Data Mining 2020 HW1 - Association Rules

— P76094321 盧宥霖

Apriori algorithm

Since the support of an itemset never exceeds the support of its subsets, we can reduce computing time by avoiding supersets af an infrequent itemset. The algorithm is implemented as shown below.

1. Scan the database and generate C1

```
def first_scan(dataset):
    C1 = {}
    for transaction in dataset:
        for item in transaction:
            item = tuple([item]) # Make it tuple in order to use it as a key of dict
            count = C1.get(item)
            if count:
                C1.update({item: count+1})
            else:
                C1.update({item: 1})
    return C1
```

2. Find frequent item sets Lk

After we get the candidates, filter out the unfrequent(which is support<minimum support)

```
def find_Lk(candidates, min_support):
    Lk = candidates.copy()
    for key in candidates:
        if Lk[key] < min_support:
            del Lk[key]
    return Lk
```

3. Find all possible combinations, generate candidates C(K+1)

```
def find_Ck(Lk, dataset):
    Ck = {}
    # List itemset in Ck
    for i in Lk:
        for j in Lk:
            # Iterate through all possible combinations
            if len(set(i) - set(j)) == 1:
                Ck.update({tuple(set(i).union(set(j))): 0})
    if len(Ck) == 0:
        return Ck
    # Scan database for support of each itemset
    for data in Ck:
        count = 0
        for transaction in dataset:
            if set(data) <= set(transaction):
                count += 1
        Ck.update({tuple(data): count})
    return Ck
```

Full apriori algorithm

Keep running the above functions and record frequent itemsets.

```
def apriori(dataset, min_support):
    candidates = first_scan(dataset)
    frequent_itemsets = []
    while candidates:
        Lk = find_Lk(candidates, min_support)
        frequent_itemsets.append(Lk)
        candidates = find_Ck(Lk, dataset)
```

The Apriori algorithm can sure tackle the task, but it is time consuming since multiple database scans are required during the process. Although it doesn't affect much facing small datasets, however, in real world applications where the size of data is often big, mining long patterns with apriori algorithm is sure a pain. Thus, we should find a way to avoid repetitve database scan and candidate generation.

Hash tree optimization

To reduce time wasted on the scanning process, we can build a hash tree instead when we are trying to get the count of each possible candidate. This can be done by changing the bottom half of find_Ck() into the following:

```
    # Build tree
    root = {}
    cur = root
    height = 0
    for candidate in Ck.keys:
        height = 0
        for item in candidate:
            if cur.get(item)==None:
                cur.update({item: {}})
                cur = cur[item]
            else:
                cur = cur[item]
            height += 1
        cur = root
    # Run thru the tree for each transaction
    for transaction in dataset:
        lst = list(combinations(transaction, height))
        for cand in lst:
            traverse_hash_tree(root, cand, [], {})
```

Here, I implemented the hash-tree-like structure using python built-in dictionaries. Then, for each transaction in the dataset, we can see if it matches the nodes in the hach tree, adding the counts to the matching nodes. The implementation looks like this:

```
def traverse_hash_tree(root, items, path, count_dict):
    if bool(root)==False: # End of recursive, push path as ans
        if count_dict.get(path):
            count_dict.update({path}:count_dict[path]+1)
        else:
            count_dict.update({path}:1)
    for item in root: # keep looping
        if item in items:
            traverse_hash_tree(root[item], items.remove(item), path.append(item), coun
        else:
            return
```

Now, we finished optimizing the Apriori algorithm. Later, we will try to avoid candidate generation in order to further improve.

FP-Growth

FP Tree construction

Scan the database and construct our fp tree. This is, in comparison to Apriori algorithm which involves multiple db scans, the only one time we need to scan the database.

Step 1. Construct frequency 1-item list

Find frequent 1-item, sorted items in frequency descending order by scanning DB.

```
def construct_frequency_list(dataset, min_support):
    items = {}
    sorted_freq_items = {}
    for transaction in dataset:
        for item in transaction:
            count = items.get(item)
            if count:
                items.update({item: count+1})
            else:
                items.update({item: 1})
    freq_items = items.copy()
    for key in items:
        if freq_items[key] < min_support:
            del freq_items[key]
    sorted_freq_items = sorted(freq_items.items(), key=lambda x: (-x[1], x[0]))
    sorted_freq_items = dict(sorted_freq_items)
    #print(sorted_freq_items)
    return sorted_freq_items
```

Step 2. Construct our itemset sorted by their frequency

```
def construct_sorted_itemset(dataset, sorted_freq_items):
    ordered = []
    for transaction in dataset:
        tmp=[]
        for item in sorted_freq_items.keys():
            if item in transaction:
                tmp.append(item)
        ordered.append(tmp)
    #print(ordered)
    return ordered
```

Step 3. Insert each transaction into fp tree

First, define our tree node stucture as following:

```
class Node:
    def __init__(self, item):
        self.item = item
        self.value = 1
        self.children = []
        self.parent = None

    def find_child(self, item):
        for child in self.children:
            if item == child.item:
                return child
        return None
```

Then, we can define our fp tree consisted by the defined nodes:

```
class FPTree:
    def __init__(self, sorted_freq_items):
        self.root = Node("root")
        self.root.value = None
        self.header_table = {}

    def insert(self, transaction):
        current_node = self.root
        for item in transaction:
            next_node = current_node.find_child(item)
            if next_node:
                current_node = next_node
                current_node.value += 1
            else:
                new_node = Node(item)
                new_node.parent = current_node
                current_node.children.append(new_node)
                current_node = new_node
                header_list = self.header_table.get(item)
                if header_list:
                    header_list.append(new_node)
                else:
                    self.header_table.update({item: [new_node]})
```

After we have our custom fp tree data structure, we can insert the transactions to build the tree:

```
tree = fp_tree.FPTree(itemset)
for transaction in itemset:
    tree.insert(transaction)
```

FP Growth

After the tree is built, we can start “growing”.

Step 1. Construct conditional pattern base

For each frequent 1-itemset, we find the set of prefix paths in FP-tree cooccurring with the suffix pattern.

```
def find_conditional_pattern(header_table, sorted_freq_items):
    conditional_pattern_base = {}
    for item in sorted_freq_items:
        patterns_list = []
        node_list = header_table.get(item)
        # Find the path of nodes in header table
        for node in node_list:
            cur = node.parent
            path = []
            # Loop til the root
            while cur:
                if cur.item != 'root':
                    path.append(cur.item)
                cur = cur.parent
            path.reverse()
            # Add current path and its weight to the
            patterns_list.append({tuple(path): node.value})
        conditional_pattern_base.update({item: patterns_list})
    # print(conditional_pattern_base)
    return conditional_pattern_base
```

Step 2. Recursively construct corresponding conditional FP-tree

```
def fp_tree_growth(prefix, value, min_support, ordered, patternList):
    itemsets = []
    # print(value)
    for item in value:
        # print(list(item.keys())[0])
        count = int(list(item.values())[0])
        while count:
            itemsets.append(list(item.keys())[0])
            count += -1
    freq_list = construct_frequency_list(itemsets, min_support)
    itemset = construct_sorted_itemset(dataset, freq_list)
    tree = fp_tree.FPTree(itemset)
    for transaction in itemset:
        tree.insert(transaction)
    if not tree.header_table:
        # print(patternList)
        return patternList
    conditional_pattern_base = find_conditional_pattern(tree.header_table,freq_list)
    for key, value in conditional_pattern_base.items():
        count = 0
        for item in value:
            count = count + int(list(item.values())[0])
        pattern = []
        item = []
        item.append(key)
        pattern.append(prefix + item)
        pattern.append(count)
        patternList.append(pattern)
        fp_tree_growth(prefix + item, value, min_support, ordered, patternList)

    return patternList
```

Full algorithm

```
dataset = preprocessor.preprocess()
min_support = 500
ptn = {}
freq_list = construct_frequency_list(dataset, min_support)
itemset = construct_sorted_itemset(dataset, freq_list)
tree = fp_tree.FPTree(itemset)
for transaction in itemset:
    tree.insert(transaction)
conditional_pattern_base = find_conditional_pattern(tree.header_table,freq_list)
for key, value in conditional_pattern_base.items():
    if value:
        patternList = fp_tree_growth(list(), value, min_support, itemset, list())
        ptn.update({key: patternList})
print(ptn)
```

Association Rule Generation

After we get the frequent patterns using apriori algorithm or fp-growth, we can then use them to generate association rules. Here, we use the `mlxtend` package to do that.

```
from mlxtend.frequent_patterns import association_rules
# Get frequent pattern using either apriori or fp-growth here
rules = association_rules(frequent_patterns ,metric='confidence', min_threshold=0.5)
print(rules[['antecedents', 'consequents', 'support', 'confidence']])
```

The rules generated on the example dataset generated by IBM generator will be shown in the next section.

Results & Comparison

To compare the difference between the two algorithms, I used the python built-in function `time.process_time()` to compute CPU time and the `psutil` package to investigate memory usage. The tests are run on the IBM generator dataset, making it easier to test on different sizes of data.
The results are shown below:

Rules generated

Here, we test the results of the 2 different algorithms, and we get the exact same frequent patterns. Thus, we can be sure that the outcome does not differ. Here, we set min_support = 0.3

Frequent Patterns

{(8,): 421, (36,): 508, (38,): 642, (47,): 326, (61,): 311, (62,): 309, (63,): 622, (69,): 543, (83,): 314, (9,): 303, (11,): 350, (14,): 308, (17,): 387, (39,): 326, (40,): 305, (43,): 301, (80,): 310, (81,): 340, (85,): 360, (87,): 605, (29,): 323, (3,): 384, (48,): 455, (28,): 405}, {(36, 38): 327, (36, 63): 329, (36, 87): 313, (38, 63): 406, (69, 38): 345, (38, 87): 412, (69, 63): 352, (87, 63): 384, (69, 87): 327, (63, 87): 384}

Association rules

#	antecedents	consequents	support	confidence
0	(36)	(38)	0.327	0.643701
1	(38)	(36)	0.327	0.509346
2	(36)	(62)	0.329	0.647638
3	(62)	(36)	0.329	0.528939
4	(36)	(86)	0.313	0.616142
5	(86)	(36)	0.313	0.517355
6	(62)	(38)	0.406	0.652733
7	(38)	(62)	0.406	0.632399
8	(68)	(38)	0.345	0.635359
9	(38)	(68)	0.345	0.537383
10	(86)	(38)	0.412	0.680992
11	(38)	(86)	0.412	0.641745
12	(68)	(62)	0.352	0.648250
13	(62)	(68)	0.352	0.565916
14	(86)	(62)	0.384	0.634711
15	(62)	(86)	0.384	0.617363
16	(68)	(86)	0.327	0.602210
17	(86)	(68)	0.327	0.540496

Time & memory analysis

Total execution time

	ntrans=0.1 nitems=0.01	ntrans=1 nitems=1
Apriori	0.0165 seconds	168.02 seconds
FP Growth	0.0391 seconds	2.02 seconds

Memory usage

	ntrans=0.1 nitems=0.01	ntrans=1 nitems=1
Apriori	24KB	2964KB
FP Growth	4KB	4904KB

With small datasets, we can see that the difference in performance between the 2 algorithm is rather small, with Apriori even faster. The reason should be that the cost of each database scan is low, making candidate generation an easy task. However, with the dataset large, FP-Growth stands out, finishing the task in less time. On the memory usage perspective, FP-Growth eats up more space building the trees.

Result on real world dataset

Now, let's try mining on real world datasets. Here, the Car Evaluation dataset on UC Irvine Machine Learning Repository is chosen. This dataset is originally used for car acceptability classification, including features of buying price, price of the maintenance, number of doors, capacity in terms of persons to carry, the size of luggage boot, and estimated safety of the car, followed by each car's acceptability label. With minimum support set to 0.4 and confidence level 0.5, we can get the following association rules:

	antecedents	consequents	support	confidence
0	(unacc)	(2)	0.438657	0.626446
1	(2)	(unacc)	0.438657	0.877315
2	(med)	(high)	0.430556	0.574074
3	(high)	(med)	0.430556	0.688889
4	(unacc)	(high)	0.408565	0.583471
5	(high)	(unacc)	0.408565	0.653704
6	(low)	(med)	0.430556	0.688889
7	(med)	(low)	0.430556	0.574074
8	(unacc)	(low)	0.456019	0.651240
9	(low)	(unacc)	0.456019	0.729630
10	(unacc)	(med)	0.502315	0.717355
11	(med)	(unacc)	0.502315	0.669753

We can see that the acceptance might be related to the number of doors and its price. For example, with only 2 doors, the car is possibly unaccepted.

Discussion - What are rules with:

- High support & High confidence
  There are often less rules found in high support & confidence setting. However, the rules in this category are more frequently seen in the dataset and could possibly provide useful insights on the data.
- High support & Low confidence
  Surprisingly, the result in this category is quite similar to the above settings. In my opinion, the reason might be that rules with high support usually also have high confidence, decreasing the importance of confidence level under high minimum support setting
- Low support & High confidence
  We can find a lot more rules in comparison to the above 2 settings. However, rules with low support probably aren't representative of the whole data.
- Low support & Low confidence
  Lots of rules generated in this setting. However, most of them are useless when analyzing the data.