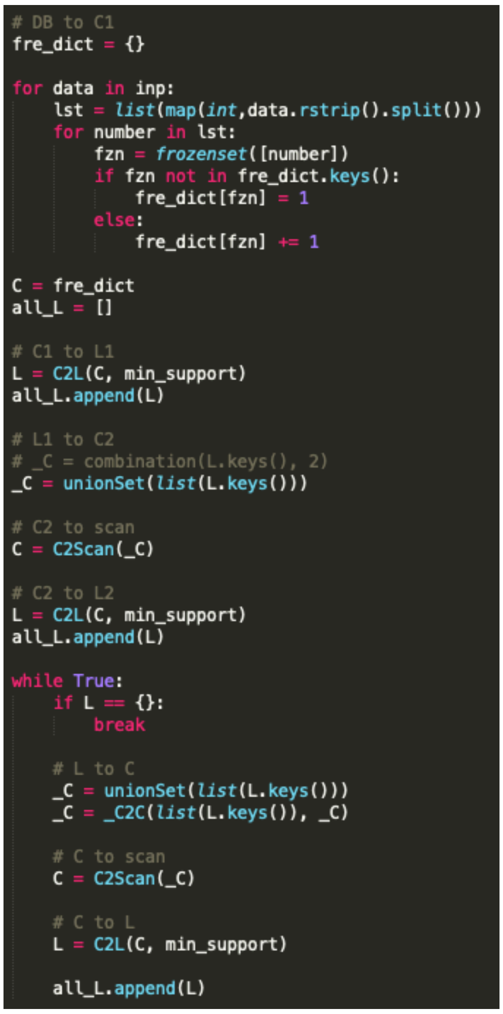
R07922141
張緣彩
18/10/2018

Data Mining Homework 1 Report

I. Implementation

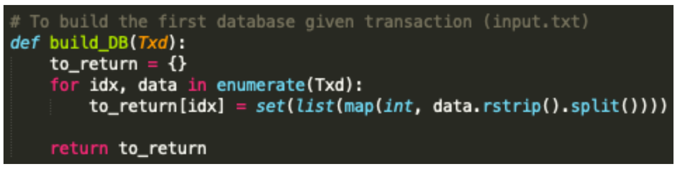
a. Apriori

My main code fragment is as the figure below, I will describe in detail in bottom.



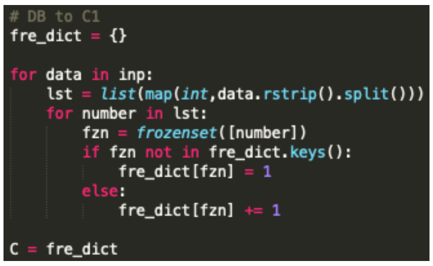
```
# DB to C1
fre_dict = {}

for data in inp:
    lst = list(map(int,data.rstrip().split()))
    for number in lst:
        fzn = frozenset([number])
        if fzn not in fre_dict.keys():
            fre_dict[fzn] = 1
        else:
            fre_dict[fzn] += 1

C = fre_dict
all_L = []

# C1 to L1
L = C2L(C, min_support)
all_L.append(L)

# L1 to C2
# _C = combination(L.keys(), 2)
_C = unionSet(list(L.keys()))

# C2 to scan
C = C2Scan(_C)

# C2 to L2
L = C2L(C, min_support)
all_L.append(L)

while True:
    if L == {}:
        break

    # L to C
    _C = unionSet(list(L.keys()))
    _C = _C2C(list(L.keys()), _C)

    # C to scan
    C = C2Scan(_C)

    # C to L
    L = C2L(C, min_support)

    all_L.append(L)
```

First, we read the data and build a database. The database is dictionary type data structure with transaction id as key and a set of itemset as value.

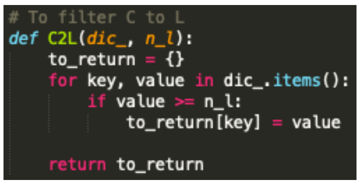


```
# To build the first database given transaction (input.txt)
def build_DB(Txd):
    to_return = {}
    for idx, data in enumerate(Txd):
        to_return[idx] = set(list(map(int, data.rstrip().split())))

    return to_return
```

Second, we iterate the all the transactions to generate C1 (all the items) and count the support of each item. Then from C1 we filter out those items that are not greater than min support (function C2L) to generate L1.



```python
# DB to C1
fre_dict = {}

for data in inp:
    lst = list(map(int,data.rstrip().split()))
    for number in lst:
        fzn = frozenset([number])
        if fzn not in fre_dict.keys():
            fre_dict[fzn] = 1
        else:
            fre_dict[fzn] += 1

C = fre_dict
```



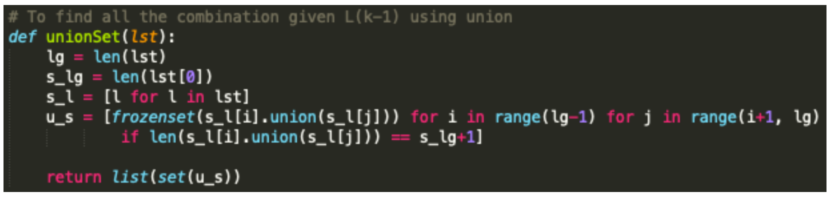
```python
# To filter C to L
def C2L(dic_, n_l):
    to_return = {}
    for key, value in dic_.items():
        if value >= n_l:
            to_return[key] = value

    return to_return
```

From L1 to generate C2 by combine any 2 items in L1 (function unionSet). For example,

a. if L2 = {AB, AC, CE} as input then unionSet will output C3 = {ABC, ACE}. (ABCE will be drop)

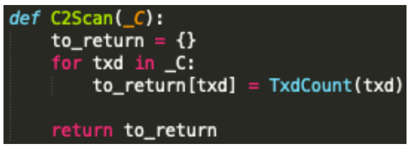
b. if L1 = {A, B, C, E} as input then unionSet will output C2 = {AB, AC, AE, BC, BE, CE}.

We don’t need to check if subsets of each item in C2 are the subset of L1.

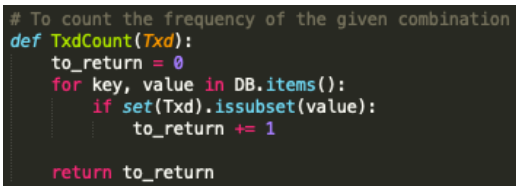


```python
# To find all the combination given L(k-1) using union
def unionSet(lst):
    lg = len(lst)
    s_lg = len(lst[0])
    s_l = [l for l in lst]
    u_s = [frozenset(s_l[i].union(s_l[j])) for i in range(lg-1) for j in range(i+1, lg)
               if len(s_l[i].union(s_l[j])) == s_lg+1]

    return list(set(u_s))
```

After generate the candidate C2, we need to scan through database and filter out those itemsets that are not greater than min support (function C2Scan, C2L) and generate L2.



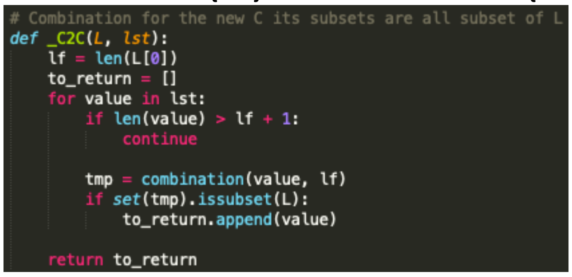
```python
def C2Scan(_C):
    to_return = {}
    for txd in _C:
        to_return[txd] = TxdCount(txd)

    return to_return
```



```python
# To count the frequency of the given combination
def TxdCount(Txd):
    to_return = 0
    for key, value in DB.items():
        if set(Txd).issubset(value):
            to_return += 1

    return to_return
```

From L2 we generate C3 using the same technique. But before we scan through the database, we have to filter out the itemsets in C3 which its subsets are not in L2 (function _C2C). For example,

a. L2 = {AB, AC, BC, BE}, Temp C3 = {ABC, ABE} then we filter out ABE since its subset {AE} is not in L2. So C3 = {ABC}.

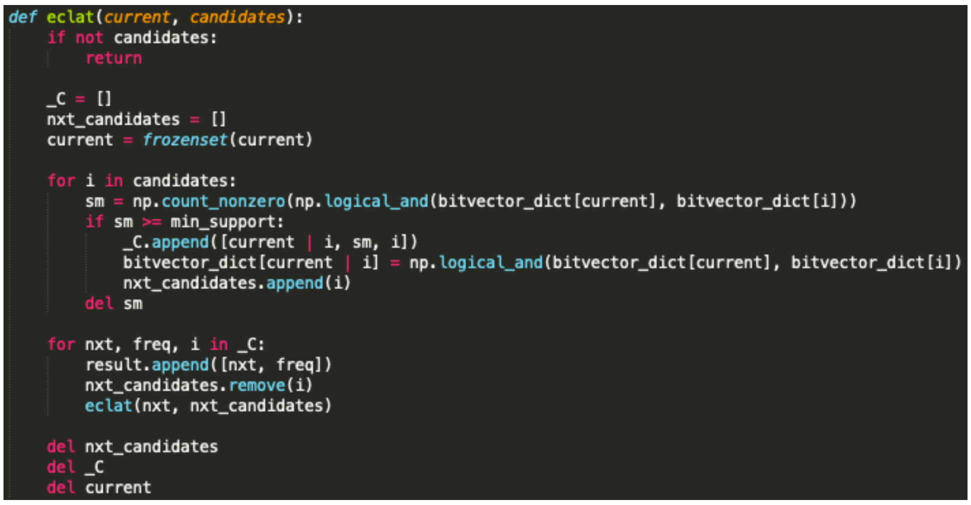


```python
# Combination for the new C its subsets are all subset of L
def _C2C(L, lst):
    lf = len(L[0])
    to_return = []
    for value in lst:
        if len(value) > lf + 1:
            continue

        tmp = combination(value, lf)
        if set(tmp).issubset(L):
            to_return.append(value)

    return to_return
```

After generate C3 we can scan through the database to generate L4. This process will keep iterate until no new L is generate.

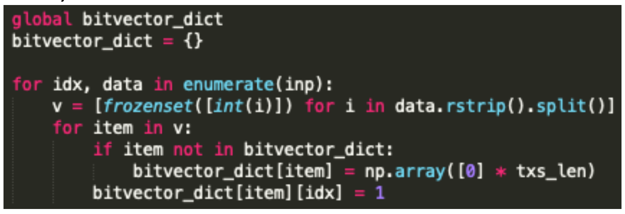
b. Eclat

My main code fragment for eclat algorithm is as figure below. Since I use recursive depth first search, the code will be short compare to apriori. I will also describe in detail in bottom.



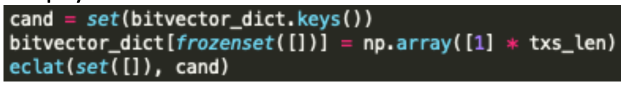
```
def eclat(current, candidates):
    if not candidates:
        return

    _C = []
    nxt_candidates = []
    current = frozenset(current)

    for i in candidates:
        sm = np.count_nonzero(np.logical_and(bitvector_dict[current], bitvector_dict[i]))
        if sm >= min_support:
            _C.append([current | i, sm, i])
            bitvector_dict[current | i] = np.logical_and(bitvector_dict[current], bitvector_dict[i])
            nxt_candidates.append(i)
        del sm

    for nxt, freq, i in _C:
        result.append([nxt, freq])
        nxt_candidates.remove(i)
        eclat(nxt, nxt_candidates)

    del nxt_candidates
    del _C
    del current
```

First, we read the data and build a vertical bit vector database.



```
global bitvector_dict
bitvector_dict = {}

for idx, data in enumerate(inp):
    v = [frozenset([int(i)]) for i in data.rstrip().split()]
    for item in v:
        if item not in bitvector_dict:
            bitvector_dict[item] = np.array([0] * txs_len)
        bitvector_dict[item][idx] = 1
```

Second, apply Eclat recursive depth first search function. Given an empty set as root.



```
cand = set(bitvector_dict.keys())
bitvector_dict[frozenset([])] = np.array([1] * txs_len)
eclat(set([]), cand)
```

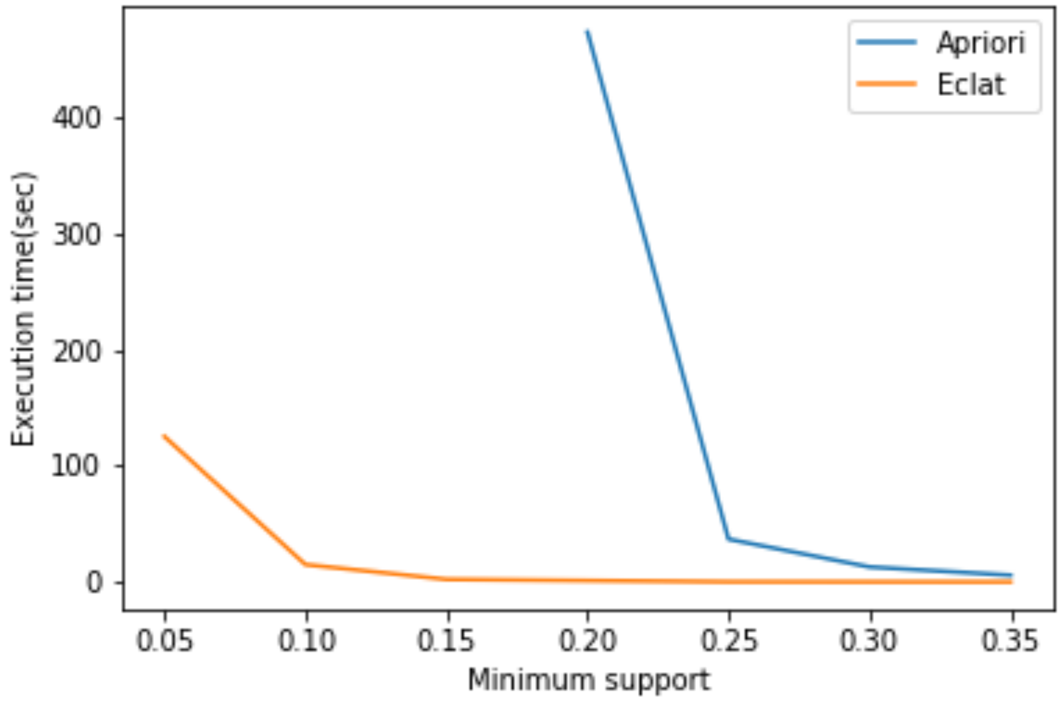
The function iterates all the itemsets. The arguments of the function are current itemset and the candidate itemsets. For example, if the itemsets we have are {A, B, C, D, E}, for current = A then the candidates of A will be {B, C, D, E}; for current = B then the candidates of B will be {C, D, E} since {AB} will be generate when current = A so that B does not need A as its candidate. For every recursive call, we will iterate the candidates and calculate the support of the new itemset (union of current and the candidate's item). If the support of the new itemset is greater than minimum support, the new itemset will become the next current and the candidate's item will be saved in the next candidates list. For performance reason, the new itemset bit vector will also be saved in the bit vector dictionary. After the iteration, we will apply the function to each next current and next candidate recursively until there is no new next candidate generated.

I found out that if I use '&' instead of np.logical_and() as logic AND, it will hurt the performance. And if I use sum() or np.sum() instead of np.count_nonzero() as to add all the intersected bit vector, it will hurt the performance really bad.

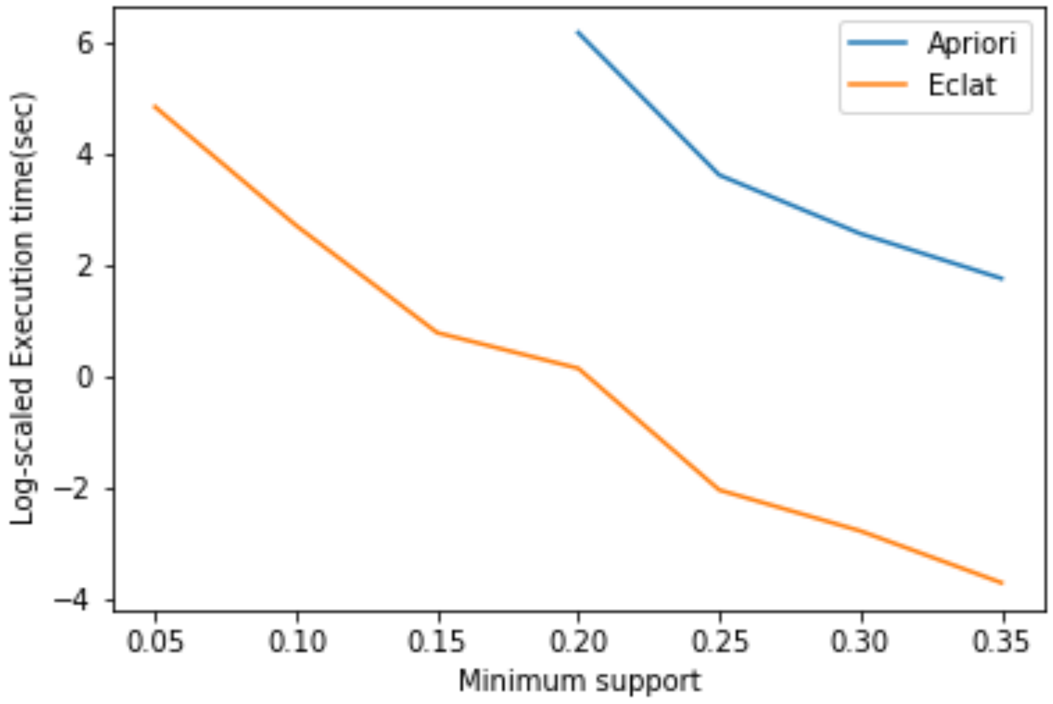
II. Graph

	0.05	0.1	0.15	0.2	0.25	0.3	0.35
Apriori	-	-	-	473.90	36.86	12.88	5.78
Eclat	125.324	14.9295	2.1841	1.1528	0.13	0.0624	0.0247

Plot in seconds



Plot in seconds (Log-scaled)


Apriori
Eclat
Log-scaled Execution time(sec)
Minimum support