assignment1 : apriori

Summary of the algorithm

structure

```
├─apriori/
|   ├─apriori.py
|   ├─utils.py
|   ├─input.txt
|   ├─output.txt
```

apriori search algorithm

1. db를 scan하면서 candidate의 support value를 얻는다.

2. 만약 어떠한 candidate도 db에서 발견되지 않는 경우, apriori 종료 후 지금까지 얻은 frequent-itemset을 반환한다.

3. support value가 minimum support보다 낮은 candidate을 제거해 frequent-itemset을 얻는다.

4. 각 단계에서 얻어진 frequent-itemset을 모두 저장한다.

5. 현재 frequent-itemset을 self-joining해 (k+1)-candidate을 generation한다.

6. (k+1)-candidate 중 subset이 frequent하지 않은 candidate 제거한다.

7. (k+1)-candidate이 존재하지 않는 경우, apriori 종료 후 지금까지 얻은 frequent-itemset을 반환한다.

8. (k+1)-candidate로 다시 1번으로 돌아간다.

association rules search algorithm

1. frequent patterns 중 길이가 긴 frequent patterns부터 탐색한다.
   - frequent itemset : $l_{i\in\{1,2,...,n\}}$이 있을 때, $len(l_1) \geq len(l_2)\ ...\ \geq len(l_n)$
2. frequent itemset l_i을 선택한 뒤, $r_i \in \{1, ..., len(l_i) - 1\}$ 길이를 갖는 l_i의 subset s_i를 생성한다.
3. s_i를 item-set으로, $(l_i - s_i)$를 associative-item-set으로 해 association rules을 추가한다.
4. 위 과정을 frequent patterns에 존재하는 모든 $l_{i\in\{1,2,...,n\}}$에서 길이 $r_{i\in\{1,2,...,n\}}$로 만들 수있는 모든 경우의 subset $s_{i\in\{1,2,...,n\}}$를 구해 association rules을 구한다.

위와 같은 algorithm을 통해 단순히 모든 frequent set에 대한 경우의 수를 구하는 것보다 훨씬 빠르게 구현할 수 있다.

위의 algorithm이 정당한 이유는 기존의 association rule을 구하기 위한 조건 중 한 가지인 $sup(item_set \cup associative_item_set) \geq min_sup$이 있기 때문이다.

해당 식이 의미하는 것은 결국 어떤 set A, B가 association rule을 만족한다면 $A \cup B$도 반드시 frequent pattern여야 한다는 것이므로, 각 frequent itemset의 disjoint set만을 이용해 association rule을 구할 수 있게 된다.

self joining algorithm

1. k-frequent_itemset $l^{(k)}_{i\in\{1,2,...,n\}}$에서 $(l^{(k)}_i, l^{(k)}_j)$ 쌍의 frequent item을 뽑는다.
2. 각 $l^{(k)}_i, l^{(k)}_j$ 의 원소를 정렬한다. 즉, $l^{(k)}_{t\in\{i,j\}}[1] \leq l^{(k)}_{t\in\{i,j\}}[2]$ ... $\leq l^{(k)}_{t\in\{i,j\}}[k]$을 만족한다.
3. 그 후 $(l^{(k)}_i[1] = l^{(k)}_j[1]) \wedge (l^{(k)}_i[2] = l^{(k)}_j[2])... \wedge (l^{(k)}_i[k-1] = l^{(k)}_j[k-1])$이 만족하는지 확인한다.
4. 마지막으로 $l^{(k)}_i[k] < l^{(k)}_j[k]$이 만족하는지 확인한 뒤, 모두 만족하는 경우 $l^{(k)}_i \cup l^{(k)}_j$를 (k+1)-candidate에 추가한다.

위와 같은 algorithm을 통해 단순히 self joining하는 것보다 더 효율적인 algorithm을 구현할 수 있다.

위의 algorithm이 정당한 이유는 해당 generation 과정을 통해 (k+1)-candidate을 구하더라도 추후에 pruning 과정을 통해 해당 candidate의 모든 subset이 frequent해야 한다는 사실을 이용한 것이다.

즉, 어떤 (k+1)-candidate의 모든 subset이 frequent하다면, 해당 frequent subset 중에서는 반드시 $1 \sim k-1$까지만 같고$(l_i[1:k-1] = l_j[1:k-1])$, $l_i[k] \neq l_j[k]$인 subset이 존재할 것이고, 이 경우에서 $l_i[k] < l_j[k]$인 경우만 고려함으로써 중복 계산을 피할 수 있게 된다.

pruning algorithm

1. self-joining을 통해 생성된 (k+1)-candidate을 돌면서 k길이의 만들 수 있는 모든 subset을 생성한다.
2. 해당 subset이 k-frequent_itemset에 없는 경우 candidate에서 제거한다.
3. 모든 (k+1)-candidate에 대해 위의 과정을 적용해 남은 candidate을 반환한다.

Detailed Description

apriori.py

```
def apriori_search(init_candidate:List[FrozenSet],db:List[str],abs_min_sup:int)->Dict[FrozenSet,int]:
  """
  apriori algorithm을 적용해 frequent patterns을 반환하는 함수

  1. db를 scan하면서 candidate의 support value를 얻는다.
  2. 만약 어떠한 candidate도 db에서 발견되지 않는 경우, apriori 종료한다.
  3. support value가 minimum support보다 낮은 candidate을 제거해
     frequent-itemset을 얻는다.
  4. 각 단계에서 얻어진 frequent-itemset을 모두 저장
  5. 현재 frequent-itemset을 self-joining해 (k+1)-candidate을 generation한다.
  6. (k+1)-candidate 중 subset이 frequent하지 않은 candidate 제거한다.
  7. (k+1)-candidate이 존재하지 않는 경우, apriori 종료한다.
  8. (k+1)-candidate로 다시 1번으로 돌아간다.
  """
  k=1
  candidate = init_candidate
  freq_pattern={}
  while True:
      ## 1. db를 scan하면서 candidate의 support value를 얻는다.
      counter = db_scan(db,candidate)
```

```
      ## 2. 만약 어떠한 candidate도 db에서 발견되지 않는 경우, apriori 종료
      if not counter:
          break
      ## 3. support value가 minimum support보다 낮은 candidate을 제거해
      ##    frequent-itemset을 얻는다.
      freq_set = get_frequent_set(counter,abs_min_sup)

      ## 4. 각 단계에서 얻어진 frequent-itemset을 모두 저장
      for freq_item in freq_set:
          freq_pattern[freq_item]=counter[freq_item]

      k+=1
      ## 5. frequent-itemset을 self-joining해 (k+1)-candidate을 generation한다.
      generated = self_joining(freq_set)
      ## 6. (k+1)-candidate 중 subset이 frequent하지 않은 candidate 제거
      candidate = pruning(generated,freq_set,k)
      ## 7. (k+1)-candidate이 존재하지 않는 경우, apriori 종료.
      if not candidate:
          break

  return freq_pattern

def association_rules_searching(frequent_pattern,num_transaction)->List:
    """
    frequent patterns을 입력받아 association rules을 반환하는 함수

    1. frequent patterns 중 길이가 긴 frequent patterns부터 탐색
    2. 1~len(l)-1까지 반복하면서 해당 길이에 맞는 l의 subset인 item_set을 생성한다.
    3. l - item_set = associative_item_set으로 만든 뒤
       item_set -> associative_item_set을 association rule로 추가
    4. 위 과정을 모든 frequent set에 대해 시행
    """
    association_rules=[]
    ## 1. frequent patterns 중 길이가 긴 frequent patterns부터 탐색
    for l in sorted(frequent_pattern,key=lambda x : len(x),reverse=True):
        ## 2. 1~len(l)-1까지 반복하면서 해당 길이에 맞는 l의 subset인 item_set을 생성
        ## 3. l - item_set = associative_item_set으로 만든 뒤,
        ##    item_set -> associative_item_set을 association rule로 추가
        for r in range(1,len(l)):
            for item_set in combinations(l,r):
                item_set = frozenset(item_set)
                associative_item_set = l-item_set
                freq_score = frequent_pattern[l]/num_transaction
                conf_score = frequent_pattern[l]/frequent_pattern[item_set]
                association_rules.append([item_set,associative_item_set,freq_score,conf_score])

    return association_rules

def main(args):
    rel_min_sup = args.rel_min_sup/100
    input_path = args.input_path
    output_path = args.output_path

    ## prepare
    with open(input_path,'r',encoding='utf-8') as f:
        db = f.readlines()
    db = list(map(preprocessing,db)) # db 내 transaction을 모두 int set으로 변경
    abs_min_sup = int(len(db)*rel_min_sup) # relative minimum support value를 absolute minimum support value로 변경
    init_candidate = [frozenset([i]) for i in range(20)] # 초기 default candidate 설정

    ## apriori algorithm
    frequent_pattern = apriori_search(init_candidate=init_candidate,
                                      db=db,abs_min_sup=abs_min_sup)

    ## association rule searching
    association_rules = association_rules_searching(frequent_pattern=frequent_pattern,num_transaction=len(db))

    ## output
    output=[]
    for out in association_rules:
        itemset = set(out[0])
        associataive_item_set = set(out[1])
        sup = round(out[2]*100,2)
```

```
        conf = round(out[3]*100,2)
        o = f"{itemset}\t{associataive_item_set}\t{sup:.2f}\t{conf:.2f}\n"
        output.append(o)

    with open(output_path,'w') as f:
        f.writelines(output)
```

utils.py

```
def preprocessing(transaction:str)->FrozenSet
"""
하나의 string transaction 데이터를 입력으로 받아 int format의 set으로 반환해주는 함수

Parameters:
  transaction(str) : db의 하나의 trasaction, str으로 표현되어 있다.

Returns:
  FrozenSet[int]
"""
  transaction = transaction.strip()
  transaction = transaction.split('\t')
  transaction = list(map(int,transaction))
  return frozenset(transaction)


def db_scan(db:List[str],candidate:List[FrozenSet])->Dict[FrozenSet,int]:
"""
전체 db를 scan하면서 candidate의 횟수를 카운팅하는 함수

Parameters:
  db(List[str]) : 전체 db list
  candidate(List[FrozenSet]) : candidate list

Returns:
  Dict[FrozenSet,int] : candidate을 key값으로, frequency를 values값으로 갖는 dict
"""
  counter={}
  for transaction in db:
      for c in candidate:
          if c.issubset(transaction):
              if counter.get(c) is None:
                  counter[c]=1
              else:
                  counter[c]+=1
  return counter


def get_frequent_set(counter:Dict[FrozenSet,int],abs_min_sup:int)->List[FrozenSet]:
"""
각 candidate의 frequency와 absolute minimum support value를
입력으로 받아 frequent itemset을 반환하는 함수.

Parameters:
  counter(Dict[FrozenSet,int]) : candidate을 key값으로, frequency를 values값으로 갖는 dict
  abs_min_sup(int) : absolute minimum support value

Returns:
  List[FrozenSet] : frequent itemset
"""

  freq_set=list()
  for itemset,freq in counter.items():
      if freq>=abs_min_sup:
          freq_set.append(itemset)
  return freq_set


def self_joining(freq_set:List[FrozenSet])->List[FrozenSet]:
```

```
"""
(k-1)-itemset을 입력으로 받아 self-joining으로 k-candidate을 생성(generation)

Parameters:
  freq_set(List[FrozenSet]) : (k-1)-itemset

Returns:
  List[FrozenSet] : k-candidate
"""
  generated=list()

  for L1,L2 in permutations(freq_set,2): # (l_i, l_j 선택)
      joinable=True
      for i,(item1,item2) in enumerate(zip(sorted(L1),sorted(L2))): # (l_i, l_j 정렬)

          # l_i[k] < l_j[k]을 만족하는지 여부 검사
          if i==len(L1)-1:
              if item1>=item2:
                  joinable=False
              break

          # l_i[1:k-1] = l_j[1:k-1]을 만족하는지 여부 검사
          if item1!=item2:
              joinable=False
              break

      if not joinable:
          continue

      # 모두 만족한다면 self-joining 후 candidate에 추가
      r = (L1|L2)
      generated.append(r)
  return generated


def pruning(generated:List[FrozenSet],prev_freq:List[FrozenSet],k:int)->List[FrozenSet]:
"""
k-candidate의 subset이 infrequent한 경우 candidate에서 제외하는 함수.

Parameters:
  generated(List[FrozenSet]) : k-candidate
  prev_freq(List[FrozenSet]) : (k-1)-itemset
  k(int) : k

Returns:
  List[FrozenSet] : subset이 infrequent한 candidate을 제외한 set
"""
  result=list()
  for candidate in generated: # 각 k-candidate을 순회
      isCandidate=True

      # 해당 candidate이 만들 수 있는 모든 (k-1) subset을 생성
      for subset in combinations(candidate,k-1):
          subset = frozenset(subset)

          # 만약 subset 중 하나라도 frequent하지 않다면 candidate에서 제거
          if subset not in prev_freq:
              isCandidate=False
              break
      if isCandidate:
          result.append(candidate)
  return result
```

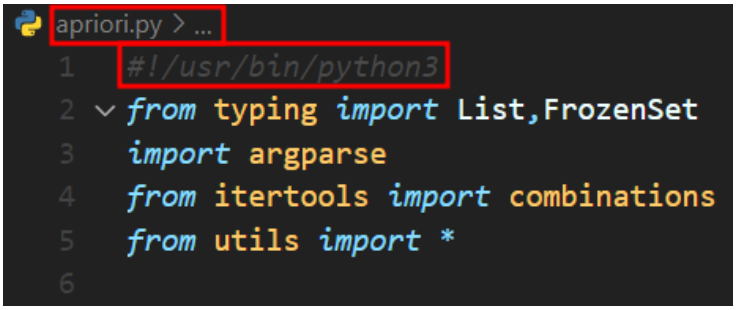
Speed

$70.61ms \pm 1.24ms$

```
moon@DESKTOP-KV9137P:~/data_science$ ./apriori.py 5 input.txt output.txt
total process time :71.03ms
moon@DESKTOP-KV9137P:~/data_science$ ./apriori.py 5 input.txt output.txt
total process time :69.78ms
moon@DESKTOP-KV9137P:~/data_science$ ./apriori.py 5 input.txt output.txt
total process time :71.79ms
moon@DESKTOP-KV9137P:~/data_science$ ./apriori.py 5 input.txt output.txt
total process time :71.86ms
moon@DESKTOP-KV9137P:~/data_science$ ./apriori.py 5 input.txt output.txt
total process time :68.62ms
```

Execute Guideline

1. `apriori.py`의 첫 줄에 python 실행 위치를 적어줍니다.



2. 현재 유저의 `apriori.py` 실행권한을 허용합니다.

```
chmod apriori.py 755
```