Data Science – Assigment1 Report

- Environment
  - Python 3.7
- Compiling
  - `C:₩Users₩User₩Desktop>apriori.py 5 input.txt output.txt`
- Summary of Algorithm
  - First, I save the given data into a variable called 'transaction' and use it to find candidates and frequent patterns. I calculate candidates and frequent patterns respectively when its length is 1. Then, I use a loop to calculate when its length is higher than 2. In a loop, I self-join the all possible candidates from previous frequent pattern, and then prune the candidates. After that, I check support of all itemset of new candidates whether their supports are higher than minimum support or not. I apply associative rules with these new frequent patterns and continue this process until there are no frequent patterns.
- Description of Codes

```
def apriori(transaction, minSup):
    total_trans = len(transaction)
    minSup = (minSup/100)*total_trans #save minimum support as a number
    line = '' # to save result

    C = set() # candidate
    L = set() # FP
    k = 1

    # C1 for k = 1
    for trx in transaction:
        for item in trx:
            if item not in C:
                C.add(item)
    C = sorted(C)
    #print('1st C:', C)

    # L1 for k = 1
    for c in C:
        if getSupport([c], transaction) >= minSup:
            L.add(c)
    #print('1st L:', L)
    L = sorted(L)
```

```
    # k >= 2
    while True:
        k += 1
        previous_L = copy.deepcopy(L)
        # do self joining first to get k+1 candidates
        C = selfJoin(L, k) # return tuple in a set
        #print(k,'st Join:', C)
        # do pruninig after self joining
        C = prune(C, previous_L, k) # return tuple in a set
        #print(k,'st Prune:', C)
        # check minimum support
        L = testSupport(C, minSup, transaction)
        #print(k, 'st FP:', L)

        # if there is no more FP, stop apriori algorithm
        if not L:
            break
        else:
            # if there is a FP, apply associative rule
            line += associationRule(L, k, minSup, total_trans, transaction)
    return line
```

- This function is a process of applying apriori algorithm. At beginning of this function, I calculate the frequent patterns when the length of itemset (variable 'k') is one. In a while loop, I self-join the candidates from previous frequent patterns(variable 'L') and prune them. When there is no frequent pattern, it stops. Otherwise, I apply an associative rule to the frequent patterns.

```
def selfJoin(C, k):
    joined_C = []

    for itemset in C:
        if k == 2:
            itemset = [itemset]
        for item in itemset:
            if item not in joined_C:
                #print(item)
                joined_C.append(item)
    joined_C = set((itertools.combinations(sorted(joined_C), k)))

    return joined_C
```

- This function is to self-join the candidates. I get all possible items whose length is 1 from previous frequent patterns whose length is k-1. Then, I use "itertools.combinations function" to find all the combinations of candidates whose length is k. I save this combination as

sorted because to prune it correctly in the next step.

```
def prune(C, previous_L, k):
    pruned_C = copy.deepcopy(C)

    for itemset in C:
        comb = set(itertools.combinations(sorted(itemset), k-1))

        if k == 2:
            for item in comb:
                if not set(item).issubset(previous_L):
                    pruned_C.remove(itemset)
                    break
        else:
            for item in comb:
                #print(item, previous_L)
                if not set((item,)).issubset(previous_L):
                    pruned_C.remove(itemset)
                    break

    return pruned_C
```

- This function is a process of pruning. When I have self-joined candidates, I prune them whether to check item's subsets are from previous frequent patterns. I sort the set to correctly remove the itemset. If not, I remove them from candidates.

```
def testSupport(itemset, minSup, transaction):
    C = copy.deepcopy(itemset) #save whole itemset first

    for item in itemset:
        if getSupport(item, transaction) < minSup:
            C.remove(item) #if its support is lower than minimum, remove it
    return C
```

- This function is to check the itemset of candidates whether they are higher the minimum support or not. If the support is lower, I remove them from candidates.

```
def getSupport(item, transaction):
    cnt = 0
    for trx in transaction:
        if set(item).issubset(set(trx)):
            cnt += 1

    return cnt
```

- This function is to get support of given itemset.

```
def associationRule(L, k, minSup, total, transaction):
    line = "" #variable to save result
    tmp = k #to save k
```

```
    for itemset in L:
        while k > 1:
            comb = list(itertools.combinations(itemset, k-
1)) #find the combinations of FP
            #print(itemset, comb)
            for item in comb:
                #print(item)
                a = set(item) # an item
                b = set(itemset) - a #find another part of the set
                cnt = 0 # to find appearance of item to find confidence
                for trx in transaction:
                    if a.issubset(set(trx)):
                        cnt += 1
                support = getSupport(itemset, transaction) #find the number of
 appearance
                confidence = (support/cnt)*100 #find percentage of confidence
                support = (support/total)*100 #find percentage of support

                line += str(a)+'\t'+str(b)+'\t'+str('%.2f'%round(support,2))+'
\t'+str('%.2f'%round(confidence, 2))+'\n'
            k -= 1 #keep repeating the procedure to find all subsets
        k = tmp

    return line
```

- This function is a process of applying associative rules to new frequent patterns and save the result in a variable called 'line'. When I get new frequent patterns, whose length is higher than or same as 2, I calculate its subset of itemset to check its support and confidence.

```
def openfile(file):
    f = open(file, 'r')
    arr = list()
    arr2 = list()
    while True:
        line = f.readline().strip()
        if not line: break
        # save give transactions as an integer
        arr.append(sorted(map(int, line.split('\t'))))
    f.close()
    return arr
```

- This function is to read the given input data set and save them. I change the given dataset from string to integer to apply apriori algorithm easily.

```
def writefile(file, result):
    f = open(file, 'w')
    f.write(result)
```

```
    f.close()
```

- This function is to save a result from apriori algorithm.

```
if __name__ == '__main__':
    minSup = int(sys.argv[1])
    file = sys.argv[2]
    output = sys.argv[3]
    transaction = openfile(file)
    result = apriori(transaction, minSup)
    writefile(output, result)
```

- This is main function, and I get all arguments, save the given dataset in openfile function, do apriori algorithm, and write the result in given output file.