Programming Assignment #1

컴퓨터소프트웨어학과 2017029470 김종원

1. Environment

- Mac OS (Monterey) M1 chip
- Python 3.10.3 (release March 16, 2022)

2. How to compile

Before compiling apriori.py, python version 3 must be installed in your system.

```
wonnx@wonnx project_1_apriori % python3 apriori.py 5 input.txt output.txt
```

- Execution file name: apriori.py
- Minimum support: 5%
- Input file name: input.txt
- Output file name: output.txt

3. Summary of algorithm

Apriori algorithm is for finding frequent item sets in a dataset for association rules. I apply an iterative approach where k-frequent item sets are used to find k+1 item sets.

- Initially, scan database once to get frequent 1-itemset.
- Generate candidate item sets of length k+1 from frequent item sets of length k by self-joining. (Self-join is a regular join, but the table is joined with itself.)
- And pruning the candidates to prevent huge number of candidates.
- After checking the support for the remaining candidates, those exceeding the minimum support are classified into a frequent pattern.
- Terminate when no frequent or candidate set can be generated.

4. Detailed description of codes

```python
5  def apriori(transaction, minimum_support):
6      # variable to save the result of apriori algorithm
7      result = ''
8
9      # converting minimum support percentage to number
10     nms = (minimum_support/100) * len(transaction)
11     candidates = set()
12     frequent_patterns = set()
13
14     # when the length of pattern is 1
15     length = 1
16     for dt in transaction:
17         for item in dt:
18             if item not in candidates:
19                 candidates.add(item)
20     candidates = sorted(candidates)
21     for itemset in candidates:
22         if get_support([itemset], transaction) >= nms:
23             frequent_patterns.add(itemset)
24     frequent_patterns = sorted(frequent_patterns)
25
26     # when the length of pattern is larger then 1
27     while True:
28         length += 1
29
30         # do self join to get candidates, which length is 'length + 1'
31         # return the tuples in the candidates set.
32         candidates = self_join(frequent_patterns, length)
33
34         # do prune to reduce the number of candidates
35         # return the reducted tuples in the candidates set.
36         previous_fp = copy.deepcopy(frequent_patterns)
37         candidates = prune(candidates, previous_fp, length)
38
39         # testing - check the support of the new candidates
40         frequent_patterns = test_support(candidates, nms, transaction)
41
42         if not frequent_patterns:
43             break
44         else: # If there is a frequent pattern, apply associative rule
45             result += association_rule(frequent_patterns, length, transaction)
46
47     return result
```

- The apriori function receives two variables: transaction and minimum support as arguments. The purpose of this function is to generate association rules by applying the apriori algorithm.
- First, I scan the transaction database to generate frequent patterns of length 1. (line 15 ~ line 24)
- As described above, from frequent pattern of length k, frequent pattern of length k+1 is created, and if no more creation is possible, the loop is terminated. (line 25 ~ line 43)
- Using the obtained frequent pattern, check whether an association rule can be created. (line 45)

```python
def get_support(itemset, transaction):
    count = 0
    for dt in transaction:
        if set(itemset) == set(itemset) & set(dt):
            count += 1
    return count
```

- The get_support function receives two variables: itemset and transaction as arguments. The purpose of this function is to return the number of transactions which include itemset.

```python
57 def self_join(frequent_patterns, length):
58     joined_candidates = list()
59     for pattern in frequent_patterns:
60         # If the length is 2, there will be patterns of length 1
61         # in the frequent patterns set, so the iteration is impossible.
62         if length == 2:
63             pattern = [pattern]
64         for item in pattern:
65             if item not in joined_candidates:
66                 joined_candidates.append(item)
67     joined_candidates = set(itertools.combinations(sorted(joined_candidates), length))
68     return joined_candidates
```

- The self_join function receives two variables: frequent patterns and length as arguments. The purpose of this function is to return joined candidate set, which is made from previous frequent patterns.
- After putting the items constituting all patterns in the frequent pattern into joined candidates list and using Python's built-in itertools to create tuples and store them in a set. (line 59 ~ line 67)

```python
def prune(candidates, previous_fp, length):
    pruned_candidates = copy.deepcopy(candidates)
    for itemset in candidates:
        # All subset of candidate itemset should be included
        # in the previous frequent pattern set.
        comb = set(itertools.combinations(sorted(itemset), length-1))

        # If the length is 2, there will be itemsets of length 1
        # the form of the items in comb is like (1,)
        if length == 2:
            for item in comb:
                if not set(item).issubset(previous_fp):
                    pruned_candidates.remove(itemset)
                    break
        # If the length is larger than 2,
        # the form of the items in comb is like (1, 2)
        else:
            for item in comb:
                if not set((item,)).issubset(previous_fp):
                    pruned_candidates.remove(itemset)
                    break
    return pruned_candidates
```

- The prune function receives three variables: candidates, previous frequent patterns and length as arguments. The purpose of this function is to return pruned candidate set, which is reduced form of candidates made through self-joining.
- If a certain pattern is to become a frequent pattern, all subset of the corresponding pattern must be included in the existing frequent pattern. In the prune function, the case in which the pattern length is 1 and the case in which the pattern length exceeds 1 were considered separately, because the result of combination is in the form of tuples.
- If a subset of a certain pattern is not included in the existing frequent pattern, it is excluded from the candidate. (line 72 ~ line 90)

```
93   # function to check the pattern's support
94   # whether it is higher than minimum support or not
95   def test_support(candidates, mns, transaction):
96       frequent_patterns = copy.deepcopy(candidates)
97       for itemset in candidates:
98           if get_support(itemset, transaction) < mns:
99               frequent_patterns.remove(itemset)
100      return frequent_patterns
```

- The test support function receives three variables: candidates, minimum support as number, and transaction as arguments. The purpose of this function is to return frequent patterns.
- First, deep copy pruned candidates in frequent pattern set. If the number of transactions containing the itemset is less than the minimum support, the itemset is removed from the frequent pattern set. (line 97 ~ line 99)

```
102  def association_rule(frequent_patterns, length, transaction):
103      line = ''  # result of the association_rule function
104      saved_length = length
105      for itemset in frequent_patterns:
106          while length > 1:
107              comb = set(itertools.combinations(itemset, length-1))
108              for item in comb:
109                  item_set = set(item)
110                  associative_item_set = set(itemset) - item_set
111                  support = get_support(itemset, transaction)/len(transaction)*100
112                  confidence = get_support(itemset, transaction)/get_support(item_set, transaction)*100
113                  line += str(item_set)+'\t'+str(associative_item_set)+'\t'+str('%.2f' % round(support, 2))+'\t'+str('%.2f' % round(confidence, 2))+'\n'
114              length -= 1
115          length = saved_length
116      return line
```

- The association rule function receives three variables: frequent patterns, length, transaction as arguments. The purpose of this function is to generate association rules using frequent patterns.
- For a frequent pattern, after obtaining a subset from length 1 to length of pattern – 1, I made an association rule by calculating support and confidence value. (line 105 ~ line 115)

```
118 if __name__ == '__main__':
119     minimnum_support = int(sys.argv[1])
120     input_file = sys.argv[2]
121     output_file = sys.argv[3]
122     transaction = list()
123     file_1 = open(input_file, 'r')
124     while True:
125         line = file_1.readline().strip()
126         if not line: break
127         transaction.append(sorted(map(int, line.split('\t'))))
128     file_1.close()
129     result = apriori(transaction, minimnum_support)
130     file_2 = open(output_file, "w")
131     file_2.write(result)
132     file_2.close()
```

- If a subset of a certain pattern is not included in the existing frequent pattern, it is excluded from the candidate. (line 72 ~ line 90)
- The above picture is the main function. As explained in the compilation method, the first argument is minimum support, the second argument is the name of input file, and the third argument is name of output file. The input file was read line by line to create a list containing transactions (line 123 ~ line 128), and the association rules created as a result of the apriori algorithm were written to the output file. (line 129 ~ line 132)

5. Testing result

input.txt

```
7	14
9
18	2	4	5	1
1	11	15	2	7	16	4	13
2	1	16
15	7	6	11	18	9	12	19	14
11	2	13	4
11	13
7	4	2	17	19	3	8	16	1
18	16	15	10	2	8	6	0	4	5
```

output.txt

```
{6}	{18}	7.20	31.86
{18}	{6}	7.20	26.09
{17}	{7}	5.40	22.69
{7}	{17}	5.40	22.50
{9}	{4}	9.00	32.37
{4}	{9}	9.00	36.59
{13}	{3}	9.00	30.41
{3}	{13}	9.00	30.00
{10}	{5}	7.20	24.83
{5}	{10}	7.20	28.57
```

- The above pictures are part of input file and output file captured. The row in the input file means one transaction, and the row in the output file means one association rule.
- As stated in the task specification, there is no duplication of items in each transaction.
- The order of association rules in the output file is random.
- The value of support and confidence are rounded to two decimal places.