Data Mining HW1

Name: 陳以瑄 Student ID: 109705001

(1) Explain how to run the code

- StepII

  Under the directory ~/DM_HW1_109705001_陳以瑄/SourceCode

  Type python StepII_apriori.py [-f file_path] [-s min_support] [-t task_id]

  Argument:

  -f file_path : relative file path, default is none

  -s min_support: min support ratio, default is 0.1

  -t task_id: 1 stands for task1, 2 stands for task2, default is 1

  Example:

```
python StepII_apriori.py -f ../Datasets/A.data -s 0.01 -t 1
```

  Runs task1 on dataset A.data, with min support 1%.

```
python StepII_apriori.py -f ../Datasets/B.data -s 0.002 -t 2
```

  Runs task2 on dataset B.data, with min support 0.2%.

- StepIII

  Under the directory ~/DM_HW1_109705001_陳以瑄/SourceCode

  Type python StepIII_eclat.py [-f file_path] [-s min_support]

  Argument:

  -f file_path : relative file path, default is none

  -s min_support: min support ratio, default is 0.1

  Example:

```
python StepIII_eclat.py -f ../Datasets/A.data -s 0.005
```

  Runs task1 on dataset A.data, with min support 0.5%.

(2) StepII

Modification of codes:

- Task1

1. Parse task flag: (line 202~208, line 222)

   In main:

```
202     optparser.add_option(
203         "-t","--task",
204         dest="task",
205         help="Task number (1 or 2)",
206         type="int",
207         default=1
208     )
```

```
222     task = options.task
```

   I add a new flag to parse augment “-t task”. Since we need to run task1 and task2, adding this flag is more efficient. If we type -t 1, it will run task 1. If we type -t 2, it will run task 2.

2. Modify “dataFromFile” to read .data: (line 157~166)

   In dataFromFile:

```
157 """Modify: read.data file"""
158 def dataFromFile(fname):
159     """Function which reads from the file and yields a generator"""
160     with open(fname, "r") as file_iter:
161         for line in file_iter:
162             line = line.strip().rstrip(",")
163             """Modify: to fit the input data 移除前面的 SID TID NITEMS """
164             trans = line.split(" ")[3:]
165             record = frozenset(trans)
166             yield record
```

   The original function is used to read .csv file, which separate the values by comma. However, the data we use is .data file, which separate values by whitespace rather than comma. Therefore, in line 164, I split the line by whitespace.

   The data generate by IBMGenerator contains SID, TID, NITEMS, and ITEMSET, but we only need ITEMSET, therefore, in line 164, I weed out the first 3 columns.

3. Modify “runApriori” to save result2 data: (line 70~75, 83~91, 100~105)

```
70      """Modify: initial support and statistics"""
71      itemSupportDict = {}
72      toStatistics = []
73      itemSet, transactionList = getItemSetTransactionList(data_iter)
74      freqSet = defaultdict(int)
75      largeSet = dict()
```

Since we need to save the number of candidates generated before and after pruning in each iteration of Apriori to statistics file, we need to save the data to some variables.

“itemSupport” saves the support of the item.

“toStatistics” save the number of candidates generated before and after pruning in each iteration.

```
83      """Modify: save 1st statistics data"""
84      k = 1
85      toStatistics.append([k, len(itemSet), len(oneCSet)])
86      total = len(oneCSet)
87      k += 1
88
89      """Modify: save the 1st support"""
90      for item in currentLSet:
91          itemSupportDict[item] = getSupport(item)
```

Before the iterate, we need to save the statistical data and support of 1-itemset. The number of candidates generated before pruning is the length of the initial itemSet, and the after-pruning one is the length of oneCSet. Also, save the total number of frequent items in the variable “total”.

```
93      if(task==1):
94          while currentLSet != set([]):
95              largeSet[k - 1] = currentLSet
96              currentLSet = joinSet(currentLSet, k)
97              currentCSet= returnItemsWithMinSupport(
98                  currentLSet, transactionList, minSupport, freqSet
99              )
100             for C_item in currentCSet:
101                 itemSupportDict[C_item] = getSupport(C_item)
102
103             """Modify: save the statistics"""
104             toStatistics.append([k, len(currentLSet), len(currentCSet)])
105             total+=len(currentCSet)
106             currentLSet = currentCSet
107             k = k + 1
```

In each iteration, we need to save the statistic data and support of the frequent itemset. The number of candidates generated before pruning is the length of currentLSet after join, and the after pruning one is the length of currentCSet. Also, we need to add the number of new candidates to the "total" variable.

4. Save results: (line 168~178, line 223~225, line 228~230)

   In saveResultToFileTask1:

```
168 """Modify: save task1 file"""
169 def saveResultToFileTask1(items,total,statistics, output_file1,output_file2):
170     """save the generated itemsets sorted by support to txt """
171     with open(output_file1, "w") as file:
172         for item, support in sorted(items, key=lambda x: x[1], reverse=True):
173             file.write("%.1f \t{%s}\n" % (support * 100, ", ".join(item)))
174     with open(output_file2, 'w') as file:
175         file.write(str(total) + '\n')
176         for sublist in statistics:
177             file.write('\t'.join(map(str, sublist)) + '\n')
178
```

   Instead of printing the result, I use this function to save the result to the file.

   In main:

```
223     input_filename = options.input.split("/")[-1].split(".")[0]
224     output_file1 = f"../OutputFile/step2_task{task:1d}_{input_filename}_{options.minS:.{str(options.minS)[::-1].find('.')}f}_result1.txt"
225     output_file2 = f"../OutputFile/step2_task{task:1d}_{input_filename}_{options.minS:.{str(options.minS)[::-1].find('.')}f}_result2.txt"
```

```
228     #printResults(items)
229     if(task==1):
230         saveResultToFileTask1(items,total,statistics, output_file1,output_file2)
```

   Set the output file name and call the "saveResultToFileTask1" function.

- Task2

1. Modify “runApriori”: (line 108~130)

```
108     else:
109         while currentLSet != set([]):
110             """Modify: closed"""
111             closed_tmp = currentLSet.copy()
112 
113             largeSet[k - 1] = currentLSet
114             currentLSet = joinSet(currentLSet, k)
115             currentCSet= returnItemsWithMinSupport(
116                 currentLSet, transactionList, minSupport, freqSet
117             )
118 
119             """Modify: check whether it is closed and save"""
120             for C_item in currentCSet:
121                 itemSupportDict[C_item] = getSupport(C_item)
122                 to_remove = set()
123                 for L_item in closed_tmp:
124                     if(L_item.issubset(C_item) and itemSupportDict[L_item] ==itemSupportDict[C_item]):
125                         to_remove.add(L_item)
126                 closed_tmp -= to_remove
127             for item in closed_tmp:
128                 closed.append((tuple(item), itemSupportDict[item]))
129             currentLSet = currentCSet
130             k = k + 1
```

Firstly, in line 111 save the unmodified currentLSet to “closed_tmp”.

In the k iteration, we must examine whether the (k-1)-frequent itemset is closed. Therefore, after generating the currentCSet, we filter the itemset in closed_tmp that none of its immediate supersets has the same support as its. (line 120 ~ 126).

Then we append the (k-1)-closed frequent itemset to “closed”. (line 127~128)

2. Save result2 to file: (line 179~185, 226, 229~232)

In saveResultToFileTask2:

```
179 """Modify: save task2 file"""
180 def saveResultToFileTask2(closed,output_file3):
181     """save the generated itemsets sorted by support to txt """
182     with open(output_file3, "w") as file:
183         file.write(str(len(closed)) + '\n')
184         for item, support in sorted(closed, key=lambda x: x[1], reverse=True):
185             file.write("%.1f \t{%s}\n" % (support * 100, ", ".join(item)))
```

I use this function to save the result to the file.

In main:

```
226    output_file3 = f"../OutputFile/step2_task{task:1d}_{input_filename}_{options.minS:.{str(options.minS)[::-1].find('.')}f}_result1.txt"
```

```
229    if(task==1):
230        saveResultToFileTask1(items,total,statistics, output_file1,output_file2)
231    else:
232        saveResultToFileTask2(closed, output_file3)
```

Set the output file name and call the “saveResultToFileTask2” function.

- Others

1. Save time: (line 188, 233~237)

```
187 if __name__ == "__main__":
188     start_time = time.time()
```

I save the start time at the very beginning.

```
229    if(task==1):
230        saveResultToFileTask1(items,total,statistics, output_file1,output_file2)
231    else:
232        saveResultToFileTask2(closed, output_file3)
233    end_time = time.time()
234    total_elapsed_time = end_time - start_time
```

The end-time is the time after the whole task, including saving the result to file.

```
236    with open("../OutputFile/time.txt", "a") as time_file:
237        time_file.write(f"{total_elapsed_time:.2f}\tstep2\ttask{task:1d}\t{input_filename}\t{options.minS:.{str(options.minS)[::-1].find('.')}f}\n")
```

Then, I save the computation time to the “time.txt”

ScreenShot of the computation time

I save the computation time in “time.txt”, and here’s the result:

- Task1 (the first column is the computation time (sec))

OutputFile > time.txt

```
1   1.80    step2   task1   A   0.01
2   4.36    step2   task1   A   0.005
3   85.74   step2   task1   A   0.002
4   629.26  step2   task1   B   0.005
5   2345.22 step2   task1   B   0.002
6   4346.17 step2   task1   B   0.0015
7   483.32  step2   task1   C   0.03
8   1350.68 step2   task1   C   0.02
9   4194.27 step2   task1   C   0.01
```

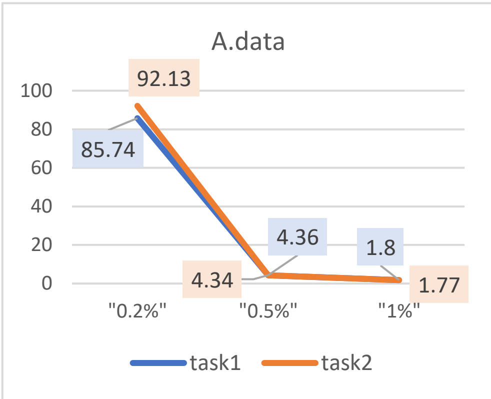
- Task2 (the first column is the computation time (sec))

```
OutputFile > time.txt
1.77     step2   task2   A   0.01
4.34     step2   task2   A   0.005
92.13    step2   task2   A   0.002
624.99   step2   task2   B   0.005
2293.94  step2   task2   B   0.002
4209.43  step2   task2   B   0.0015
489.84   step2   task2   C   0.03
1304.48  step2   task2   C   0.02
4289.57  step2   task2   C   0.01
```

- Compare the computation time

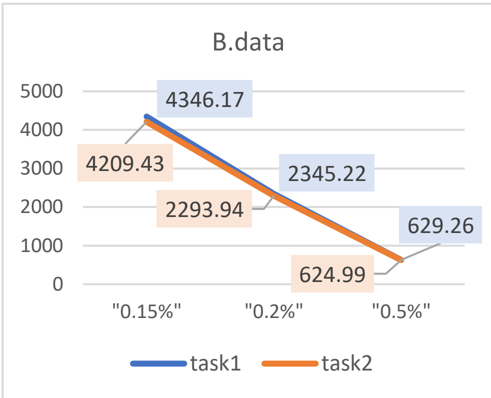
(1) A.data (1000 transactions)

minS	task1	task2	Ratio
0.2%	85.74	92.13	107.4528
0.5%	4.36	4.34	99.54128
1%	1.8	1.77	98.33333



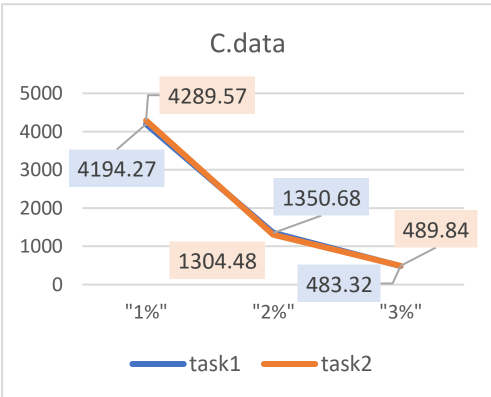
(2) B.data (100000 trnsactions)

minS	task1	task2	Ratio
0.15%	4346.17	4209.43	96.85378
0.2%	2345.22	2293.94	97.81342
0.5%	629.26	624.99	99.32143



(3) C.data (1000000 transactions)

minS	task1	task2	Ratio
1%	4194.27	4289.57	102.2721
2%	1350.68	1304.48	96.5795
3%	483.32	489.84	101.349



The restrictions

- The apriori algorithm is efficient for dealing with small datasets, such as A.data, which only has 1000 transactions. It can finish the task with 1% min support in 1.8 seconds. However, as the dataset becomes larger, such as the C.data, which has 1000000 transactions, it took 4194.27 seconds to mine the frequent itemsets with 1% min support. In this case, the data increases 1000 times, the computation time becomes 2330 times.
- The apriori algorithm is less efficient when the min support threshold is small. For example, when dealing with C.data, which has 1000000 transactions, it took 489.32 seconds to find out the frequent itemsets with min support of 3%, however, it took 4194.27 seconds to mine the frequent itemsets with 1% min support.

Problems encountered in mining:

- Since the apriori algorithm is less efficient when dealing with large datasets, or small min support thresholds, it took pretty much time to finish all the tasks.

(3) StepIII

Descriptions of mining algorithm:

- Algorithm:

  Eclat, Equivalence Class Clustering and Bottom-Up Lattice Traversal.

  It is a non-candidate base mining algorithm, using a vertical data format and a depth-first traversal of a lattice structure to directly find frequent itemsets without explicitly generating and testing candidates.
- Relevant references

  http://adrem.uantwerpen.be/~goethals/software/files/eclat.py

- Program flow

1. Data preparation and vertical data representation:

```
"""Modify: read .data file
        get the # of transactions"""
data = {}
f = open(options.input, 'r')
for row in f:
    trans += 1
    for item in row.split()[3:]:
        if item not in data:
            data[item] = set()
        data[item].add(trans)
f.close()
```

Read the data from the given .data file.

Then, transform the transaction list into a vertical format where each item maintains a list of transaction IDs in which it appears.

Example:

Original datasets:

Transaction ID	Items
1	A,B
2	A,C,D
3	B,D
4	A,B,C,E

Vertical format

Item	Transaction ID
A	1,2,4
B	1,3,4
C	2,4
D	2,3
E	4

2. Initialization:

```
    eclat([], sorted(data.items(), key=lambda item: len(item[1]), reverse=True))
```

Start with the vertical format of the initial transaction list and keep track of frequent itemsets and their support counts.

3. Eclat Algorithm Iteration:

The Eclat algorithm operates in a recursive manner to find frequent itemsets.

```python
def eclat(prefix, items):
    # for each item in the list
    while items:
        # create Equivalence Class itids
        i, itids = items.pop()
        isupp = len(itids)
        # check if the item support >= minimum support threshold
        if isupp >= minsup:
            # if so, the item is a frequent itemset.
            frequent_itemset = sorted(prefix + [i])
            frequent_itemsets.append((frequent_itemset, isupp))
            # find more frequent itemsets that contain the current item.
            # by combining current item and other items in the EC.
            suffix = []
            for j, ojtids in items:
                jtids = itids & ojtids
                if len(jtids) >= minsup:
                    suffix.append((j, jtids))
            # Recursively call eclat() on the transactions of the EC
            # EC: the remaining transactions after removing the current item.
            eclat(prefix + [i], sorted(suffix, key=lambda item: len(item[1]), reverse=True))
```

For each item in the list of unique items:

Create an Equivalence Class for the item, which is a set of transactions containing the item.

If the support count of this item is greater than or equal to the minimum support threshold:

The item itself is a frequent itemset.

Form combinations with the current item and other items in the EC to find more frequent itemsets that contain the current item.

Recursively call the Eclat algorithm on the transactions of the remaining transactions after removing the current item.

Differences/Improvements in your algorithm

- Differences/improvements of Eclat compared to Apriori.

1. Eclat uses depth-first search, while Apriori uses breath-first search. Therefore, Eclat is more efficient since it does not involve the repeated scanning of the data to compute the individual support values.
2. Eclat uses a vertical data structure, which is more memory-efficient

compared to the horizontal data structure used by Apriori.

3. Instead of generating a large number of candidates like Apriori, which can be computationally expensive. Eclat doesn't generate candidate itemsets explicitly. It leverages the vertical data structure to find intersections of itemsets more efficiently.

- Modifications made on the original algorithm.

1. Parse flag: (line 46~60)

In main:

```
46     """Modify: Add command options"""
47     optparser = OptionParser()
48     optparser.add_option(
49         "-f", "--inputFile", dest="input", help="filename containing csv", default=None
50     )
51     optparser.add_option(
52         "-s",
53         "--minSupport",
54         dest="minS",
55         help="minimum support value",
56         default=0.1,
57         type="float",
58     )
59
60     (options, args) = optparser.parse_args()
```

Just like what we do in Step II, I use the OptionParser to parse the file path and min support ratio.

2. Modify to read IBMGenerator's data: (line 72)

In main:

```
66         """Modify: read .data file
67                 get the # of transactions"""
68         data = {}
69         f = open(options.input, 'r')
70         for row in f:
71             trans += 1
72             for item in row.split()[3:]:
73                 if item not in data:
74                     data[item] = set()
75                 data[item].add(trans)
76         f.close()
```

The data generate by IBMGenerator contains SID, TID, NITEMS, and ITEMSET, but we only need ITEMSET, therefore, in line 72, I weed out the first 3 columns.

3. Modify min support count to min support ratio: (line71, 83)

   In main:

```
69      f = open(options.input, 'r')
70      for row in f:
71          trans += 1
```

```
82  """Modify: input minS ratio """
83  minsup = options.minS * trans
```

   The origin code parses in the min support count, however, in this task, we parse in the min support ratio. Therefore, when reading the file, I count the number of transactions (line 71). Then, I multiplied the input min support ratio by the number of transactions to get the min support count.(line 83)

4. Modify "eclat" to save the frequent itemsets data: (line 5, 17~18)

```
5   """Modify: global var to save the frequent itemsets"""
6   frequent_itemsets = []
7
8   def eclat(prefix, items):
9       # for each item in the list
10      while items:
11          # create Equivalence Class itids
12          i, itids = items.pop()
13          isupp = len(itids)
14          # check if the item support >= minimum support threshold
15          if isupp >= minsup:
16              # if so, the item is a frequent itemset.
17              frequent_itemset = sorted(prefix + [i])
18              frequent_itemsets.append((frequent_itemset, isupp))
```

   The original code just prints the frequent itemset on screen, however, in this homework, we need to save the data to a file. Therefore, I added a global variable "frequent_itemsets" to save the data.(line 5) Each time when a new frequent itemset is generated, I append it to the list.(line 18)

5. Save results: (line 36~41, 85~87, 90)

   In saveResultToFileTask1:

```python
"""Modify: save task1 file"""
def saveResultToFileTask1(items, output_file1,trans):
    """save the generated itemsets sorted by support to txt """
    with open(output_file1, "w") as file:
        for item, support in sorted(items, key=lambda x: x[1], reverse=True):
            file.write("%.1f \t{%s}\n" % (support*100/trans, ", ".join(item)))
```

Instead of printing the result, I use this function to save the result to the file.

In main:

```python
"""Modify: write result to file"""
input_filename = options.input.split("/")[-1].split(".")[0]
output_file1 = f"../OutputFile/step3_task1_{input_filename}_{options.minS:.{str(options.minS)[::-1].find('.')}f}
```

```python
90      saveResultToFileTask1(frequent_itemsets, output_file1,trans)
```

Set the output file name and call the “saveResultToFileTask1” function.

6. Save time: (line 45,89~96)

```python
44  if __name__ == "__main__":
45      start_time = time.time()
```

I save the start time at the very beginning.

```python
89      eclat([], sorted(data.items(), key=lambda item: len(item[1]), reverse=True))
90      saveResultToFileTask1(frequent_itemsets, output_file1,trans)
91
92      end_time = time.time()
93      total_elapsed_time = end_time - start_time
```

The end-time is the time after the whole task, including saving the result to file.

```python
95  with open("../OutputFile/time.txt", "a") as time_file:
96      time_file.write(f"{total_elapsed_time:.2f}\tstep3\ttask1\t{input_filename}\t{options.minS:.{str(options.min
```

Then, I save the computation time to the “time.txt”

Computation time

- Paste the screenshot of the computation time

  I save the computation time (sec) in “time.txt”, and here’s the result:

OutputFile > time.txt

```
0.02    step3   task1   A   0.01
0.03    step3   task1   A   0.005
0.10    step3   task1   A   0.002
2.28    step3   task1   B   0.005
2.75    step3   task1   B   0.002
2.93    step3   task1   B   0.0015
6.02    step3   task1   C   0.03
9.92    step3   task1   C   0.02
18.85   step3   task1   C   0.01
```

- Speed up

(1) A.data (1000 transactions)

Min support	Apriori	Eclat	Speed up
0.2%	85.74	0.1	99.88337
0.5%	4.36	0.03	99.31193
1%	1.8	0.02	98.88889

(2) B.data (100000 transactions)

Min Support	Apriori	Eclat	Speed up
0.15%	4346.17	2.93	99.93258
0.2%	2345.22	2.75	99.88274
0.5%	629.26	2.28	99.63767

(3) C.data (1000000 transactions)

Min Support	Apriori	Eclat	Speed up
1%	4194.27	18.85	99.55058
2%	1350.68	9.92	99.26556
3%	483.32	6.02	98.75445

The Eclat speeds up 99% of the computation time in almost every case.

Discuss the scalability of your algorithm in terms of the size of the dataset

As the result shown above, Eclat has good performance in every setting of the dataset and minimal support threshold in this homework. However, I have done some research online, and some say that the Eclat algorithm can only work on small or medium datasets, like those in this homework. If the transaction ID list is too large, the Eclat algorithm may run out of memory.