CS550: Massive Data Mining and Learning
Homework 1

Due 11:59pm Saturday, March 2, 2019

Only one late period is allowed for this homework (11:59pm Sunday 3/3)

Submission Instructions

Assignment Submission Include a signed agreement to the Honor Code with this assignment. Assignments are due at 11:59pm. All students must submit their homework via Sakai. Students can typeset or scan their homework. Students also need to include their code in the final submission zip file. Put all the code for a single question into a single file.

Late Day Policy Each student will have a total of *two* free late days, and for each homework only one late day can be used. If a late day is used, the due date is 11:59pm on the next day.

Honor Code Students may discuss and work on homework problems in groups. This is encouraged. However, each student must write down their solutions independently to show they understand the solution well enough in order to reconstruct it by themselves. Students should clearly mention the names of all the other students who were part of their discussion group. Using code or solutions obtained from the web is considered an honor code violation. We check all the submissions for plagiarism. We take the honor code seriously and expect students to do the same.

Discussion Group (People with whom you discussed ideas used in your answers):

On-line or hardcopy documents used as part of your answers:

I acknowledge and accept the Honor Code.

*(Signed) Xinyang Wang*______________________________

If you are not printing this document out, please type your initials above.

Answer to Question 1

1. Code for question 1 is included in question-1.zip including mapper.py, reducer.py.
2. Total structure of algorithm: mapper.py — hadoop-streaming-2.7.3.2.6.3.0-235.jar — reducer.py

   In mapper.py, we first read the flie from sys.stdin. Then split it into userID and friend group corresponding to this userID. Finally form pairs between userID and friend group with output 1 and pairs among friend group with output 0. Output is like the following:

```
0,2     1
0,3     1
0,4     1
2,3     0
2,4     0
3,4     0
```

   In reducer.py, we first read info from sys.stdin(actually from output of mapper.py). For every pair((k1,k2),flag), we update the dictionary(dictionary is in the form of 'k1':'k2':[count, flag]. After all pairs are processed, for each 'k1' in dictionary, we recomend 10 'k2' in the decreasing order of count if k1 and k2 are not friends.

3. Below are the required recommendations for user 924, 8941, 8942, 9019, 9020, 9021, 9022, 9990, 9992, 9993.

```
924     439,2409,6995,11860,15416,43748,45881
8941    8943,8944,8940
8942    8939,8940,8943,8944
9019    9022,317,9023
9020    9021,9016,9017,9022,317,9023
9021    9020,9016,9017,9022,317,9023
9022    9019,9020,9021,317,9016,9017,9023
9990    13134,13478,13877,34299,34485,34642,37941
9992    9987,9989,35667,9991
9993    9991,13134,13478,13877,34299,34485,34642,37941
```

Answer to Question 2(a)

Let's say the Pr(B) equals 1 (which means every basket contains B). In this situation,
conf($A \rightarrow B$) = Pr(B|A) = P(AB)/P(A) = P(A)/P(A) = 1
Although the confidence is 1, it can not represent that item A and item B are associated.
While from the formula of Lift and Conviction, we can see that both of them takes the Pr(B) into consideration. Thus they avoid this drawback

Answer to Question 2(b)

Confidence is not symmetrical:
conf(A $\rightarrow$ B) = Pr(B|A)= P(AB)/P(A), while conf(B $\rightarrow$ A) = Pr(A|B) = P(AB)/P(B).
Because P(A) may not equal P(B), conf(A $\rightarrow$ B) may not equal conf(B $\rightarrow$ A).

Lift is symmetrical:
Lift(A $\rightarrow$ B) = conf(A $\rightarrow$ B)/S(B) = (sup(AB)/sup(A))/(sup(B)/N) = (sup(AB)*N)/(sup(A)sup(B))
We can easily see that Lift(B $\rightarrow$ A) also equals (sup(AB)*N)/(sup(A)sup(B)), so Lift(A $\rightarrow$ B) = Lift(B $\rightarrow$ A)

Conviction is not symmetrical
conv(A $\rightarrow$ B) = (N-sup(B))sup(A)/(N*(sup(A)-sup(AB))), while conv(B $\rightarrow$ A) = (N-sup(A))sup(B)/(N*(su
sup(AB))), so they do not equal.

Answer to Question 2(c)

Confidence and Conviction are desirable.
We can see that when if B occurs every time A occurs. Then we have:
conf(A $\rightarrow$ B) = 1
conv(A $\rightarrow$ B) = +inf (exception: B occurs in every basket)
Lift(A $\rightarrow$ B) depends on S(B)

Answer to Question 2(d)

Below is the code structure for question 2(d). We iteratively read in lines of browsing.txt, generating pairs according to items in each line (in this step, we need to judge if the support of item bigger than the threshold and sort every two items before generating pairs). Finally, we prune the false positive.

```
def get_double(self, file):
    '''In this step, we do not generate doubles according \
    to singles, we generate through input file to save \
    one step'''
    for line in self.loadfile(file):
        if len(line) < 2:    #if len<2, then no doubles
            continue
        for i in range(len(line)):
            if line[i] in self.items_single and \
            self.items_single[line[i]] >= self.support:
                for j in range(i+1,len(line)):
                    if line[j] in self.items_single and \
                    self.items_single[line[j]] >= self.support:
                        l = sorted([line[i], line[j]])
                        #print(l)
                        #input()
                        self.items_double.setdefault(l[0], {})
                        self.items_double[l[0]].setdefault(l[1], 0)
                        self.items_double[l[0]][l[1]] += 1

    for key in list(self.items_double):
        for key1 in list(self.items_double[key]):
            if self.items_double[key][key1] < self.support:
                del self.items_double[key][key1]

def association_rule_double(self):
    confidence = []
    for key in self.items_double:
        for key1 in self.items_double[key]:
            confidence.append([key,key1,self.items \
            _double[key][key1]/self.items_single[key]])
            confidence.append([key,key1,self.items \
            _double[key][key1]/self.items_single[key1]])
    confidence = sorted(confidence, key=lambda x: x[1])
    confidence = sorted(confidence, key=lambda x: x[0])
    confidence = sorted(confidence, key=lambda x: x[2], \
    reverse = True)
    confidence = confidence[:5]

    for double in confidence:
        print(double[0],'\t',double[1],'\t\t',double[2])
```

Below is the top 5 rules ordered in decreasing order:

```
DAI93865      FRO40251            1.0
FRO40251      GRO85051            0.999176276771005
FRO40251      GRO38636            0.9906542056074766
ELE12951      FRO40251            0.9905660377358491
DAI88079      FRO40251            0.9867256637168141
```

Answer to Question 2(e)

Below is the code structure for question 2(e). We iteratively read in lines of browsing.txt, generating triples according to items in each line (in this step, we need to judge if the support of item bigger than the threshold and sort every three items before generating pairs). Finally, we prune the false positive.

```
def get_triple(self,file):
    '''    '''
    for line in self.loadfile(file):
        if len(line) < 3:    #if len<3, then no triples
            continue
        for i in range(len(line)):
            if line[i] in self.items_single and self.items \
            _single[line[i]] >= self.support:
                for j in range(i+1,len(line)):
                    if line[j] in self.items_single and \
                    self.items_single[line[j]] >= self.support:
                        for k in range(j+1,len(line)):
                            if line[j] in self.items_single and \
                            self.items_single[line[j]] >= self.support:
                                l = sorted([line[i], line[j], line[k]])
                                #print(l)
                                #input()
                                self.items_triple. \
                                setdefault(l[0], {})
                                self.items_triple[l[0]]. \
                                setdefault(l[1], {})
                                self.items_triple[l[0]][l[1]]. \
                                setdefault(l[2], 0)
                                self.items_triple[l[0]][l[1]][l[2]] += 1

    #prune the unfrequent triples
    for key in list(self.items_triple):
        for key1 in list(self.items_triple[key]):
            for key2 in list(self.items_triple[key][key1]):
                if self.items_triple[key][key1][key2] < self.support:
                    del self.items_triple[key][key1][key2]

def association_rule_triple(self):
    confidence = []
    for key in self.items_triple:
        for key1 in self.items_triple[key]:
            for key2 in self.items_triple[key][key1]:
                confidence.append([key,key1,key2,self.items_ \
                triple[key][key1][key2]/self.items_double[key][key1]])
                confidence.append([key,key2,key1,self.items_ \
                triple[key][key1][key2]/self.items_double[key][key2]])
                confidence.append([key1,key2,key,self.items_ \
                triple[key][key1][key2]/self.items_double[key1][key2]])
    confidence = sorted(confidence, key=lambda x: x[2])
    confidence = sorted(confidence, key=lambda x: x[1])
    confidence = sorted(confidence, key=lambda x: x[0])
    confidence = sorted(confidence, key=lambda x: x[3],reverse = True)
    confidence = confidence[:5]

    for triple in confidence:
        print(triple[0],'+',triple[1],'\t',triple[2],'\t\t',triple[3])
```

Below is the top 5 rules ordered in decreasing order:

```
DAI23334 + ELE92920       DAI62779                      1.0
DAI31081 + GRO85051       FRO40251                      1.0
DAI55911 + GRO85051       FRO40251                      1.0
DAI62779 + DAI88079       FRO40251                      1.0
DAI75645 + GRO85051       FRO40251                      1.0
```

Answer to Question 3(a)

We get k rows out of n rows, so the probability of a row is selected is $\frac{k}{n}$. Thus the probability of a row is not selected is $1 - \frac{k}{n} = \frac{n-k}{n}$.

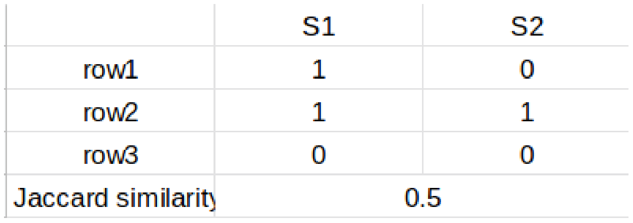
we have m 1's that are not selected, so the probability is $\underbrace{\frac{n-k}{n} * \cdots * \frac{n-k}{n}}_{m} = (\frac{n-k}{n})^m$

Answer to Question 3(b)

According to the requirement of the question, We have:
$(\frac{n-k}{n})^m \leq e^{-10}$
As we already know for large x:
$e^{-1} = (1 - \frac{1}{x})^x$
We can set x = n/k (n is much larger than k), and we get:
$e^{-1} = (1 - \frac{1}{n/k})^{n/k}$, so:
$e^{-10} = (1 - \frac{1}{n/k})^{10n/k}$
Combined with the first formula, we can get:
$(\frac{n-k}{n})^m \leq (1 - \frac{1}{n/k})^{10n/k}$
because $\frac{n-k}{n} \leq 1$, so:
$m \geq 10n/k$, which is:
$k \geq \frac{10n}{m}$

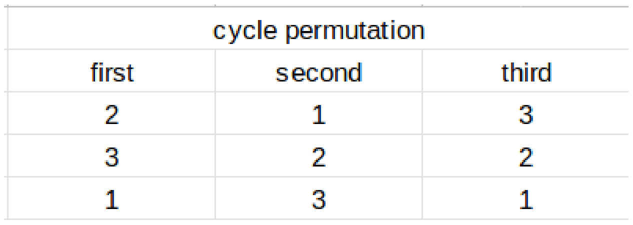
Answer to Question 3(c)

Below we have an example illustrated by p1, p2, p3. And as we can see, the similarity obtained from signature matrix does not equal to Jaccard Similarity



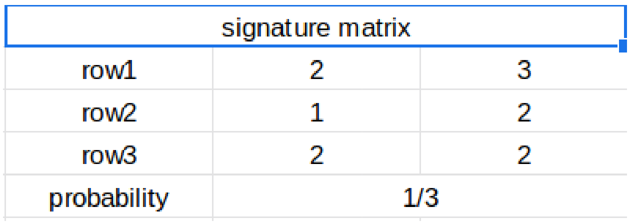
	S1	S2
row1	1	0
row2	1	1
row3	0	0
Jaccard similarit	0.5	

Figure 1: Jaccard Similarity



cycle permutation		
first	second	third
2	1	3
3	2	2
1	3	1

Figure 2: cycle permutation



signature matrix		
row1	2	3
row2	1	2
row3	2	2
probability	1/3	

Figure 3: signature matrix