FREQUENT ITEMSET MINING PROJECT

DATA ANALYTICS
2018102048 - Gowthami Gongati
2018111013 - Sriharshitha Bondugula

Introduction

In this project, we did the implementation of frequent itemset mining algorithms; Apriori and FPgrowth.

Data

For the same, we used the datasets from SPMF: Open dataset library which contains the itemsets in the numbered format and in the form of text files. We chose 2 datasets and implemented the algorithms on both these datasets. Following are the links for the same;

http://www.philippe-fournier-viger.com/spmf/datasets/mushrooms.txt
(MUSHROOM DATASET)

http://www.philippe-fournier-viger.com/spmf/datasets/BMS1_spmf
(BMSWebView1)

http://www.philippe-fournier-viger.com/spmf/datasets/MSNBC.txt
(MSNBC)

1) APRIORI ALGORITHM

Data cleaning :

We did the necessary data preprocessing to get the processed data from the text file. We named/labeled the first column as ‘items’.

Mushroom dataset: 8416 transactions and 119 distinct items.

	Standard
1	items
2	1 5 12 21 23 25 36 39 42 53 56 57 67 71 79 88 90 94 97 104 11
3	1 5 12 21 23 25 36 39 42 53 56 57 67 71 79 88 90 94 97 104 10
4	1 5 12 21 23 25 36 39 42 50 56 57 67 71 79 88 90 94 97 104 11
5	1 5 12 21 23 25 36 39 42 50 56 57 67 71 79 88 90 94 97 104 10
6	1 5 12 21 23 25 36 39 42 44 56 57 67 71 79 88 90 94 97 104 11
7	1 5 12 21 23 25 36 39 42 44 56 57 67 71 79 88 90 94 97 104 10

BMSWebView1 dataset: 59601 transactions and 497 distinct items.

	Standard
1	items
2	10307 -1 10311 -1 12487 -1 -2
3	12559 -1 -2
4	12695 -1 12703 -1 18715 -1 -2
5	10311 -1 12387 -1 12515 -1 12691 -1 12695 -1 12699 -1 12703
6	10291 -1 12523 -1 12531 -1 12535 -1 12883 -1 -2
7	12523 -1 12539 -1 12803 -1 12819 -1 -2

Implementation of the algorithm :

We wrote a function perform_apriori. This function takes data and minimum support count as input.

Single itemsets are obtained and the ones that have support count more than the parameter passed (**min_support_count**) are stored in a list. This is then appended to the main data frame (apriori_data) which the function will return at the end. Then, the itemsets of different sizes are generated in a loop and the frequent ones are obtained by using a table data structure (2D array).

The following is the intermediate data frame (d) that is created to find the counts of each 2-itemset.

```
             0     1         2     3    ...        296   297   298         299
0     (128, 1)  None  (128, 5)  None    ...  (97, 120)  None  None  (104, 120)
1     (128, 1)  None  (128, 5)  None    ...  (97, 120)  None  None  (104, 120)
2     (128, 1)  None  (128, 5)  None    ...  (97, 120)  None  None  (104, 120)
3     (128, 1)  None  (128, 5)  None    ...  (97, 120)  None  None  (104, 120)
4     (128, 1)  None  (128, 5)  None    ...  (97, 120)  None  None  (104, 120)
...        ...   ...       ...   ...    ...        ...   ...   ...         ...
8411      None  None      None  None    ...       None  None  None        None
8412      None  None      None  None    ...  (97, 120)  None  None  (104, 120)
8413      None  None      None  None    ...       None  None  None        None
8414      None  None      None  None    ...  (97, 120)  None  None  (104, 120)
8415      None  None      None  None    ...       None  None  None        None
```

The same is repeated to the itemsets of all the sizes which are the combinations of single items that are frequent. combinations(**single_items_set, length**) is used for the same where **single_items_set** is the set of single items that are frequent.

	0	1	2	3	...	2296	2297	2298	2299
0	None	(128, 1, 5)	None	None	...	None	None	(97, 104, 120)	None
1	None	(128, 1, 5)	None	None	...	None	None	(97, 104, 120)	None
2	None	(128, 1, 5)	None	None	...	None	None	(97, 104, 120)	None
3	None	(128, 1, 5)	None	None	...	None	None	(97, 104, 120)	None
4	None	(128, 1, 5)	None	None	...	None	None	(97, 104, 120)	None
...	...	...	...	...	...	...	...	...	...
8411	None	None	None	None	...	None	None	None	None
8412	None	None	None	None	...	None	None	(97, 104, 120)	None
8413	None	None	None	None	...	None	None	None	None
8414	None	None	None	None	...	None	None	(97, 104, 120)	None
8415	None	None	None	None	...	None	None	None	None

The above tables are the data structures we used for optimisation.

These are stored in a data frame named **d**.

The counts of each column are taken and the sets with a count more than that of min_support_count are added to the data frame apriori_data.

The loop breaks when there are no frequent itemsets of any length and the final data frame is returned. It looks like the following;

(Running the implemented algorithm with a min_support_count of 5050 on the mushroom dataset.)

	items	support_count	set_size
0	36	8200.0	1
1	38	6824.0	1
2	41	5880.0	1
3	67	5316.0	1
4	71	5076.0	1
5	90	8416.0	1
6	94	8216.0	1
7	97	7768.0	1
8	(97, 36)	7576.0	2
9	(97, 38)	6464.0	2
10	(97, 41)	5232.0	2
11	(97, 90)	7768.0	2
12	(97, 94)	7568.0	2
13	(67, 36)	5124.0	2
14	(67, 90)	5316.0	2
15	(67, 94)	5124.0	2
16	(36, 38)	6608.0	2
17	(36, 41)	5664.0	2
18	(36, 90)	8200.0	2
19	(36, 94)	8192.0	2
20	(38, 90)	6824.0	2
21	(38, 94)	6632.0	2
22	(71, 90)	5076.0	2
23	(41, 90)	5880.0	2
24	(41, 94)	5688.0	2
25	(90, 94)	8216.0	2
26	(97, 36, 38)	6272.0	3
27	(97, 36, 90)	7576.0	3
28	(97, 36, 94)	7568.0	3
29	(97, 38, 90)	6464.0	3
30	(97, 38, 94)	6272.0	3
31	(97, 41, 90)	5232.0	3
32	(97, 90, 94)	7568.0	3
33	(67, 36, 90)	5124.0	3
34	(67, 36, 94)	5124.0	3
35	(67, 90, 94)	5124.0	3
36	(36, 38, 90)	6608.0	3
37	(36, 38, 94)	6608.0	3
38	(36, 41, 90)	5664.0	3
39	(36, 41, 94)	5664.0	3
40	(36, 90, 94)	8192.0	3
41	(38, 90, 94)	6632.0	3
42	(41, 90, 94)	5688.0	3
43	(97, 36, 38, 90)	6272.0	4
44	(97, 36, 38, 94)	6272.0	4
45	(97, 36, 90, 94)	7568.0	4
46	(97, 38, 90, 94)	6272.0	4
47	(67, 36, 90, 94)	5124.0	4
48	(36, 38, 90, 94)	6608.0	4
49	(36, 41, 90, 94)	5664.0	4
50	(97, 36, 38, 90, 94)	6272.0	5

```
Time taken by standard apriori is  12.731229066848755
Given minimum support =  5050
```

We also implemented apriori using the inbuilt-libraries to verify the output. The output when we run the inbuilt implementation with a min_support of 0.6 is as follows (Equivalent to a min_support_count of 5050 as 5050/8416 is 0.6 approximately.)

```
     support            itemsets
0   0.974335                (36)
1   0.810837                (38)
2   0.698669                (41)
3   0.631654                (67)
4   0.603137                (71)
5   1.000000                (90)
6   0.976236                (94)
7   0.923004                (97)
8   0.785171            (38, 36)
9   0.673004            (41, 36)
10  0.608840            (36, 67)
11  0.974335            (90, 36)
12  0.973384            (94, 36)
13  0.900190            (97, 36)
14  0.810837            (90, 38)
15  0.788023            (94, 38)
16  0.768061            (97, 38)
17  0.698669            (90, 41)
18  0.675856            (94, 41)
19  0.621673            (97, 41)
20  0.631654            (90, 67)
21  0.608840            (94, 67)
22  0.603137            (90, 71)
23  0.976236            (94, 90)
24  0.923004            (90, 97)
25  0.899240            (94, 97)
26  0.785171        (90, 38, 36)
27  0.785171        (94, 38, 36)
28  0.745247        (97, 38, 36)
29  0.673004        (90, 41, 36)
30  0.673004        (94, 41, 36)
31  0.608840        (90, 36, 67)
32  0.608840        (94, 36, 67)
33  0.973384        (94, 90, 36)
34  0.900190        (97, 90, 36)
35  0.899240        (94, 97, 36)
36  0.788023        (94, 90, 38)
37  0.768061        (90, 97, 38)
38  0.745247        (94, 97, 38)
39  0.675856        (90, 94, 41)
40  0.621673        (97, 41, 90)
41  0.608840        (94, 90, 67)
42  0.899240        (90, 94, 97)
43  0.785171    (94, 90, 38, 36)
44  0.745247    (97, 90, 38, 36)
45  0.745247    (94, 97, 38, 36)
46  0.673004    (90, 94, 41, 36)
47  0.608840    (94, 90, 36, 67)
48  0.899240    (97, 94, 90, 36)
49  0.745247    (90, 94, 97, 38)
50  0.745247  (97, 90, 94, 36, 38)
Time taken by inbuilt apriori is  0.07065773010253906
Given minimum support =  0.6
```

Both the ways, we get the same output. The time taken by the inbuilt function is very much lesser than the algorithm implemented.

Optimisation of the algorithm :

The optimisations we implemented are the following:

1) *Translation reduction*:

   As a further optimisation we have done transaction reduction, i.e the transactions that do not have any of the frequent k-itemsets cannot contain any frequent (k + 1) itemsets. The following part of the code implements the same. Here dataframe **d** (previously mentioned) contains a table that stores if a subset is present in a tuple. The tuples which do not contain any of them have NaN for all the columns. Indices of all those tuples are stored and removed from the original data table.

```
is_NaN = d.isnull()
row_has_NaN = d[is_NaN.all(axis=1)].index.tolist()
# print(row_has_NaN)
data = data.drop(row_has_NaN,axis=0)
```

2) *Hashing*:

   We have directly implemented hashing to find the counts of each itemset by creating a table (2D array). (We did not implement Apriori with loops)

3) The tuples which have a length 'l' cannot contain frequent itemsets of length 'k' if l<k. Hence, all such tuples are removed.

```
data = data[data['set_size'] >= length]
```

The following shows the outputs before optimisation and after optimisation

```
                         items  support_count  set_size
0                            1         4488.0         1
1                          120         4064.0         1
2                           24         5040.0         1
3                           36         8200.0         1
4                           38         6824.0         1
..                         ...            ...       ...
186       (97, 90, 38, 41, 94)         4016.0         5
187       (67, 36, 90, 38, 94)         4232.0         5
188       (67, 36, 90, 71, 94)         4034.0         5
189       (36, 90, 38, 41, 94)         4352.0         5
190   (97, 36, 90, 38, 41, 94)         4016.0         6

[191 rows x 3 columns]
Time taken by standard apriori is  103.15224528312683
Given minimum support =  4000
```

```
                         items  support_count  set_size
0                            1         4488.0         1
1                          120         4064.0         1
2                           24         5040.0         1
3                           36         8200.0         1
4                           38         6824.0         1
..                         ...            ...       ...
186       (97, 90, 38, 41, 94)         4016.0         5
187       (67, 36, 90, 38, 94)         4232.0         5
188       (67, 36, 90, 71, 94)         4034.0         5
189       (36, 90, 38, 41, 94)         4352.0         5
190   (97, 36, 90, 38, 41, 94)         4016.0         6

[191 rows x 3 columns]
Time taken by optimised apriori is  99.93356823921204
Given minimum support =  4000
```

We have not implemented it without the hashing optimisation. So the optimised apriori includes transaction reduction and the other optimisation.

The following table shows the times taken by both the algorithms for different values of minimum support count for the mushroom data.

Minimum support	Standard apriori (includes hash table)	Optimised apriori with hashing, transition reduction, and additional optimisation	Number of frequent itemsets
4000	103.15224528312683	99.93356823921204	191
4300	48.729281187057495	48.65313833893212	139
5000	15.041770458221436	14.910938739776611	59
5300	12.26511287689209	12.199238718248765	41

The following table shows the times taken by both the algorithms for different values of minimum support count for the BMSWebview1 data.

Minimum support	Standard apriori (includes hash table)	Optimised apriori with hashing, transition reduction, and additional optimisation	Number of frequent itemsets
800	117.48995041847229	74.40363836288452	42
900	81.48569846153259	53.22280240058899	36
1000	60.48276209831238	39.54233741760254	31
1100	38.15256643295288	26.291325330734253	25

Analysis :

- The improvement seen is very less for the mushroom dataset when compared to the BMSWebview1 data set. The reason for the same may be that the itemsets that are frequent are present in almost all the tuples (in a majority of them). As a result, fewer transactions are being removed and optimisation is not that effective.
- As we increase the value of min_sup the improvement is decreasing. The number of transactions with frequent itemsets increases. As a result, fewer transactions will be removed each time reducing the effectiveness.
- The third optimisation works well for the BMSWebview1 dataset as it contains transactions of all kinds of lengths and the average length of each transaction is 2.42 in the mushroom dataset it is 23. For the same reason, this optimisation also works well on the BMSWebview1 dataset.

Outputs for BMSWebview1 dataset for min_sup=1100 :

```
      support        itemsets
0   0.033707         (10295)
1   0.046929         (10307)
2   0.039781         (10311)
3   0.057868         (10315)
4   0.019580         (10335)
5   0.023305         (10877)
6   0.020100         (12431)
7   0.034379         (12483)
8   0.038053         (12487)
9   0.024966         (12621)
10  0.030083         (12663)
11  0.029999         (12679)
12  0.023859         (12695)
13  0.032684         (12703)
14  0.020369         (12715)
15  0.021258         (12723)
16  0.018674         (12819)
17  0.019798         (12831)
18  0.022735         (12875)
19  0.060788         (12895)
20  0.027114         (32213)
21  0.061375         (33449)
22  0.060603         (33469)
23  0.020151         (34893)
24  0.020201  (33449, 33469)
Time taken by inbuilt apriori is  1.0683331489562988
Given minimum support =  0.0185
```

```
             items  support_count  set_size
0            10295         2009.0         1
1            10307         2797.0         1
2            10311         2371.0         1
3            10315         3449.0         1
4            10335         1167.0         1
5            10877         1389.0         1
6            12431         1198.0         1
7            12483         2049.0         1
8            12487         2268.0         1
9            12621         1488.0         1
10           12663         1793.0         1
11           12679         1788.0         1
12           12695         1422.0         1
13           12703         1948.0         1
14           12715         1214.0         1
15           12723         1267.0         1
16           12819         1113.0         1
17           12831         1180.0         1
18           12875         1355.0         1
19           12895         3623.0         1
20           32213         1616.0         1
21           33449         3658.0         1
22           33469         3612.0         1
23           34893         1201.0         1
24  (33449, 33469)         1204.0         2
Time taken by optimised apriori is  27.531821727752686
Given minimum support =  1100
```

```
             items  min_support_count  set_size
0            10295             2009.0         1
1            10307             2797.0         1
2            10311             2371.0         1
3            10315             3449.0         1
4            10335             1167.0         1
5            10877             1389.0         1
6            12431             1198.0         1
7            12483             2049.0         1
8            12487             2268.0         1
9            12621             1488.0         1
10           12663             1793.0         1
11           12679             1788.0         1
12           12695             1422.0         1
13           12703             1948.0         1
14           12715             1214.0         1
15           12723             1267.0         1
16           12819             1113.0         1
17           12831             1180.0         1
18           12875             1355.0         1
19           12895             3623.0         1
20           32213             1616.0         1
21           33449             3658.0         1
22           33469             3612.0         1
23           34893             1201.0         1
24  (33449, 33469)             1204.0         2
Time taken by standard apriori is  43.91935062408447
Given minimum support =  1100
```

2) FP-GROWTH ALGORITHM

The shortcomings of the *Apriori Algorithm*:

a. Using Apriori needs a generation of candidate itemsets. These itemsets may be large in number if the itemset in the database is huge.
b. Apriori needs multiple scans of the database to check the support of each itemset generated and this leads to high costs.

In FP-growth, a frequent pattern is generated without the need for candidate generation. FP growth algorithm represents the database in the form of a tree called a frequent pattern tree or FP tree. This tree structure will maintain the association between the itemsets. The database is fragmented using one frequent item. This fragmented part is called “pattern fragment”. The itemsets of these fragmented patterns are analyzed. Thus with this method, the search for frequent itemsets is reduced comparatively.

Optimisation:

Merging strategy is used to optimise in time and space while generating conditional pattern bases during pattern-growth mining for the *FP-growth algorithm*.

Class variables :

```
class fp_node(object):
    def __init__(self, value, cnt, parent):
        self.parent = parent
        self.value = value
        self.cnt = cnt
        self.visited = False
        self.link = None
        self.children = []
```

a. Firstly, we have to sort the **frequent item-set keys** to mine in the increasing order.

```
m=self.freq.keys()
sorteds = sorted(m, key=lambda x: self.freq[x])
```

b. Maintain an array to check that the current node is **visited** already or not and a **conditional path list** to maintain the paths. as (self.visited)

```python
for ind in suff:
    freq = ind.cnt
    if ind.visited == False: ind.visited = True
    else:
        for i in range(0, freq):
            conditional_sub.append(conditional_base_paths[ind])
        continue
```

c. After that traverse for each node in the **suffix order**, if the node is visited for the first time then append the path of the node into conditional_sub list and repeat until its count.
d. Then for each node in that path find it's a parent and continues until we hit the root node and store its path in **conditional_base_path array**.

```python
for i in range(0, len(path)):
    if parent.parent != None:
      conditional_base_paths[parent] = path[i+1:]
      parent = parent.parent
    else: break
```

e. At the end **mine the conditional subtree** which was stored in conditional_sub array and get the itemsets and print it.

Outputs comparing Optimised and unoptimised:

These are the outputs we got for the **MSNBC_SPMF.txt** file for support = 0.8 and frequent itemset length greater than 5.

```
FP Growth without Optimization
The frequent itemsets for support cnt =  0.8
Support_cnt= 4 Itemset= (1, 2, 4, 6, 7, 10, 10)
Support_cnt= 4 Itemset= (1, 2, 2, 4, 6, 10, 10)
Support_cnt= 12 Itemset= (1, 4, 10, 10, 10, 10, 10)
Support_cnt= 2 Itemset= (1, 10, 10, 10, 10, 10, 10)
Support_cnt= 6 Itemset= (1, 1, 1, 1, 1, 2, 2)
Support_cnt= 1080 Itemset= (1, 1, 1, 2, 2, 14, 14)
Support_cnt= 4 Itemset= (4, 7, 9, 9, 9, 9, 9)
Support_cnt= 40 Itemset= (6, 6, 7, 7, 7, 8, 8)
Support_cnt= 286 Itemset= (4, 4, 7, 13, 14, 14, 14)
Support_cnt= 56 Itemset= (4, 4, 7, 7, 14, 14, 14)
Support_cnt= 56 Itemset= (4, 4, 7, 7, 13, 14, 14, 14)
Support_cnt= 154 Itemset= (4, 7, 7, 13, 14, 14, 14)
Support_cnt= 42 Itemset= (4, 7, 7, 7, 14, 14, 14)
Support_cnt= 42 Itemset= (4, 7, 7, 7, 13, 14, 14, 14)
Support_cnt= 880 Itemset= (4, 7, 13, 13, 14, 14, 14)
Support_cnt= 42 Itemset= (7, 7, 7, 13, 14, 14, 14)
Support_cnt= 56 Itemset= (4, 4, 7, 7, 13, 14, 14)
Support_cnt= 42 Itemset= (4, 7, 7, 7, 13, 14, 14)
Support_cnt= 160 Itemset= (4, 4, 9, 9, 9, 9, 9)
Support_cnt= 12 Itemset= (13, 13, 14, 14, 14, 14, 14)
Support_cnt= 6 Itemset= (13, 13, 13, 14, 14, 14, 14)
The time taken for normal FP Growth =  0.16387967599985132
```

```
Support_cnt= 4 Itemset= (1, 2, 4, 6, 7, 10, 10)
Support_cnt= 4 Itemset= (1, 2, 2, 4, 6, 10, 10)
Support_cnt= 12 Itemset= (1, 4, 10, 10, 10, 10, 10)
Support_cnt= 2 Itemset= (1, 10, 10, 10, 10, 10, 10)
Support_cnt= 6 Itemset= (1, 1, 1, 1, 1, 2, 2)
Support_cnt= 1080 Itemset= (1, 1, 1, 2, 2, 14, 14)
Support_cnt= 4 Itemset= (4, 7, 9, 9, 9, 9, 9)
Support_cnt= 40 Itemset= (6, 6, 7, 7, 7, 8, 8)
Support_cnt= 286 Itemset= (4, 4, 7, 13, 14, 14, 14)
Support_cnt= 56 Itemset= (4, 4, 7, 7, 14, 14, 14)
Support_cnt= 56 Itemset= (4, 4, 7, 7, 13, 14, 14, 14)
Support_cnt= 154 Itemset= (4, 7, 7, 13, 14, 14, 14)
Support_cnt= 42 Itemset= (4, 7, 7, 7, 14, 14, 14)
Support_cnt= 42 Itemset= (4, 7, 7, 7, 13, 14, 14, 14)
Support_cnt= 880 Itemset= (4, 7, 13, 13, 14, 14, 14)
Support_cnt= 42 Itemset= (7, 7, 7, 13, 14, 14, 14)
Support_cnt= 56 Itemset= (4, 4, 7, 7, 13, 14, 14)
Support_cnt= 42 Itemset= (4, 7, 7, 7, 13, 14, 14)
Support_cnt= 160 Itemset= (4, 4, 9, 9, 9, 9, 9)
Support_cnt= 12 Itemset= (13, 13, 14, 14, 14, 14, 14)
Support_cnt= 6 Itemset= (13, 13, 13, 14, 14, 14, 14)
The time taken with merging strategy = 0.12518233899936604
```

The following table shows the times taken by both the algorithms for different values of minimum support count for the MSNBC data.

Minimum support	Normal FP-growth	Optimised FP-growth with merging strategy
0.8	0.16387967599985132	0.12518233899936604
0.7	0.24285525500090444	0.1259647080013383
0.6	0.25072952000118676	0.12629417599883163
0.5	0.26331629199921736	0.12733943800008274

The following table shows the times taken by the optimised algorithm for different values of minimum support count for the mushroom dataset. We ran only on optimised as the other one was using the whole RAM.

Minimum support	Optimised FP-growth with merging strategy
3000	0.5058177320000254
4000	0.1964621319999651
5000	0.07334636600000977
500	28.62137

The following table shows the times taken by the optimised algorithm for different values of minimum support count for the BMSWebview1 dataset. We ran only on optimised as the other one was using the whole RAM.

Minimum support	Optimised FP-growth with merging strategy
800	0.06219835599995349
900	0.05970546599996851
1000	0.058474137999837694
1100	0.057245221000130186

Analysis :

- The optimised algorithm takes the same amount of time for almost all the values of min_sup.
- As the min_sup is **decreased**, it is evident that the time is taken **decreases** for both the algorithms.
- With the decrease in min_sup, the effectiveness in optimisation is increasing.
- The algorithm performs well on **BMSWebview1** and **MSNBC** because the length of frequent itemsets is **lesser** in these datasets which is not true for the mushroom dataset. The size of data in MSNBC is larger than that in BMSWebview1 and hence the algorithm performs better on BMSWebview1.
- As the mushroom dataset is a sparse dataset, i.e almost all itemsets are equally frequent, this algorithm works comparatively bad for this dataset. Because for sparse datasets, the tree grows bigger and traversal takes comparatively more time.

3) ANALYSIS

- The output tables for apriori were inserted previously.
- Comparing both the algorithms, the **FP-Growth** algorithm performs **well** and hence is **efficient** than apriori. The reason for our results may be that we have not implemented the proper hashing optimisation which would have made the apriori algorithm more efficient.
- The **FPGrowth** algorithm works comparatively **very better** on MSNBC as it is a **dense** dataset and as there are long frequent itemsets that makes it **difficult** for **Apriori**.
- But, if **optimized** correctly, Apriori should work better than Fpgrowth on the mushroom dataset as a large tree makes FPgrowth comparatively less efficient.
- The value of minimum support also plays a major role in the space and time that each of the algorithms takes. The lesser values of minimum support result in huge trees which results in the consumption of more space. We have used a table data structure in our implementation which consumes more space too for lesser values of minimum support. But, with a correct hashing data structure, Apriori consumes lesser space than FPGrowth.