Data Science [ITE4005]

Programming Assignment #1

: find association rules using the Apriori algorithm

2015004120
Software Engineering
Park Seonha

0. Environment

- OS : Windows
- Language : Python3 (version : 3.4.4)
- Executable file : apriori.py (../Programming_Assignment_1/project_apriori/apriori.py)

1. Summary of Algorithm

This programming assignment's goal is "find association rules using the Apriori algorithm." So I approached this assignment in two parts, "implement of apriori algorithm" and "make association rules with all frequent itemsets."

To implement apriori algorithm, I referenced the pseudo-code of apriori algorithm on class slides. This algorithm takes following steps, and I will explain this steps with small example which the number of transactions is 4 and minimum support is 2.

ⓐ make C_1(1 - candidate itemset) by scanning all transactions.

1 3 4 2 3 5 1 2 3 5 2 5 small example	Scanning all transactions and count supports of each item_id. C_1 = [{1}: 2, {2}: 3, {3}: 3, {4}: 1, {5}: 3]

ⓑ make L_1(1 - frequent itemset) by eliminate infrequent itemsets in C_1

C_1 = [{1}: 2, {2}: 3, {3}: 3, ~~{4}: 1~~, {5}: 3] , minimum support = 2
Since minimum support is 2, {4} is infrequent itemset and others are frequent itemset. Add frequent itemsets in empty L_1 and the result is, L_1 = [{1}: 2, {2}: 3, {3}: 3, {5}: 3]

ⓒ after k is 2, make C_k using L_{k-1} (do self-join and pruning infrequent itemsets)

L_1 = [{1}, {2}, {3}, {5}]
C_2 = $L_1 \bowtie L_1$ = [{1,2}, {1,3}, {1,5}, {2,3}, {2,5}, {3,5}] All subsets of itemset in C_2 are in L_1, I don't have to prune infrequent itemset. and get supports of each itemset in C_2, the result of this stage is C_2 = [{1,2}: 1, {1,3}: 2, {1,5}: 1, {2,3}: 2, {2,5}: 3, {3,5}: 2]

ⓓ make L_k pick up frequent itemsets in C_k

C_2 = [~~{1,2}: 1~~, {1,3}: 2, ~~{1,5}: 1~~, {2,3}: 2, {2,5}: 3, {3,5}: 2] , minimum support = 2
{1,2} and {1,5} are not frequent set, so eliminate those itemsets. L_2 = [{1,3}: 2, {2,3}: 2, {2,5}: 3, {3,5}: 2]

ⓔ repeat ⓒ and ⓓ while L_{k-1} is not empty set.

L_2 = [{1,3}, {2,3}, {2,5}, {3,5}] C_3 = $L_2 \bowtie L_2$ = [{1,2,3}, {1,3,5}, {2,3,5}] {1,2,3} is infrequent set because its subset {1,2} is infrequent. [Apriori pruning principle] Same reason, {1,3,5} is infrequent set because of its subset {1,5}. C_3 = [{2,3,5} : 2] , minimum support = 2 {2,3,5} is frequent set so L_3 is, L_3 = [{2,3,5} : 2]
L_3 = [{2,3,5}] C_4 = $L_3 \bowtie L_3$ = $\varnothing$, so L_4 is empty set, and algorithm end.

If L_{k-1} is empty set, C_k will be empty set and L_k will be also empty, so this algorithm ends in terminal condition [L_{k-1} is empty set].

The result of this apriori algorithm is set of all frequent itemsets and supports.

L = [{1}: 2, {2}: 3, {3}: 3, {5}: 3, {1,3}: 2, {2,3}: 2, {2,5}: 3, {3,5}: 2, {2,3,5}: 2]

Second part is making association rule with set of frequent itemsets L. In this process, I don't need 1-frequent itemset since the formula of generating association rule is $s \rightarrow (t-s)$ with non-empty set t and its non-empty subset s. So, I can remove 1-frequent itemsets, and the result is below.

L' = [{1,3}: 2, {2,3}: 2, {2,5}: 3, {3,5}: 2, {2,3,5}: 2]

For each itemset in L', make all non empty subsets and its difference set. Also, $P(t-s\|s)$ is $P(t)/P(s)$, and it is each association rule's confidence.

{1,3} subset : {1}, {3}	{1} -> {3} {3} -> {1}	0.5 0.5	1 0.67
{2,3} subset : {2}, {3}	{2} -> {3} {3} -> {2}	0.5 0.5	0.67 0.67
{2,5} subset : {2}, {5}	{2} -> {5} {5} -> {2}	0.75 0.75	1 1
{3,5} subset : {3}, {5}	{3} -> {5} {5} -> {3}	0.5 0.5	0.67 0.67
{2,3,5} subset : {2}, {3}, {5}, {2,3}, {2,5}, {3,5}	{2} -> {3,5} {3} -> {2,5} {5} -> {2,3} {2,3} -> {5} {2,5} -> {3} {3,5} -> {2}	0.5 0.5 0.5 0.5 0.5 0.5	0.67 0.67 0.67 1 0.67 1

2. Description of codes

get command line arguments

```python
import sys

#to change raw input to command line argument
minsup_param = int(sys.argv[1])
input_file = sys.argv[2]
output_file = sys.argv[3]

minsup = minsup_param / 100
```

To see code top to bottom, before start main process, program get command line arguments(minimum support, input file name, output file name) with **sys** module.

implemented funtions

```python
# change item_set between list and string {[item_id],[item_id],...[item_id]}
def list_to_str(item_set):
    brace = "{"
    for item_id in item_set:
        brace = brace + str(item_id)
        brace += ","
    brace = brace[:-1]
    brace += "}"
    return brace

def str_to_list(str_list):
    str_list = str_list[1:-1]
    new_list = list(map(int, str_list.split(",")))
    return new_list
```

This code is written in Python3 and because of its specialty, I have to define some functions to implement this assignment. Since Python can't make [key : list, value : int] dictionary, I made functions which can change list to string and list-formed string to list. Since I implement this functions, candidate itemset and frequent itemset are implemented as [key : string(list-formed), value : int], and also available set operations.

list_to_str(item_set) gets list as parameter, and return list-formed string. It is same as assignment item set output format. For example if list_to_str([1, 3, 4]) executed, it returns string "{1,3,4}".

In reverse, function *str_to_list(str_list)* gets string parameter like "{3,7,8}". After remove braces("{", "}") and commas, this function returns list [3, 7, 8].

```python
# make subset of itemset
def jin_subset(item_set):
    result_set = [[]]
    for x in item_set:
        result_set.extend([y + [x] for y in result_set])
    # this is empty set
    result_set.pop(0)
    # this is not proper-subset
    result_set.pop(len(result_set)-1)
    for it in result_set:
        it.sort()
    return result_set
```

jin_subset(item_set) function gets list as parameter, and returns list of all subset of parameter. If jin_subset([1,2,3]) executed, 33 line and 34 line makes all subset of item_set ([$\varnothing$, [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]), and after 36 line and 38 line, this function removes empty set and set which is exactly same with parameter set. Now, there are only remain proper subset, return this set of proper subsets after sorting each proper subsets. In this case, this function returns [[1], [2], [1,2], [3], [1,3], [2,3]].

```python
# return 1 if set1 and set2 is same set, return 0 other case
def set_comp(set1, set2):
    result = 0
    if len(set(set1).difference(set(set2))) == 0:
        if len(set(set2).difference(set(set1))) == 0:
            result = 1
    return result
```

set_comp(set1,set2) gets two sets as parameter, and checks these two sets are exactly same or not. If $A-B$ is empty set and $B-A$ is also empty set, they are same set because all elements are in $A\cap B$. So this function designed to check $set1-set2$ and if it has zero length, in other words, if it is empty set, check if $set2-set1$ is empty set or not. This function returns 1 if two sets are same, other case, returns 0.

```python
52 #change round function because python round is not sa-sa-o-ip
53 def new_round(num):
54     new_num = num*1000
55     #if num is 13.255 -> new num is 13255
56     over_num = int(new_num/1000)
57     first_num = int((new_num%1000)/100)
58     second_num = int((new_num%100)/10)
59     third_num = int(new_num%10)
60     upper = 0
61     if third_num >= 5:
62         second_num = second_num + 1
63     if second_num > 9:
64         second_num = second_num - 10
65         first_num = first_num + 1
66     if first_num > 9:
67         first_num = first_num - 10
68         upper = 1
69
70     ret_num = int(over_num) + int(upper) + round(float(first_num/10),1) + round(float(second_num/100),2)
71     ret_num = round(ret_num,2)
72     return ret_num
```

new_round(num) is the function gets a floating point number as parameter and return its two decimal place rounded number. Since python internal round function implied 'round to nearest even' way and human uses 'round to nearest integer' way, I defined new round off function. In 'round to nearest integer' way, we don't use after three decimal place, so if input is $ab.cdefg_{(10)}$, after 54 line, it changes $abcde.fg_{(10)}$ and separated $ab_{(10)}$(over_num), $c_{(10)}$(first_num), $d_{(10)}$(second_num), $e_{(10)}$(third_num).

round off third_num and regrouping with second_num, first_num, and merge them as $ab_{(10)}+c/10_{(10)}+d/100_{(10)}$ = $ab.cd_{(10)}$. After merging, to prevent long tail numbers comes from floating point number's inaccuracy like 13.10000000000000135, I use round function at two decimal place and it doesn't affect return value's accuracy because we only left number on two decimal place, so under two decimal place never raise up.

Import input data and parsing

```
75 # open input data file and store in transaction list
76 with open(input_file) as f:
77     input_data = f.readlines()
78 input_data = [d.strip() for d in input_data]
79
```

after 75 line, main process of this program runs. Open input file and read file as [str, ...] format. Since all transaction strings finish "₩n", I used strip() remove newline.

```
# parse each input lines to item_ids and get the number of total transactions
transactions = []
total_trans = 0
for input_line in input_data:
    trans_list = list(map(int, input_line.split("\t")))
    transactions.append(trans_list)
    total_trans += 1
```

Since transaction stored like "1₩t2₩t3₩t5", parse this string to list of integer. *transactions* is the list of integer list. and while appending each transaction, count total number of transactions(*total_trans*).

After this process, this program will have [[item_id, item_id, ...], ...] list and the number of transactions, and they are necessary to execute apriori algorithm.

Start apriori algorithm

```
# Start apriori algorithm
cand = []
freq = []
```

cand is the list which will contain C_1 ... C_k in dictionary type. And *freq* is the list which will contain L_1 ... L_k in dictionary type. So their structure is list of dictionary whose key is string(list-formed) and value is integer.

```
# make C_1 (1-candidate itemset) by scanning transactions
# since list cannot be key of dictionary in python3, I changed sorted itemset to string
cand_1 = {}
for trans in transactions:
    for item_id in trans:
        itemset = []
        itemset.append(item_id)
        itemset_key = list_to_str(itemset)
        if itemset_key not in cand_1:
            cand_1[itemset_key] = 1
        else:
            cand_1[itemset_key] += 1
cand.append(cand_1)
```

First step of apriori algorithm is making C_1 (1- candidate set) by using *transactions*.

scan all transactions in *transactions*, make list *itemset*, and change *itemset* list to string to use as the key of dictionary in Python. (In python, list cannot be the key of dictionary because it is not hashable type) After make *itemset_key*, check if C_1 already has itemset_key. If it exists, count up its value, doesn't exist, add new key and value to C_1 whose key is *itemset_key*, value is 1.

After generate dictionary C_1, append C_1 into list *cand*. (cand[0] = C_1)

```
# make L_1 (1-frequent itemset) by pruning C_1
freq_1 = {}
for key, value in cand_1.items():
    if value / total_trans >= minsup:
        freq_1[key] = value
freq.append(freq_1)
```

L_1 is also dictionary whose key is string and value is integer. To generate L_1 using C_1, check if itemset's support overs minimum support or not.

After generate dictionary L_1, also append L_1 in *freq* list.

```
k = 2
```

```
# for(k=2 ; L_k != empty set ; k++)
while k <= total_trans and len(freq[k - 2]) != 0:
    # make cand_k(C_k) from freq_km(L_k-1) joining itself and pruning
    cand_k = {}
    freq_km = freq[k - 2]

    # self-joining step
    joined_fkm = []
    for k1, v1 in freq_km.items():
        for k2, v2 in freq_km.items():
            kj_s = set(str_to_list(k1)).union(set(str_to_list(k2)))
            kj_l = list(kj_s)
            kj_l.sort()
            if len(kj_l) != k:
                continue
            else:
                if list_to_str(kj_l) not in joined_fkm:
                    joined_fkm.append(list_to_str(kj_l))
                else:
                    continue
```

After make C_1 and L_1, we can get C_k with using L_{k-1}. (k≥2) This loop will execute while L_{k-1} is not empty set.

cand_k is C_k, *freq_km* is L_{k-1}. And we need to self-join L_{k-1}. Since we are generating k-candidate itemset, union two set in L_{k-1}, and translate it to list and remain sets whose size is *k*. If it can be candidate itemset, append that list in *joined_fkm* as string format.

In this result, all candidate itemsets (whether it is frequent or not) are in *joined_fkm*.

```
138     # pruning step
139     # check its subset in freq_km, freq_kmm ...
140     # make each joined_fkm's non-empty subset
141     pruned_fkm = []
142     for j_set in joined_fkm:
143         subs_fkm = jin_subset(str_to_list(j_set))
144         prune = -1
145         for subset in subs_fkm:
146             prune = 1
147             for fset in freq:
148                 for key_fset , val_fset in fset.items():
149                     if set_comp(str_to_list(key_fset), subset) == 1:
150                         prune = 0
151             if prune == 1:
152                 break
153         if prune == 1:
154             continue
155         else :
156             pruned_fkm.append(j_set)
157     for p_set in pruned_fkm:
158         cand_k[p_set] = 0
```

```
160     # for each transactions
161     # increment the count of each candidate in C_k which are contained in t
162     for candidate, value in cand_k.items():
163         cand_list = str_to_list(candidate)
164         for transact in transactions:
165             if len(list(set(cand_list).difference(set(transact)))) == 0:
166                 cand_k[candidate] += 1
```

After make *joined_fkm*, we need to prune infrequent itemset. According to Apriori pruning principle, if a subset of set S is infrequent, S must be infrequent set. Using this principle, I check sets in *joined_fkm* should be pruned or not. Check all non-empty subsets of each set in *joined_fkm*, and if any subset is not in *freq*, that set will be pruned.

After pruning, make C_k with *pruned_fkm*. Since *cand_k* is key-value dictionary and I don't know each of their support, all candidate itemset's support initialized 0 and get counts check *transactions*. If a transaction includes candidate itemset (this can be checked by $candidate - transaction$ is empty set or not), that candidate counts up.

```
168     # make freq_k(L_k) in cand_k(C_k) with minsup
169     freq_k = {}
170     for candidate, value in cand_k.items():
171         if cand_k[candidate] / total_trans >= minsup:
172             freq_k[candidate] = value
```

Now, C_k is generated. And generating L_k is available. Check all itemsets in C_k if its support is same or over minimum support. If it is, add that itemset in *freq_k*.

```
174     cand.append(cand_k)
175     freq.append(freq_k)
176     k += 1
```

Append C_k and L_k to *cand* and *freq*. increase *k*, return to top of loop, and generate C_{k+1} and L_{k+1} during L_k is not empty set. if 'while'(line 118) ends, apriori algorithm part is end.

making association rules

```python
freq.pop(len(freq)-1)
```

Since while terminology condition is L_{k-1} is empty set, freq[size of freq – 1] is empty itemset dictionary. So remove last item of *freq*.

```python
decomp_freq_dic = {}
decomp_freq = []
for fset in freq:
    for key, value in fset.items():
        if len(str_to_list(key)) > 1:
            decomp_freq.append(str_to_list(key))
        decomp_freq_dic[key] = value
```

Structure of *freq* is [[{string:int}, …], …] (list of the list of dictionary whose key is string and value is integer). It is too complicate so it needs to be decomposed.
I decomposed freq to *decomp_freq* (string array), *decomp_freq_dic*(dictionary whose key is string and value is integer). for all dictionary list in *freq*, append its itemset to decomposed list and dictionary. And 1-frequent itemset has no non-empty proper subset, I removed itemsets in L_1 to check the size of list is bigger than 1.

```python
#make association rules with each subset and write in output file
for fset in decomp_freq:
    set_size = len(fset)
    all_subset = jin_subset(fset)
    for subset in all_subset:
        lhs = subset
        rhs = list(set(fset).difference(set(lhs)))
        rhs.sort()

        s = decomp_freq_dic[list_to_str(fset)] / total_trans
        c = (decomp_freq_dic[list_to_str(fset)] / total_trans) / (decomp_freq_dic[list_to_str(list(lhs))] / total_trans)
        s *= 100
        c *= 100
        out_f.write(list_to_str(list(lhs)) + "\t" + list_to_str(list(rhs)) + "\t" + str(new_round(s)) + "\t" + str(new_round(c))+"\n")
```

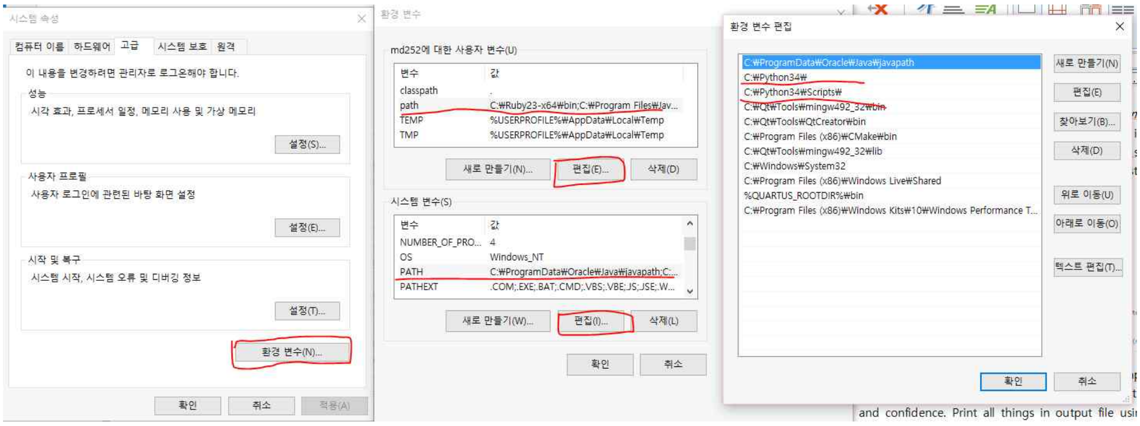
For frequent itemset in *decomp_freq*, make all non-empty proper subset with function jin_subset(), get difference set of each subset using set operation in python, calculate support(s) and confidence(c). Print all things in output file using write(). Since python round() function doesn't return value we expecting, I defined *new_round()* function at top of code, and used here.

3. Instructions for compiling this code

This code is written in Python3 and tested in Python 3.4.4. To run this code, we need python 3.4.4 interpreter.

We can get python 3.4.4 in (https://www.python.org/downloads/release/python-344/)

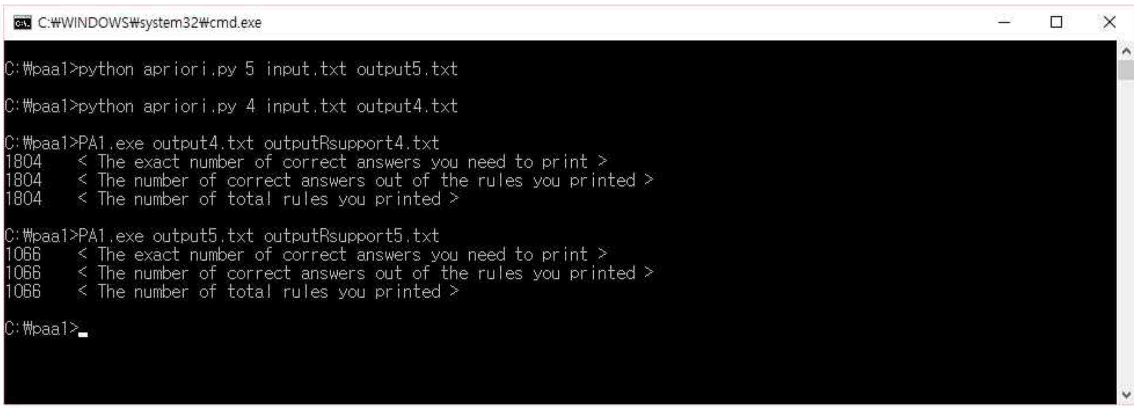
If python2 is already installed in PC, change path Python2x to Python34 in advanced system settings – environment variables.



After environment variables setting, we can get python3 default in cmd.

In code directory(python file is in /project_apriori), we can run this program on cmd "python apriori.py [minimum support] [input file name] [output file name]".

4. Other Specifications



In testing, some computers take quite long time to generate result because of their performance. (more than 40 seconds in minimum support 5%) It has additional condition in while loop (k is smaller than total_trans), so it can't make infinite loop, just wait until it will finish.