Apriori assignment report

2015004248
Jo Hyung Jung

1. Overview

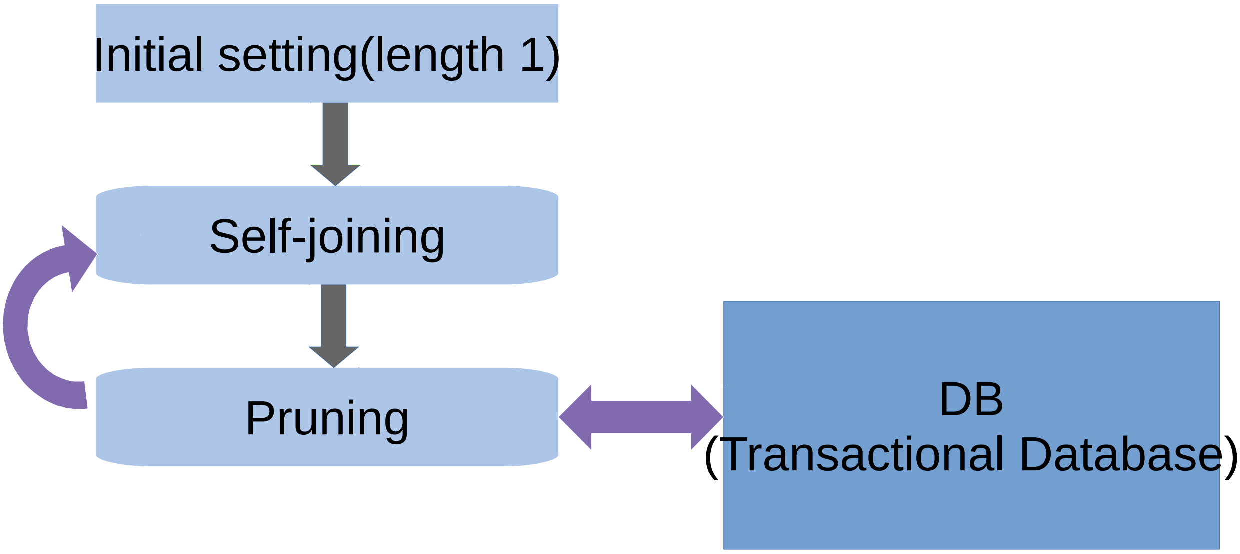
'Apriori' is a candidate generation and test approach used in frequent pattern and association rule mining. By using 'apriori pruning principle', which denotes if there is any infrequent item set then its super set should not be generated, it can reduce many candidates and thus perform efficiently.
In this assignment, I got a chance that help me to deeply think about data mining method by implementing 'apriori algorithm'.

2. Run-time environment

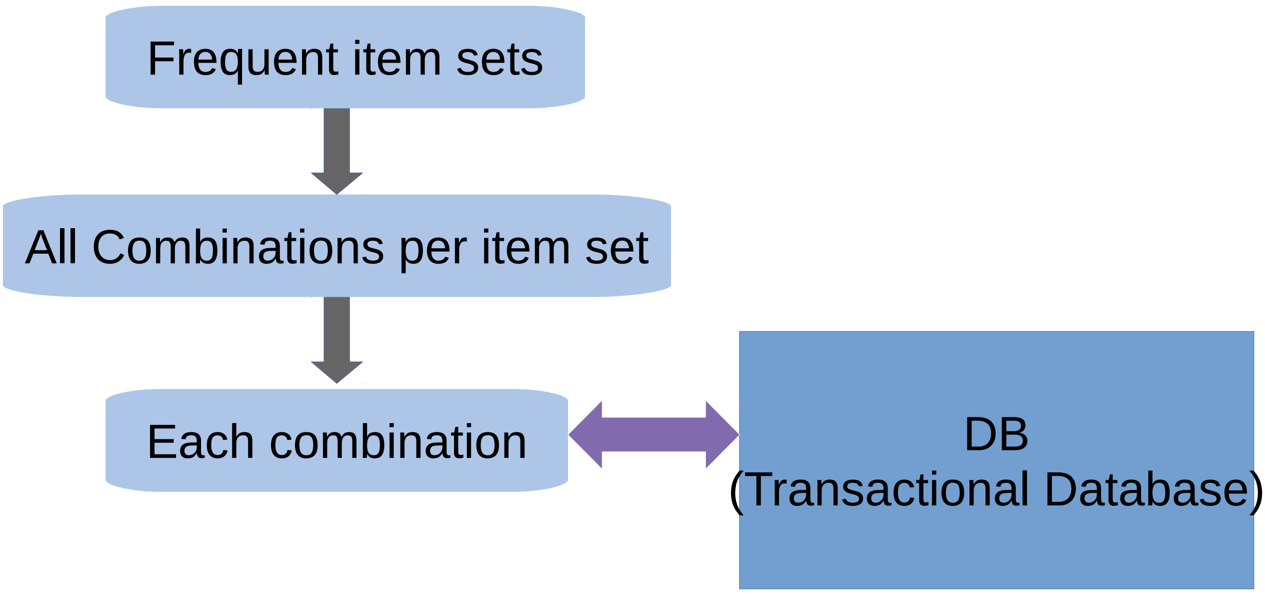
- OS: Ubuntu 18.04
- Language: Python 3.6
- Libraries:
    - Python sys module: for getting command-line arguments.
    - Python itertools module: for generating possible combinations in iterable object.

3. Summary of algorithm

'Apriori' algorithm consists of two stage repetition.
i) Generate candidates by length [Self-Joining]
ii) Prune candidates against data base [Pruning]
Self joining stage uses 'apriori pruning principle'. By using this, it can reduce huge amount of candidates which is infrequent and thus perform faster.
Pruning stage verifies candidates which generated in self-joining stage with minimum support. It needs scanning whole data base to calculate candidates' support.
This two stage repetition continues until there are no other candidates. Verified item sets are saved with its support value.
Also in this assignment, algorithm should generate proper association rules with its support and confidence. In this step, frequent item sets generate all possible combinations by each item. Additional scanning in whole data base is required in each combination. Association rules are saved in proper format.
Below is the flow chart of this algorithm.



[Figure1: generating frequent item sets]



[Figure2: generating association rules]

4. Analysis of code

i) Data Structures

I use mainly 3 data structures to store transactional database, candidates, and frequent item sets.

- table: list of sorted sets

'table' stores all transactional database which is read from input file. It stores each transaction as a set which can easily do self-joining by '|' operator.

ex)

```
[{14, 7}, {9}, {1, 2, 4, 5, 18}, {1, 2, 4, 7, 11, 13, 15, 16}, {16, 1, 2}, {6, 7, 9, 11, 12, 14, 15, 18, 19}, {2, 11, 4, 13}, {11, 13}, {1, 2, 3, 4, 7, 8, 16, 17, 19},
{9}, {10}, {1, 3, 4, 8, 9, 13, 16}, {0, 6, 8, 11, 16}, {3, 6, 8, 16, 17, 19}, {1, 2, 4, 9, 10, 13, 15, 18, 19}, {1, 4, 5, 6, 9, 10, 12, 13}, {1, 6, 10, 12, 13, 15, 16,
{7}, {17, 7}, {3, 5, 7, 8, 9, 13, 16, 18}, {8, 2}, {0, 2, 6, 8, 10, 11, 13}, {3, 4, 8, 10, 17, 19}, {1, 2, 10, 11, 14, 16}, {16}, {8, 19}, {0, 1, 4, 7, 8, 13, 14, 18,
 11, 13, 14, 15, 16, 19}, {2, 3, 4, 8, 9, 16, 18}, {3, 5, 7, 8, 13, 16}, {0, 3, 8, 16, 18}, {0, 1, 2, 9, 12, 16, 17, 18, 19}, {17}, {2, 6, 8, 11, 12, 13, 14, 15, 16},
 {0, 3, 8, 9, 10, 12, 16}, {8, 1}, {2, 3, 5, 6, 8, 9, 12, 15, 16, 17}, {1, 8, 9, 12, 16}, {1, 5, 9, 12, 14, 15}, {18}, {8, 16, 3, 19}, {5, 6, 8, 9, 13, 15, 16, 17}, {1,
 {1, 5, 8, 12, 13, 16}, {1, 2, 4, 6, 9, 10, 13, 15, 16}, {4, 5, 10, 12, 13, 14, 19}, {2, 5, 6, 7, 8, 11, 13, 14, 17, 18}, {4, 12, 13, 14, 19}, {7}, {3, 8, 10, 12, 13,
```

- cand_list: list of lists of sets

'cand_list' stores all candidates by its length. Each list has specific length of sets. These sets are pruned later.

ex)

```
[{1}, {2}, {3}, {5}]
[{1, 3}, {2, 3}, {2, 5}, {3, 5}]
[{2, 3, 5}]
```

- freq_list: list of dictionaries(key: tuple, value: float)

'freq_list' stores all frequent item sets. Each dictionary has specific length of tuple as a key. Each key's value is its support.

ex)

```
[{(1,): 50.0, (2,): 75.0, (3,): 75.0, (5,): 75.0}, {(1, 3): 50.0, (2, 3): 50.0, (2, 5): 75.0, (3, 5): 50.0}, {(2, 3, 5): 50.0}]
```

ii) Functions

I use 3 functions to perform self-joining, pruning, and generating association rules.

- prune

```
'''
    Feature: Prune candidates by support and length
    Arguments: cand(type: list[set])
    Return value type: dictionary(key: tuple, value: float]
'''
def prune(cand):
    # Make counting table
    cnt_table = {idx: 0 for idx, _ in enumerate(cand)}
    for transaction in table:
        for idx,itemset in enumerate(cand):
            # Check whether each transaction is superset of each itemset
            if transaction >= itemset:
                cnt_table[idx] += 1
    # Filter by minimum support
    ret = {tuple(cand[key]): val/len(table)*100 for key, val in cnt_table.items() if val>=min_sup}

    return ret

```

Feature: Prune candidates by support and length
Arguments: cand → list of sets
Return value: ret → dictionary(key: tuple, value: float)

In this function, validate process is executed.

First, create counting table by using given candidates list. I use dictionary as this table, key for index of each candidate and value for its counts.

Second, scan the whole data base to count each candidate. Nested for loop is used.

Finally, filter the candidates with minimum support. I use python filter built-in function to compute easily. However, set can’t be used as a key in dictionary, I convert its type set to tuple. Its support is stored as value.

- self_join

```python
'''
    Feature: Generate candidates by self-joining
    Arguments: itemdict(type: dictionary(key: tuple, value: float)), length(type: integer)
    Return value type: list[set]
'''
def self_join(itemdict,length):
    # Convert type: tuple->set
    itemsets = list(map(set,itemdict.keys()))
    # Generate candidates by set union operator
    ret = [itemsets[i]|itemsets[j] for i in range(len(itemsets)) for j in range(i+1,len(itemsets))]
    # Filter by length
    ret = list(filter(lambda x: len(x)==length, ret))
    # Remove duplicates
    ret = list(set(tuple(sorted(itemset)) for itemset in ret))
    # Convert type: tuple->set
    ret = list(map(set,ret))
    return ret
```

Feature: Generate candidates by self-joining
Arguments: itemdict → dictionary(key: tuple, value: float) / length → integer
Return value: ret → list of sets

In this function, self-joining process is executed.
First, convert argument's type tuple to set. By doing this, set operator '|'(union) can be used to create candidates easily.
Second, generate all possible candidates by using set union operator.
Third, filter generated candidates with specific length. I use python built-in filter function to compute easily.
Finally, remove duplicates using some python tricks. As 'set' data structure doesn't allow duplicates, I use 'set' to remove duplicates. By applying 'set' to sorted tuple, all possible duplicates are removed easily.

- generate_association_rule

```python
'''
    Feature: Gerneate association rules
    Arguments: freq_list(type: list[dictionary(key: tuple. value: float)])
    Return value type: list[string]
'''
def generate_association_rule(freq_list):
    ret = []
    for itemdict in freq_list:
        for itemset,val in itemdict.items():
            # Exception: length is less or equal than 1
            if len(itemset) <= 1:
                break
            # Save support value
            sup = val
            # Convert type: tuple->set
            itemset = set(itemset)
            for length in range(1,len(itemset)):
                # Generate all combinations
                total_combinations = combinations(itemset,length)
                for combination in total_combinations:
                    # Convert type: tuple->set
                    left = set(combination)
                    right = itemset - left
                    left_cnt=0
                    for transaction in table:
                        if transaction >= left:
                            left_cnt += 1
                    # Calculate confidence
                    conf = sup*len(table)/left_cnt
                    # Save result as string
                    ret.append("{}\t{}\t{:.2f}\t{:.2f}\n".format(str(left),str(right),sup,conf))
    return ret
```

Feature: Generate association rules
Arguments: freq_list → list of dictionaries(key: tuple, value: float)
Return value: ret → list of strings

In this function, association rules are generated.

First, make all possible combinations by using python itertools.combinations function. Each item set's support value is also stored.

Second, divide each combination into left side and right side. Notice that each side is converted to 'set' type for '-'operator(complement) and '>=' operator(is_superset).

Third, calculate confidence by scanning data base. Support and confidence are calculated as [# of (A&B) / # of total transaction] and [# of (A&b) / # of A] in association rule A → B so we can easily compute confidence by using 'support' and '# of A'.

Finally, each formatted string is stored in list. In this format, each side is enclosed in curly braces. Also, support and confidence is rounded in 2 decimal places.

iii) Code Flow

```python
if __name__ == "__main__":
    '''
        Initialize arguments
        [Type]
            min_sup: float(ratio)->integer(count)
            input_file: string
            output_file: string
    '''
    # Save command line argument
    min_sup, input_file, output_file = sys.argv[1:]
    # Convert type: string->float
    min_sup = float(min_sup)/100

    '''
        Read input file
        & Create transaction table
        [Type]
            table: list[sorted sets]
    '''
    with open(input_file, 'r') as f:
        table = [set(sorted(map(int,line.split()))) for line in f.readlines()]

    # Convert: ratio->count
    min_sup = len(table)*min_sup

    '''
        Generate candidates(Self-joining)
        & Prune candidates
        => Generate frequent item sets depending on its length
        [Type]
            itmes: set
            cand_list: list[list[set]]
            freq_list: list[dictionary(key:tuple, value=float(support))]
            max_length: integer
    '''
    # Create candidates: length 1
    items = {item_id for transaction in table for item_id in transaction}
    cand_list = [[{item_id} for item_id in items]]
    freq_list = [prune(cand_list[0])]
    max_length = 1
```

1. Save command-line arguments, modify minimum support's type
2. Create data base by reading input file, modify minimum support(Ratio to actual counts).
3. Create candidates (length 1)
4. Verify created candidates using 'prune' function (length 1)

Now, we are ready to get into loop.

```python
    # Create candidates depending on its length
    while True:
        cand_list.append(self_join(freq_list[-1],max_length+1))
        tmp = prune(cand_list[-1])
        if not tmp:
            break
        freq_list.append(tmp)
        max_length += 1

    '''
        Generate association rules
        [Type]
            association_list: list[string]
    '''
    association_list = generate_association_rule(freq_list)
    # Write results in output file
    with open(output_file, 'w') as f:
        for line in association_list:
            f.write(line)
```

5. Repeat creating candidates(using 'self_join' function) and verifying them(using 'prune' function) until there is no other candidates.
6. Generate association rules using 'generate_association_rule' function
7. Write the results in to output file

5. Instructions for execution

```
jsense@jsense:~/2020_ITE4005_2015004248/project_apriori$ python3 apriori.py 5 input.txt output.txt
jsense@jsense:~/2020_ITE4005_2015004248/project_apriori$ python3 apriori.py 50 sample_input.txt sample_output.txt
```

Type ‘python3 “minimum support” “input file name” “output file name”’ in terminal.

6. Test result

To verify code, I make simple sample file that can be easily figured out. This sample is in our lecture note. (Convert ‘A’,’B’,’C’… in to ‘1’,’2’,’3’…) I verified this output file by calculating its support and confidence manually.

```
{1}     {3}     50.00   100.00
{3}     {1}     50.00   66.67
{2}     {3}     50.00   66.67
{3}     {2}     50.00   66.67
{2}     {5}     75.00   100.00
{5}     {2}     75.00   100.00
{3}     {5}     50.00   66.67
{5}     {3}     50.00   66.67
{2}     {3, 5}  50.00   66.67
{3}     {2, 5}  50.00   66.67
{5}     {2, 3}  50.00   66.67
{2, 3}  {5}     50.00   100.00
{2, 5}  {3}     50.00   66.67
{3, 5}  {2}     50.00   100.00
```

After successful sample test, I tested actual input file. Here is the result.

```
{8, 3}  {16, 17}        5.80    22.48
{8, 17} {16, 3} 5.80    48.33
{16, 3} {8, 17} 5.80    23.02
{16, 17}        {8, 3}  5.80    44.62
{17, 3} {8, 16} 5.80    76.32
{8, 16, 3}      {17}    5.80    24.17
{8, 16, 17}     {3}     5.80    65.91
{8, 17, 3}      {16}    5.80    85.29
{16, 17, 3}     {8}     5.80    93.55
{8}     {16, 3, 13}     7.40    16.37
{16}    {8, 3, 13}      7.40    17.45
{3}     {8, 16, 13}     7.40    24.67
{13}    {8, 16, 3}      7.40    25.00
{8, 16} {3, 13} 7.40    24.50
{8, 3}  {16, 13}        7.40    28.68
{8, 13} {16, 3} 7.40    51.39
{16, 3} {8, 13} 7.40    29.37
{16, 13}        {8, 3}  7.40    53.62
{3, 13} {8, 16} 7.40    82.22
```

I got 1066 association rules in given input text file.

7. Reference

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. VLDB, 1994.