Leonidas Boutsikaris 2776

Algorithms for big data
Monday, 18 May 2020

2nd set of exercises

Association rules and the A-priori algorithm

Association Rule learning is a rule based machine learning method which is used to discover interesting relationships hidden in large data-sets. These rules do not necessarily extract users preference, but rather find relationships between set of elements present in different entries in data-sets. Association Rule Mining is sometimes referred to as “Market Basket Analysis”.

Running the script: “Python3 main.py”

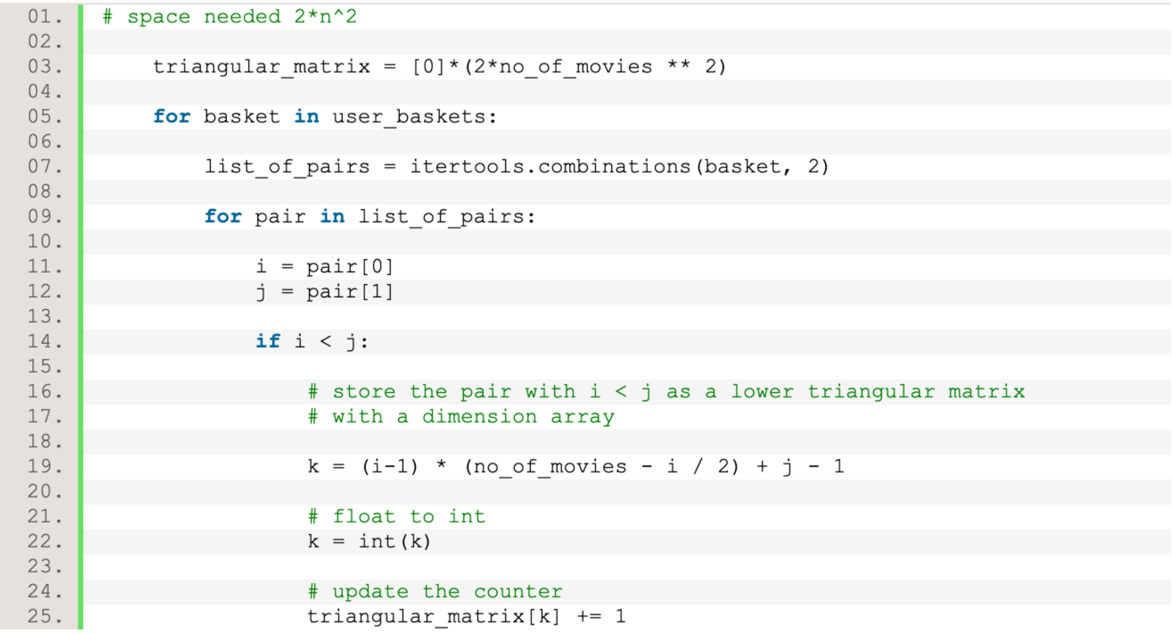
Data preprocessing

At first we need to create our sets. In our occasion we will analyse imdb ratings and how each users rated a movie. These will be our “baskets”. The $CreateMovieBaskets$ function does exactly this thing having an input file of the ratings in a csv format. The function iterates over the csv as it creates and finally returns the $user\_baskets$ list. A second function named $ReadMovies$ reads a file that contains all movies as single items with their information, converts it into a pandas dataframe and returns it for use.

The A-priori algorithm

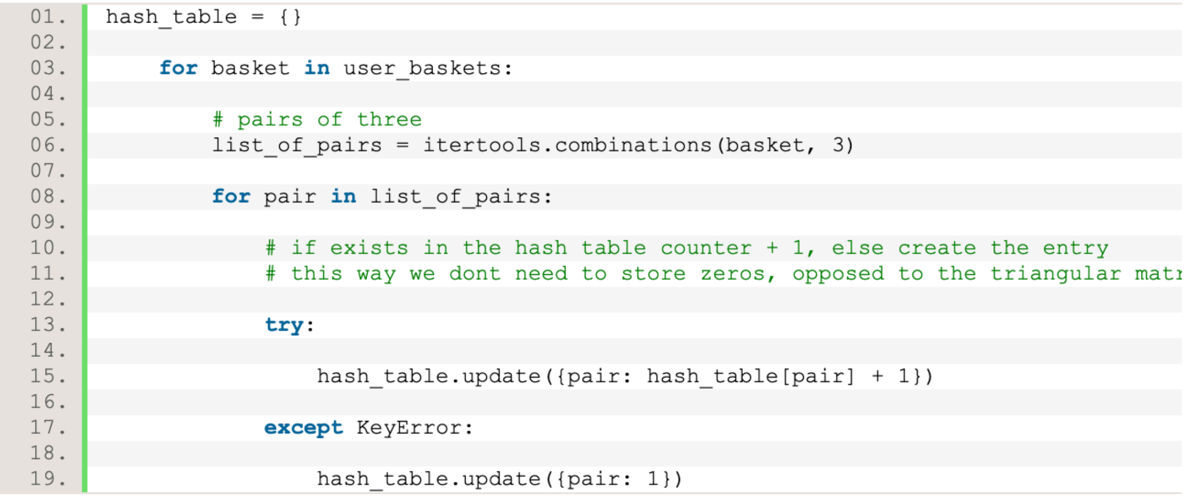
The first step in generation of association rules is to get all the frequent items of a set. This cannot be done with a brute force approach. There are some techniques that use memory loading such as the 1-dimensional triangular matrix or the hash table approach. The triangular matrix stores each counter of a pair in the position of $a[k]$ where $k = (i-1)(n - \frac{i}{2} + j - 1$. In result the pair counters are stored lexicographically.

The hash table counts triples for every basket we have. In contrast with the triangular matrix the hash method does not need to count nothing if the counter of a pairs is zero. Bellow we can see two snippets of these memory techniques.



```
# space needed 2*n^2

    triangular_matrix = [0]*(2*no_of_movies ** 2)

    for basket in user_baskets:

        list_of_pairs = itertools.combinations(basket, 2)

        for pair in list_of_pairs:

            i = pair[0]
            j = pair[1]

            if i < j:

                # store the pair with i < j as a lower triangular matrix
                # with a dimension array

                k = (i-1) * (no_of_movies - i / 2) + j - 1

                # float to int
                k = int(k)

                # update the counter
                triangular_matrix[k] += 1
```

Triangular Matrix

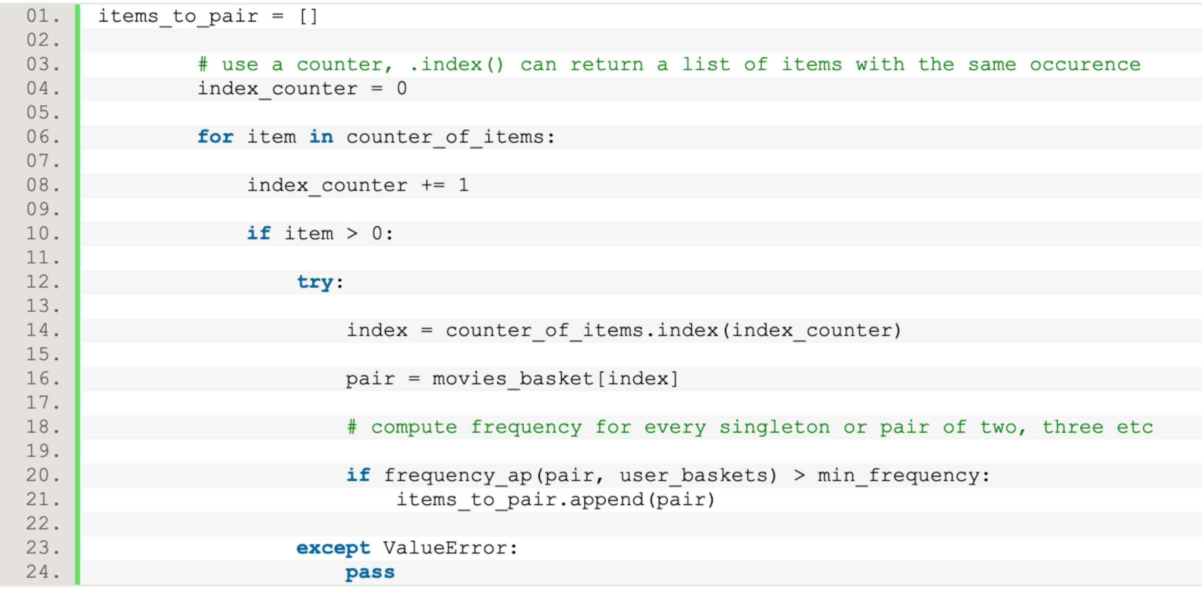


```
hash_table = {}

    for basket in user_baskets:

        # pairs of three
        list_of_pairs = itertools.combinations(basket, 3)

        for pair in list_of_pairs:

            # if exists in the hash table counter + 1, else create the entry
            # this way we dont need to store zeros, opposed to the triangular matr

            try:

                hash_table.update({pair: hash_table[pair] + 1})

            except KeyError:

                hash_table.update({pair: 1})
```

Hash table

The A-priori principle helps us with making this search efficient. The main principle that holds is that all subsets of a frequent item set must also be frequent. This allows us to prune all the supersets of an item set which does not satisfy the minimum threshold condition for the *support* (= the fraction of the number of occurrences of all items in a set with the total number of the item sets). It helps us identify the rules worth considering.

The A-priori starts by generating singletons, pairs of two, pairs of three, etc until there are no bigger common pairs of common items while keeping only those that have a support of a certain threshold. Bellow we can see a snippet.

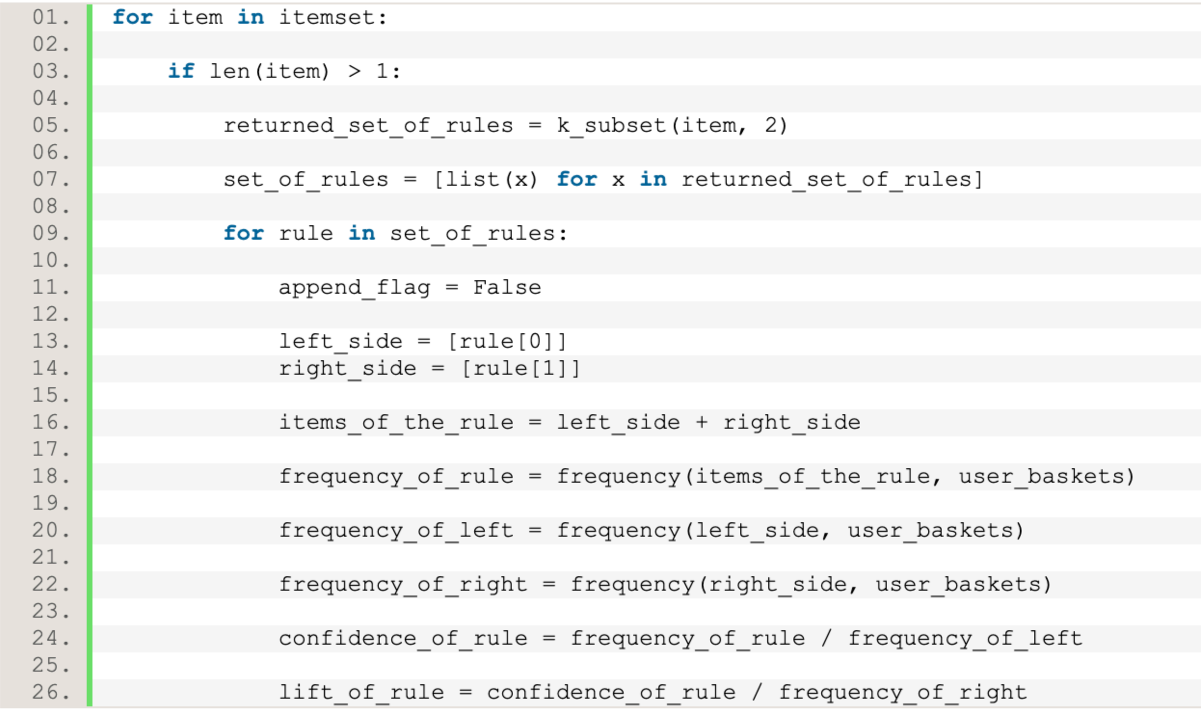


```
items_to_pair = []

        # use a counter, .index() can return a list of items with the same occurence
        index_counter = 0

        for item in counter_of_items:

            index_counter += 1

            if item > 0:

                try:

                    index = counter_of_items.index(index_counter)

                    pair = movies_basket[index]

                    # compute frequency for every singleton or pair of two, three etc

                    if frequency_ap(pair, user_baskets) > min_frequency:
                        items_to_pair.append(pair)

                except ValueError:
                    pass
```

Computing wich items to pair

Creating association rules

Once the frequent item sets are generated, identifying rules out of them is comparatively easy. From a list of all possible candidate rules, we aim to identify rules that fall above a minimum confidence level. Bellow we can see a snippet and how the metrics are computed.



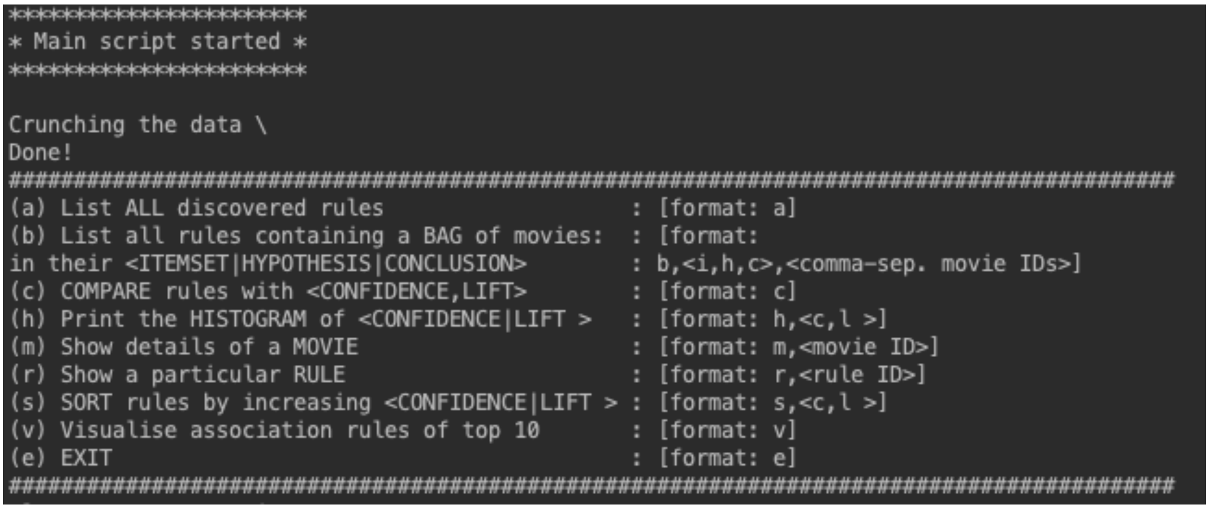
```
for item in itemset:

    if len(item) > 1:

        returned_set_of_rules = k_subset(item, 2)

        set_of_rules = [list(x) for x in returned_set_of_rules]

        for rule in set_of_rules:

            append_flag = False

            left_side = [rule[0]]
            right_side = [rule[1]]

            items_of_the_rule = left_side + right_side

            frequency_of_rule = frequency(items_of_the_rule, user_baskets)

            frequency_of_left = frequency(left_side, user_baskets)

            frequency_of_right = frequency(right_side, user_baskets)

            confidence_of_rule = frequency_of_rule / frequency_of_left

            lift_of_rule = confidence_of_rule / frequency_of_right
```

Rules generation

Results

Bellow we can see the available options and the results each one of them generates.
Parameters:

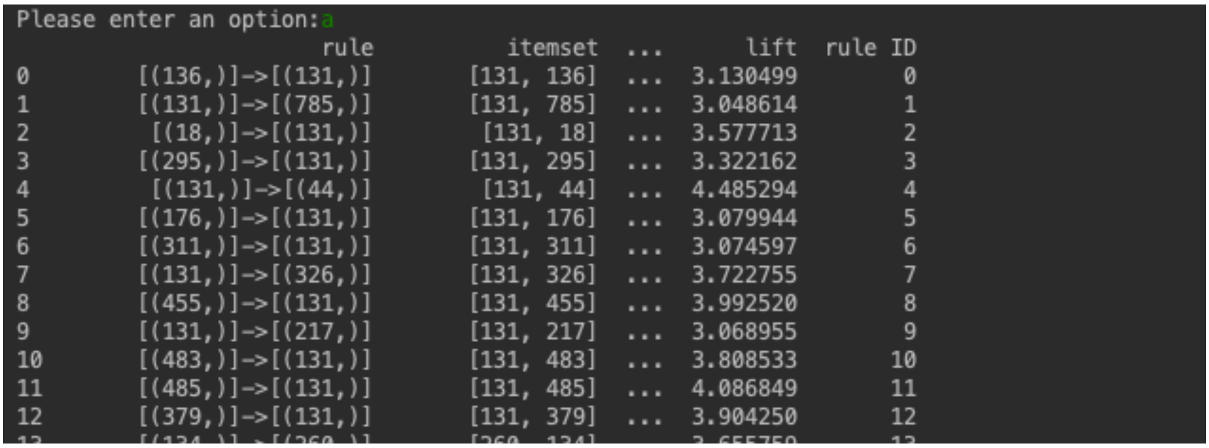
$$min\_conf = 0.1, \quad min\_lift = 3, \quad max\_lift = 6, \quad min\_length = 2$$



```
***********************
* Main script started *
***********************

Crunching the data \
Done!
##########################################################################################
(a) List ALL discovered rules                    : [format: a]
(b) List all rules containing a BAG of movies:   : [format:
in their <ITEMSET|HYPOTHESIS|CONCLUSION>         : b,<i,h,c>,<comma-sep. movie IDs>]
(c) COMPARE rules with <CONFIDENCE,LIFT>         : [format: c]
(h) Print the HISTOGRAM of <CONFIDENCE|LIFT >    : [format: h,<c,l >]
(m) Show details of a MOVIE                      : [format: m,<movie ID>]
(r) Show a particular RULE                       : [format: r,<rule ID>]
(s) SORT rules by increasing <CONFIDENCE|LIFT >  : [format: s,<c,l >]
(v) Visualise association rules of top 10        : [format: v]
(e) EXIT                                         : [format: e]
##########################################################################################
```

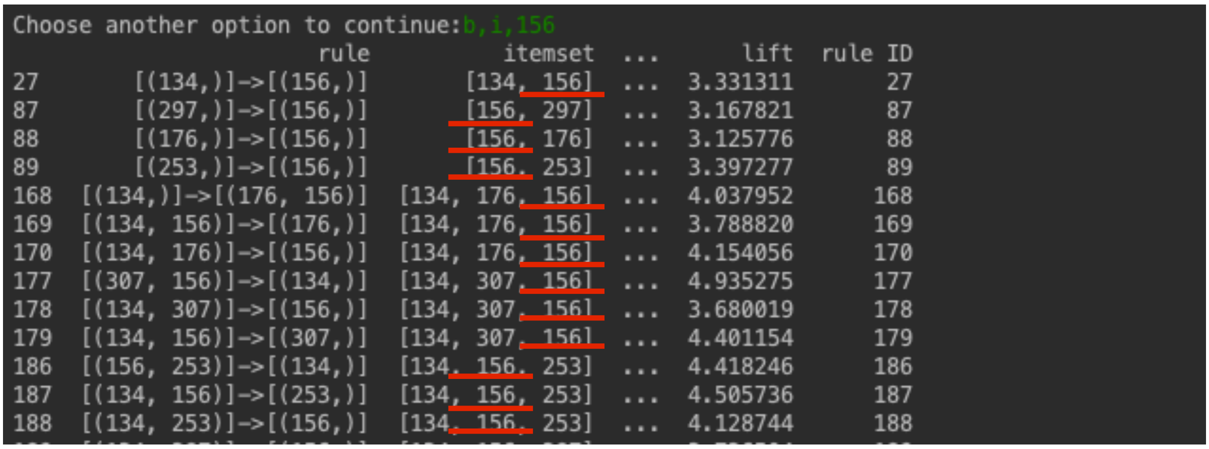
Main options

Presenting the discovered rules



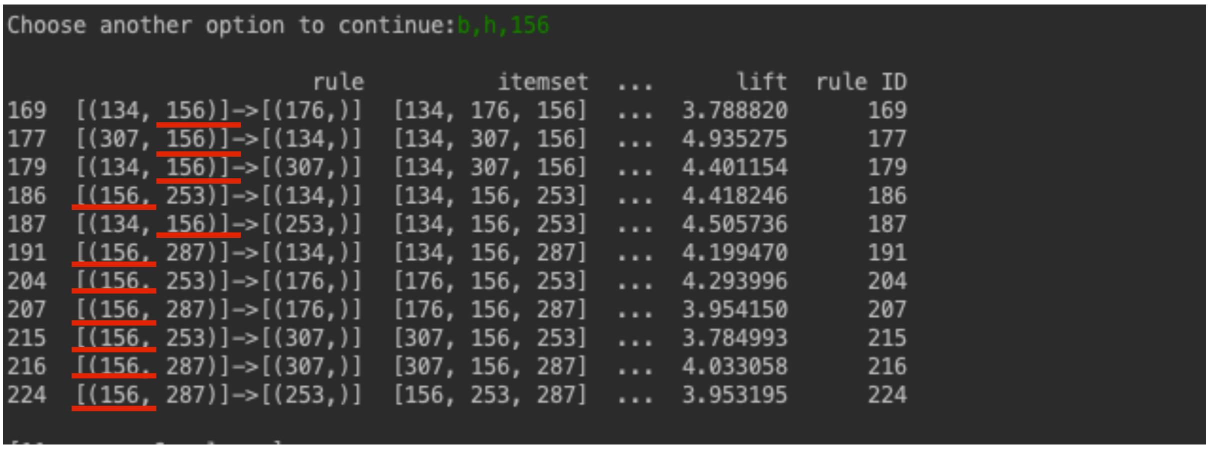
```
Please enter an option:a
                         rule          itemset  ...      lift  rule ID
0         [(136,)]->[(131,)]       [131, 136]  ...  3.130499        0
1         [(131,)]->[(785,)]       [131, 785]  ...  3.048614        1
2          [(18,)]->[(131,)]        [131, 18]  ...  3.577713        2
3         [(295,)]->[(131,)]       [131, 295]  ...  3.322162        3
4          [(131,)]->[(44,)]        [131, 44]  ...  4.485294        4
5         [(176,)]->[(131,)]       [131, 176]  ...  3.079944        5
6         [(311,)]->[(131,)]       [131, 311]  ...  3.074597        6
7         [(131,)]->[(326,)]       [131, 326]  ...  3.722755        7
8         [(455,)]->[(131,)]       [131, 455]  ...  3.992520        8
9         [(131,)]->[(217,)]       [131, 217]  ...  3.068955        9
10        [(483,)]->[(131,)]       [131, 483]  ...  3.808533       10
11        [(485,)]->[(131,)]       [131, 485]  ...  4.086849       11
12        [(379,)]->[(131,)]       [131, 379]  ...  3.904250       12
```

Presenting discovered rules of movie 156 that exists in the item set of rules



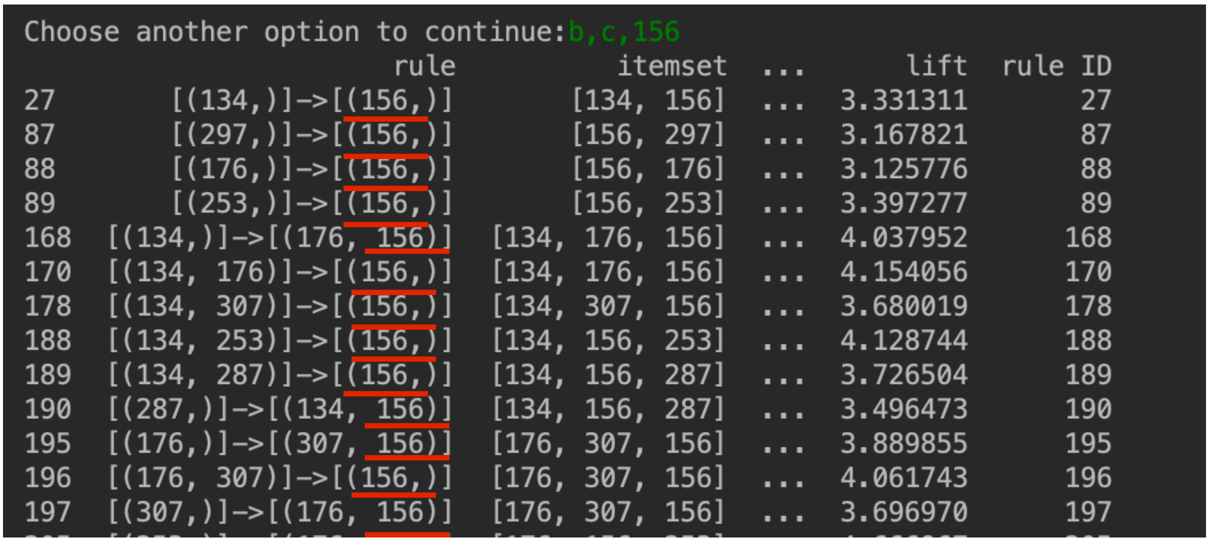
```
Choose another option to continue:b,i,156
                         rule          itemset  ...      lift  rule ID
27        [(134,)]->[(156,)]       [134, 156]  ...  3.331311       27
87        [(297,)]->[(156,)]       [156, 297]  ...  3.167821       87
88        [(176,)]->[(156,)]       [156, 176]  ...  3.125776       88
89        [(253,)]->[(156,)]       [156, 253]  ...  3.397277       89
168   [(134,)]->[(176, 156)]  [134, 176, 156]  ...  4.037952      168
169   [(134, 156)]->[(176,)]  [134, 176, 156]  ...  3.788820      169
170   [(134, 176)]->[(156,)]  [134, 176, 156]  ...  4.154056      170
177   [(307, 156)]->[(134,)]  [134, 307, 156]  ...  4.935275      177
178   [(134, 307)]->[(156,)]  [134, 307, 156]  ...  3.680019      178
179   [(134, 156)]->[(307,)]  [134, 307, 156]  ...  4.401154      179
186   [(156, 253)]->[(134,)]  [134, 156, 253]  ...  4.418246      186
187   [(134, 156)]->[(253,)]  [134, 156, 253]  ...  4.505736      187
188   [(134, 253)]->[(156,)]  [134, 156, 253]  ...  4.128744      188
```

Presenting discovered rules of movie 156 that exists in the hypothesis of rules



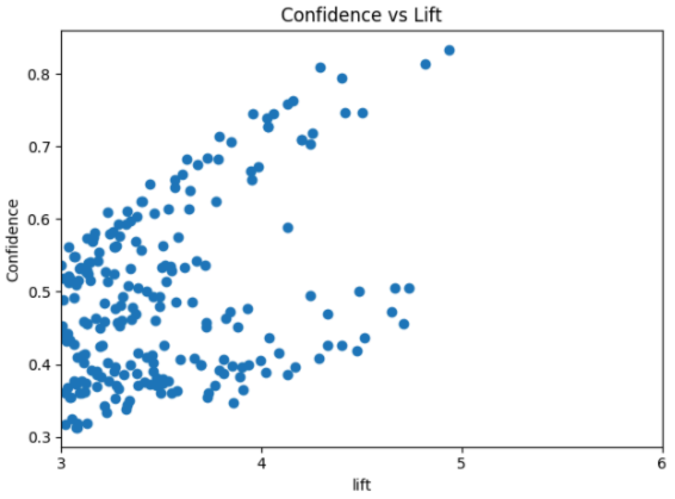
```
Choose another option to continue:b,h,156

                         rule          itemset  ...      lift  rule ID
169  [(134, 156)]->[(176,)]  [134, 176, 156]  ...  3.788820      169
177  [(307, 156)]->[(134,)]  [134, 307, 156]  ...  4.935275      177
179  [(134, 156)]->[(307,)]  [134, 307, 156]  ...  4.401154      179
186  [(156, 253)]->[(134,)]  [134, 156, 253]  ...  4.418246      186
187  [(134, 156)]->[(253,)]  [134, 156, 253]  ...  4.505736      187
191  [(156, 287)]->[(134,)]  [134, 156, 287]  ...  4.199470      191
204  [(156, 253)]->[(176,)]  [176, 156, 253]  ...  4.293996      204
207  [(156, 287)]->[(176,)]  [176, 156, 287]  ...  3.954150      207
215  [(156, 253)]->[(307,)]  [307, 156, 253]  ...  3.784993      215
216  [(156, 287)]->[(307,)]  [307, 156, 287]  ...  4.033058      216
224  [(156, 287)]->[(253,)]  [156, 253, 287]  ...  3.953195      224
```

Presenting discovered rules of movie 156 that exists in the conclusion of rules



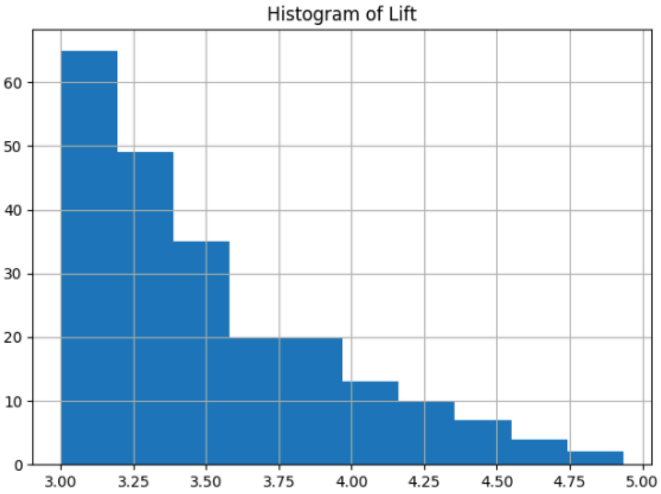
```
Choose another option to continue:b,c,156
                        rule          itemset  ...      lift  rule ID
27         [(134,)]->[(156,)]       [134, 156]  ...  3.331311       27
87         [(297,)]->[(156,)]       [156, 297]  ...  3.167821       87
88         [(176,)]->[(156,)]       [156, 176]  ...  3.125776       88
89         [(253,)]->[(156,)]       [156, 253]  ...  3.397277       89
168   [(134,)]->[(176, 156)]  [134, 176, 156]  ...  4.037952      168
170   [(134, 176)]->[(156,)]  [134, 176, 156]  ...  4.154056      170
178   [(134, 307)]->[(156,)]  [134, 307, 156]  ...  3.680019      178
188   [(134, 253)]->[(156,)]  [134, 156, 253]  ...  4.128744      188
189   [(134, 287)]->[(156,)]  [134, 156, 287]  ...  3.726504      189
190   [(287,)]->[(134, 156)]  [134, 156, 287]  ...  3.496473      190
195   [(176,)]->[(307, 156)]  [176, 307, 156]  ...  3.889855      195
196   [(176, 307)]->[(156,)]  [176, 307, 156]  ...  4.061743      196
197   [(307,)]->[(176, 156)]  [176, 307, 156]  ...  3.696970      197
```

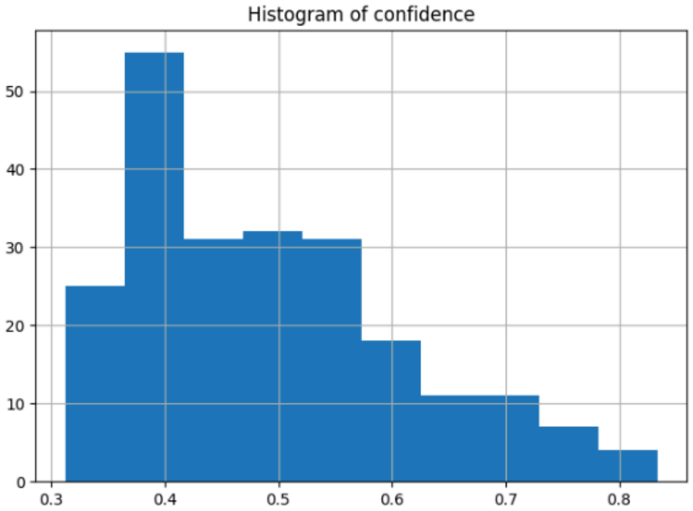
Comparing rules with Confidence and Lift



We can clearly see that as the lift is getting bigger the number of items are getting smaller. Even though that some items remain approximately at the same level the confidence of the rest is clearly higher.

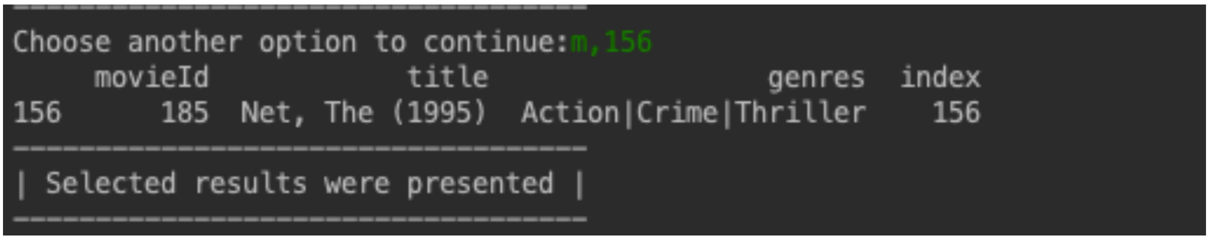
Histograms of Confidence and Lift





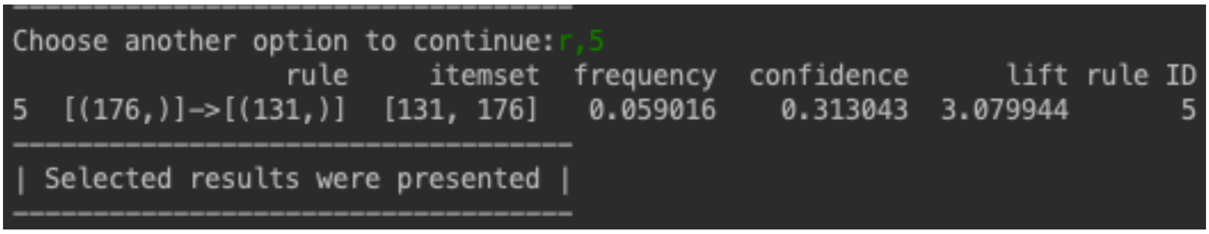
A small number of items have a high Lift and Confidence number, that means, a small number of rules are interesting. Exactlv what we want.

Showing movie information



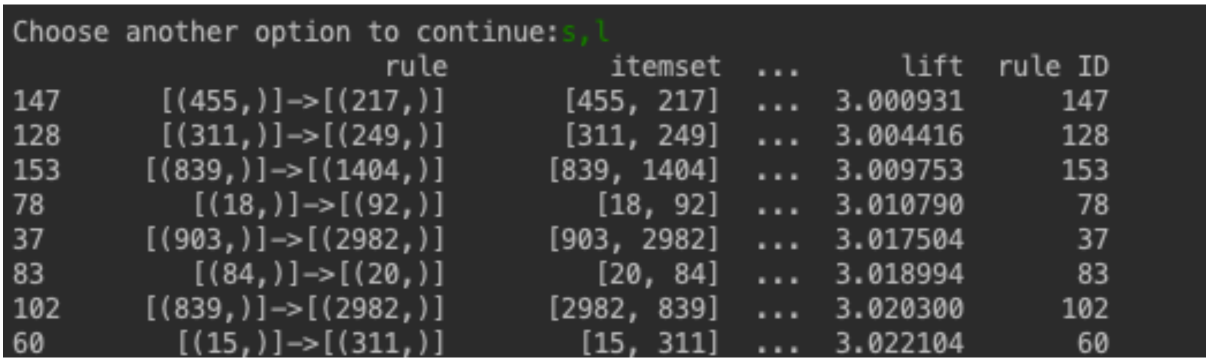
```
------------------------------------
Choose another option to continue:m,156
     movieId             title                   genres  index
156      185  Net, The (1995)  Action|Crime|Thriller    156
------------------------------------
| Selected results were presented |
------------------------------------
```

Showing rules information



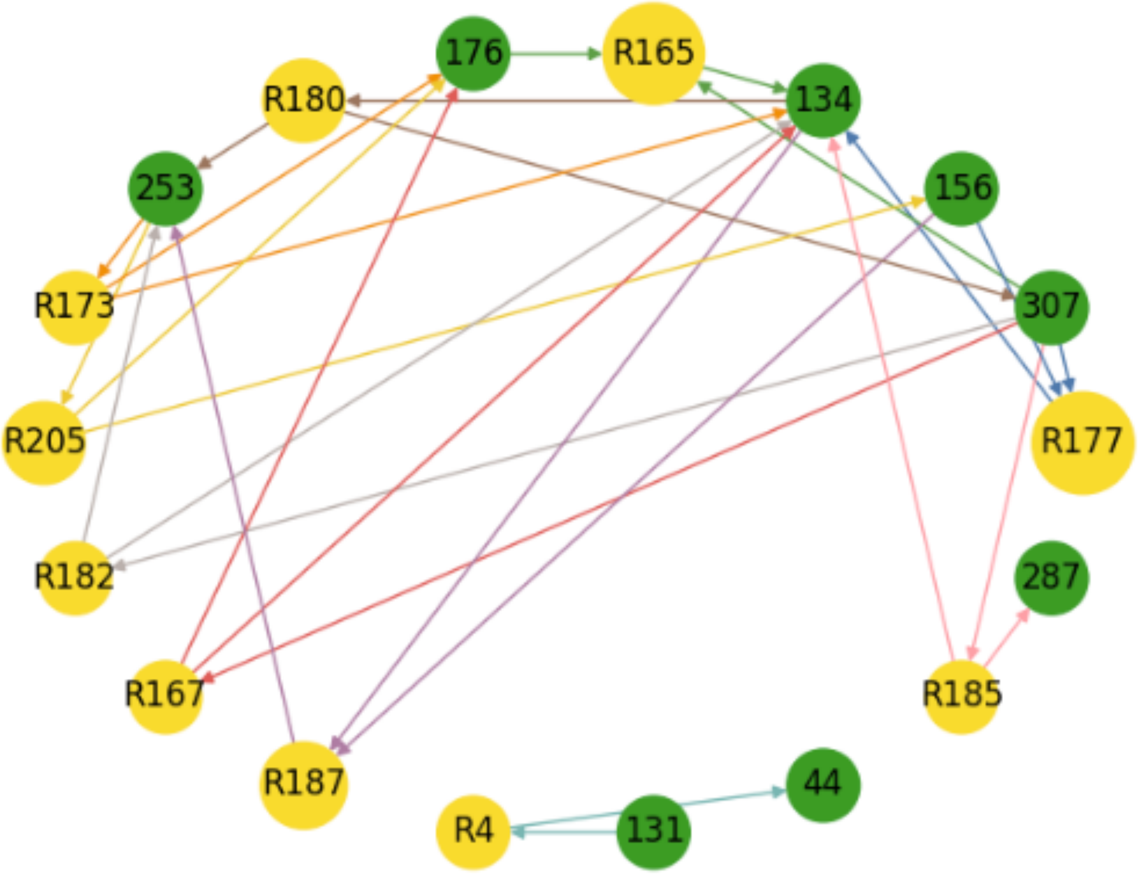
```
-----------------------------------
Choose another option to continue:r,5
                rule     itemset  frequency  confidence      lift rule ID
5  [(176,)]->[(131,)]  [131, 176]   0.059016    0.313043  3.079944       5
-----------------------------------
| Selected results were presented |
-----------------------------------
```

Sorting according to lift or confidence



```
Choose another option to continue:s,l
                         rule           itemset  ...       lift  rule ID
147       [(455,)]->[(217,)]        [455, 217]  ...   3.000931      147
128       [(311,)]->[(249,)]        [311, 249]  ...   3.004416      128
153     [(839,)]->[(1404,)]       [839, 1404]  ...   3.009753      153
78          [(18,)]->[(92,)]          [18, 92]  ...   3.010790       78
37      [(903,)]->[(2982,)]       [903, 2982]  ...   3.017504       37
83          [(84,)]->[(20,)]          [20, 84]  ...   3.018994       83
102     [(839,)]->[(2982,)]       [2982, 839]  ...   3.020300      102
60        [(15,)]->[(311,)]         [15, 311]  ...   3.022104       60
```

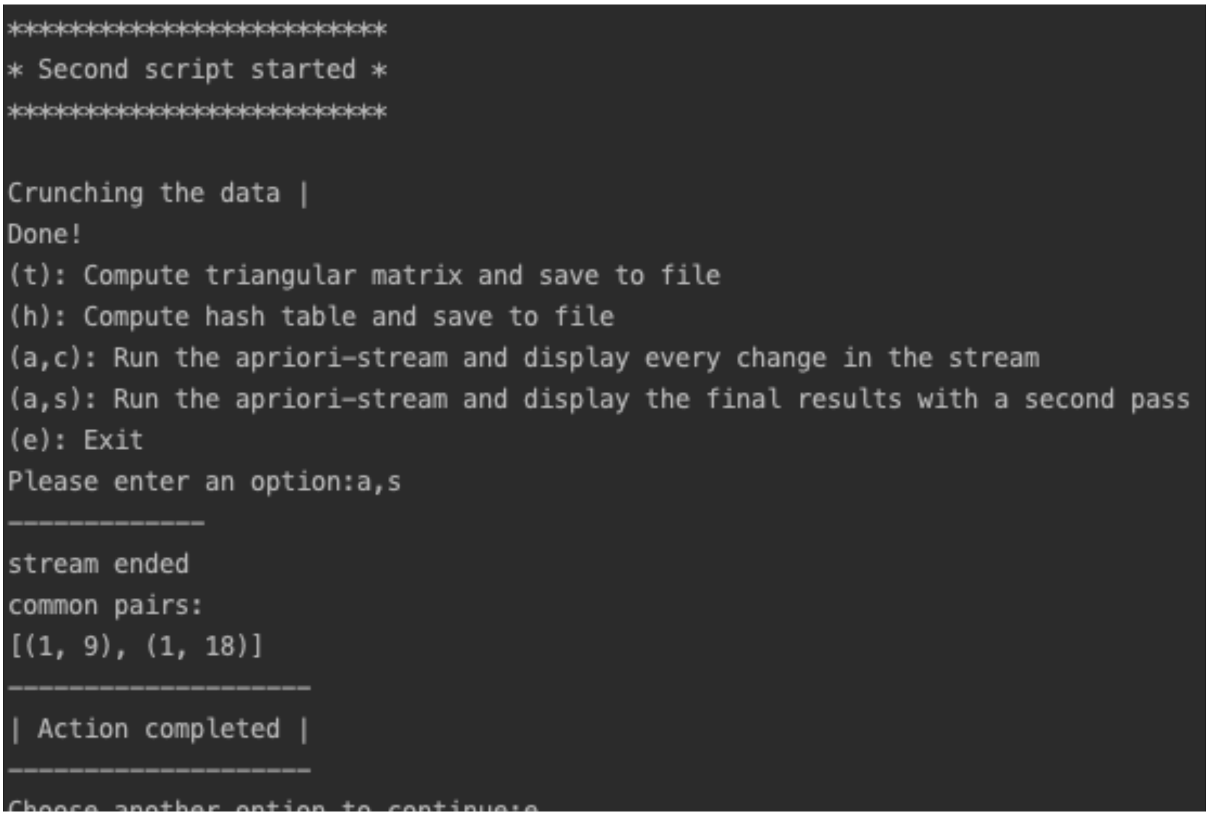
Graph relation between the movies and the rules. Node rule is bigger if it has bigger confidence thus being more important and notable.



The A-priori algorithm in a stream of item sets.

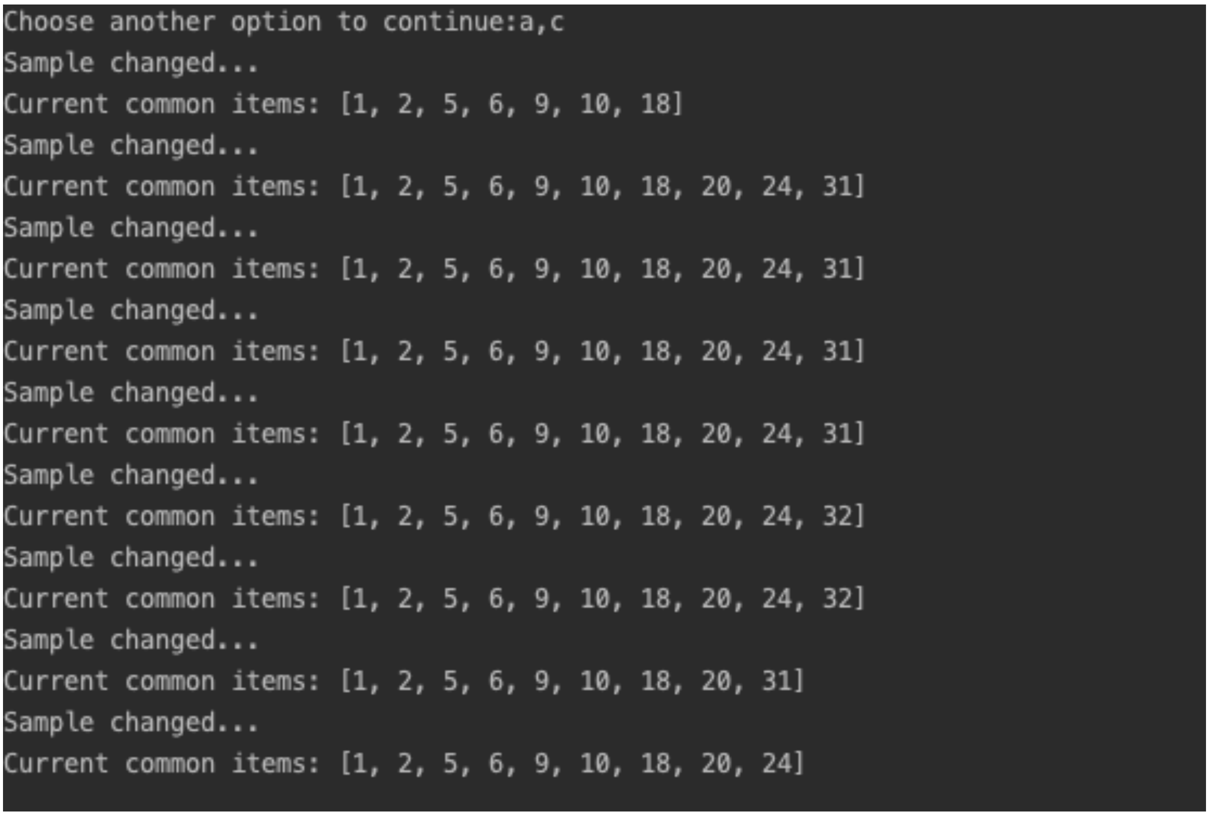
This concept applies to a uniform reservoir sampling. In situations that we have a live stream of data, in our case baskets, we want to sample this data with an equal probability, thus a uniform probability. This is achieved with the reservoir sampling. At the beginning the first n baskets fill the available place of the memory, after this is done a random number is drawn with a range from zero to the current iteration. If the number is bellow n then the n^{th} item will be replaced with the new item that came from the stream. If it is higher the new item will be discarded. In our implementation we have two options. Return the current common pairs the priori algorithm found if there is a change in the sample or after the end of the stream take a second pass and return the common pairs it found. The second script provides this in an option menu.
Note: the second script also provides the triangular and hash methods.

Running the script: “Python3 sampled_apriori_and_memory_techniques.py”



```
*************************
* Second script started *
*************************

Crunching the data |
Done!
(t): Compute triangular matrix and save to file
(h): Compute hash table and save to file
(a,c): Run the apriori-stream and display every change in the stream
(a,s): Run the apriori-stream and display the final results with a second pass
(e): Exit
Please enter an option:a,s
-------------
stream ended
common pairs:
[(1, 9), (1, 18)]
--------------------
| Action completed |
--------------------
Choose another option to continue:e
```

Running example 1



```
Choose another option to continue:a,c
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24, 31]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24, 31]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24, 31]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24, 31]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24, 32]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24, 32]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 31]
Sample changed...
Current common items: [1, 2, 5, 6, 9, 10, 18, 20, 24]
```

Running example 2

Notes:

1) Association rules are saved in a csv file after discovering them.

2) Item sets are stored in a txt file after discovering them.

3) Sampled A-priori, triangular matrix and hash table results are stored in a pickle file. Pickle serializes a single object at a time, and reads back a single object, the pickled data is recorded in sequence on the file.

4) The scrips provide basic input handling for wrong inputs and catches most of the wrong arguments.