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.

```python
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	0.1	6074.08
	0.2	1994.86
	0.3	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*.

```python
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
    ...
```

```python
    # 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.

```python
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	0.1	6007.42	98.90%
	0.2	1962.21	98.36%
	0.3	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 is an 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

References

[1] J. Heaton, "Comparing dataset characteristics that favor the apriori, eclat or fp-growth frequent itemset mining algorithms," pp. 1–7, 2016.