PROGRAMMING ASSIGNMENT #1 : APRIORI Algorithm

2020113200 Kyeong-A Kim

#1. Development Environment

- OS : Mac OS
- Language : Python 3.8.3
- Code Editor : Visual Studio Code, Juypyter notebook

#2. Summary of Algorithm

Apriori is an algorithm for frequent item set mining learning over relation database. It called Association Rule or Market basket analysis. Here is a method of this algorithm.

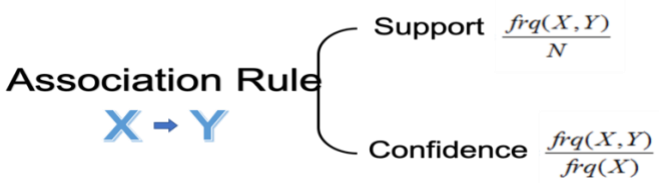
First, Scan DB once to get frequent 1-itemset. 1-itemset means that has only one itemset such as {A }, {B}, {C} etc..

Second, Count support that satisfied bigger than initial minimum support. Then, generate candidate itemsets of length (K+1) from frequent itemsets of length K. At this time, use Self-Joining to generate candidate.

Third, Test the candidates against DB to get K+1 Frequent item sets. It called Pruning deleting items that do not satisfy the minimum support.

Fourth, Calculate Support and Confidence for each Frequent itemset.

Last, The Important thing is that iterate when no frequent or candidate set can be generated.



*** Support :** probability that a transaction contains X U Y
*** Confidence** : conditional probability that a transaction having not only X but also Y

#3. Detail description of codes

Now, I'll introduce my algorithm step by step. I divided algorithm codes by functions.

1) Data Load

```
def load_data():
    global item_list # This is initial list which contain all of itemsets.
    item_list = []

    with open('input.txt', 'r') as f:
        lines = f.read().split('\n') # Read by a line-break
        for line in lines:
            line = line.split('\t') # Read by a space
            item_list.append(line)
    return item_list
```

This is Data Load Function that can read ‘input.txt’ file. Read by a line-break and a space.
I used ‘with open~’ logic instead of ‘read_csv’ for text file.

2) Initial Frequent Itemset

This function is making dictionary that counting the item numbers.

- Make a default dictionary.
- Count items line by line.
- And then pass next function to filter under minimum support.

```
def init_freq_set():
    global item_list
    localSet = defaultdict(int) # defaultdict : default of dictionary

    for line in item_list:
        for item in line:
            if item not in localSet:
                localSet[item] = 1
            else:
                localSet[item] += 1
    return pruning_min_sup(localSet)
```

3) Pruning(=Remove) Under Minimum Support

This function is to pick itemset which is larger than minimum support.

- Convert minimum support standard to minimum number of items because candidate[key] is also number of items. Finally, _itemSet contains itemsets which is larger than minimum cnt.
- Terminate when there is no more items in _itemSet.

```
def pruning_min_sup(candidate):
    global item_list
    cnt = min_support * len(item_list) # min_support = cnt/len(item_list)
    _itemSet = {key: candidate[key] for key in candidate.keys()
                if candidate[key] >= cnt}
#   print(_itemSet)

    if len(_itemSet) < 1: # Terminate when there is no more _itemSet
        print('Terminate')
        return _itemSet
    else:
        return _itemSet # Get itemSet which is bigger than minimum support.
```

4) Self-Joining Frequent itemset

This function is very important. Make a combination like union.

But, {1} -> {2} and {2} -> {1} are different. These support and confidence are quite different.

- If making combination of length is two, combine with previous frequent itemset (lengh=1).
- If making combination of length is over two, combine with previous frequent itemset and the more previous frequent itemset without overlap.

```python
# 4) Self-Joining Frequent itemset
def self_joining(length, prev_freq_set):
    join_list = list()

    if length == 2: #two candidates
        for item in itertools.combinations(prev_freq_set, length):
            # union of previous frequent itemsets.
            join_list.append(item)
        return list(map(set, join_list))

    else: #over three candidates
        for item_set in prev_freq_set:
            # Prevent dupli%cated frequent items.
            for item in item_set:
                if item not in join_list:
                    join_list.append(item)

        for item in itertools.combinations(join_list, length):
            # union of frequent itemsets over three candidates.
            join_list.append(item)
        candidate = list(map(set, join_list))

        return candidate
```

5) Pruning

- Compare K+1 candidate made by self-joining and K candidate.
- If all combination of K+1 candidate is in item list, append to _itemSet.
- If not, remove candidate. Finally, check whether candidate's support is over minimun support or not. Here is an example.

- L3 = {abc, abd, acd, ace, bcd}
- self-joining L3*L3 is {abcd, acde}.
- all combination of abcd is in L3 . It includes {abc, acd, bcd, abd}.
- But, all combination of acde is not in L3. Only include {acd, ace}. So, preclude acde element.

```python
# 5) Pruning
def pruning(length, prev_freq_set, candidate):
    global item_list
    _itemSet = dict()

    if length == 2 : #two candidates
        temp = list()
        for item in prev_freq_set:
            temp.append(list([item,])) # For append two digits itemset.
        prev_freq_set = temp

    else: #over three candidates
        join_list = list()
        for item in prev_freq_set:
            join_list.append(set(item))
        prev_freq_set = join_list
        # Both of list become previous freq itemset.

    for item_set in candidate:
        cnt = 0 # compare frequent set and previous frequent set.
        for item in list(itertools.combinations(item_set, length - 1)):
            if length == 2:
                item = list(item)
            else:
                item = set(item)

            if item not in prev_freq_set:
                break
            cnt = cnt + 1

        if cnt == length:
            _itemSet[tuple(item_set)] = 0

    for key in _itemSet.keys(): # For next frequent set.
        for line in item_list:
            if set(key) <= set(line):
                _itemSet[key] = _itemSet[key] + 1

    return pruning_min_sup(_itemSet)
```

6) Association rule

Calculate support, confidence about all frequent itemsets in _itemSet.
Iterate using while conditional when there is no frequent itemset.

```
# 6) association_rule
# calculate support, confidence about all frequent-itemset.
def association_rule(length, _itemSet):
    for item_set, freq in _itemSet.items():
        frequent_set_len = length
        while frequent_set_len > 1: #iterate when no frequent itemset
            combi = list(itertools.combinations(item_set,
                                                frequent_set_len-1))
            for item in combi:
                item = set(item)
                remain = set(item_set) - set(item)
                # difference for counterpart combi
                support = freq / len(item_list) * 100

                cnt_item = 0
                for line in item_list:
                    if set(line) >= item:
                        cnt_item = cnt_item + 1

                confidence = freq / cnt_item * 100

                # string -> int
                item = set(map(int, item))
                remain = set(map(int, remain))

                # print format : rounded to two decimal places
                line = str(item) + '\t' + str(remain) + '\t' +
                        str('%.2f' % round(support, 2)) + '\t' +
                            str('%.2f' % round(confidence, 2)) + '\n'
                save_result(line)
                print(line)
            frequent_set_len -= 1
```

7) Save result

```
# 7) Save output
def save_result(line):
    with open(sys.argv[3], 'a') as f: # Save fourth argument name.
            f.write(line)
```

#4. Instructions for compiling

- Open a new command prompt.
- Write this following command :
  python {file name} {min_support} {input.txt} {output.txt}

#5. Result (Compiling source code)

Setting minimum support = 2