Apriori Algorithm

과목명.	데이터 사이언스
담 당.	김 상 욱 교수님
제출일.	2021년 03월 18일
공과대학	소프트웨어전공
4 학년,	학 번. 2016024884
이 름.	송준익

HANYANG UNIVERSITY

목 차

1. Algorithm Description

이번 과제에서 제가 구현한 알고리즘을 간단히 정리하면 아래와 같습니다.

1. 파일을 스캔하여 input data의 크기에 맞는 행렬 생성
2. 행렬에 input data 정리
3. Input data를 scan하여 frequent한 1-itemset 찾음
4. 1-itemset의 combination을 이용하여 frequent한 2-itemset 찾음
5. 그 후, while문을 이용하여 3개 이상의 item을 가지는 frequent한 itemset이 없을 때까지 itemset을 찾아 출력함.

해당 알고리즘을 Python을 이용하여 구현하였습니다.

2. Code Description

이번 과제에서 Apriori algorithm의 구현을 위해 사용한 방법은 아래와 같습니다.

먼저, main 함수에서는 기본적인 파일 입력과 Apriori 함수 호출을 수행합니다.

```
def main(mins, strin, strout):
    global mat
    f = open(strin, 'r')
    lines = f.readlines()
    cnt = len(lines)
    max_items = -1  # The number of items in the biggest line

    # Read file to generate matrix
    for l in lines:
        spliter = l.split('₩t')

        idx = 0
        for particle in spliter:
            idx+=1

        if(idx > max_items):
            max_items = idx

    mat = np.full((cnt, max_items), -1)

    # filling matrix
    row = 0
    for l in lines:
        spliter = l.split('₩t')
        col = 0
        for particle in spliter:
            mat[row][col] = int(particle)
            col += 1
        row += 1

    f.close
    apriori(cnt, mins, max_items, strout)
```

1. file의 input data를 matrix에 저장하기 위해 먼저 파일을 읽어와 총 input line의 개수(cnt)와 모든 line들 중에 가장 input이 많은 line의 input(max_items)을 계산합니다.
2. 해당 값들을 이용해 input을 저장할 matrix를 만들고, 그 matrix에 input data를 저장합니다.
3. 그 후, apriori 함수를 실행합니다.

```
def apriori(lines, minsup, max, strout):
    global mat
    start_time = time.time()

    print("Start Apriori...")
    f = open(strout, 'w')

    freq = []
    support_cnts = np.zeros(int(np.max(mat))+1)

    # First Round => Finding frequent items (1)

    for j in range(0,lines):
        for i in range(0, 1+int(np.max(mat))):
            if i in mat[j][:max]:
                support_cnts[i]+=1
    for i in range(0, 1+int(np.max(mat))):
        sup = (float(support_cnts[i]) / float(lines)) * 100.0
        if(sup >= float(minsup)):
            freq.append(i)
```

4. Apriori 함수가 실행되면, 먼저 matrix를 돌면서 frequent한 1-itemset을 찾습니다.

5. Frequent한 1-item들을 frequent 배열에 저장합니다.

```
# Second Round => Finding frequent items (2)
if(np.size(freq) == 0):
    return
idx = 2

# Generating candidates
candi = list(combinations(freq,2))
candisize = len(candi)
confi_a = np.zeros(candisize)
confi_b = np.zeros(candisize)
support_cnts = np.zeros(candisize)
freq = []


for j in range(0,lines):
    for c in range(0, len(candi)):
        if candi[c][0] in mat[j][:max]:
            confi_a[c] += 1
        if candi[c][1] in mat[j][:max]:
            confi_b[c] += 1
        if candi[c][0] in mat[j][:max] and candi[c][1] in mat[j][:max]:
            support_cnts[c]+=1
```

6. 다음으로는 2-itemset을 찾기 위한 알고리즘입니다.

7. 먼저, frequent한 1-item들로 2개짜리 조합을 생성합니다.

8. 그 다음, matrix를 돌면서 각 조합의 개수를 셉니다. 이때, 각 confidant를 구하기 위해 조합 내 각각의 원소 개수 또한 셉니다.

```
for c in range(0, len(candi)):
    sup = (float(support_cnts[c]) / float(lines)) * 100.0
    if(sup >= float(minsup)):
        freq.append(candi[c])
        confi_a[c] = (float(support_cnts[c]) / float(confi_a[c])) * 100.0
        confi_b[c] = (float(support_cnts[c]) / float(confi_b[c])) * 100.0
        f.write("{" + str(candi[c][0]) + "}₩t" + "{" + str(candi[c][1]) + "}₩t"
                + str(format(sup,".2f")) + "₩t" + str(format(confi_a[c], ".2f")) + "₩n")
        f.write("{" + str(candi[c][1]) + "}₩t" + "{" + str(candi[c][0]) + "}₩t"
                + str(format(sup,".2f")) + "₩t" + str(format(confi_b[c], ".2f")) + "₩n")
```

9. 마지막으로, 각 조합의 support가 minimum support 이상이라면, output file에 출력합니다.

```
# Over 3 Round
while(idx <= max and fsize>0):
    printed = []
    idx += 1
    candi = []
    fsize = len(freq)
    numset = []
    if(idx % 2 == 1):
        search = int((idx-1)/2)
    else:
        search = int(idx/2)

    # Generating candidates
    for fr in freq: # Frequent number set : to check all combinations
        for i in fr:
            if(i not in numset):
                numset.append(i)
    combi = list(combinations(numset, idx))

    for com in combi: # Checking all combinations' combinations to see if it is frequent
        tmp_list = list(com)
        tmp_com_list = list(combinations(tmp_list, idx-1))
        ifadd = True
        for t in tmp_com_list:
            if(tuple_sort(t) not in freq):
                ifadd = False
        if(ifadd):
            candi.append(tuple_sort(com))
```

10. 3개 이상의 itemset은 while문을 통해 처리했습니다.

11. Frequent한 n-itemset을 구하기 위해 frequent한 n-1-itemset을 이용해 n개의 원소를 가지는 조합을 생성합니다.

12. 그 다음, 이렇게 생성된 조합 각각의 sub-combination이 frequent한지 판단합니다. 모든 sub-combination이 frequent해야만, candidate에 추가합니다.

```
support_cnts = np.zeros(len(candi))
candisize = len(candi)
for j in range(0,lines):
    for k in range(0, candisize):
        if(find(candi[k], mat[j][:max])):
            support_cnts[k] += 1

for s in range(0, len(candi)-1):
    sup = (float(support_cnts[s]) / float(lines)) * 100.0
    if(sup >= float(minsup)):
        if(candi[s] not in freq):
            freq.append(candi[s])
        target = []
        asso = []

        for i in range(1,search+1):
            test_set = list(combinations(candi[s], i))
            for test in test_set:
                target.append(test)
        for tg in target:
            asc = ()
            for num in candi[s]:
                if(num not in tg):
                    asc += (num,)
            asso.append(asc)
```

13. candidate들에 대해 support를 계산합니다, 그 후, minimum support 이상의 support를 가지는 것들에 대해서만, 출력을 위해 target과 association으로 나누는 작업을 수행합니다.

```
for j in range(0,lines):
    tsize = len(target)
    for m in range(0, tsize):
        if(find(target[m], mat[j][:max])):
            confi_tg[m] += 1
        if(find(asso[m], mat[j][:max])):
            confi_as[m] += 1
for k in range(0, tsize):
    if(target[k] not in printed):
        confi_tg[k] = (float(support_cnts[s]) / float(confi_tg[k])) * 100.0
        confi_as[k] = (float(support_cnts[s]) / float(confi_as[k])) * 100.0
```

14. 그 다음은 target과 association을 찾아 confidant를 계산합니다. 출력 과정은 자명하기에 생략하였습니다.

```
def find(tuple, list):
    for num in tuple:
        if(num not in list):
            return False
    return True

def tuple_sort(tuple):
    tmp_list = sorted(tuple)
    tpl = ()
    for num in tmp_list:
        tpl += (num,)
    return tpl
```

15. 이번 과제를 수행하기 위해 가벼운 함수 2개를 사용하였습니다. List 내에 tuple이 존재하는지 판단하는 find함수와, tuple을 정렬해주는 tuple_sort 함수를 직접 생성하여 사용했습니다.

```
import numpy as np
import sys
from itertools import combinations
import time
```

16. 해당 과제를 수행하는데 사용된 open library입니다.

3. Compiling Description

Python을 사용했으므로, 특별한 점은 없습니다.

위와 같이 입력하면 아주 잘 돌아갑니다.

4. Specifications and Testing

특별한 사항은 없습니다. 다만, 실행 확인을 위해 위 사진과 같은 메시지가 출력되도록 했습니다.