Data Science Assignment #1

Find association rules using the Apriori algorithm

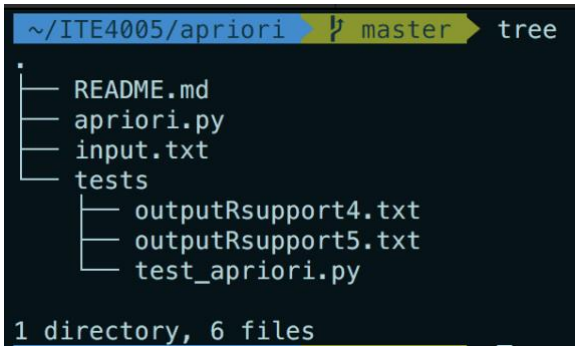
2015004857 이영수

1) Summary of algorithm

1. Build sparse tree
2. Run apriori algorithm
   i. Set itemsets to all possible subset whose length is 1
   ii. Calculate supports of itemsets
   iii. Append itemsets to result which is satisfying minimum support
   iv. Generate candidates by self-joining frequent items on previous step
   v. If there are no candidates, go to step 3
   vi. Set itemsets to candidates and repeat step 3-ii.
3. Create association rules from frequent patterns
4. Print rules

2) Code Description

Codes are written by Python3.



Project consists of python file that implements apriori, input file, and test files of apriori. *`test_apriori`* is file that asserts output value of apriori.py and the answer set. There are two answer set, one is result when minimum support is 4, and the other is 5.

aprioiri.py

There is one global function, named `itemset_hash`, that gets itemset as param and return string value for hashing. And there is one class, Apriori, that implements procedure of apriori algorithm.

```
1   import sys
2   from decimal import Decimal, ROUND_HALF_UP
3
4   def itemset_hash(itemset) :
5       return ','.join(str(x) for x in itemset)
6
7   class Apriori:
8
9       # 2d list of item (list of transactions)
10      transactions = []
11
12      # Set of transaction items
13      item_list = set()
14
15      # Matrix consisted by 0 or 1
16      sparse_matrix = []
17
```

`init` function of Apriori class get minimum support, input file, output file, and then run procedures.

```
18      def __init__(self, minimum_support, input_file, output_file):
19          """ Initialize apriori
20
21          :param minimum_support: ex) 5 -> 5%
22          :param input_file: Opened file object to read
23          :param output_file: Opened file object to write
24          """
25          self.minimum_support = minimum_support
26          self.transactions = []
27          self.item_list = set()
28          self.sparse_matrix = []
29
30          for line in input_file.readlines() :
31              if line[-1] == '\n': line = line[0:-1]
32              t = line.split('\t')
33
34              self.transactions.append(t)
35              self.item_list |= set(t) # Add new item ids
36
37          self.item_list = list(self.item_list) # Fix order
```

After initializing variables, each line of file is appended to *transaction* variable, and *item_list* variable consists all item in transaction.

Next, building sparse matrix is processed. To optimize space of array, sequential index is set to every item, and matrix consists new index.

```
40          #####
41          # Build sparse matrix
42          #####
43          id_of_item = dict(zip(self.item_list, range(0, len(self.item_list))))
44
45          for transaction in self.transactions:
46              self.sparse_matrix.append([0] * len(self.item_list))
47
48              for item in transaction:
49                  self.sparse_matrix[-1][id_of_item[item]] = 1
```

Then, get all itemsets and supports with apriori algorithm. After getting this, generate all association rules by itemsets.

```
51          #####
52          # Get association rules
53          #####
54          rules = self.all_association_rules(
55              *self.get_itemsets_and_supports()
56          )
```

Finally, save these association rules to file. Each rule is printed with required format, and support and confidence are rounded with half-up method.

```
58          #####
59          # Save association rules
60          #####
61          for itemset, ass_itemset, sup, conf in rules :
62              sup2 = sup / len(self.transactions) * 100
63              conf2 = conf * 100
64
65              output_file.write("%s\t%s\t%.2f\t%.2f\n" % (
66                  self.pretty_itemset(itemset),
67                  self.pretty_itemset(ass_itemset),
68                  Decimal(sup2 * 100).quantize(0, ROUND_HALF_UP) / 100,
69                  Decimal(conf2 * 100).quantize(0, ROUND_HALF_UP) / 100,
70              ))
71
72          print("%d2 rules are created" % len(rules))
73
```

Next function is `get_itemsets_and_suuports`, which is core part of apriori algorithm. At first step, set initial itemsets as all item list whose length is 1. And then calculate supports of itemsets, filtering frequent itemsets, get candidates, set itemsets to candidates, and repeating these steps until length of candidate becomes 0.

```
75      def get_itemsets_and_supports(self):
76          """ Get frequent itemsets by apriori algorithm
77          """
78          frequent_itemsets = []
79          frequent_supports = []
80
81          itemsets = [[i] for i in range(0, len(self.item_list))]
82          k = 0
83          while True :
84              supports = self._calc_supports(itemsets)
85              cur_frequent_itemsets, cur_frequent_supports = self._get_frequent_itemsets_and_supports(itemsets, supports)
86
87              frequent_itemsets += cur_frequent_itemsets
88              frequent_supports += cur_frequent_supports
89
90              candidates = self._get_candidates(cur_frequent_itemsets, k+1)
91
92              #self._print_itemsets(itemsets, supports)
93
94              if len(candidates) == 0 :
95                  break
96              else :
97                  itemsets = candidates
98                  k += 1
99
100         return frequent_itemsets, frequent_supports
```

Next function is `all_association_rules` that is called in `__init__` function. It generates association rules from frequent itemsets and supports.

```
102     def all_association_rules(self, frequent_itemsets, supports) :
103         """
104         Get all association rules from itemsets
105         :param frequent_itemsets: List of itemsets
106         :return: List of tuple(itemset, associative_itemset, support, confidence)
107         """
108         support_table = {}
109         for i in range(0, len(frequent_itemsets)):
110             support_table[itemset_hash(frequent_itemsets[i])] = supports[i]
111
112         ret = []
113
```

(rest of function is in next page)

To reduce time complexity of this function, make support table whose key is hash of itemset, so searching time for support of each itemset is O(1).

To generate powerset of itemset, it uses recursive function. Flag[i] means current powerset contain element in index i. At terminal iteration, set p_set(itemset) by positive flag values, and set q_set(associative itemset) by negative flag values. Then calculate support and confidence using support table, and put values to *ret* variable.

```
114         def recurs(whole_itemset):
115             flag = [0] * len(whole_itemset)
116 
117             def powerset(depth):
118                 if len(whole_itemset) <= 1:
119                     return
120 
121                 if depth == len(whole_itemset):
122                     p_set = []
123                     q_set = []
124 
125                     for i in range(0, depth):
126                         if flag[i]: p_set.append(whole_itemset[i])
127                         else:       q_set.append(whole_itemset[i])
128 
129                     if len(p_set) == 0 or len(q_set) == 0:
130                         return
131 
132                     sup = support_table[itemset_hash(whole_itemset)]
133                     conf = sup / support_table[itemset_hash(p_set)]
134 
135                     ret.append((p_set, q_set, sup, conf))
136                     return
137 
138                 flag[depth] = 1
139                 powerset(depth + 1)
140 
141                 flag[depth] = 0
142                 powerset(depth + 1)
143 
144             powerset(0)
```

`recurs` function is called for each frequent itemset, and finally return `ret` value that contains all *association rules*.

```
146         for itemset in frequent_itemsets:
147             recurs(itemset)
148 
149         return ret
150 
```

`_calc_supports` function is used in `get_itemsets_and_supports` function, that calculate supports by itemsets using sparse matrix.

```
151     def _calc_supports(self, itemsets):
152         """ Calculate supports by itemsets
153 
154         :param itemsets: List of itemset
155         :return: List or support value of each itemset
156         """
157         supports = [0] * len(itemsets)
158 
159         for row in self.sparse_matrix:
160             for no, itemset in enumerate(itemsets):
161                 exists = True
162                 for item_id in itemset:
163                     if not row[item_id]:
164                         exists = False
165                         break
166 
167                 if exists:
168                     supports[no] += 1
169 
170         return supports
```

`_get_frequent_itemsets_and_supports` function is also used in `get_itemsets_and_supports` function, that only returns itemsets which satisfy minimum support.

```
172     def _get_frequent_itemsets_and_supports(self, itemsets, supports):
173         """ Get frequent itemsets by condition `minimum_support`
174 
175         :param itemsets: List of itemset
176         :param supports: List or support value returned from `_calc_supports`
177         :return: Tuple of (List of itemset) and (List of support)
178         """
179         ret_itemsets = []
180         ret_supports = []
181 
182         for i, itemset in enumerate(itemsets):
183             if self._satisfying_support(supports[i]):
184                 ret_itemsets.append(itemset)
185                 ret_supports.append(supports[i])
186 
187         return (ret_itemsets, ret_supports)
188 
```

`_satisfying_support` function simply returns whether support is larger than minimum_support in class.

```
216     def _satisfying_support(self, support):
217         """ Return whether this value satisfying condition
218 
219         :param support: Number
220         :return: Boolean
221         """
222         return (support / len(self.transactions)) >= self.minimum_support
223 
```

`_get_candidates` function is used in `get_itemsets_and_supports` function, that returns candidates whose length equals with "length of itemsets + 1". It uses apriori algorithm to generate candidates.

```
189     def _get_candidates(self, itemsets, k):
190         """ Get candidates on next step
191 
192         :param itemsets: List of itemset
193         :param k: Step number
194         :return: List of itemset
195         """
196         candidates = []
197         for i in range(0, len(itemsets)):
198             for j in range(i + 1, len(itemsets)):
199                 itemset1 = itemsets[i]
200                 itemset2 = itemsets[j]
201 
202                 valid = True
203                 for l in range(0, k - 1):
204                     if itemset1[l] != itemset2[l]:
205                         valid = False
206                         break
207 
208                 if not itemset1[k - 1] < itemset2[k - 1]:
209                     valid = False
210 
211                 if valid:
212                     candidates.append(sorted(set(itemset1) | set(itemset2)))
213 
214         return candidates
```

`_print_itemsets` and `pretty_itemset` functions are used for debugging or printing output.

```
224     def _print_itemsets(self, itemsets, supports):
225         """ Print itemsets and supports for debug
226         """
227         print("----------------")
228         for i, itemset in enumerate(itemsets):
229             print("%s: %.2f <%s>" % (
230                 self.pretty_itemset(itemset),
231                 supports[i] / len(self.transactions) * 100,
232                 'Yes' if self._satisfying_support(supports[i]) else 'No',
233             ))
234 
235         print("----------------")
236 
237     def pretty_itemset(self, itemset):
238         """ Get string version of itemset like {0,1,4}
239         """
240         return "{%s}" % ','.join([str(i) for i in
241             sorted([int(self.item_list[x]) for x in itemset])
242         ])
```

Lastly, create Apriori class instance if program is called directly. Arguments are parsed from `sys.argv`.

```
245 if __name__ == '__main__':
246     if len(sys.argv) != 4:
247         print("Usage: python apriori.py <minimum_support> <input_file> <output_file>")
248         sys.exit(-1)
249 
250     Apriori(
251         int(sys.argv[1]) / 100,
252         open(sys.argv[2], 'r'),
253         open(sys.argv[3], 'w'),
254     )
255 
```

3) Instruction for compiling

Codes are written by python3, so please not to run with python2.

```
$ python3 apriori.py 5 input.txt output5.txt
```

You can also run tests by below instruction. (pytest is required)