HW1 0416246 王彥茹

Task1.

1. what transaction you define

(板橋's temperature, north power usage)

using inner join to combine the result of two tables(temperature, power) on MySQL and export the result to a .csv file

```
select temperature , power_usage,  w_Time
from
(select w.obsTime as w_Time, date(w.obsTime) as w_date, hour(w.obsTime) as w_hour, w.value as temperature
 from weather w
 where date(w.obsTime) != '0000-00-00'
 && date(w.obsTime) >= date'2016-09-27'
 && date(w.obsTime) <= date'2017-07-03'
 && w.locationName='BANQIAO,板橋') t1

inner join

 (select date(create_date) as p_date, hour(p.create_time) as p_hour, p.north_usage as power_usage
  from taipower p
  where date(p.create_date) >= date'2016-09-27'
  && date(p.create_date) <= date'2017-07-03') t2

  on t1.w_date = t2.p_date && t1.w_hour = t2.p_hour;
```

Import the data to jupyter notebook by read_csv(), and put the data into pandas.DataFrame

In [1]:
```python
import numpy as np
import pandas as pd

file1 = 'transcation1.csv'

raw_data1 = pd.read_csv(file1,header=0, delimiter=',')

data1 = np.array(raw_data1)

transct1 = pd.DataFrame(data1[:,0:2], columns = ['temp', 'usage'])
transct1['temp'] = transct1['temp'].apply(pd.to_numeric)
transct1['usage'] = transct1['usage'].apply(pd.to_numeric)


transct1
```

Out[1]:

	temp	usage
0	26.6	841.3
1	26.0	826.4
2	25.9	789.5
3	25.8	773.6
4	26.9	778.0
5	26.6	783.1

2. what discretization method you use

a. Equal Frequency Binning

First, using numpy.linspace to create bins according the maximum and minimum of each data.(make sure that every value can has a proper bin) Then, discretize the data by numpy.digitize()

```
#Equal Frequency Binning
bins = np.linspace(20.0, 35.0, num=4)
discr_temp = np.digitize(transct1['temp'], bins, right=False)
print(discr_temp)
```

```
[2 2 2 ..., 2 2 2]
```

```
np.linspace(630.0, 1305.0, num=8, retstep=True)
```

```
(array([  630.        ,   726.42857143,   822.85714286,   919.28571429,
        1015.71428571,  1112.14285714,  1208.57142857,  1305.        ]),
 96.428571428571431)
```

```
#bins_usage = np.linspace(630.0, 1305.0, num=8) low support confidence
bins_usage = np.linspace(630.0, 1305.0, num=4)
discr_usg = np.digitize(transct1['usage'], bins_usage, right=False)
print(discr_usg)
```

```
[1 1 1 ..., 3 3 3]
```

Change the data type to string to make implementing algorithm more convenient

```
dataset = pd.DataFrame(tmp, columns = ['temp', 'usage'])
dataset['temp'] = dataset['temp'].apply(str)
dataset['usage'] = dataset['usage'].apply(str)
dataset['temp'] = "temp" + dataset['temp']
dataset['usage']= "usage" + dataset['usage']
dataset
```

	temp	usage
0	temp2	usage1
1	temp2	usage1
2	temp2	usage1
3	temp2	usage1
4	temp2	usage1
5	temp2	usage1

b. **Standardization plus Equal Frequency Binning**

Using normalization formula $Z = \frac{X - \mathbf{E}[X]}{\sigma(X)}$ to normalize the data, and then do the binning. The idea why I use normalization is from the hw0 . We were asked to do amplitude scaling to the data, it's obvious that the data's range shrink a lot after the transformation. Therefore, I think maybe it would help when finding association rules since it transform the data into a smaller range.

Do the normalization first

```
#Amplitude
mean_t = np.mean(transct1['temp'])
mean_u = np.mean(transct1['usage'])

std_t= np.std(transct1['temp'])
std_u= np.std(transct1['usage'])

trans2 = (transct1['temp'] - mean_t) / std_t
trans2_u = (transct1['usage'] - mean_u) / std_u

trans2_u
```

```
0      -0.429362
1      -0.513289
2      -0.721133
3      -0.810693
4      -0.785909
5      -0.757182
```

Then, do what have been done above to the normalized data

```
#Equal Frequency Binning
bt2 = np.linspace(-2.0, 2.5, num=3)
discr2_t = np.digitize(trans2, bt2, right=False)

bu2 = np.linspace(-1.5, 3.0, num = 4)
discr2_u = np.digitize(trans2_u, bu2, right=False)

#change the data type to string
df2 = pd.DataFrame({'temp':discr2_t, 'usage': discr2_u})
df2['temp'] = df2['temp'].apply(str)
df2['usage'] = df2['usage'].apply(str)
df2['temp'] = "T" + df2['temp']
df2['usage']= "U" + df2['usage']
df2
```

	temp	usage
0	T2	U1
1	T2	U1
2	T2	U1
3	T2	U1

3. what algorithm you use

a. **Apriori** (https://github.com/luoyetx/Apriori/blob/master/README.md)
b. **FP growth**(https://github.com/evandempsey/fp-growth)

4. what rules you discover

transaction applied **equal frequency binning** discretization method

```
from apriori import Apriori
minsup = 0.1
minconf = 0.4

ap = Apriori(dataset.as_matrix(), minsup, minconf)
ap.run()# run algorithm
ap.print_frequent_itemset()
ap.print_rule()
```

```
======================================================
Frequent itemset:
(temp1)  support = 0.378
(usage3)  support = 0.182
(temp2)  support = 0.314
(temp0)  support = 0.21
(usage2)  support = 0.396
(usage1)  support = 0.392
(temp1, usage2)  support = 0.167
(temp2, usage2)  support = 0.123
(usage1, temp0)  support = 0.114
(temp1, usage1)  support = 0.189
(temp2, usage3)  support = 0.103
======================================================
======================================================
Rules:
(usage2) ==> (temp1)  confidence = 0.421
(temp1) ==> (usage2)  confidence = 0.442
(temp0) ==> (usage1)  confidence = 0.543
(usage1) ==> (temp1)  confidence = 0.483
(temp1) ==> (usage1)  confidence = 0.501
(usage3) ==> (temp2)  confidence = 0.569
======================================================
```

Rules (Wall time: 489 ms)

temp1	**20~25 °C**	**usage2**	**about 726 ~ 822**
temp2	**25~30 °C**	**usage2**	**about 726 ~ 822**
temp0	**12.4~20 °C**	**usage1**	**about 630 ~ 726**
temp1	**20~25 °C**	**usage1**	**about 630 ~ 726**
temp2	**25~30 °C**	**usage3**	**about 822~919**

(12.4°C is the lowest temperature)

transaction applied **normalization** plus **equal frequency binning**

```
=======================================================
Frequent itemset:
(U2)  support = 0.397
(U1)  support = 0.48
(T2)  support = 0.411
(U3)  support = 0.088
(T1)  support = 0.557
(U2, T2)  support = 0.192
(U1, T1)  support = 0.327
(T2, U3)  support = 0.087
(T1, U2)  support = 0.197
(U1, T2)  support = 0.131
=======================================================
=======================================================
Rules:
(T2) ==> (U2)  confidence = 0.468
(U2) ==> (T2)  confidence = 0.485
(T1) ==> (U1)  confidence = 0.588
(U1) ==> (T1)  confidence = 0.683
(U3) ==> (T2)  confidence = 0.982
(U2) ==> (T1)  confidence = 0.497
=======================================================
```

Rules (Wall time: 263 ms)

T2	**25~35.3 °C**	**U2**	**about 918 ~ 1184**
T1	**14.5~25 °C**	**U1**	**about 651~ 918**
T2	**25~35.3 °C**	**U3**	**about 1184 ~ 1450**
T1	**14.5~25 °C**	**U2**	**about 918 ~ 1184**
T2	**25~35.3 °C**	**U1**	**about 651~ 918**

transaction applied **equal frequency binning** discretization method

```python
import pyfpgrowth
patterns = pyfpgrowth.find_frequent_patterns(dataset.as_matrix(), 2)
rules = pyfpgrowth.generate_association_rules(patterns, 0.3)

print("patterns :")
print(patterns)

print("rules : ")
print(rules)
```

```
patterns :
{('temp4', 'usage3'): 3, ('temp4', 'usage4'): 4, ('temp2', 'usage0'): 2, ('temp1', 'usage0'): 18, ('temp0', 'usage0'): 39, ('te
mp2', 'usage4'): 6, ('temp3', 'usage4'): 63, ('temp3', 'usage1'): 10, ('temp3', 'usage2'): 86, ('temp3', 'usage3'): 266, ('temp
1', 'usage3'): 79, ('temp2', 'usage3'): 459, ('temp0', 'usage2'): 387, ('temp0', 'usage1'): 507, ('temp2', 'usage1'): 381, ('te
mp2', 'usage2'): 545, ('temp1', 'usage2'): 741, ('temp1', 'usage1'): 840, ('usage1',): 1738, ('usage2',): 1759}
rules :
{('usage2',): (('temp1',), 0.4212620807276862), ('usage1',): (('temp1',), 0.48331415420023016)}
```

Rules (Wall time: 443 ms)

temp1	20~25 °C	usage2	about 726 ~ 822
temp1	20~25 °C	usage1	about 630 ~ 726

transaction applied **normalization** plus **equal frequency binning**

```
patterns :
{('T0',): 140, ('U0',): 158, ('T1', 'U0'): 138, ('U3',): 391, ('T2', 'U3'): 384, ('T2', 'U2'): 854, ('T1', 'U2'): 874, ('T2',):
1825, ('T2', 'U1'): 583, ('U1',): 2128, ('T1', 'U1'): 1453, ('T1',): 2471}
rules :
{('U0',): (('T1',), 0.8734177215189873), ('U3',): (('T2',), 0.9820971867007673), ('T2',): (('U1',), 0.31945205479452055), ('T
1',): (('U1',), 0.5880210441116956), ('U1',): (('T1',), 0.6828007518796992)}
```

Rules (Wall time: 332 ms)

T1	14.5~25 °C	U0	about 575 ~651
T2	25~35.3 °C	U3	about 1184 ~ 1450
T2	25~35.3 °C	U1	about 651~ 918
T1	14.5~25 °C	U1	about 651~ 918

5. what did you learned, and comparison between different methods

from the rules above we can notice that there are some positive correlation between temperature and power usage but the association rule can’t specify in what range of temperature associated with what range of power usage.(ex. According to the table above, T1 is associated with both U0 and U1)

run time comparison (using %%time): I didn’t use the same support threshold since the way that support is use in the algorithm is different(one by percentage, the other by count).Hence, I won’t compare the time efficiency of two algorithm implementation here although we know that fg-growth should be faster than apriori according to the lectures in class. Comparing the run time of same algorithm with different discretization method we can realize that it speeds up after doing normalization. Even so, the rules we found didn’t change dramatically which may imply that this discretization could increase the efficiency and keep some accuracy at the same time.

memory usage of the dataframe is about 69.4KB

Task2.

1. what transaction you define

(north supply, south supply, center supply, east supply)

Select south supply, north supply, center supply, east supply from power table on MySQL and export the result to a .csv file

```
select south_supply, center_supply, north_supply, east_supply
from power
```

Import the data to jupyter, and put the data into pandas.DataFrame

```
file = 'transcation2.csv'
raw_data = pd.read_csv(file,header=0, delimiter=',')
data = np.array(raw_data)

transct2 = pd.DataFrame(data, columns = ['south_supply','center_supply','north_supply','east_supply'])

print(transct2.info())
print(transct2)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6550 entries, 0 to 6549
Data columns (total 4 columns):
south_supply     6550 non-null float64
center_supply    6550 non-null float64
north_supply     6550 non-null float64
east_supply      6550 non-null float64
dtypes: float64(4)
memory usage: 204.8 KB
None
      south_supply  center_supply  north_supply  east_supply
0            839.6          733.0         648.4         14.0
1            827.3          725.4         601.3         14.0
2            821.5          759.0         500.9         13.9
3            744.2          717.1         516.1         13.9
4            746.0          691.6         530.2          8.7
5            751.7          703.0         546.3          4.7
```

2. what discretization method you use (same as task1)

a. Equal Frequency Binning

since east supply only range from 0.2 to 17.6, I set only two bins 0~10 and 10~20

```python
#Equal Frequency Binning
discrt = pd.DataFrame(columns = ['south_supply','center_supply','north_supply','east_supply'])

south_bin = np.linspace(700, 1400, num=5) #create bin by min_supply and high_supply
north_bin = np.linspace(500, 1300, num=5)
center_bin = np.linspace(400, 1200, num=5)
east_bin = np.array([10, 20])

discrt['south_supply'] = np.digitize(transct2['south_supply'].as_matrix() , south_bin, right=False)
discrt['north_supply'] = np.digitize(transct2['north_supply'].as_matrix() , north_bin, right=False)
discrt['center_supply'] = np.digitize(transct2['center_supply'].as_matrix() , center_bin, right=False)
discrt['east_supply'] = np.digitize(transct2['east_supply'].as_matrix() , east_bin, right=False)
discrt
```

	south_supply	center_supply	north_supply	east_supply
0	1	2	1	1
1	1	2	1	1
2	1	2	1	1
3	1	2	1	1
4	1	2	1	0
5	1	2	1	0
6	1	2	1	0

Change the data type to string

```python
discrt['south_supply'] = discrt['south_supply'].apply(str)
discrt['north_supply'] = discrt['north_supply'].apply(str)
discrt['center_supply'] = discrt['center_supply'].apply(str)
discrt['east_supply'] = discrt['east_supply'].apply(str)

discrt['south_supply'] = "S" + discrt['south_supply']
discrt['north_supply'] = "N" + discrt['north_supply']
discrt['center_supply'] = "C" + discrt['center_supply']
discrt['east_supply'] = "E" + discrt['east_supply']
discrt
```

	south_supply	center_supply	north_supply	east_supply
0	S1	C2	N1	E1
1	S1	C2	N1	E1
2	S1	C2	N1	E1
3	S1	C2	N1	E1
4	S1	C2	N1	E0
5	S1	C2	N1	E0
6	S1	C2	N1	E0
7	S1	C2	N1	E0

b. Standardization plus Equal Frequency Binning

Do the normalization first

```
#Amplitude
south_mean = np.mean(transct2['south_supply'])
north_mean = np.mean(transct2['north_supply'])
center_mean = np.mean(transct2['center_supply'])
east_mean = np.mean(transct2['east_supply'])

south_std = np.std(transct2['south_supply'])
north_std = np.std(transct2['north_supply'])
center_std = np.std(transct2['center_supply'])
east_std = np.std(transct2['east_supply'])
```

```
am = pd.DataFrame(columns = ['south','center','north','east'])
am['south'] = (transct2['south_supply'] - south_mean) / south_std
am['north'] = (transct2['north_supply'] - north_mean) / north_std
am['center'] = (transct2['center_supply'] - center_mean) / center_std
am['east'] = (transct2['east_supply'] - east_mean) / east_std
am
```

	south	center	north	east
0	-1.040459	-0.172675	-1.372328	1.557295
1	-1.126401	-0.219975	-1.648152	1.557295
2	-1.166927	-0.010860	-2.236108	1.531586
3	-1.707039	-0.271632	-2.147095	1.531586
4	-1.694462	-0.430335	-2.064524	0.194721
5	-1.654635	-0.359385	-1.970240	-0.833637

Do binning and change the data type to string

```
#Equal Frequency Binning after amplitude
discrt2 = pd.DataFrame(columns = ['south','center','north','east'])

south_bin2 = np.linspace(-2, 3, num=5) #create bin by min_supply and high_supply
north_bin2 = np.linspace(-2.5, 2.5, num=5)
center_bin2 = np.linspace(-2.5, 2.6, num=5)
east_bin2 = np.linspace(-1.5, 2.5, num=5)

discrt2['south'] = np.digitize(am['south'].as_matrix() , south_bin2, right=False)
discrt2['north'] = np.digitize(am['north'].as_matrix() , north_bin2, right=False)
discrt2['center'] = np.digitize(am['center'].as_matrix() , center_bin2, right=False)
discrt2['east'] = np.digitize(am['east'].as_matrix() , east_bin2, right=False)
#discrt2
```

```
discrt2['south'] = discrt2['south'].apply(str)
discrt2['north'] = discrt2['north'].apply(str)
discrt2['center'] = discrt2['center'].apply(str)
discrt2['east'] = discrt2['east'].apply(str)

discrt2['south'] = "S" + discrt2['south']
discrt2['north'] = "N" + discrt2['north']
discrt2['center'] = "C" + discrt2['center']
discrt2['east'] = "E" + discrt2['east']
discrt2
```

south	center	north	east
S1	C2	N1	E4
S1	C2	N1	E4
S1	C2	N1	E4
S1	C2	N1	E4
S1	C2	N1	E2
S1	C2	N1	E1
S1	C2	N1	E1
S1	C2	N1	E1
S1	C2	N1	E1
S1	C2	N1	E1
S1	C2	N1	E1
S0	C2	N1	E1
S0	C2	N1	E1

3. what algorithm you use (same as task1)

a. **Apriori**
b. **FP growth**

4. what rules you discover

transaction applied **equal frequency binning** discretization method

```
======================================================
Frequent itemset:
(E1)  support = 0.299
(C2)  support = 0.378
(S2)  support = 0.425
(S1)  support = 0.24
(C3)  support = 0.387
(S3)  support = 0.253
(E0)  support = 0.701
(N2)  support = 0.399
(N3)  support = 0.324
(N3, E0)  support = 0.206
(N3, C3)  support = 0.222
(E0, C3)  support = 0.224
(E0, N2)  support = 0.292
(S2, N2)  support = 0.2
(S2, E0)  support = 0.311
(C2, N2)  support = 0.239
(E0, C2)  support = 0.294
======================================================
======================================================
Rules:
(N3) ==> (E0)  confidence = 0.638
(N3) ==> (C3)  confidence = 0.685
(N2) ==> (E0)  confidence = 0.731
(S2) ==> (E0)  confidence = 0.731
(C2) ==> (N2)  confidence = 0.632
(C2) ==> (E0)  confidence = 0.778
======================================================
```

Rules (Wall time: 1.3s)

N3	**900~1100**	**E0**	**0~10**
N3	**900~1100**	**C3**	**800~1000**
N2	**700~900**	**E0**	**0~10**
S2	**875~1050**	**E0**	**0~10**
C2	**600~800**	**N2**	**700~900**
C2	**600~800**	**E0**	**0~10**

transaction applied **normalization** plus **equal frequency binning**

```
(S2, C2)  support = 0.19
(N3, C3)  support = 0.24
(S2, E1)  support = 0.189
(S2, N3)  support = 0.193
(S2, C3)  support = 0.191
(S2, N2)  support = 0.198
(C2, N2)  support = 0.239
(S3, C3)  support = 0.148
(E2, C3)  support = 0.157
(N2, E1)  support = 0.173
(S1, N2)  support = 0.161
(C2, E1)  support = 0.177
(S2, N3, C3)  support = 0.134
======================================================
======================================================
Rules:
(N3) ==> (C3)  confidence = 0.678
(C2) ==> (N2)  confidence = 0.65
(S3) ==> (C3)  confidence = 0.627
(S1) ==> (N2)  confidence = 0.636
(S2, C3) ==> (N3)  confidence = 0.7
(S2, N3) ==> (C3)  confidence = 0.694
======================================================
```

Rules (Wall time: 992 ms)

N3	**883~1096**	**C3**	**767~974**		
N2	**669~883**	**C2**	**564~767**		
S3	**1060~1238**	**C3**	**767~974**		
N2	**669~883**	**S1**	**702~881**		
S2	**881~1060**	**C3**	**767~974**	**N3**	**883~1096**

transaction applied **equal frequency binning** discretization method

```
patterns :
{('N4',): 811, ('N1',): 968, ('E0', 'N1'): 765, ('C1',): 1141, ('C1', 'E0'): 995, ('C2', 'S1'): 713, ('N2', 'S1'): 905, ('E0',
 'S1'): 976, ('N3', 'S3'): 849, ('C3', 'S3'): 1035, ('E0', 'S3'): 1203, ('E1', 'N2'): 704, ('E1', 'S2'): 750, ('E1', 'N3'): 76
8, ('C3', 'E1'): 1069, ('N3', 'S2'): 1038, ('E0', 'N3'): 1352, ('C3', 'E0', 'N3'): 822, ('C3', 'N3'): 1452, ('C2', 'N2', 'S2'):
802, ('C2', 'E0', 'S2'): 1096, ('C2', 'N2'): 1563, ('C2', 'E0', 'N2'): 1198, ('C2', 'E0'): 1926, ('C3', 'S2'): 971, ('C3', 'E
0'): 1465, ('N2', 'S2'): 1312, ('E0', 'N2', 'S2'): 1012, ('E0', 'N2'): 1910, ('S2',): 2785, ('E0', 'S2'): 2035, ('E0',): 4589}
rules :
{('N1',): (('E0',), 0.7902892561983471), ('C1',): (('E0',), 0.8720420683610868), ('E0', 'N3'): (('C3',), 0.6079881656804734),
 ('N2', 'S2'): (('E0',), 0.7713414634146342), ('C2', 'E0'): (('N2',), 0.6220145379023884), ('C2', 'N2'): (('E0',), 0.7664747280
870121), ('E0', 'N2'): (('C2',), 0.6272251308900524), ('S2',): (('E0',), 0.7307001795332136)}
```

Rules (Wall time: 1.54s)

N1	**500~700**	**E0**	**0~10**		
C1	**400~600**	**E0**	**0~10**		
S2	**875~1050**	**E0**	**0~10**		
C3	**800~1000**	**N3**	**900~1100**	**E0**	**0~10**
N2	**700~900**	**S2**	**875~1050**	**E0**	**0~10**
C2	**600~800**	**N2**	**700~900**	**E0**	**0~10**

transaction applied **normalization** plus **equal frequency binning**

```
###pyfpgrowth after amlitude
patterns = pyfpgrowth.find_frequent_patterns(discrt2.as_matrix(), 700)
rules = pyfpgrowth.generate_association_rules(patterns, 0.6)

print("patterns :")
print(patterns)

print("rules : ")
print(rules)
```

```
patterns :
{('N1',): 735, ('E4',): 820, ('N4',): 840, ('C1',): 848, ('E3',): 1179, ('N3', 'S3'): 797, ('C3', 'S3'): 967, ('C2', 'S1'): 81
7, ('N2', 'S1'): 1054, ('C2', 'E2'): 721, ('E2', 'N2'): 783, ('E2', 'N3'): 839, ('E2', 'S2'): 848, ('C3', 'E2'): 1028, ('N3',
 'S2'): 1263, ('C3', 'N3', 'S2'): 877, ('C3', 'N3'): 1573, ('C2', 'E1'): 1160, ('C2', 'S2'): 1242, ('C2', 'N2', 'S2'): 788, ('C
2', 'N2'): 1565, ('E1', 'N2'): 1131, ('E1', 'S2'): 1237, ('N2',): 2649, ('N2', 'S2'): 1298, ('C3',): 2779, ('C3', 'S2'): 1253,
 ('S2',): 2885}
rules :
{('C3', 'S2'): (('N3',), 0.6999201915403033), ('N3', 'S2'): (('C3',), 0.6943784639746635), ('C2', 'S2'): (('N2',), 0.6344605475
040258), ('N2', 'S2'): (('C2',), 0.6070878274268104)}
```

Rules (Wall time: 600 ms)

C3	**767~974**	**S2**	**881~1060**	**N3**	**883~1096**
C2	**564~767**	**S2**	**881~1060**	**N2**	**669~883**

5. what did you learned, and comparison between different methods

From the rules above we can expect that the amount of north, south and center usually are usually around the mean(bin[2], bin[3] bins=5) which really means nothing it's just common sense. The reason why E0 occurs a lot in the rules with discretization method 1, but not method 2 might be that the bins number I set is too big, so the datas scatter into bins and couldn't meet the support threshold.

run time comparison: It costs much more time for the table of 6550*4 datas than doing the same algorithm in task1. Also, it's almost two times faster whether using normalization before running the algorithm or not.

memory usage of the dataframe is about 204.8KB

Task3.

1. what transaction you define

(time(hour), north usage)

Export the data from MySQL to .csv file

Import the data to jupyter, and put the data into pandas.DataFrame

```
file1 = 'transcation3.csv'

raw_data1 = pd.read_csv(file1,header=0, delimiter=',')

data = np.array(raw_data1)

transct3 = pd.DataFrame(data, columns = ['date','hour', 'north_usage'])
transct3['hour'] = transct3['hour'].apply(pd.to_numeric)
transct3['north_usage'] = transct3['north_usage'].apply(pd.to_numeric)

print(transct3.info())
print(transct3)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6523 entries, 0 to 6522
Data columns (total 3 columns):
date           6523 non-null object
hour           6523 non-null int64
north_usage    6523 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 153.0+ KB
None
             date  hour  north_usage
0      2016-09-27    12        841.3
1      2016-09-27    13        826.4
2      2016-09-27    14        789.5
3      2016-09-27    15        773.6
4      2016-09-27    16        778.0
5      2016-09-27    17        783.1
```

2. what discretization method you use (same as task1)

a. Equal Frequency Binning

make time into 8 slots , 3 hours for each slot(ex. 12~14, 15~17...etc)

```
#Equal Frequency Binning
discrt = pd.DataFrame(columns = ['time','usage'])

time_bin = np.array([2, 5, 8, 11, 14, 17, 20, 23]) #create bin by min_supply and high_supply
usage_bin = np.linspace(600, 1500, num=4)

discrt['time'] = np.digitize(transct3['hour'].as_matrix() , time_bin, right=True)
discrt['usage'] = np.digitize(transct3['north_usage'].as_matrix() , usage_bin, right=False)
```

```
discrt['time'] = discrt['time'].apply(str)
discrt['usage'] = discrt['usage'].apply(str)

discrt['time'] = "T" + discrt['time']
discrt['usage'] = "U" + discrt['usage']

discrt
```

	time	usage
0	T4	U1
1	T4	U1
2	T4	U1
3	T5	U1
4	T5	U1
5	T5	U1

b. Standardization plus Equal Frequency Binning

Do the normalization first, then do binning, and then change the data type to string. But on north usage only since the hour data is discrete itself

```
#Amplitude
usage_mean = np.mean(transct3['north_usage'])
usage_std = np.std(transct3['north_usage'])

am_usage = (transct3['north_usage'] - usage_mean) / usage_std
```

```
#Equal Frequency Binning after amplitude
discrt2 = pd.DataFrame(columns = ['time','usage'])
usage_bin2 = np.linspace(-1.5, 2.5, num=5)

discrt2['usage'] = np.digitize(am_usage.as_matrix() , usage_bin2, right=False)
discrt2['time'] = discrt['time']
discrt2['usage'] = discrt2['usage'].apply(str)
discrt2['usage'] = "U" + discrt2['usage']
```

	time	usage
0	T4	U1
1	T4	U1
2	T4	U1
3	T5	U1
4	T5	U1
5	T5	U1
6	T6	U1
7	T6	U1
8	T6	U1

3. what algorithm you use (same as task1)

a. Apriori

b. FP growth

4. what rules you discover

transaction applied **equal frequency binning** discretization method

```
Frequent itemset:
(U2)  support = 0.427
(T7)  support = 0.126
(T2)  support = 0.124
(T3)  support = 0.124
(T5)  support = 0.125
(U3)  support = 0.176
(T4)  support = 0.124
(T0)  support = 0.125
(U1)  support = 0.396
(T1)  support = 0.125
(T6)  support = 0.126
(U1, T0)  support = 0.082
(U2, T3)  support = 0.065
(U1, T1)  support = 0.108
(T5, U2)  support = 0.064
(U1, T2)  support = 0.092
(T6, U2)  support = 0.076
(T7, U2)  support = 0.071
(T4, U2)  support = 0.061
======================================================
======================================================
Rules:
(T0) ==> (U1)  confidence = 0.656
(T1) ==> (U1)  confidence = 0.867
(T2) ==> (U1)  confidence = 0.744
(T6) ==> (U2)  confidence = 0.601
======================================================
```

Rules (Wall time: 887 ms)

T0	**0,1,2**	**U1**	**600~825**
T1	**3,4,5**	**U1**	**600~825**
T2	**6,7,8**	**U1**	**600~825**
T6	**18,19,20**	**U2**	**825~1050**

transaction applied **normalization** plus **equal frequency binning**

```
========================================================
Frequent itemset:
(U2)  support = 0.344
(T7)  support = 0.126
(T2)  support = 0.124
(T3)  support = 0.124
(T6)  support = 0.126
(T5)  support = 0.125
(U3)  support = 0.207
(T4)  support = 0.124
(T0)  support = 0.125
(U1)  support = 0.313
(T1)  support = 0.125
(U4)  support = 0.097
(U1, T0)  support = 0.072
(U1, T1)  support = 0.075
(U1, T2)  support = 0.076
(T6, U2)  support = 0.056
(T7, U2)  support = 0.056
========================================================
========================================================
Rules:
(T1) ==> (U1)  confidence = 0.605
(T2) ==> (U1)  confidence = 0.611
========================================================
```

Rules (Wall time: 1.2 s)

T1	**3,4,5**	**U1**	**667~875**
T2	**6,7,8**	**U1**	**667~875**

transaction applied **equal frequency binning** discretization method

```
patterns :
{('T3',): 807, ('T2',): 810, ('T2', 'U1'): 603, ('T4',): 812, ('T1',): 813, ('T1', 'U1'): 705, ('T5',): 815, ('T0',): 818, ('T
0', 'U1'): 537, ('T7',): 823, ('T6',): 825, ('U3',): 1147, ('U1',): 2586, ('U2',): 2785}
rules :
{('T2',): (('U1',), 0.7444444444444445), ('T1',): (('U1',), 0.8671586715867159), ('T0',): (('U1',), 0.656479217603912)}
```

Rules (Wall time: 761ms)

T2	**6,7,8**	**U1**	**600~825**
T1	**3,4,5**	**U1**	**600~825**
T0	**0,1,2**	**U1**	**600~825**

transaction applied **normalization** plus **equal frequency binning**

```
patterns :
{('U0',): 259, ('T1', 'U0'): 168, ('T6', 'U4'): 114, ('T3', 'U4'): 133, ('T5', 'U4'): 178, ('T4', 'U4'): 193, ('T3', 'U1'): 12
3, ('T3', 'U3'): 236, ('T3', 'U2'): 314, ('T2', 'U2'): 212, ('T2', 'U1'): 495, ('T4', 'U1'): 104, ('T4', 'U3'): 237, ('T4', 'U
2'): 278, ('T1', 'U2'): 153, ('T1', 'U1'): 492, ('T5', 'U3'): 256, ('T5', 'U2'): 282, ('T0', 'U2'): 271, ('T0', 'U1'): 472, ('T
7', 'U1'): 192, ('T7', 'U3'): 250, ('T7', 'U2'): 366, ('T6', 'U3'): 281, ('T6', 'U2'): 367, ('U3',): 1348, ('U1',): 2040, ('U
2',): 2243}
rules :
{('U0',): (('T1',), 0.6486486486486487)}
```

Rules (Wall time: 858 ms)

T1	**3,4,5**	**U0**	**575~667**

5. what did you learned, and comparison between different methods

I choose the third transaction to be time and north power usage is because I thought that there might be some association between them like the way temperature and power usage does.

Form the rules above we can notice that the power usage in north area tend to be around 600~800 when it’s late at night and in early morning. But the support is pretty low, so the ‘rules’ actually are not so rules just have slightly higher probability. I assume the low support it caused by too less data. This makes me wonder what if we have about ten years or more data, maybe we can find out the rules of power usage in different season and at different time that may can give a hand on making a better power manage plan.

run time comparison: It hard for me to explain why doing normalization before implementing algorithm increases the run time. What I observe is that the support count from the fp-growth algorithm decrease. I assume that when I want to find something about pattern. Although normalization shrink the range of the data which make us easier to do discretization, it also eliminate some characteristic of the data at the same time. Hence, in this case, normalization doesn’t really help when finding association rules.

memory usage of the dataframe is about 102KB