CS412: Introduction to Data Mining
Hanfei Lin
Oct. 8th, 2017

Question 1

Answers:

a. 1024 cuboids are there in the full data cube.
b. The complete cube will contain 2813 distinct aggregated (i.e., non-base) cells.
c. An iceberg cube will contain 128 distinct aggregated cells, if the condition of the iceberg cube is count > 2.
d. The closed cell with count = 3 has 7 non-star dimensions.

Explanation:

a. This is just the definition of cuboid.
   # of cuboid= 2^(# of attributes) = 2^10 = 1024
b. # of base cells = 3
   # of duplicate cells = 3*2^7
   (from this three copies, we save one and remove two. I.e. remove 2*2^7)
   # of total cells = 3*2^10
   Therefore the answer is 3*2^10-2^8-3 = 2813.
c. Considering the sharing 7 attributes c4, c5…, c10, they will form the iceberg of count = 3. Eg. (*, *, *, c4, …, c10), etc… Therefore the answer is 2^7 = 128.
d. (*, *, *, c4, c5, c6, c7, c8, c9, c10)

Question 2

Answers:

a. There are 24 cuboids in this cube.
   $T = \prod_{i=1}^{n}(L_i + 1)$ = (2 + 1) * (1 + 1) * (1 + 1) * (1 + 1) = 24
b. There are 48 cells in the cuboid (Location(city), Category, Rating, Price).
c. Now let's drill up by climbing up in the Location dimension, from City to State. There are 34 cells in the cuboid (Location(State), Category, Rating, Price).
d. Further, there are 23 cells in the cuboid (*, Category, Rating, Price).
e. The count for the cell (Location(state) = 'Illinois', *, rating = 3, Price = 'Moderate') is 2.
f. The count for the cell (Location(city) = 'Chicago', Category='food', *, *) is 2.

Code:
To solve this problem, I use the *Pandas* Python library to perform quasi-SQL manipulation over the Q2data.csv
The following are codes with comments:

```
# Question 2
import pandas as pd
import numpy as np

# Import data
df = pd.read_csv('Q2data.csv', names=['BId',
'State', 'City', 'Category', 'Price','Rating'])

# b.
dataCitCatRatPri = df.groupby(['City',
'Category', 'Price', 'Rating']).agg(['count'])
print('b. There are {} cells in in the
cuboid (Location(city), Category, Rating,
Price).'.format(len(dataCitCatRatPri.index)))
```

```
# c.
dataStaCatRatPri = df.groupby(['State',
'Category', 'Price', 'Rating']).agg(['count'])
print('c. There are {} cells in in the
cuboid (Location(State), Category, Rating,
Price).'.format(len(dataStaCatRatPri.index)))

# d.
dataCatRatPri = df.groupby(['Category', 'Price',
'Rating']).agg(['count'])
print('d. There are {} cells in in the
cuboid (*, Category, Rating,
Price).'.format(len(dataCatRatPri.index)))

# e.
dataE = df.loc[(df['State'] == 'Illinois') &
(df['Rating']==3) & (df['Price']=='moderate')]
print('e. The count for the
cell (Location(state) = ’Illinois’, *, rating =
3, Price = ’moderate’) is
{}.'.format(len(dataE.index)))

# f.
dataF = df.loc[(df['City'] == 'Chicago') &
(df['Category']=='food')]
print('f. The count for the cell (Location(city)
= ’Chicago’, Category=’food’, *,
*) is {}.'.format(len(dataF.index)))
```

Question 3

Answers:

a.

1. The number of frequent patterns is 30.
2. The number of frequent patterns with length 3 is 8.
3. The number of max patterns is 7.

b.

1. The number of frequent patterns is 55.
2. The number of frequent patterns with length 3 is 20.
3. The number of max patterns is 6.
4. The confidence measure of the association rule (C, E) -> A is 0.679
   sup(A, C, E) / sup(C, E) = 38 / 56 = 0.679
5. The confidence measure of the association rule (A, B, C) -> E is 0.742
   sup(A, B, C, E) / sup(A, B, C) = 23 / 31 = 0.742

Code:

To solve this problem, I implemented the Apriori algorithm in Python and used other customized helper function to get the answer.
The following are codes with comments:

```
# Question 3
# Implement a simple Apriori algorithm(from
pseudocode in lecture
# notes) in Python


# Import data
with open('Q3data') as data_file:
    # Save each transaction as a Python set in a
Python list called TDB.
    TDB = [set(record.split()) for record in
data_file.readlines()]
```

```
    # apriori is function that performs the
Apriori algorithm with
    # minimum support minsup.
    # returns a freqItemsetPool that contains
all FP, grouped by length,
    # organized as: FP -> support
    def apriori(minsup):
        DBSize = len(TDB)
        # k is the length for FP(number of
element in a FP)
        k = 1
        # a Python dictionary that saves: FP ->
support,
        # each freqItemset contains only FPs
with same length
        freqItemset = {}
        # a Python list that saves freqItemset
of all length
        freqItemsetPool = []

        # Find the 1-length freqItemset of FPs.
        for T in TDB:
            for item in T:
                if (item,) in freqItemset:
                    freqItemset[(item,)] += 1
                else:
                    freqItemset[(item,)] = 1

        temp = {}

        # Remove 1-length patterns with supports
that less than minsup
        for itemset in freqItemset:
            if freqItemset[itemset] >= minsup:
                temp[itemset] =
freqItemset[itemset]
        freqItemset = temp

        # Save 1-length freqItemset into
freqItemsetPool
        freqItemsetPool.append(freqItemset)

        # Perform Apriori algorithm until
freqItemset is empty
        # I.e. no larger FPs can be found.
        while freqItemset:
            # Accumulate the length of FP
            k += 1

            # Find all k-length FP candidates
from (k-1)-length FP
            candidates = set()
            freqSets = list(freqItemset.keys())
            for i in range(len(freqSets) - 1):
                for j in range(i + 1,
len(freqSets)):
                    temp = set(freqSets[i] +
freqSets[j])
                    if len(temp) == k:

candidates.add(tuple(sorted(tuple(temp))))
            candidates =
sorted(list(candidates))

            # Find k-length freqItemset of FPs.
            # Similar to the above 1-length
situation.
            freqItemset = {}
            for candidate in candidates:
                freqItemset[candidate] = 0
            for T in TDB:
                for candidate in candidates:
                    if
T.issuperset(set(candidate)):
                        freqItemset[candidate]
+= 1
            temp = {}

            # Remove 1k-length patterns with
supports that less than minsup
            for itemset in freqItemset:
                if freqItemset[itemset] >=
minsup:
                    temp[itemset] =
freqItemset[itemset]
            # Save k-length freqItemset into
freqItemsetPool
            freqItemset = temp
            if freqItemset:

freqItemsetPool.append(freqItemset)
        return freqItemsetPool


    # ThisIsNotMaxPattern inherits from
Exception, and is used to
    # jump out of multiple loops if we find out
that a is FP is
    # not max pattern.
    class ThisIsNotMaxPattern(Exception):
        pass
    # numOfMaxPatterns takes a freqItemsetPool
and returns its
    # number of max patterns
    def numOfMaxPatterns(freqItemsetPool):
        num = 0

        # enumerate all FP in the
freqItemsetPool, compare each
        # with larger FP and see if it is a sub-
pattern. If no
        # super-pattern is found, then the FP is
counted for one
        # max pattern.
        for i in range(len(freqItemsetPool) -
1):
            curItemset = freqItemsetPool[i]
            for curFP in curItemset:
                # This try is for jumping out of
multiple loops
                try:
                    for j in range(i + 1,
len(freqItemsetPool)):
                        compItemset =
freqItemsetPool[j]
                        for compFP in
compItemset:
                            if
set(curFP).issubset(set(compFP)):
                                raise
ThisIsNotMaxPattern
                    num += 1
                except ThisIsNotMaxPattern:
                    continue
        num +=
len(freqItemsetPool[len(freqItemsetPool) - 1])
        return num


    # assoConfidMeasure takes freqItemsetPool,
fp_base and fp_infer
    # and return the confidence rule from
fp_base to infer fp_infer
    def assoConfidMeasure(freqItemsetPool,
fp_base, fp_infer):
        try:
            sup_base =
freqItemsetPool[len(fp_base) - 1][fp_base]
            sup_infer =
freqItemsetPool[len(fp_infer) - 1][fp_infer]
        except KeyError:
            print('Warning: Can not find the
frequenty pattern(s)!')
            return 0
        return round(float(sup_infer) /
float(sup_base), 3)


    # a
    freqItemsetPool20 = apriori(20)
    print('Prob a')
    print('The number of frequent patterns
is {}.'.format(
        sum([len(itemset) for itemset in
freqItemsetPool20])))
    print('The number of frequent patterns with
length 3 is {}.'.format(
        len(freqItemsetPool20[2])))
    print('The number of max patterns
is {}.'.format(
        numOfMaxPatterns(freqItemsetPool20)))
    # b
    freqItemsetPool10 = apriori(10)
    print('Prob b')
    print('The number of frequent patterns is
{}.'.format(
        sum([len(itemset) for itemset in
freqItemsetPool10])))
    print('The number of frequent patterns with
length 3 is {}.'.format(
        len(freqItemsetPool10[2])))
    print('The number of max patterns
is {}.'.format(
        numOfMaxPatterns(freqItemsetPool10)))
    print('The confidence measure of the
association rule (C, E) -> A is {:.3f}'.format(
        assoConfidMeasure(freqItemsetPool10,
('C', 'E'), ('A', 'C', 'E'))))
    print('The confidence measure of the
association rule (A, B,
C) -> E is {:.3f}'.format(
        assoConfidMeasure(freqItemsetPool10,
('A', 'B', 'C'), ('A', 'B', 'C', 'E'))))
```