Data Mining Assignment 1
Association Rule Mining

Bo-Han Chen (陳柏翰)
Student ID:312551074
bhchen312551074.cs12@nycu.edu.tw

Experiment Environment & Usage

Environment

The Experiment environment is based on the work station of Information Technology Center, the details are as follows:

- OS: CentOS Stream release 8
- Hardware: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
- Python 3.9.17
- The computation time is recorded by *time.process_time()* function

Usage

The command for executing the program of step2 & 3 is shown as follows:

```
# step2
python apriori.py -f [inputFile] -t [task] -s [support]
# step3
python myEclat.py -f [inputFile] -s [support]
```

I wrote a script for running the association rule mining program, whcich can run the algorithm with all task/support/dataset options, and the execution time will be recorded in the log file named *result.log*. Take the script of step2 for example, the script *run.sh* is shown as follows:

```
#!/bin/bash
dataset_folder="../dataset"
log_file_path="../result"
declare -a dataset_arr=("datasetA.data" "datasetB.data" "datasetC.data")
declare -a task_arr=(1 2)
declare -a sup_arrA=(0.2 0.5 0.1)
declare -a sup_arrB=(0.5 0.2 0.5)
declare -a sup_arrC=(0.1 0.2 0.3)

for task in "${task_arr[@]}"
do
    for sup_idx in 0 1 2
    do
        for dataset in "${dataset_arr[@]}"
        do
            if [ $dataset == 'datasetA.data' ]
            then
                sup_arr=("${sup_arrA[@]}")
            elif [ $dataset == 'datasetB.data' ]
            then
                sup_arr=("${sup_arrB[@]}")
            else [ $dataset == 'datasetC.data' ]
                sup_arr=("${sup_arrC[@]}")
            fi
            data_path=$dataset_folder/$dataset
            sup="${sup_arr[$sup_idx]}"
            echo "running $dataset on task $task with support: $sup"
```

```
            time python apriori.py -f $data_path -t $task -s $sup | tee -a $log_file_path/result.log
        done
    done
done
```

The usage of *run.sh*:

```
# executing run.sh script
./run.sh
```

Step2: Apriori Algorithm

Task1: Mining Frequent Itemsets

In this part, I add two functions *writeTask1_1* and *writeTask1_2* to write the frequent itemsets to the txt file based on the original Apriori algorithm. The code is shown as follows:

```
def runApriori_1(data_iter, case, minSupport):
  itemSet, transactionList = getItemSetTransactionList(data_iter)

  freqSet = defaultdict(int)
  largeSet = dict()
  # initialize the number of candidate itemset before and after pruning
  canNumSetBf = [len(itemSet)]
  canNumSetAf = []

  oneCSet= returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet)
  canNumSetAf.append(len(oneCSet))

  currentLSet = oneCSet
  k = 2
  while currentLSet != set([]):
      largeSet[k - 1] = currentLSet
      currentLSet = joinSet(currentLSet, k)
      # get the number of candidate itemset before pruning
      canNumSetBf.append(len(currentLSet))
      currentCSet= returnItemsWithMinSupport(
          currentLSet, transactionList, minSupport, freqSet
      )
      # get the number of candidate itemset after pruning
      canNumSetAf.append(len(currentCSet))
      currentLSet = currentCSet
      k = k + 1
  .
  .
  .
  # write the frequent itemsets and number of candidate to file
  writeTask1_1(toRetItems, case, minSupport)
  writeTask1_2(canNumSetBf, canNumSetAf, case, minSupport)
```

In *writeTask1_1* function, the frequent itemsets will be sorted by support and be written to the file.

```
def writeTask1_1(items, case, sup):
  """write the generated itemsets sorted by support to file"""
  write_line = ''
  for itemset, support in sorted(items, key=lambda x: x[1], reverse = True):
      item_str = ""
      for item in itemset:
          item_str = item_str + str(item) + ','
      item_str = item_str.strip(',')
      write_line += "%.1f\t{%s}\n" %(support * 100, item_str)
  with open('../result/' + 'step2' + '_task1_' + case + '_' + str(sup) + '_result1.txt', mode = 'w') as write_file:
      write_file.write(write_line)
```

In *writeTask1_2* function, the number of candidate itemsets before and after pruning will be written to the file.

```
def writeTask1_2(canNumSetBf, canNumSetAf, case, sup):
    """write the number of candidate itemsets before and after pruning to file"""
    write_line = str(sum(canNumSetAf)) + '\n'
    for idx in range(len(canNumSetBf)):
        write_line += "%s\t%s\t%s\n" %(str(idx + 1), str(canNumSetBf[idx]), str(canNumSetAf[idx]))
    with open('../result/' + 'step2' + '_task1_' + case + '_' + str(sup) + '_result2.txt', mode = 'w') as write_file:
        write_file.write(write_line)
```

The computation time of task1 is shown as follows (concluded from the *result.log* file):

Dataset	Minimum Support (%)	Computation Time (sec)
A	0.2	143.79
	0.5	6.72
	1.0	2.79
B	0.15	6861.15
	0.2	3823.43
	0.5	1111.96
C	1.0	6074.08
	2.0	1994.86
	3.0	729.53

Table 1: Computation Time of Task1

As we can see above, the computation time increases considerablely when the minimum support (*min_sup*) decreases. Take dataset A for example, and the computation time of $min\_sup = 0.5\%$ is 95% faster than $min\_sup = 0.2\%$, and the computation time of $min\_sup = 1.0\%$ is 98% faster than $min\_sup = 0.2\%$.
To explain this phenomenon, we can analyze *result2.txt* file to find out the reason. Comparing the number of candidate k-itemsets (L_k) of each iteration among $min\_sup = 0.5\%$ and $min\_sup = 0.2\%$, we can observe that with higher minimum support, fewer frequent k-itemsets (F_k) will remain in each iteration, which leads to fewer procedure to calculate the support of itemsets in L_{k+1}.

Task2: Mining All Frequent Closed Itemsets

In this task, I first check whether the frequent itemset is closed or not by *checkClosed* function in each iteration, and then write the closed frequent itemsets to the file by *writeTask2*.

```
def runApriori_2(data_iter, case, minSupport):
    itemSet, transactionList = getItemSetTransactionList(data_iter)
    ...
    k = 2
    # save the closed frequent itemsets in each iteration
    closedSet = dict()
    while currentLSet != set([]):
        largeSet[k - 1] = currentLSet
        currentLSet = joinSet(currentLSet, k)
        currentCSet= returnItemsWithMinSupport(
            currentLSet, transactionList, minSupport, freqSet
        )
        # check whether the frequent itemset is closed or not
        # passing the frequent itemset in last iteration and current iteration
        closedSet[k - 1] = checkClosed(largeSet[k-1], currentCSet, freqSet)
        currentLSet = currentCSet
        k = k + 1
    ...
```

```
    # write the closed frequent itemsets to file
    closedItems = []
    for key, value in closedSet.items():
        closedItems.extend([(tuple(item), getSupport(item)) for item in value])

    writeTask2(closedItems, case, minSupport)
```

In *checkClosed* function, each itemset of F_{k-1} will be compared with each itemset of F_k, if the latter one is a superset of the former and the support of the latter is larger or equal (equal, precisely) to the former one, then we can say that itemset is not closed.

```
def checkClosed(canLevelPre, canLevelCur, freqSet):
    # first assume that all itemsets of previous iteration are closed
    closedSetPre = canLevelPre.copy()
    for item_pre in canLevelPre:
        for item_cur in canLevelCur:
            # if item_cur is a superset of item_pre
            # and the support of item_cur is larger than item_pre
            # then the latter one is not closed
            if item_pre.issubset(item_cur) and freqSet[item_pre] <=
  freqSet[item_cur]:
                closedSetPre.remove(item_pre)
                break
    return closedSetPre
```

The computation time of task2 and the comparison with task1 is shown as follows:

Dataset	Minimum Support (%)	Computation Time (sec)	Ratio of Computation Time (%)
A	0.2	157.117	109.26%
	0.5	6.65	98.95%
	1.0	2.70	96.77%
B	0.15	7094.64	103.40%
	0.2	3730.72	97.57%
	0.5	1137.05	102.25%
C	1.0	6007.42	98.90%
	2.0	1962.21	98.36%
	3.0	717.23	98.31%

Table 2: Computation Time of Task2

With low *min_sup* (take datasetA with $min\_sup = 0.2\%$ for example), we can observe that task2 is obviously slower than task1, since there are more itemsets in F_{k-1} and F_k, and there will also have more iteration in the while loop, which cause more check procedure in *checkClosed* function. Sometimes the computation of task2 is even faser than task1, by observing the *result.log* file, we can find out such condition is caused by the number of iteration, in other words, if there is fewer iteration, the extra computation of *checkClosed* is nearly negligible.

Step3: Eclat Algorithm

For task3, I choose Eclat mining algorithm to mine the frequent itemsets. In this section, I will first introduce the Eclat algorithm, then explain its advantages compared to Apriori algorithm, finally analysis the experiment result.

Introduction

Eclat algorithm [1] [2] is a depth-first-based association mining algorithm using the vertical database, instead of calculating the support of each itemset by traversing the whole trasaction list, Eclat algorithm uses the intersection of TID_Sets, which results in more efficient computation.

Program Flow

The pueudocode of Eclat algorithm is shown as follows:

Algorithm 1: My Eclat Algorithm Overview

```
input  : Transaction list T, minimum support Sup_min
output: Frequent itemsets F

// build the vertical database VDB from T
for tran in T do
    for item in tran do
        append tran to VDB[item]
    end
end
// get the frequent 1-itemsets F_1 from VDB
for item in VDB do
    if |VDB[item]| ≥ Sup_min then
        append item to F_1
    end
end
// mine the frequent itemsets recursively
F = []
for item in F_1 do
    EclatRecursive(item, VDB[item], idx(item))
end
```

First the vertical database VDB will be built from the trasaction list T, then the frequent 1-itemsets F_1 will be generated from VDB. Finally, the frequent itemsets will be mined recursively by *EclatRecursive* function. Why we need to build F_1 before calling *EclatRecursive* function will be discussed later.
The following algorithm represents the *EclatRecursive* function:

Algorithm 2: EclatRecursive Function

```
input  : frequent itemset i, tid set SET_i, index of i's last item IDX_i

for j ← IDX_i + 1 to |F_1| do
    SET_ij = SET_i ∩ VDB[j]
    if |SET_ij| ≥ Sup_min then
        i_new = i ∪ F_1[j]
        append i_new to F
        EclatRecursive(i_new, SET_ij, j)
    end
end
```

The *EclatRecursive* function will traverse F_1 from i's last item. For each item $F_1[j]$, the function will first intersect its' tid list $VDB[j]$ and SET_i, the result SET_{ij} represents the transactions that contain both i and $F_1[j]$. If the size of SET_{ij}, which means the new itemset's support, is larger than Sup_{min}, then the union of i and $F_1[j]$, i_{new}, will become the new frequent itemset. Finally the *EclatRecursive* function will be called recursively to find the new frequent with i_{new} as the prefix.

Eclat Algorithm vs. Apriori

The differences between Eclat algorithm and Apriori algorithm are the approach of searching the frequent itemsets and the data structure of the database.

For Apriori algorithm, the frequent itemsets will be searched by breadth-first search (BFS), which lead to the high-cost computation of counting support. By using vertical database and depth-first search (DFS), Eclat algorithm can reduce the computation time of counting support even the candidate number is larger than Apriori.

Time Complexity in the Worse Case

Given a dataset with $ntrans$ transaction, $nitems$ items and average transaction length $tlen$, we can analyze the complexity of Eclat algorithm and Apriori. Initially, the time complexity of both algorithms are $O(ntrans \times tlen)$ for building the itemset list and vertical database.
By searching frequent itemsets by Apriori, the time complexity is $O(nitems \times ntrans \times tlen)$, since in the worse case, there will have $nitems$ iterations, and for each iteration, Apriori will traverse the whole trasaction list ($O(ntrans \times tlen)$) to calculate the support for corresponding itemsets.
For Eclat algorithm, the time complexity is $O(2^{nitems} - 1) \times O(ntrans)$ searching the frequent itemsets, since in the worst case, the algorithm will perform intersection operation $2^{nitems} - 1$ times, and for each intersection operation, the time complexity will be $O(ntrans)$ in the worse case.

Experiment

The computation time of task3 and the comparison with task1 is shown as follows:

Dataset	Minimum Support (%)	Computation Time (sec)	Speedup Percentage (%)
A	0.2	2.92	97.96%
	0.5	0.19	97.17%
	1.0	0.06	97.84%
B	0.15	67.09	99.02%
	0.2	40.32	98.94%
	0.5	6.93	99.37%
C	1.0	65.37	98.92%
	2.0	27.88	98.60%
	3.0	13.61	98.13%

Table 3: Computation Time of Step3

As we can see above, the computation time of each dataset and minimums support setting is much faster than Apriori algorithm. By checking the *result2.txt* file, we can see that the with larger candidate number before pruning, the speedup percentage will get higher, which means the support counting method of Eclat is more efficient than Apriori.

Scalability of Eclat Algorithm

For testing the scalability of Eclat algorithm, I executing the Eclat program with fixed minimum support and datset with different $ntrans$, the result is shown as follows:

Minimum Support (%)	Dataset	Computation Time (sec)
0.1	A	1407.57
	B	124.47
	C	2864.68
0.15	A	2.94
	B	67.09
	C	1061.82
0.2	A	2.92
	B	40.32
	C	656.96
0.5	A	0.19
	B	6.93
	C	116.97
1.0	A	0.06
	B	3.81
	C	65.37
0.2	A	0.02
	B	1.83
	C	27.88
0.3	A	0.01
	B	0.92
	C	13.61

Table 4: Computation Time of Scalability Test

It's obviously to observe that the computation time will increase considerablely when the number of transaction increases. Since the *ntrans* will affect the computation time of building the vertical database and the length of each itemset's tid list, which will lead to longer computation time.
The other observation is that with $Sup_{min} = 0.1\%$, the computation time of dataset A is abnormally large, so the following section will discuss this phenomenon.

Restriction of Eclat Algorithm

To explain the abnormal result of datasetA with $Sup_{min} = 0.1\%$ in the previous section, I think the main reason is the complexity of intersection operation in Eclat algorithm. Since datasetA not only has lower *ntrans* setting compared to datasetB and datasetC, but also lower *nitems*, so each itemset will have more appearance in the whole trasaction list, which resulted in larger tid list of each itemset in the vertical database. As we discussed in section 3.3.1, the time complexity of Eclat is mainly affected by the number of intersection operation and the size of tid list be intersected. So we can concluded that with fixed *ntrans* annd *tlen*, the computation time of Eclat algorithm will become slower if the *nitems* of dataset gets lower.
To verifiy the inference above, I generate extra dataset with fixed *ntrans*/*tlen* and different *nitems*. Details of datasets for experiment in this section are shown as follows (the setting of datasetF is same as datasetB):

Dataset	ntrans	tlen	nitems
D	100000	10	200
E	100000	10	400
F	100000	10	600
G	100000	10	800

With different Sup_{min} setting, the computation time of executing Eclat algorithm on dataset D/E/F/G is shown as follows:

Minimum Support (%)	Dataset	Computation Time (sec)
0.1	D	256.56
	E	147.04
	F	124.47
	G	104.75
0.15	D	140.01
	E	84.09
	F	61.34
	G	39.70
0.2	D	93.50
	E	57.26
	F	36.70
	G	22.40
0.5	D	25.23
	E	11.57
	F	6.55
	G	6.31
1.0	D	9.40
	E	3.96
	F	3.55
	G	3.81
0.2	D	3.05
	E	1.90
	F	1.71
	G	1.18
0.3	D	1.64
	E	1.14
	F	0.80
	G	0.42

Table 5: Computation Time of Dataset with Different $nitems$

As we can see in the table above, with fixed Sup_{min}, the computation time is getting higher while the $nitems$ is getting lower. If the Sup_{min} gets higher, since the number of intersection operation will decrease, so the difference of computation time becomes smaller. Finally we can conclude that the appearance of each itemset in the whole trasaction list will become a performance restriction of Eclat algorithm.

References

[1] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, Wei Li, et al. New algorithms for fast discovery of association rules. In *KDD*, volume 97, pages 283–286, 1997.

[2] Christian Borgelt. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 2(6):437–456, 2012.