

Apriori

Environment & How to run

OS: Mac OS, Windows 10

Language: Python 3.9

Run with the command:

```
// In the Apriori folder.  
  
python apriori.py [minimum_support] [input_file_name] [output_file_name]  
  
Ex) python apriori.py 5 input.txt output.txt
```

Input file format

```
[item_id]\t[item_id]\n  
[item_id]\t[item_id]\t[item_id]\t[item_id]\t[item_id]\n  
[item_id]\t[item_id]\t[item_id]\t[item_id]\n  
...
```

- A row is a transaction.
- item_id is a numerical value.
- There is no duplication of items in each transaction.

Ex)

```
7    14  
9  
18 2  4  5  1
```

```

1  11 15 2  7  16 4  13
2   1 16
15 7  6  11 18 9  12 19 14
11 2  13 4
...

```

Output file format

```
[item_set]\t[associative_item_set]\t[support(%)]\t[confidence(%)]\n
```

```
[item_set]\t[associative_item_set]\t[support(%)]\t[confidence(%)]\n
```

...

Ex)

```

{14} {7} 7.60 29.69
{7} {14} 7.60 31.67
{9} {14} 8.60 30.94
{14} {9} 8.60 33.59
{1} {14} 8.20 27.52
{14} {1} 8.20 32.03
...
{8} {16,11} 8.20 18.14
{11} {16,8} 8.20 29.93
{16,8} {11} 8.20 27.15
{16,11} {8} 8.20 67.21
...

```

- Will use braces to represent item sets: {[item_id],[item_id],...}

How it works

Mining frequent pattern

1. Given input data, counting from the transaction DB, make a length 1 candidate set.
2. From the candidate set, eliminate elements under minimum support to make a length 1 frequent pattern set.

3. Take all possible two-element combinations from length 1 frequent patterns to make a length 2 candidate set.
4. Iterating transaction from transaction DB, from the length 2 candidate set, take each element and count it if it is in the transaction.
5. From the length 2 candidate set, Eliminate elements under minimum support. then we can get a length 2 frequent pattern set.
6. Repeat {1 - 5} till an n-th frequent pattern set does not appear.
7. Union all the frequent pattern sets from length 1 to n.

```
# run()
# ...
transaction_db = []
candidate_patterns_1 = {} # candidate_k means Candidate item set of size k

for transaction in csv_reader:
    transaction = frozenset(map(lambda x: int(x), transaction))

    for item in transaction:
        frozenset_item = frozenset({item})
        if candidate_patterns_1.get(frozenset_item) is None:
            candidate_patterns_1[frozenset_item] = 1
        else:
            candidate_patterns_1[frozenset_item] += 1
    transaction_db.append(transaction)

support_to_count = ceil(transaction_db.__len__() * support)

frequent_patterns_1 = dict(filter(lambda x: x[1] >= support_to_count, candidate_patterns_1.items()))

total_frequent_patterns = apriori(transaction_db, frequent_patterns_1, support_to_count)
association_rules = mine_association_rule(transaction_db, total_frequent_patterns)
create_txt(csv_writer, association_rules)
# ...
```

- Used frozenset type to use it as the key of dictionary type (which takes only immutable objects as a key).
- To reduce DB scan, made the length 1 candidate set whilst csv_reader reads txt file.

```

def apriori(transaction_db, frequent_patterns_1, support_to_count):
    candidate = {}
    current_frequent_patterns = frequent_patterns_1
    total_frequent_patterns = []

    trial = 1

    while True:
        for combination in combinations(current_frequent_patterns.keys(), 2):
            combination_set = reduce(frozenset.union, combination)

            if combination_set.__len__() != trial + 1:
                continue

            is_candidate = True

            if trial != 1:
                for sub_combination in combinations(combination_set, trial):
                    sub_combination_set = frozenset(sub_combination)

                    flag = False

                    for frequent_pattern_set in current_frequent_patterns:
                        if sub_combination_set == frequent_pattern_set:
                            flag = True
                            break

                    if not flag:
                        is_candidate = False

            if is_candidate:
                candidate[combination_set] = 0

        for item_set in candidate: # calculate support
            for transaction in transaction_db:
                flag = True
                for item in item_set:
                    if item not in transaction:
                        flag = False
                        break
                if flag:
                    candidate[item_set] += 1

        current_frequent_patterns = dict(filter(lambda x: x[1] >= support_to_count, candidate.items()))
        total_frequent_patterns.extend(list(current_frequent_patterns.items()))
        candidate.clear()

        trial += 1

    if len(current_frequent_patterns) == 0:
        break

```

```
return total_frequent_patterns
```

- Skipping trial 1st lap, reduced the number of n-th candidates by checking whether **the subset of combination_set** exists in the previous n-1th frequent pattern set.
 - if the subset does not exist, by apriori pruning principles, the **target combination set** cannot be frequent.

Mining association rules

1. Take one frequent pattern element from the union set.
2. Make all possible combinations from the selected frequent pattern.
i.e.) nC1, nC2, nC3, ... , nCn-1
3. With each combination, make $p \rightarrow q$ relationship.
 - p is the set of selected objects
 - q is the complement of p
4. Iterating transaction from transaction DB, count $p \rightarrow q$ relationship to calculate confidence.
5. With the support and confidence, make association rule.
6. Repeat {1 - 5} till we iterate all the elements in the union set.

```
def mine_association_rule(transaction_db, total_frequent_patterns):
    association_rules = []

    for frequent_pattern in total_frequent_patterns:
        for given_event_item_count in range(1, len(frequent_pattern[0])):
            frozenset_frequent_pattern = frequent_pattern[0]
            for given_event in combinations(frozenset_frequent_pattern, given_event_item_count):
                given_event_in_transaction_count = 0
                confidence_count = 0
                given_event_set = frozenset(given_event)
                complement_of_given_event = frozenset_frequent_pattern.difference(given_event_set)

                for transaction in transaction_db:
                    flag = True
```

```

        for element in given_event:
            if element not in transaction:
                flag = False
                break
        if flag:
            given_event_in_transaction_count += 1
            confidence_flag = True
            for element in complement_of_given_event:
                if element not in transaction:
                    confidence_flag = False
                    break
            if confidence_flag:
                confidence_count += 1

        confidence = (confidence_count / given_event_in_transaction_count) * 100
        support = (frequent_pattern[1] / transaction_db.__len__()) * 100
        association_rules.append((given_event, tuple(complement_of_given_event), support, confidence))

    return association_rules

```

Creating output file

```

def create_txt(csv_writer, association_rules):
    for association in association_rules:
        p = f'{{{",".join(map(lambda x: str(x), association[0]))}}}'
        q = f'{{{",".join(map(lambda x: str(x), association[1]))}}}'

        support = f'{association[2]:.2f}'
        confidence = f'{association[3]:.2f}'


        csv_writer.writerow([p, q, support, confidence])

```

- Used 'sth'.join({iterable}) considering string concatenation performance issue.

What is the most efficient string concatenation method in python?

As per John Fouhy's answer, don't optimize unless you have to, but if you're here and asking this question, it may be precisely because you have to. In my case, I needed assemble some URLs from string

 <https://stackoverflow.com/questions/1316887/what-is-the-most-efficient-string-concatenation-method-in-python>

