 1, 5], [0, 1, 2], [0, 1, 3], [1, 2, 3], [0, 2, 5], [0,
2, 3], [3, 4, 5], [2, 4, 5], [0, 3, 4], [0, 3, 5], [1, 4, 5]]

[[0, 1, 2]]

[[1, 2, 3], [0, 1, 3], [0, 1, 4], [0, 2, 3], [1, 3, 4], [0, 2, 4], [2, 3, 4],
[1, 2, 4], [0, 3, 4], [0, 1, 2]]
```

Part C: Suppose instead that our $k = 2$ item counts were stored in a list of the form e.g. `Pairs_Counts` = `[[0,1,12], [0,2,0], [0,3,11], ..., [7,8,103]]`

Where each element is a triple storing the two item indices and their count, $[i, j, c_{ij}]$.

Create a function `Cand_Trips_List` that takes in a list of pairs counts and returns all valid candidate triples as a list.

Some usage notes:

- The first input argument is `Pairs_Counts`, a zero-indexed list of triples.
- The second input argument is the positive integer support threshold `s`.
- The underlying itemset is 0-indexed, so e.g. `[0,1,3]` is a valid triple.
- The return array `Candidates` should be a list of 3-element lists, as above.

You should **not** convert the input list `Pairs_Counts` into a triangular array as part of your function. After all, sometimes we use the list format for pairs because it saves memory compared to the triangular array format! You may be able to borrow heavily from the logic of your first function, though!

[17]:
```
def Cand_Trips_List(Pairs_Counts, s):
    # trying to find k = 3 candidates combinations
    k              = 3

    # we can discard any candidate pairs that don't meet our support threshold
    pruned_indices = prune_pairs2(Pairs_Counts, s)

    # from the pruned candidate pairs, we get all the possible candidate triples
    triples        = get_all_triples(pruned_indices)

    pi_in_triples  = get_triple_subsets(triples, pruned_indices, k)

    counts         = get_counts(pi_in_triples, k)

    return return_candidate_triples(counts, triples)

def prune_pairs2(Pairs_Counts, s):
    # simple list comprehension to return all pairs as a set for ease of use in
 ↪the future
    return [set([i[0], i[1]]) for i in Pairs_Counts if i[2] > s]
```

Part D: Do the test case again. Below is the list reprentation of the same matrix M from part B.

Input the given list into your function to verify that it returns the correct valid triples at $s = 1$ and $s = 6$.

[18]:
```
Pairs_Counts=[[0,1,10], [0,2,7], [0,3,3], [0,4,2],\
             [1,2,6],[1,3,4], [1,4,3],\
             [2,3,3],[2,4,6],\
```

```python
             [3,4,0]]

'''
Check that...
'''
# Cand_Trips(Triang_Counts, 1) returns all the possible triples except those␣
↪that contain BOTH items 4 and 5.
print(Cand_Trips(Triang_Counts, 1), '\n')

# Wanted to see how I would move forward with the problem making sure I have␣
↪the proper number of sets to make
# a quad.
# print(Cand_Trips(Triang_Counts2, 1), '\n')

# Cand_Trips(Triang_Counts, 6) returns only the triple [[0,1,2]].
print(Cand_Trips(Triang_Counts, 6), '\n')

# Wanted to see how I would move forward with the problem making sure I have␣
↪the proper number of sets to make
# a quad, figured I'd also see if I could come up with a smaller example to␣
↪work with
# print(Cand_Trips(Triang_Counts2, 6), '\n')
```

```
[[1, 2, 3], [0, 1, 3], [0, 1, 4], [0, 2, 3], [0, 2, 4], [1, 2, 4], [0, 1, 2]]

[[0, 1, 2]]
```

Part E Describe *in words* how you would generalize your code in part D to work for generating candidate quadruples $[i_1, i_2, i_3, i_4]$ from an input list of triples counts (each element of the form $[i, j, k, c_{ijk}]$).

Back to top # Problem 2 (Practice: A-Priori; 25 pts)

Consider the Online Retail data set provided in `onlineretail.csv`. This includes over 500,000 purchases from an online retailer.

We want to use the baskets (marked by `InvoiceNo`) and the items (marked by `StockCode` and/or `Description`) to perform an item basket analysis.

This data set is small enough to run directly from main memory, so you may do that if you wish. You may also complete this problem using only the first 100,000 entries of the .csv if you wish for shorter computational time. Be very explicit which you are using.

a) There are some odd entries in the data set. Make sure that you're discarding any transactions and items with no `Description`, non-positive `Quantity`, or non-positive

Unit Price.

[19]:
```python
df = pd.read_csv('onlineretail.csv', encoding='unicode_escape')
print("Size initially:", len(df.index)) # 541909
print("Based on this, we only had", len(df.InvoiceNo.unique()), "number of␣
↪unique purchases, based on InvoiceNo")

# Removing negative Quantities
df = df[df.Quantity >= 0]
print("Size after removing non-negative Quantities:", len(df.index)) # 531285

# Removing negative Unit Prices
df = df[df.UnitPrice >= 0]
print("Size after removing non-negative UnitPrice:", len(df.index)) # 531283

# Removing 'some' non-existent Descriptions
df = df.dropna(subset=['Description'])
print("Size after removing non-existent Description:", len(df.index)) # 530691

print("Now after cleaning the data, we only had", len(df.InvoiceNo.unique()),␣
↪"number of unique purchases, based on InvoiceNo")

'''So it's confirmed I've dropped every row that has invalid values based on␣
↪part A.'''
df.tail(15)
```

```
Size initially: 541909
Based on this, we only had 25900 number of unique purchases, based on InvoiceNo
Size after removing non-negative Quantities: 531285
Size after removing non-negative UnitPrice: 531283
Size after removing non-existent Description: 530691
Now after cleaning the data, we only had 20134 number of unique purchases, based
on InvoiceNo
```

[19]:
```
       InvoiceNo StockCode                        Description  Quantity  \
541894    581587     22631           CIRCUS PARADE LUNCH BOX         12
541895    581587     22556      PLASTERS IN TIN CIRCUS PARADE        12
541896    581587     22555          PLASTERS IN TIN STRONGMAN        12
541897    581587     22728          ALARM CLOCK BAKELIKE PINK         4
541898    581587     22727           ALARM CLOCK BAKELIKE RED         4
541899    581587     22726         ALARM CLOCK BAKELIKE GREEN         4
541900    581587     22730         ALARM CLOCK BAKELIKE IVORY         4
541901    581587     22367    CHILDRENS APRON SPACEBOY DESIGN         8
541902    581587     22629                SPACEBOY LUNCH BOX         12
541903    581587     23256         CHILDRENS CUTLERY SPACEBOY         4
541904    581587     22613        PACK OF 20 SPACEBOY NAPKINS        12
541905    581587     22899        CHILDREN'S APRON DOLLY GIRL         6
541906    581587     23254       CHILDRENS CUTLERY DOLLY GIRL         4
```

```
541907    581587     23255  CHILDRENS CUTLERY CIRCUS PARADE         4
541908    581587     22138    BAKING SET 9 PIECE RETROSPOT          3

             InvoiceDate  UnitPrice  CustomerID Country
541894  12/9/2011 12:50       1.95     12680.0  France
541895  12/9/2011 12:50       1.65     12680.0  France
541896  12/9/2011 12:50       1.65     12680.0  France
541897  12/9/2011 12:50       3.75     12680.0  France
541898  12/9/2011 12:50       3.75     12680.0  France
541899  12/9/2011 12:50       3.75     12680.0  France
541900  12/9/2011 12:50       3.75     12680.0  France
541901  12/9/2011 12:50       1.95     12680.0  France
541902  12/9/2011 12:50       1.95     12680.0  France
541903  12/9/2011 12:50       4.15     12680.0  France
541904  12/9/2011 12:50       0.85     12680.0  France
541905  12/9/2011 12:50       2.10     12680.0  France
541906  12/9/2011 12:50       4.15     12680.0  France
541907  12/9/2011 12:50       4.15     12680.0  France
541908  12/9/2011 12:50       4.95     12680.0  France
```

So we have roughly $530,000$ rows, but only around $20,000$ baskets. This definitely makes sense that the amount of purchases is much lower than total items bought. However, this means we have an average of around $\frac{530{,}000}{20{,}000}$ of 26.5 items per basket. We probably don't have 26.5 items in every basket but, unless I'm brazenly misunderstanding the dataset, it's something to keep in mind. In fact, I ran a test with my new-found knowledge of generators down below, and we have a reasonably similar amount of baskets with 26 or more items and baskets with less than 26 items. Further, we have more baskets that are smaller than 26, than we do baskets that are equal or larger than 26.

---

[20]:
```python
total = 0
low   = 0
high  = 0

l = 26
h = 26
for i in (df.groupby('InvoiceNo')['StockCode'].apply(list).values):
    total += 1
    if len(i) < l:
        low += 1
    if len(i) >= h:
        high += 1
print("Total baskets", total)
print("Baskets with fewer than {} items: {}". format(l, low))
print("Baskets with {} or more items: {}". format(h, high))
```

```
Total baskets 20134
```

```
Baskets with fewer than 26 items: 14246
Baskets with 26 or more items: 5888
```

b) For our first iteration, we will use just `StockCode` for the items. Use `StockCode` to create a table of frequent single items at 1% support threshold. For convenience on this part of the problem and part c), you may choose to discard all items with non-integer values in `StockCode`. You may use Python's native classes to set up your lookup functions/tables. Was 1% an appropriate support threshold? Describe why or why not.

[21]:
```python
'''
So we lose roughly 55,000 rows when we do this, but I think I can make it work␣
↪without because I'm gonna
be using sets.
'''
####################################################################
# df = df[pd.to_numeric(df['StockCode'], errors='coerce').notnull()]
# print(len(df.index)) # 477605
####################################################################

num_baskets       = len(df.InvoiceNo.unique())
support_threshold = 0.01
s                 = int(num_baskets * support_threshold)

print("An item is frequent if it appears in {} or more baskets".format(s), '\n')

d = {}
for basket in (df.groupby('InvoiceNo')['StockCode'].apply(list).values):
    for item in basket:
        if item not in d.keys():
            d[item] = 0
        d[item] += 1

print("Number of items in the dictionary before pruning: {}".format(len(d)))
for key in d.copy():
    if d[key] < s:
        returned_value = d.pop(key, False)
        if not returned_value:
            print("Couldn't remove key: {}, with value {}".format(key,␣
↪returned_value))
print("Number of items in the dictionary after pruning: {}".format(len(d)))
sorted_d = [(key, val) for key, val in sorted(d.items(), key=lambda x: x[1],␣
↪reverse=True)]
print(sorted_d[:10])
```

```
An item is frequent if it appears in 201 or more baskets

Number of items in the dictionary before pruning: 3925
```

```
Number of items in the dictionary after pruning: 837
[('85123A', 2270), ('85099B', 2115), ('22423', 2019), ('47566', 1707), ('20725',
1595), ('84879', 1489), ('22197', 1426), ('22720', 1399), ('21212', 1370),
('20727', 1328)]
```

c) Use A-priori to find all frequent pairs of items from your set of frequent items in a). Use whatever support threshold you feel is most appropriate. Report the confidences of the two association rules corresponding to the most frequent item pair.

[22]:
```
class APriori:
    def __init__(self, cleaned_df, basket_column, item_column, k_tuples,␣
↪support_threshold):
        self.df              = cleaned_df
        self.k               = k_tuples
        self.n               = 0
        # trying to dip my toes in error/exception handling... kinda messy at␣
↪the moment
        try:
            length          = len(cleaned_df[basket_column].unique())
        except KeyError as a:
            raise NameError("Basket Column is invalid!") from a
        else:
            self.n_baskets = length
            self.s          = int(support_threshold * length)
            self.b_col      = basket_column
        try:
            _ = cleaned_df[item_column]
        except KeyError as b:
            raise NameError("Item Column is invalid!") from b
        else:
            self.i_col      = item_column

    def performAPriori(self, use_hashing=False):

        def first_pass():
            counts = {}
            # counting step that I am resuing from before
            for basket in (self.df.groupby(self.b_col)[self.i_col].apply(list).
↪values):
                for item in basket:
                    if item not in counts.keys():
                        counts[item] = 0
                    counts[item] += 1
            # DEBUG PURPOSES
#             print("Total number of single items: {}".format(len(counts)))
            return prune(counts)
```

```
#         def tri_arr_index(i, j):
#             return i * ( self.n - ( ( i + 1 ) / 2 ) ) + j - i - 1

        def second_pass(single_counts):
            keys          = single_counts.keys()
            double_counts = {}
            # loop through each basket
            for i, basket in enumerate((self.df.groupby(self.b_col)[self.i_col].
↪apply(list).values)):
                # loop through each item only if its a frequent item
                length = len(basket)
                for i1 in (range(length)):
                    # loop through each item only if its a frequent item and
                    # it's not equal to item in the loop above
                    for i2 in (range(i1+1, length)):
                        # weird stuff to make sure duplicate keys are not added␣
↪to dict
                        ks = double_counts.keys()
                        t  = (basket[i1], basket[i2])
                        if t not in ks:
                            double_counts[t] = 0
                        double_counts[t] += 1
                # DEBUG PURPOSES
#                 print("{}/{}".format(i+1, self.n_baskets))
            # DEBUG PURPOSES
#             print("Total number of pairs: {}".format(len(double_counts)))
            return prune(double_counts), confidence(single_counts,␣
↪double_counts)

        def confidence(old_counts, new_counts):
            confidences = []
            for s_old, c_old in old_counts.items():
                set_old = set([s_old]) if type(s_old) != tuple else set(s_old)
                for s_new, c_new in new_counts.items():
                    set_new = set(s_new)
                    # DEBUG PURPOSES
#                     print(set_old, set_new)
                    if set(set_old).issubset(set_new):
                        d_set      = set_new.difference(set_old)
                        difference = list(d_set)[0] if len(d_set) == 1 else␣
↪tuple(d_set)
                        tup        = tuple([(c_new / c_old), "{} -> {}".
↪format(s_old, difference), c_new])
                        confidences.append(tup)
                        c_diff     = old_counts[difference]
                        tup        = tuple([(c_new / c_diff), "{} -> {}".
↪format(difference, s_old), c_new])
```

```
                        confidences.append(tup)
            return confidences

        def prune(counts):
            for key in counts.copy():
                if counts[key] < self.s:
                    try:
                        del counts[key]
                    except KeyError:
                        print("Couldn't remove key: {}".format(key))
            self.n = len(counts)
            return counts

        if use_hashing:
            pass

        frequent_items                  = first_pass()
        frequent_pairs, confidences     = second_pass(frequent_items)

        sorted_freq_pairs  = [(key, val) for key, val in sorted(frequent_pairs.
 ↪items(), key=lambda x: x[1], reverse=True)]
        sorted_confidences = [(i[0], i[1]) for i in sorted(confidences,␣
 ↪key=lambda x: x[2], reverse=True)]
#         sorted_confidences = [(i[0], i[1]) for i in confidences.
 ↪sort(key=lambda x: x[2], reverse=True)]

        for k in range(2, self.k):
            pass
          # DEBUG PURPOSES
#         print("Total number of frequent items: {}".
 ↪format(len(frequent_items)))
#         print("Total number of frequent pairs: {}".
 ↪format(len(frequent_pairs)))
#         for _ in range(self.k_tuples - 1):
        return sorted_freq_pairs, sorted_confidences
```

---

3 Note:

I suppose I kind of misinterpretted the problem and I repeat part a. in my `APriori` class. I'm going to need to add an option for adding a dictionary of counts, so we can bypass the `first_pass()` function.

Furthermore, I've been at this for absolute hours, and I can't seem to fix the bug that some items appear more as a frequent pair with another item than either item appears singly. **I got it!!!**

Finally, I still have to implement the rest of the algorithm to get frequent k-tuples, but that'll be for another day because I'm already really late.

---

[23]:
```
ap = APriori(df, "InvoiceNo", "StockCode", 2, 0.01)

freq_pairs, confidences = ap.performAPriori() # 348638

print(freq_pairs[:10])
print(confidences[:10])
```

```
[(('22697', '22698'), 613), (('22386', '85099B'), 542), (('22697', '22699'),
527), (('22411', '85099B'), 466), (('85099B', 'DOT'), 461), (('21931',
'85099B'), 455), (('20725', '20727'), 440), (('22698', '22699'), 414),
(('85099B', '85099C'), 394), (('22726', '22727'), 393)]
[(0.5877277085330777, '22697 -> 22698'), (0.7720403022670025, '22698 -> 22697'),
(0.7720403022670025, '22698 -> 22697'), (0.5877277085330777, '22697 -> 22698'),
(0.4378029079159935, '22386 -> 85099B'), (0.25626477541371157, '85099B ->
22386'), (0.25626477541371157, '85099B -> 22386'), (0.4378029079159935, '22386
-> 85099B'), (0.48616236162361626, '22699 -> 22697'), (0.5052732502396932,
'22697 -> 22699')]
```

[24]:
```
'''Most Frequent Pair'''
print("Most frequent pair is: {}\n".format(freq_pairs[0]))
'''Most Frequent Pair''''s association rules
print("The most frequent pair's confidences are:\n{}\nand\n{}".
 ↪format(confidences[0], confidences[1]))
```

```
  File "<ipython-input-24-05f60db6c8bc>", line 3
    '''Most Frequent Pair''''s association rules
                             ^
SyntaxError: invalid syntax
```

d) Use a hash table to hash items from their `Descriptions`. Include a check to minimize and fix any collisions, as in nb08.

I started to get this implemented but the notebook oddly didn't save (which is almost better, it was a disaster). Too many collision, I tweaked the `hash_func` from notebook 08 and had even more collisions

---

[ ]:
```
```

e) Use A-priori to find all frequent items and all frequent pairs of items from either your hashed data set in part c) or via using Python's native dictionary methods for lookup tables.

A little confused about this problem, I didn't know if you wanted us to use `Description` here. Also, part d) was asking about hashing, my implementation for part c) relies on dictionaries and sets mostly.

---

f) Did any frequent items appear in part e) that did not in part c)? If so, list them.

[ ]:

```

```