CS145 Howework 5

Important Note: HW4 is due on **11:59 PM PT, Dec 4 (Friday, Week 9)**. Please submit through GradeScope.

Print Out Your Name and UID

Name: Kevin Li, UID: 405200619

Before You Start

You need to first create HW5 conda environment by the given `cs145hw5.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw5.yml
conda activate hw4
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw5.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as some important hyperparameters) that you are allowed to edit (between STRART/END YOUR CODE HERE), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

In [1]:
```
import numpy as np
import pandas as pd
import sys
import random
import math
import matplotlib.pyplot as plt
from graphviz import Digraph
from IPython.display import Image
from scipy.stats import multivariate_normal
%load_ext autoreload
%autoreload 2
```

If you can successfully run the code above, there will be no problem for environment setting.

1. Frequent Pattern Mining for Set Data (25 pts)

Table 1

TID	Items
1	b,c,j
2	a,b,d
3	a,c
4	b,d
5	a,b,c,e
6	b,c,k
7	a,c
8	a,b,e,i
9	b,d
10	a,b,c,d

Given a transaction database shown in Table 1, answer the following questions. Let the parameter `min_support` be 2.

Questions

1.1 Apriori Algorothm (16 pts) .

Note: This is a "question-answer" style problem. You do not need to code anything and you are required to calculate by hand (with a scientific calculator). Find all the frequent patterns using Apriori Algorithm.

a. C_1
b. L_1
c. C_2
d. L_2

e. C_3
f. L_3
g. C_4
h. L_4

C1
{'a',}: 6
{'b',}: 8
{'c',}: 6
{'d',}: 4
{'e',}: 2
{'i',}: 1
{'j',}: 1
{'k',}: 1

L1

{'a',}: 6
{'b',}: 8
{'c',}: 6
{'d',}: 4
{'e',}: 2

==========

C2

{'a', 'b'}: 4
{'a', 'c'}: 4
{'a', 'd'}: 2
{'a', 'e'}: 2
{'b', 'c'}: 4
{'b', 'd'}: 4
{'b', 'e'}: 2
{'c', 'd'}: 1
{'c', 'e'}: 1
{'d', 'e'}: 0

L2

{'a', 'b'}: 4
{'a', 'c'}: 4
{'a', 'd'}: 2
{'a', 'e'}: 2
{'b', 'c'}: 4
{'b', 'd'}: 4
{'b', 'e'}: 2

==========

C3

{'a', 'b', 'c'}: 2
{'a', 'b', 'd'}: 2
{'a', 'b', 'e'}: 2

L3

{'a', 'b', 'c'}: 2
{'a', 'b', 'd'}: 2
{'a', 'b', 'e'}: 2

==========

C4 is empty

L4 is empty

==========

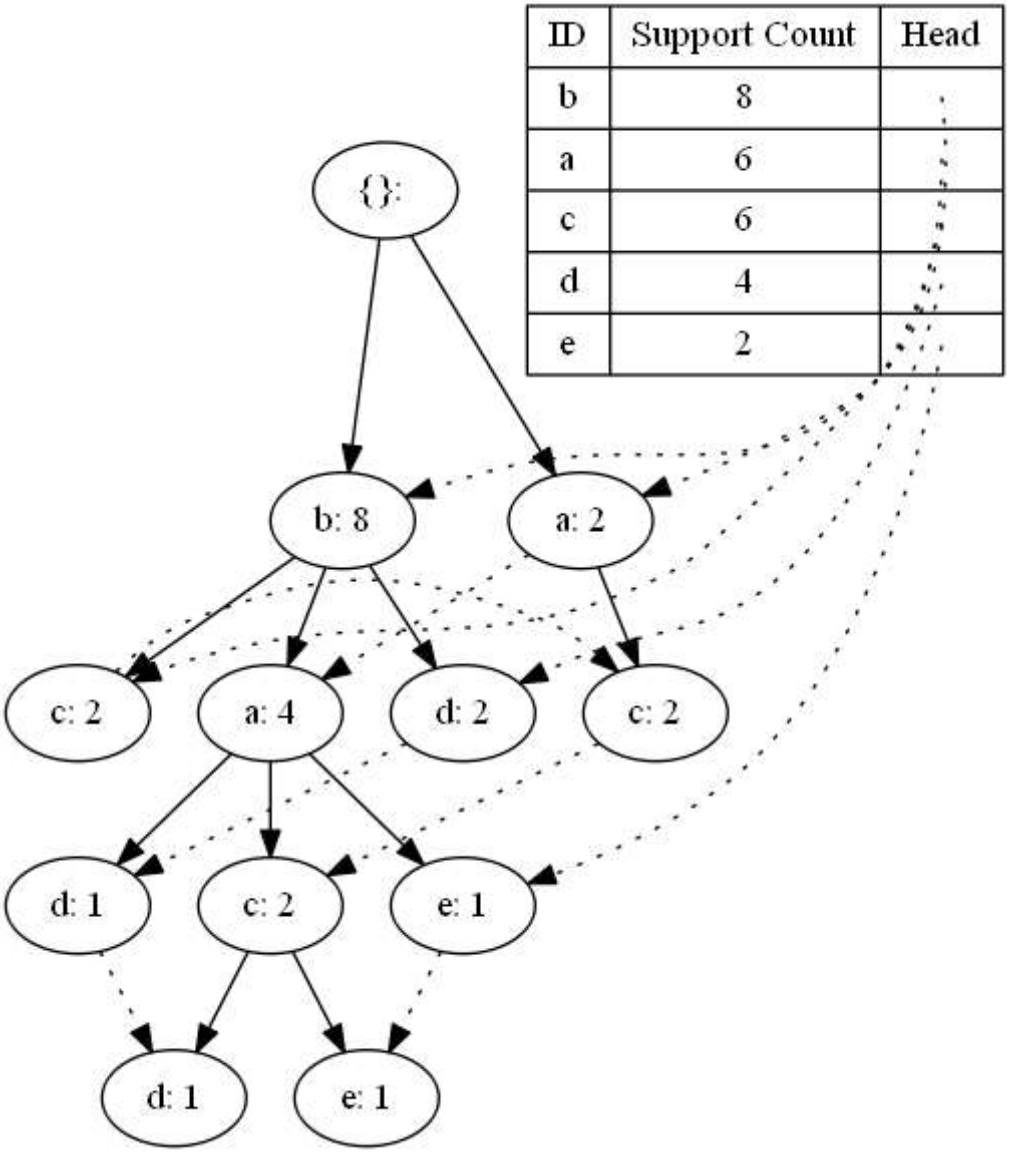
1.2 FP-tree (9 pts)

(a)Construct the FP-tree of the table.
(b) For the item d, show its conditional pattern base (projected database) and conditional FP-tree You may use Package `graphviz` to generate graph (https://graphviz.readthedocs.io/en/stable/manual.html (https://graphviz.readthedocs.io/en/stable/manual.html)) (Bonus point: 5pts) or draw by hand.

(c) Find frequent patterns based on d's conditional FP-tree

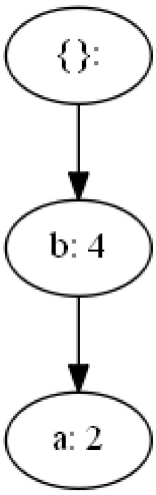
In [25]:
```
import numpy as np
from graphviz import Digraph
from collections import defaultdict, deque

min_sup = 2
db = [
    ['b', 'c', 'j'],
    ['a', 'b', 'd'],
    ['a', 'c'],
    ['b', 'd'],
    ['a', 'b', 'c', 'e'],
    ['b', 'c', 'k'],
    ['a', 'c'],
    ['a', 'b', 'e', 'i'],
    ['b', 'd'],
    ['a', 'b', 'c', 'd']
]

db_items = np.array(list(''.join(''.join(t) for t in db)))
items, counts = np.unique(db_items, return_counts=True)
item_data = sorted(zip(list(-counts), list(items)))
_, item_order = zip(*item_data)
frequents = [(item, -count) for count, item in item_data if -count >= min_sup]
frequent_names, frequent_counts = list(zip(*frequents))

class Node:
    names = defaultdict(int)
    def __init__(self, name):
        self.name = name
        self.value = 0
        self.children = {}
        self.hidden_name = name + str(Node.names[name])
        Node.names[name] += 1

    def add(self, item):
        if item in self.children:
            child = self.children[item]
        else:
            child = Node(item)
            self.children[item] = child
        child.value += 1
        return child

root = Node('{}')
root.value = ''
for tx in db:
    n = root
    for item in frequent_names:
        if item in tx:
            n = n.add(item)

dot = Digraph()
dot.node(root.hidden_name, f'{root.name}: {root.value}')

frequent_nodes = defaultdict(list)
fringe = deque([root])
```

```
while fringe:
    to_expand = fringe.popleft()
    if to_expand.name in frequent_names:
        frequent_nodes[to_expand.name].append(to_expand.hidden_name)

    children = list(to_expand.children.values())
    fringe.extend(children)
    for child in children:
        dot.node(child.hidden_name, f'{child.name}: {child.value}')
        dot.edge(to_expand.hidden_name, child.hidden_name)

table_label = '{ID|' + '|'.join(frequent_names) + '}|' + \
    '{Support Count|' + '|'.join(map(str, frequent_counts)) + '}|' + \
    '{Head|<' + '>|<'.join(frequent_names) + '>}'
dot.node('table', label=table_label, shape='record')

dot.attr('edge', style='dotted', constraint='false')
between_edges = [[[item[i], item[i+1]] for i in range(len(item) - 1)] for item ir
dot.edges([e for item_edges in between_edges for e in item_edges])
dot.attr('edge', tailclip='false')
head_edges = [[f'table:<{item}>:c', nodes[0]] for item, nodes in frequent_nodes.i
dot.edges(head_edges)

with open('graph1.dot', 'w') as f:
    f.write(dot.source)

dot
```



In [26]:

```
dot=Digraph()
dot.node('0','{}:')
dot.node('1','b: 4')
dot.node('2','a: 2')
dot.edges([['0', '1'],['1', '2']])

with open('graph2.dot', 'w') as f:
    f.write(dot.source)

dot
```



For part a, I wrote code to generate the fp-tree and header table based on setting min_support and db at the beginning

for part b, I got the conditional pattern base by reading the fp-tree and moving up from each "d" node.

2. Apriori for Yelp (50 pts)

In `apriori.py` , fill the missing lines. The parameters are set as `min_suppport=50` and `min_conf` = 0.25, and `ignore_one_iter_set=True` . Use the Yelp data `yelp.csv` and `id_nams.csv` , and run the following cell and report the frequent patterns and rules associated with it.

In [19]:
```
#No need to modify
from hw5code.apriori import *
input_file = read_data('./data/yelp.csv')
min_support = 50
min_conf = 0.25
items, rules = run_apriori(input_file, min_support, min_conf)
name_map = read_name_map('./data/id_name.csv')
print_items_rules(items, rules, ignore_one_item_set=True, name_map=name_map)
```

```
item:
"Holsteins Shakes & Buns","Wicked Spoon" 51
item:
"Wicked Spoon","Earl of Sandwich" 52
item:
"Secret Pizza","Wicked Spoon" 52
item:
"Wicked Spoon","The Cosmopolitan of Las Vegas" 54
item:
"Wicked Spoon","Mon Ami Gabi" 57
item:
"Wicked Spoon","Bacchanal Buffet" 63

------------------------ RULES:
Rule:
"Secret Pizza" "Wicked Spoon" 0.2561576354679803
Rule:
"The Cosmopolitan of Las Vegas" "Wicked Spoon" 0.27692307692307694
Rule:
"Holsteins Shakes & Buns" "Wicked Spoon" 0.3148148148148148
```

What do these results mean? Do a quick Google search and briefly interpret the patterns and rules mined from Yelp in 50 words or less.

We see that there we find patterns related Secret Pizza, Wicked Spoon, The Cosmopolitan of Las Vegas, and Holsteins Shakes and Buns.

This makes a lot of sense upon searching them up, as all except Secret Pizza are part of the restaurant collection associated with The Cosmopolitan (a hotel in Las Vegas). Secret Pizza is also closely related, being a nearby restaurant. Since we're on Yelp, it is unsurprising to see these in close proximity because it is a review site, and people are likely to encounter these together and write about them together when reviewing locations.

3. Correlation Analysis (10 pts)

Note: This is a "question-answer" style problem. You do not need to code anything and you are required to calculate by hand (with a scientific calculator).

Table 2

---	Beer	No Beer	Total

---	Beer	No Beer	Total
Nuts	150	700	850
No Nuts	350	8800	9150
Total	500	9500	10000

Table 2 shows how many transactions containing beer and/or nuts among 10000 transactions.

Answer the following questions:

3.1 Calculate `confidence` , `lift` and `all_confidence` between buying beer and buying nuts.
3.2 What are you conclusions of the relationship between buying beer and buying nuts? Justify your conclusion with the previous measurements you calculated in 3.1.

3.1

confidence(beer -> nuts) = sup_count(beer, nuts) / sup_count(beer) confidence(beer -> nuts) = 150/850 confidence(beer -> nuts) = 0.18

confidence(nuts -> beer) = sup_count(nuts, beer) / sup_count(nuts) confidence(nuts -> beer) = 150/500 confidence(nuts -> beer) = 0.30

lift(beer, nuts) = P(beer, nuts) / (P(beer) * P(nuts)) lift(beer, nuts) = (150/10000) / ((500/10000) * (850/10000))

all_confidence(beer, nuts) = min(confidence(beer -> nuts), confidence(nuts -> beer)) all_confidence(beer, nuts) = min(0.18, 0.30) all_confidence = 0.18

3.2

There exists a positive correlation between buying beer and nuts. We can see that the lift value is significantly over 1 (although there are a lot of no beer/no nut purchases which may bias it upwards) and that the confidence measures of beer -> nuts and nuts -> beer are much greater than the individual supports of nuts and beer (which are 0.05 and 0.085 respectively)

4. Sequential Pattern Mining (GSP Algorithm) (15 pts)

Note: This is a "question-answer" style problem. You do not need to code anything and you are required to calculate by hand (with a scientific calculator).

4.1 For a sequence $s =< ab(cd)(ef) >$, how many events or elements does it contain? What is the length of s? How many non-empty subsequences does s contain?
4.2 Suppose we have
$L_3 = \{< (ac)e >, < b(cd) >, < bce >, < a(cd) >, < (ab)d >, < (ab)c >\}$, as the requent 3-sequences, write down all the candidate 4-sequences C_4 with the details of the join and pruning steps.

4.1

It contains 4 events. Its length is 6. It contains 2^6-1 = 63 non-empty subsequences

4.2

JOINING CANDIDATES:

The joining table looks like this: (only the valid ones are filled in) A valid join is when dropping the first item from s1 is the same as dropping the second item from s2

	<(ac)e>	**<b(cd)>**	**<a(cd)>**	**<(ab)d>**	**<(ab)c>**
<(ac)e>					
<b(cd)>					
<a(cd)>					
<(ab)d>					
<(ab)c>		<(ab)(cd)>	<(ab)ce>		

This gives us the two candidates <(ab)(cd)> and <(ab)ce>.

PRUNING CANDIDATES:

We prune by checking if each 3-length subseq of the candidates is in L3. This holds for <(ab)(cd)> but not for <(ab)ce> since no super sequence of is in L3.

Thus, we end up with the following:

C4 = <(ab)(cd)>

5 Bonus Question (10 pts)

1.In FP-tree, what will happen if we use `ascending` instead `descending` in header table?
2.Describe CloSpan (`Mining closed sequential patterns: CloSpan (Yan, Han & Afshar @SDM'03)`). Compare with algorithms we discussed in class.

5.1

We use descending order to make the more frequent items higher and the less frequent items leaves. If we switch this order, we would build it with frequent items as the leaves and less frequent ones near the root. This would make FP-tree inefficient, since the point is that pattern bases are small and efficient because as you approach the leaves, the frequency get lower.

5.2

CloSpan is an attempt to improve PrefixSpan. To that end, it does the same general procedure of finding frequent items, projecting them onto the database and recursing on the projected database. The difference is in the pruning of the search space to seach faster and more effectively (e.g. using techniques like looking for common prefixes).

End of Homework 5 :)

After you've finished the homework, please print out the entire `ipynb` notebook and four `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Also this time remember assign the pages to the questions on GradeScope

```python
from itertools import chain, combinations,islice
from collections import defaultdict
from time import time
import pandas as pd
import operator


def run_apriori(infile, min_support, min_conf):
    """
    Run the Apriori algorithm. infile is a record iterator.
    Return:
        rtn_items: list of (set, support)
        rtn_rules: list of ((preset, postset), confidence)
    """
    one_cand_set, all_transactions = gen_one_item_cand_set(infile)

    set_count_map = defaultdict(int)  # maintains the count for each set

    one_freq_set, set_count_map = get_items_with_min_support(
        one_cand_set, all_transactions, min_support, set_count_map)

    freq_map, set_count_map = run_apriori_loops(
        one_freq_set, set_count_map, all_transactions, min_support)

    rtn_items = get_frequent_items(set_count_map, freq_map)
    rtn_rules = get_frequent_rules(set_count_map, freq_map, min_conf)

    return rtn_items, rtn_rules


def gen_one_item_cand_set(input_fileator):
    """
    Generate the 1-item candidate sets and a list of all the transactions.
    """
    all_transactions = list()
    one_cand_set = set()
    for record in input_fileator:
        transaction = frozenset(record)
        all_transactions.append(transaction)
        #========================#
        # STRART YOUR CODE HERE  #
        #========================#
        for item in transaction:
            one_cand_set.add(frozenset([item]))
        #========================#
        #   END YOUR CODE HERE   #
        #========================#
    return one_cand_set, all_transactions


def get_items_with_min_support(item_set, all_transactions, min_support,
                               set_count_map):
    """
    item_set is a set of candidate sets.
    Return a subset of the item_set
    whose elements satisfy the minimum support.
    Update set_count_map.
    """
    rtn = set()
    local_set = defaultdict(int)
```

```
61
62      for item in item_set:
63          for transaction in all_transactions:
64              if item.issubset(transaction):
65                  set_count_map[item] += 1
66                  local_set[item] += 1
67
68      #========================#
69      # STRART YOUR CODE HERE  #
70      #========================#
71      for item, count in local_set.items():
72          if count >= min_support:
73              rtn.add(item)
74      #========================#
75      #   END YOUR CODE HERE   #
76      #========================#
77
78
79
80      return rtn, set_count_map
81
82
83  def run_apriori_loops(one_cand_set, set_count_map, all_transactions,
84                        min_support):
85      """
86      Return:
87          freq_map: a dict
88              {<length_of_set_l>: <set_of_frequent_itemsets_of_length_l>}
89          set_count_map: updated set_count_map
90      """
91      freq_map = dict()
92      current_l_set = one_cand_set
93      i = 1
94      #========================#
95      # STRART YOUR CODE HERE  #
96      #========================#
97      while (current_l_set != set([])):
98          freq_map[i] = current_l_set
99          current_l_set = join_set(current_l_set, i)
100         current_c_set, set_count_map = get_items_with_min_support(current_l_set,
    all_transactions, min_support, set_count_map)
101         current_l_set = current_c_set
102
103         i += 1
104     #========================#
105     #   END YOUR CODE HERE   #
106     #========================#
107
108     return freq_map, set_count_map
109
110
111 def get_frequent_items(set_count_map, freq_map):
112     """ Return frequent items as a list. """
113     rtn_items = []
114     for key, value in freq_map.items():
115         rtn_items.extend(
116             [(tuple(item), get_support(set_count_map, item))
117              for item in value])
118     return rtn_items
119
```

```
120
121 def get_frequent_rules(set_count_map, freq_map, min_conf):
122     """ Return frequent rules as a list. """
123     rtn_rules = []
124     for key, value in islice(freq_map.items(),1,None):
125         for item in value:
126             _subsets = map(frozenset, [x for x in subsets(item)])
127             for element in _subsets:
128                 remain = item.difference(element)
129                 if len(remain) > 0:
130                #=======================#
131                # STRART YOUR CODE HERE  #
132                #=======================#
133                      confidence = set_count_map[item] / set_count_map[element]
134                #=======================#
135                #   END YOUR CODE HERE   #
136                #=======================#
137                      if confidence >= min_conf:
138                          rtn_rules.append(
139                              ((tuple(element), tuple(remain)), confidence))
140     return rtn_rules
141
142
143 def get_support(set_count_map, item):
144     """ Return the support of an item. """
145     #=======================#
146     # STRART YOUR CODE HERE  #
147     #=======================#
148     sup_item = set_count_map[item]
149     #=======================#
150     #   END YOUR CODE HERE   #
151     #=======================#
152     return sup_item
153
154
155 def join_set(s, l):
156     """
157     Join a set with itself .
158     Return a set whose elements are unions of sets in s with length==l.
159     """
160     #=======================#
161     # STRART YOUR CODE HERE  #
162     #=======================#
163     join_set=set()
164     ls = list(s)
165     for i, s1 in enumerate(ls):
166         for j in range(i+1, len(s)):
167             joined = s1.union(ls[j])
168             if len(joined) == l + 1:
169                 join_set.add(joined)
170     #=======================#
171     #   END YOUR CODE HERE   #
172     #=======================#
173     return join_set
174
175
176 def subsets(x):
177     """ Return non =-empty subsets of x. """
178     return chain(*[combinations(x, i + 1) for i, a in enumerate(x)])
179
```

```
180
181 def print_items_rules(items, rules, ignore_one_item_set=False, name_map=None):
182     for item, support in sorted(items, key=operator.itemgetter(1)):
183         if len(item) == 1 and ignore_one_item_set:
184             continue
185         print ('item: ')
186         print (convert_item_to_name(item, name_map), support)
187     print ('\n------------------------ RULES:')
188     for rule, confidence in sorted(
189             rules, key=operator.itemgetter(1)):
190         pre, post = rule
191         print ('Rule: ')
192         print( convert_item_to_name(pre, name_map), convert_item_to_name(post,
    name_map),confidence)
193
194
195 def convert_item_to_name(item, name_map):
196     """ Return the string representation of the item. """
197     if name_map:
198         return ','.join([name_map[x] for x in item])
199     else:
200         return str(item)
201
202
203 def read_data(fname):
204     """ Read from the file and yield a generator. """
205     file_iter = open(fname, 'rU')
206     for line in file_iter:
207         line = line.strip().rstrip(',')
208         record = frozenset(line.split(','))
209         yield record
210
211
212 def read_name_map(fname):
213     """ Read from the file and return a dict mapping ids to names. """
214     df = pd.read_csv(fname, sep=',\t ', header=None, names=['id', 'name'],
215                      engine='python')
216     return df.set_index('id')['name'].to_dict()
217
218
219
```