project 1 report

2017030464 한현진

OS : Window 10 **언어버전** : Python 3.10.1

IDE : Pycharm Community Edition 2021.2.2 / 2021.3.2

apriori 알고리즘을 구현해봤습니다.

함수는 따로 나누진 않고 그냥 메인함수에서 작동하게 했습니다.

```
# 입력 받아오는 부분
args = sys.argv[1:]
min_sup = int(args[0])
try:
    input = open(args[1], 'r', encoding='utf-8')
except FileNotFoundError:
    print("*** input 파일이 없습니다. ***\n")
    return 0
output = open(args[2], 'w', encoding='utf-8')
```

처음에 명령프롬프트에서 파일을 실행할 때, 명세에서 주어진대로 입력을 받아서 minimum support와 input파일 output파일을 차례로 받아왔습니다.

인풋파일이 없으면 그냥 종료하게 만들었습니다.

```
# 인풋 자르고 트랜잭션으로 해가지고 다 넣는 모습
rdr = csv.reader(input, delimiter='\t')
transaction = []
transactions = []
tran_count = 0
dict = {}
sub_dict = {}
for line in rdr:
    for i in line:
        transaction.append(int(i))
        if tuple([int(i), ]) in sub_dict:
            sub_dict[tuple([int(i), ])] += 1
        else:
            sub_dict[tuple([int(i), ])] = 1
    transaction_tp = tuple(transaction)
    transaction = []
    transactions.append(transaction_tp)
    tran_count += 1
transactions_np = np.array(transactions, dtype=object)
```

rdr : 인풋파일을 탭간격으로 잘라내기 위한 것입니다.

transaction : 인풋파일 줄별로 있는 숫자들을 묶은 리스트입니다.

transactions : 트랜잭션들을 묶은 리스트입니다.

tran_count : 트랜잭션이 몇 개이었는지 세기 위한 변수입니다.

dict : apriori 규칙에 의해 살아남은 집합(튜플,key)이 트랜잭션 셋에 얼마나 있는지 개수를 알기 위한 변수.

sub_dict : apriori 규칙에 의해 살아남은 집합(튜플,key)이 트랜잭션 셋에 얼마나 있는지 개수를 알기 위한 변수인데, dict와 다른점은, 아래나오겠지만 while문을 돌때마다 초기화됨. 즉, 집합(튜플)의 인자 개수마다 그때그때 해당하는 애들을 잠깐 담아두는 변수.

transaction_tp: 각 트랜잭션을 tuple 형식으로 표현한 것을 저장하기 위한 변수

처음 인풋을 받아오면, 인풋파일에서 라인별로 다 자르고 tab 간격으로 다 잘라서 파이썬에서 사용가능한 dictionary 와 dictionary에 넣기 위한 튜플을 만들어서 각 튜플별로 몇 개나 있는지, 튜플을 key, 개수를 value로 해서 dictionary로 만들었습니다. 그리고 후에 비교를 위해 인풋파일을 처음부터 끝까지 읽을 때 줄별로 트랜잭션을 만들어서 그걸 튜플로 만들고, 그 튜플들을 모은 리스트를 만들어서 그것을 넘파이배열로 만들었습니다.

```python
element_count = 1

while True:
    if element_count != 1:
        # 살아남은 집합들 합집합해서 우선 집합 요소 개수로 맞는 애들만 살려놈 (self joining)
        key_sets_sj = []
        for i in range(len(keys)):
            for j in range(i + 1, len(keys)):
                key_set = set(list(keys[i])).union(set(list(keys[j])))
                if len(key_set) == element_count and key_set not in key_sets_sj:
                    key_sets_sj.append(key_set)
        # 프루닝 과정
        key_sets = []
        for key_set in key_sets_sj:
            subset_list = list(combinations(key_set, element_count - 1))
            new_subset_list = []
            for subset_element in subset_list:
                subset_element_list = list(subset_element)
                subset_element_list.sort()
                new_subset_list.append(tuple(subset_element_list))
            subset = set(new_subset_list)
            if subset == set(keys).intersection(subset):
                key_sets.append(key_set)
        # 구해낸 집합이 트랜잭션에 몇개나 있을까 확인 중
        sub_dict = {}
        for key_set in key_sets:
            for i in range(len(transactions_np)):
                bool_index = (key_set == set(list(transactions_np[i])).intersection(key_set))
                if bool_index:
                    key_set_list = list(key_set)
                    key_set_list.sort()
                    key_set_tuple = tuple(key_set_list)
                    # key_set_tuple = tuple(key_set)
                    if key_set_tuple in sub_dict:
                        sub_dict[key_set_tuple] += 1
                    else:
                        sub_dict[key_set_tuple] = 1
```

element_count : 현재 apriori 규칙에 맞는지 확인하기위해 선별한 key의 인자의 개수를 나타내기 위한 변수. 1일 때는 2페이지에 작성한 코드로 이미 선별작업이 끝나서 위의 코드는 필요가 없다.

위의 코드는 아래코드가 동작을 한번 해야 정상작동이 되므로 아래코드부터 잠깐 설명하겠습니다.

```
# 서포트 안맞는애들 다 걸러내는 중
keys = list(sub_dict.keys())
values = list(sub_dict.values())
del_list = []
for i in range(len(values)):
    support = values[i] / tran_count * 100
    if support < min_sup:
        del_list.append(keys[i])
for i in del_list:
    del sub_dict[i]

# 서포트 안맞는 애들 다 걸러냈는데 결국 남는게 없으면 더가도 답없으니까 그만둔다.
if len(sub_dict) == 0:
    break

# 서포트 맞는 애들만 이제 dict에 넣어줌. 왜냐면 이따가 컨피던스구할때 각 집합별로 개수 알아야해서 저장중
keys = list(sub_dict.keys())
values = list(sub_dict.values())
for i in range(len(keys)):
    dict[keys[i]] = values[i]

for key in keys:
    for i in range(1, element_count):
        subset = list(combinations(set(list(key)), i))
        for tuple_x_ns in subset:
            set_x_ns = set(tuple_x_ns)
            set_y_ns = set(key) - set_x_ns
            x_list = list(set_x_ns)
            x_list.sort()
            tuple_x = tuple(x_list)
            output.write(str(set_x_ns) + '\t' + str(set_y_ns) + '\t' + "{:.2f}".format(dict[key] / tran_count * 100)
                         + '\t' + "{:.2f}".format(dict[key] / dict[tuple_x] * 100) + '\n')
element_count += 1
```

del_list: 조건에 맞지 않는 key 값을 모아둔 리스트

위의 while 문 안에 같이 포함된 코드인데, sub_dict 딕셔너리에서 key 와 value 값 배열들을 따로 분류해서 리스트로 만듭니다.

리스트의 개수는 keys 와 values 는 다 같고 짝처럼 인덱스에 해당하는 위치도 같기 때문에, value 의 길이로 포문을 만들어서 value 즉, 트랜잭션에 얼마나 있는지 개수 비율을 계산해서 support 를 구하고, 그 support 가 minimum support 보다 작으면, apriori 에 해당하지 않으므로 해당 인덱스의 keys 값을 찾아서 del_list 에 추가해줍니다.

그리고 del_list 에 있는 튜플들을 sub_dict 에서 삭제해줍니다.

만약 이때, sub_dict 에 남은 것들이 없다면, 더 이상 support 를 만족하는 집합이 없다는 것이므로 while 문을 깨고 나옵니다.

그것이 아니라면, 갱신된 sub_dict 에서 다시 keys 와 values 를 뽑아서 dict 에 추가해줍니다.

후에, key 에 있는 튜플들을 리스트로만들고 집합으로 만들어서 subset 을 만들어줍니다.
이 때, subset 의 element 개수는 1 개부터 제일많을개수 -1 개로 설정해줍니다.

그리고 차집합도 구해주고 output 파일에 subset 과 key 의 집합과 subset 의 차집합인 또다른 subset 을 작성해줄 준비를 해주고, 제가 아직 설명을 못드렸는데 dict 에 쓰여진 모든 key 는 집합을 리스트로 만들고 정렬하여 튜플로 만든것이기 때문에 그 과정을 거쳐서 튜플을 만들고
support 는 현재집합(key)이 트랜잭션에 속한 개수 / 전체 트랜잭션개수
confidence 는 현재집합(key)이 트랜잭션에 속한 개수 / subset 이 트랜잭션에 속한 개수
로 구해줍니다. subset 이 트랜잭션에 속한 개수는 어떻게 아느냐? 그건 바로 dict 에 지금까지 기록해놨기 때문에 찾아서 확인하면 됩니다. 이것을 위해 dict 를 작성해 왔던 것입니다.

그럼 element_count 가 1 이 아닌 그 이상일 때 코드로 다시 돌아가서 확인해 보자면,

```
# 살아남은 집합들 합집합해서 우선 집합 요소 개수로 맞는 애들만 살려놈 (self joining)
key_sets_sj = []
for i in range(len(keys)):
    for j in range(i + 1, len(keys)):
        key_set = set(list(keys[i])).union(set(list(keys[j])))
        if len(key_set) == element_count and key_set not in key_sets_sj:
            key_sets_sj.append(key_set)
```

keys 는 이 아래에 무조건적으로 나오게 되어있습니다.
바로 선별된 sub_dict 의 key 의 리스트였습니다.
이 key 의 리스트를 통해서 for 문으로 각 키들을 합친 집합을 구해줍니다.
이 때, 자기자신끼리는 합치지 않고, 개수는 element_count 에 해당하는 즉 element_count 가 2 일때는, 2 개의 element 로 구성된 집합만을 알고 싶다 이런 느낌으로 선별해줍니다. 그리고 만약 이미 리스트안에 셀프조이닝된 집합이 있다면 굳이 또 넣을 필요없으니까 넣지않습니다.
key_set 은 집합형태라서 순서가 없어서 {1,2} 나 {2,1} 이나 똑같다고 볼거라고 예상했습니다.

```
# 프루닝 과정
key_sets = []
for key_set in key_sets_sj:
    subset_list = list(combinations(key_set, element_count - 1))
    new_subset_list = []
    for subset_element in subset_list:
        subset_element_list = list(subset_element)
        subset_element_list.sort()
        new_subset_list.append(tuple(subset_element_list))
    subset = set(new_subset_list)
    if subset == set(keys).intersection(subset):
        key_sets.append(key_set)
```

프루닝 과정에선 결국 그렇게 만들어낸 집합의 element 개수보다 1 개만큼 더 작은 element 개수를 가진 subset 이 모두 dict 에 있냐 없냐 그것을 알고 싶은 것이기 때문에, 우선 모든 self joining 된 집합을 하나씩 탐색합니다.

하나씩 subset_list 를 만들고, subset_list 에서 subset_element 를 뽑아내서 리스트로 만들어주고 정렬을 한다음에, 튜플로만들어서 new_subset_list 에 넣어줍니다. 그리고 그렇게 만든 new_subset_list 를 subset 으로 튜플의 집합으로 만들어줍니다. 이 과정을 왜 했느냐, key 는 튜플로 저장되고, 집합은 순서가 없기 때문에 리스트로 만들어서 정렬하는 과정이 없다면, (11, 3) (3, 11) 처럼 집합은 같지만, 튜플은 달라서 서로 인식하지 못하는 경우가 생길 수 있기 때문입니다. 아무튼 이렇게 subset 을 만들었다면, 이것이 keys 에 모두 있는 지 확인하기 위해 keys 를 set 으로 만들고 subset 과 교집합을 해서 subset 이 온전히 나온다면 key_sets 리스트에 추가하여 완전히 걸러진 key_sets 을 완성해주면 됩니다.

```python
# 구해낸 집합이 트랜잭션에 몇개나 있을까 확인 중
sub_dict = {}
for key_set in key_sets:
    for i in range(len(transactions_np)):
        bool_index = (key_set == set(list(transactions_np[i])).intersection(key_set))
        if bool_index:
            key_set_list = list(key_set)
            key_set_list.sort()
            key_set_tuple = tuple(key_set_list)
            # key_set_tuple = tuple(key_set)
            if key_set_tuple in sub_dict:
                sub_dict[key_set_tuple] += 1
            else:
                sub_dict[key_set_tuple] = 1
```

여기서 드디어 넘파이로 만든 넘파이 배열이 사용됩니다.
bool_index 실습시간때 봤던 것을 대충 응용해봤습니다.
선별된 집합이 집합화된 트랜잭션과 교집합이 되는지 아닌지 확인하는 것이 bool_index 입니다.
만약 교집합이라면 True 가 나올 것이므로 true 가 나왔다면, 해당 집합은 한 개라도 트랜잭션에 있다는 뜻이니까, key_set 을 우선 리스트로 만들고 그것을 정렬하여 다시 튜플로만들어서 sub_dict 에 없다면 1 이라는 value 값과 sub_dict 에 추가해주고 이미 존재한다면 그 개수를 1 개씩 늘리기 위해 value 값에 +1 만큼 해줍니다.

이후의 일은 위에 정리한대로 굴러가며 그렇게 끝날때까지 굴러갑니다.

실행방법

```
C:₩workspace₩2022_ite4005_2017030464₩apriori>python apriori.py 5 input.txt output.txt
C:₩workspace₩2022_ite4005_2017030464₩apriori>python apriori.py 4 input.txt output.txt
C:₩workspace₩2022_ite4005_2017030464₩apriori>python apriori.py 5 input.txt output.txt
C:₩workspace₩2022_ite4005_2017030464₩apriori>python apriori.py 4 input.txt output.txt
```

파이썬 파일이라서 실행폴더까지 들어가서
python apriori.py min_sup 인풋파일 아웃풋파일
로 명령어를 치시면 될 것 같습니다.

Any other specification of your implementation and testing

piazza 에도 있는 질문인데, 파이썬의 round 방식과 c#의 round 방식간에 차이가 있어서 그런지 채점프로그램을 돌린 결과 minimum support 가 4 인 경우는 4 개, 5 인경우는 2 개 정도 답안에 차이가 있었습니다.

내 답안	/	주어진 output
90.62		90.63
15.62		15.63
15.62		15.63
28.12		28.13

4 인경우는 위 4 개의 confidence 값이 나오는 경우가 다른 것으로 확인했습니다.
이는 채점할 때 관계가 없다고 하셔서 따로 고치지는 않았습니다.