

Implementation of Apriori Algorithm

2018008395 박정호

1. Algorithm

기본적인 구조는 강의에서 설명된 알고리즘에서 크게 벗어나지 않았다. 다만, Transaction Database(이하 *TDB*) Scan 횟수를 단계별로 한번씩만으로 한정하고, *confidence*를 구하기 위해 부분집합을 구할 때, 비트마스킹을 사용하는 등, 연산 횟수를 줄이려 노력했다.

Apriori Algorithm은 크게 두가지 단계로 나뉜다. frequent pattern인지 판단할 candidate를 선정하는 과정과 이 candidate들 중 frequent pattern만을 골라내는 과정이 바로 그것이다. 새로운 frequent pattern을 찾기 전까지 이 과정을 반복하는데, 매 단계마다 candidate의 크기는 1씩 늘어난다(길이 k의 candidate set을 C_k 로 기술하겠다.).

첫번째 candidate인 C_1 은 *TDB*를 scan하는 과정에서 찾을 수 있는데, 단순히 scan 중에 발견된 *item_id* 들을 하나씩 집합으로 두면 된다. C_1 이외의 candidate C_k 는 길이 k-1의 frequent pattern (길이 k의 frequent pattern set을 F_k 로 기술하겠다.)을 self merge하는 방식으로 구할 수 있다. (frequent pattern의 subpatterns는 모두 frequent pattern이라는 규칙을 이용)

다만 이렇게 구한 C_k 가 모두 frequent pattern이라는 보장이 없으므로, 이 중 frequent pattern을 골라내야 한다. 이는 다시 *TDB*를 scan하면서 각 패턴 p 가 부분집합으로 포함된 Transaction의 비율 (이하 *support(p)*)이 전제된 *minimum support* 이상인지를 확인하는 방식으로 진행한다. 즉 매 필터링 과정마다 1번의 *TDB* scan이 발생한다. 여기서 *support(p)*의 정의는 다음과 같다.

$$support(p) = \frac{cnt(p)}{|TDB|}$$

여기서 *cnt(p)*는 p 를 포함하는 transaction의 개수이다. 위 수식의 값이 *minimum support* 이상이어야만, frequent pattern이라고 할 수 있다.

여기까지가 frequent pattern을 **Apriori Algorithm**으로 구하는 과정이다. 하지만 과제에서 요구한 것은 여기서 association rules를 구하는 것이므로 이후에 하나의 과정이 더 필요하다. $F_k (k > 1)$ 에 대해서 각 패턴을 두 개의 서로소 집합으로 나누고, 이에 대한 *confidence*를 구해야 하기 때문이다. 비트마스킹을 이용해서 가능한 모든 조합의 서로소 집합을 구하고, **Apriori Algorithm**을 이용해서 구해둔 *cnt(p)*를 이용해서 *confidence*를 구했다. 여기서 패턴 p 에서 분할된 두 서로소 집합 A, B 에 대해, $A \rightarrow B$ 의 *confidence(A → B)*는 다음과 같다.

$$confidence(A \rightarrow B) = \frac{cnt(A \cup B)}{cnt(A)} = \frac{cnt(p)}{cnt(A)}$$

2. Implementation

1. 입력 처리

```
# read input file,
# sort each transaction (to simplify merge)
def input_process(filename):
    file = open(filename)
    TDB = []

    while True:
        line = file.readline()
        if not line:
            break
        txn = list(map(int, line.split('\t')))
        txn.sort()
        TDB.append(txn)
    file.close()
    return TDB
```

단순히 주어진 filename을 이용해 파일을 read mode로 열고, 한 줄씩 읽어내려가면서 transaction을 하나의 리스트로 모으는 함수이다. 이때 transaction을 단순히 문자열로 두지 않고, 파이썬의 built-in function들을 이용해 int list로 만들어서 저장했다. 또한, 각 transaction을 정렬해서 이후의 과정에서 binary search를 사용할 수 있도록 했다.

2. Apriori 알고리즘

1. 범용 Binary Search

```
# binary search in sorted list
def search(arr, x):
    idx = bisect_left(arr, x)
    return idx!=len(arr) and arr[idx]==x
```

부분 집합 여부를 판단하는 일이 잦은 **Apriori Algorithm**의 특성 상, python의 in operator와 같은 탐색 연산을 개선할 필요가 있었다. 상술했듯이, 필자의 코드는 pattern들을 모두 정렬된 꼴로 저장하기 때문에 이분 탐색을 통해서 탐색 연산을 개선할 수 있었다.

이 연산은 후술할 get_candidate 함수에서 self merge로 구해진 함수의 부분집합이 모두 기존 frequent pattern에 포함되는지 확인하는 과정과, 역시 후술할 frequent_filter 함수에서 어떤 패턴이 transaction의 부분집합인지 확인하는 과정에서 쓰였다.

2. candidate set 구하기

```
# get initial frequent pattern set
def get_frequent_1_itemset(TDB, min_sup):
    num_txn = len(TDB)
    cnt = {}
    patterns = {}
    for txn in TDB:
        for item_id in txn:
            if not ((item_id,) in cnt):
                cnt[(item_id,)] = 0
            cnt[(item_id,)] += 1

    for pattern in cnt:
        if cnt[pattern]/num_txn*100 >= min_sup:
            patterns[pattern] = cnt[pattern]

    return patterns
```

주어진 *TDB*와 *minimum support*를 가지고 F_1 을 뽑는 과정이다. 여기서 return하는 것이 pattern을 모은 list가 아니라 dict인 것을 볼 수 있는데, 이는 나중에 association rule을 구할 때, $cnt(p)$ 를 저장해놓는 편이 추가적인 연산을 줄일 수 있기 때문이다. 또한, 여기서는 in 연산자를 굳이 search로 바꾸지 않았는데, 이는 in 연산자의 피연산자 cnt가 단순 list가 아닌 dict이기 때문에 이미 좋은 시간복잡도로 탐색을 할 수 있기 때문이다.

```
# get candidate for frequent pattern
# get candidate for frequent pattern
# k is candidate's length
# self merge is used
def get_candidate(frequent_patterns, k):
    size = len(frequent_patterns)
    i = 0
    j = 0
    candidates = []

    while i < size:
        j = 0
        while j < i:
            fp1 = frequent_patterns[i]
            fp2 = frequent_patterns[j]
            j+=1
            merged_pattern = tuple(sorted(list(set(fp1+fp2))))

            if len(merged_pattern)!=k:
                continue
```

```

        flag = 0
        for pattern in frequent_patterns:
            temp_flag = 0
            for item in merged_pattern:
                if search(pattern, item):
                    temp_flag+=1
            if temp_flag == k-1:
                flag+=1
        if flag == k:
            candidates.append(merged_pattern)
        i+=1

    return list(set(candidates))

```

F_k 를 이용해서 C_{k+1} 을 구하는 함수이다. 인자로 주어지는 frequent_patterns가 F_k 이며, 이를 이중으로 탐색하면서 가능한 모든 조합의 pattern을 merge한다. 이때는 한번 set으로 만들었다가 다시 tuple로 만드는 식으로 중복을 제거한다. 이때, 합쳐진 tuple의 크기가 k가 아니라면 이를 무시한다. 만약 길이가 k라면, 다시 frequent_patterns를 탐색하는데, 이때 이 tuple의 부분집합 중 길이가 k-1인 tuple이 frequent_patterns 안에 모두 존재하는지 찾는다. 이 과정에서 위에서 구현한 범용 이분탐색 함수 search를 활용한다.

모든 candidate를 구한 후, 이들 중 중복을 제거하기 위해서 만들어진 list를 한번 set으로 바꾼 다음, 다시 list로 변환해서 return한다.

3. frequent pattern만 골라내기

```

# check whether candidate is frequent or not
# and gather frequent patterns into one list
def frequent_filter(TDB, min_sup, candidates):
    num_txn = len(TDB)
    frequent_patterns = {}
    cnt = {}
    for txn in TDB:
        for pattern in candidates:
            flag = len(pattern)
            for item in pattern:
                if search(txn, item):
                    flag -= 1

            if flag != 0:
                continue

            if not (pattern in cnt):
                cnt[pattern] = 0

```

```

        cnt[pattern] += 1

    for pattern in cnt:
        if cnt[pattern]/num_txn*100 >= min_sup:
            frequent_patterns[pattern] = cnt[pattern]

    return frequent_patterns

```

C_k 에서 F_k 를 골라내는 함수이다. `get_frequent_1_itemset`과 마찬가지로, association rule을 구할 때 사용하기 위해 `frequent_patterns`는 단순 list가 아닌 출현빈도를 매핑, 저장한 dict로 두었다. 여기서는 *TDB*를 scan하면서 각 candidate가 transaction의 부분집합인지를 체크한다. 이렇게 출현빈도를 모두 구한 후, *minimum support* 이상인지를 체크해서 `frequent_pattern`에 해당 패턴만 저장하여 return한다.

4. Master Apriori Function

```

# master APRIORI process
# using all functions above, it returns all frequent patterns
def APRIORI_process(TDB, min_sup):
    now_frequent_set = get_frequent_1_itemset(TDB, min_sup)
    frequent_patterns = {}
    k = 1
    while len(now_frequent_set) > 0:
        k+=1
        candidates = get_candidate(list(now_frequent_set.keys()), k)
        frequent_patterns.update(now_frequent_set)
        now_frequent_set = frequent_filter(TDB, min_sup, candidates)

    return frequent_patterns

```

위에서 언급된 모든 함수를 이용하는 **Apriori Algorithm**의 master function이다. 실제로 main함수에서는 이 함수만을 호출하며, 상술된 함수들은 모두 이 함수에서 호출된다. 가장 먼저 F_1 을 구한 다음, F_k 의 수가 0이 될 때까지 반복문을 돌리면서 새로운 candidate를 구하고, 이 candidate를 이용해서 F_{k+1} 을 구한다. 또한 이렇게 구해지는 frequent pattern set들을 모두 최종적으로 return할 `frequent_patterns`에 더하는 식으로 진행된다.

이 함수도 물론, 이후에 association rule을 구할 때 사용하기 위해 `frequent_patterns`를 dict로 관리하고 있다. 이 안에는 frequent pattern과 그 출현 빈도 $cnt(p)$ 를 매핑해서 저장하고 있다.

3. Association Rule 구하기

```
# get association rules from frequent_patterns
# using bitmask to find all possible subsets
def get_association_rules(TDB, frequent_patterns):
    num_txn = len(TDB)
    association_rules = []
    for pattern in frequent_patterns:
        for bitmask in range(1<<len(pattern)):
            loop = 0
            sub1 = ()
            sub2 = ()
            while loop < len(pattern) :
                if bitmask % 2 == 1:
                    sub1+=(pattern[loop],)
                else:
                    sub2+=(pattern[loop],)
                bitmask //= 2
                loop+=1
            if len(sub1)==0 or len(sub2)==0:
                continue
            support = frequent_patterns[pattern]/num_txn*100
            confidence = frequent_patterns[pattern]/frequent_patterns[sub1]*100

            association = (sub1, sub2, support, confidence)
            association_rules.append(association)

    return association_rules
```

Apriori Algorithm만을 가지고는 frequent patterns만 구할 수 있다. Association rules를 구하기 위해서는 추가적으로 다른 알고리즘을 구현해야 했다. 2진수 00001~11111까지($1 \sim 2^k-1$, k 는 pattern의 길이)의 비트마스크(0일 경우 포함하지 않고, 1일 경우 포함됨)를 사용해서 가능한 모든 경우의 수의 부분집합을 가져왔다. 이때 frequent pattern의 subpattern은 당연히 frequent pattern이므로, 인자로 들어온 frequent_patterns에 포함됨이 보장된다.

$$confidence(A \rightarrow B) = \frac{cnt(A \cup B)}{cnt(A)} = \frac{cnt(p)}{cnt(A)}$$

*confidence*를 구하는 식은 introduction에서 소개한 식을 그대로 사용했는데, 이는 *confidence*를 구하는 조건부 확률 식에서 분자와 분모의 공통분모가 $|TDB|$ 로 동일했기 때문이다. 따라서 $cnt(p)$ 만 가지고도 정확한 *confidence*를 구할 수 있었다.

return하는 것은 association rule을 이루는 pattern A, B, 이 association rule의 *support*, *confidence*를 모두 저장한 tuple의 list이다. 이 값을 사용해서 이후의 출력 파일 구성에 사용한다.

4. 출력 처리

1. item set의 string formatting

```
# formatting function for item set
def set_to_str(item_set):
    formatted = '{'
    for item in item_set:
        formatted+=str(item)
        formatted+=','
    formatted = formatted[:-1]
    formatted += '}'

    return formatted
```

과제 명세의 item set 출력 형식이 “{a,b,c...,z}”와 같았으므로 이러한 형식에 맞게 문자열을 만들 필요가 있었다. 기존 tuple 형식 그대로 출력할 경우 “(a, b, c, d...,z)”와 같이 출력할 것이기 때문이다. 따라서 문자열에 item과 쉼표를 하나하나씩 넣은 뒤, 마지막 쉼표를 제거하고 중괄호를 닫는 방식으로 출력 형식을 맞췄다.

2. 파일 출력

```
#write output file
def output_process(filename, association_rules):
    file = open(filename, mode='w')
    for association in association_rules:
        line = set_to_str(association[0])
        line += '\t'
        line += set_to_str(association[1])
        line += '\t'
        line += '%.2f' % (association[2])
        line += '\t'
        line += '%.2f' % (association[3])
        line += '\n'

        file.write(line)
    file.close()
```

상술한 set_to_str함수를 이용해서 형식을 맞추고, output file에 구한 association rules를 출력하는 함수이다. 역시 format string을 이용해서 실수형 값의 출력 시 소수점 아래 두번째 자리까지만 출력되도록 했다.

5. main 함수

```
def main(argv):
    if len(argv)<4:
        print('PLEASE, give 3 arguments (minimum support, input filename, output filename)')
        return

    TDB = input_process(argv[2])
    start = time.time()
    frequent_patterns = APRIORI_process(TDB, float(argv[1]))
    association_rules = get_association_rules(TDB, frequent_patterns)
    print('Apriori Time:', time.time()-start)
    output_process(argv[3], association_rules)
```

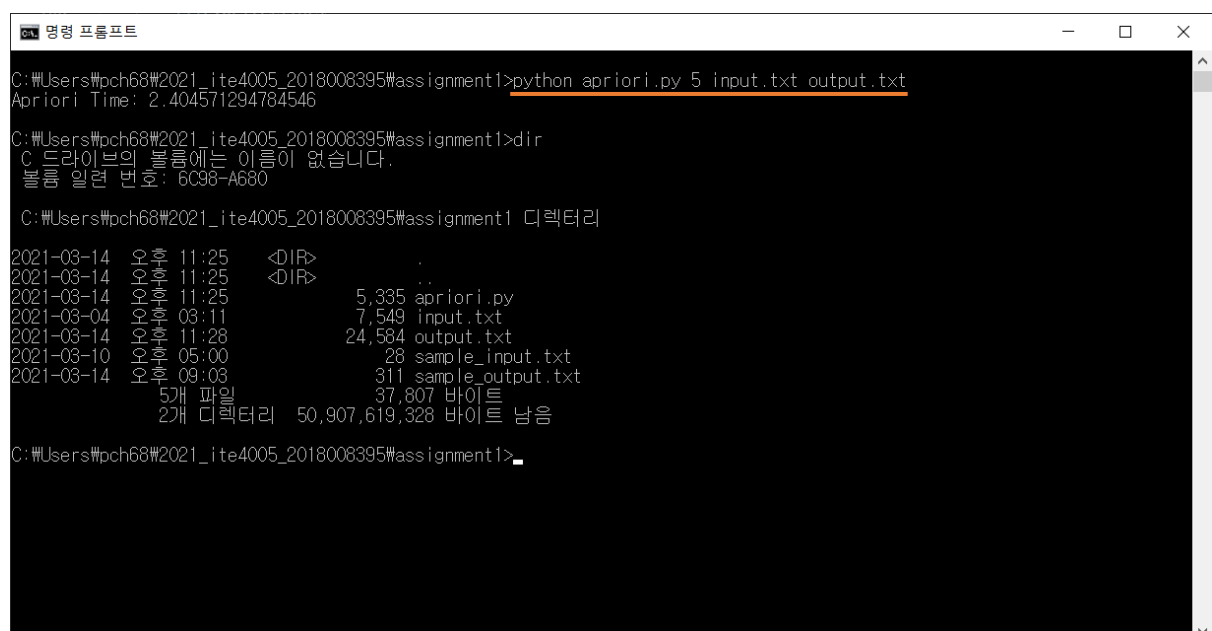
위에서 구현된 모든 함수를 이용하는 함수이다. I/O 시간을 배제하고 순수하게 필자가 구현한 알고리즘의 성능을 보기 위해 input_process와 output_process를 제외한 시간을 측정했다.

3. Usage

우선 전체 코드는 python으로 작성되었으며, 버전은 3.8.2이다.

python의 built-in library만 사용했기 때문에 numpy 등의 별도 외부 라이브러리의 설치는 필요없다.

또한 python은 인터프리터 언어이기에 별도의 컴파일 명령어나 Makefile을 쓰지 않았으며, 아래의 밑줄 그어진 부분의 명령어로 실행할 수 있다. 필자는 .py 파일에 대한 환경변수 설정이 되어있지 않았기 때문에 파일명만으로 실행하는 것이 불가능했음을 밝힌다.



```
명령 프롬프트
C:\Users\pch68\2021_ite4005_2018008395\assignment1>python apriori.py 5 input.txt output.txt
Apriori Time: 2.404571294784546

C:\Users\pch68\2021_ite4005_2018008395\assignment1>dir
C 드라이브의 볼륨에는 이름이 없습니다.
볼륨 일련 번호: 6C98-A680

C:\Users\pch68\2021_ite4005_2018008395\assignment1 디렉터리

2021-03-14 오후 11:25 <DIR> .
2021-03-14 오후 11:25 <DIR> ..
2021-03-14 오후 11:25 5,335 apriori.py
2021-03-04 오후 03:11 7,549 input.txt
2021-03-14 오후 11:28 24,584 output.txt
2021-03-10 오후 05:00 28 sample_input.txt
2021-03-14 오후 09:03 311 sample_output.txt
                5개 파일 37,807 바이트
                2개 디렉터리 50,907,619,328 바이트 남음

C:\Users\pch68\2021_ite4005_2018008395\assignment1>
```