CSE 601
Data Mining & Bioinformatics
Report
Association Analysis

Submitted By:

Mohammed Kamran Syed (msyed3)

Shreya Bhargava (shreyabh)

Objective:

Implementing Apriori Algorithm: Generation of frequent itemsets whose support is more than given minimum threshold support value.
Association: Generation of association rules from the given frequent item sets. The rules are generated according to a specified query.

Introduction:

Apriori Algorithm:

Apriori algorithm uses frequent itemsets to generate association rules. It is based on the concept that a subset of a frequent itemset must also be a frequent itemset. Frequent Itemset is an itemset whose support value is greater than or equal to a threshold value (min support). Support can be calculated as follows:

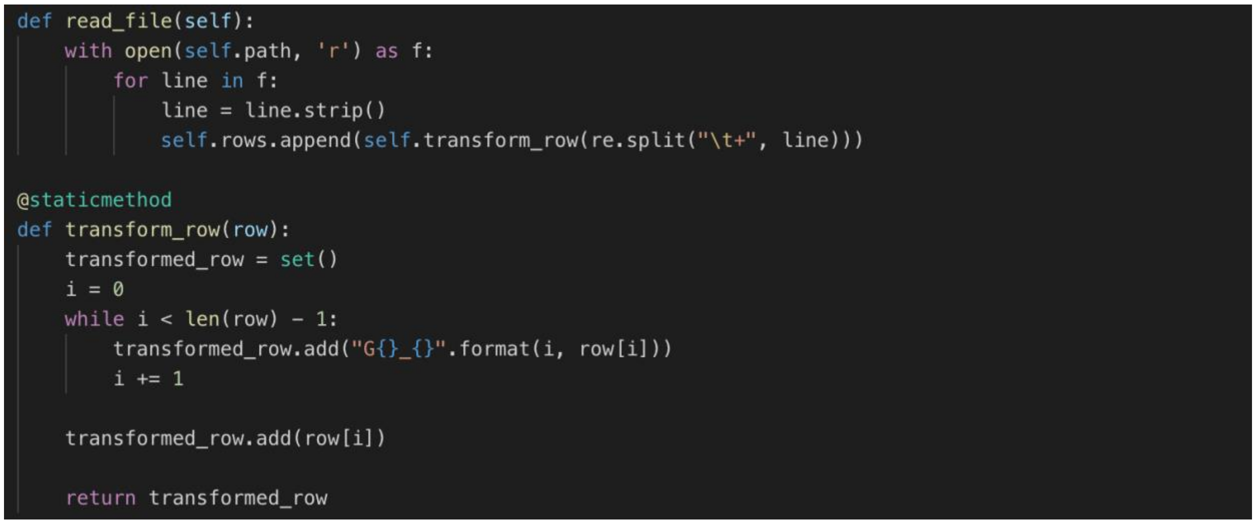
Support = Total no. of rows in which the given itemset was present / Total no. of rows

Or simply put, fraction of transactions that contain the given itemset.

"If an itemset is frequent then all of its subsets will be also frequent". Therefore, by using anti-monotone we property we can conclude that "If an item set is infrequent then all of its supersets will be also infrequent". Therefore, when we create combinations of data items then we do not need to compute support for all of the combinations. Instead, when we create combinations of data items, then for each level of itemset lattice we can check the support count. If any of the nodes of the itemset lattice has a support below the given threshold, then we can eliminate all the supersets of that node.

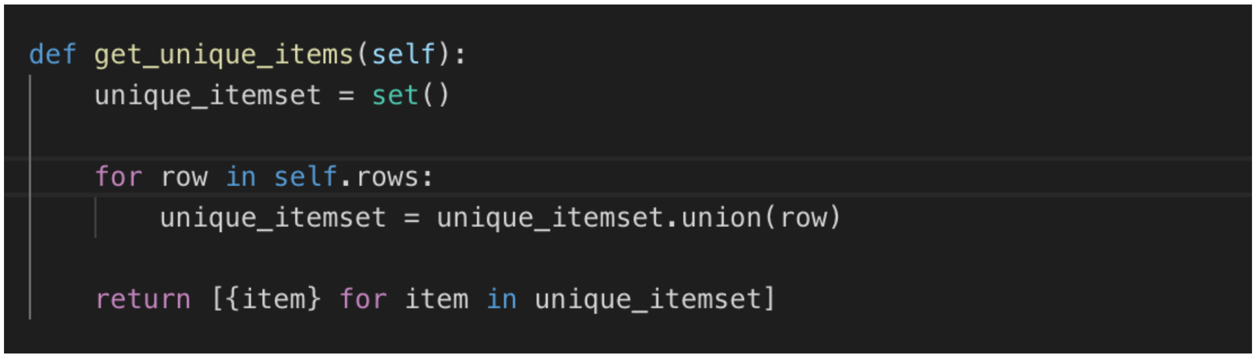
Steps:

- Read the data file from the "Data" directory and Column Numbers to the all the values except for the last column. Last column values are added without column numbers.



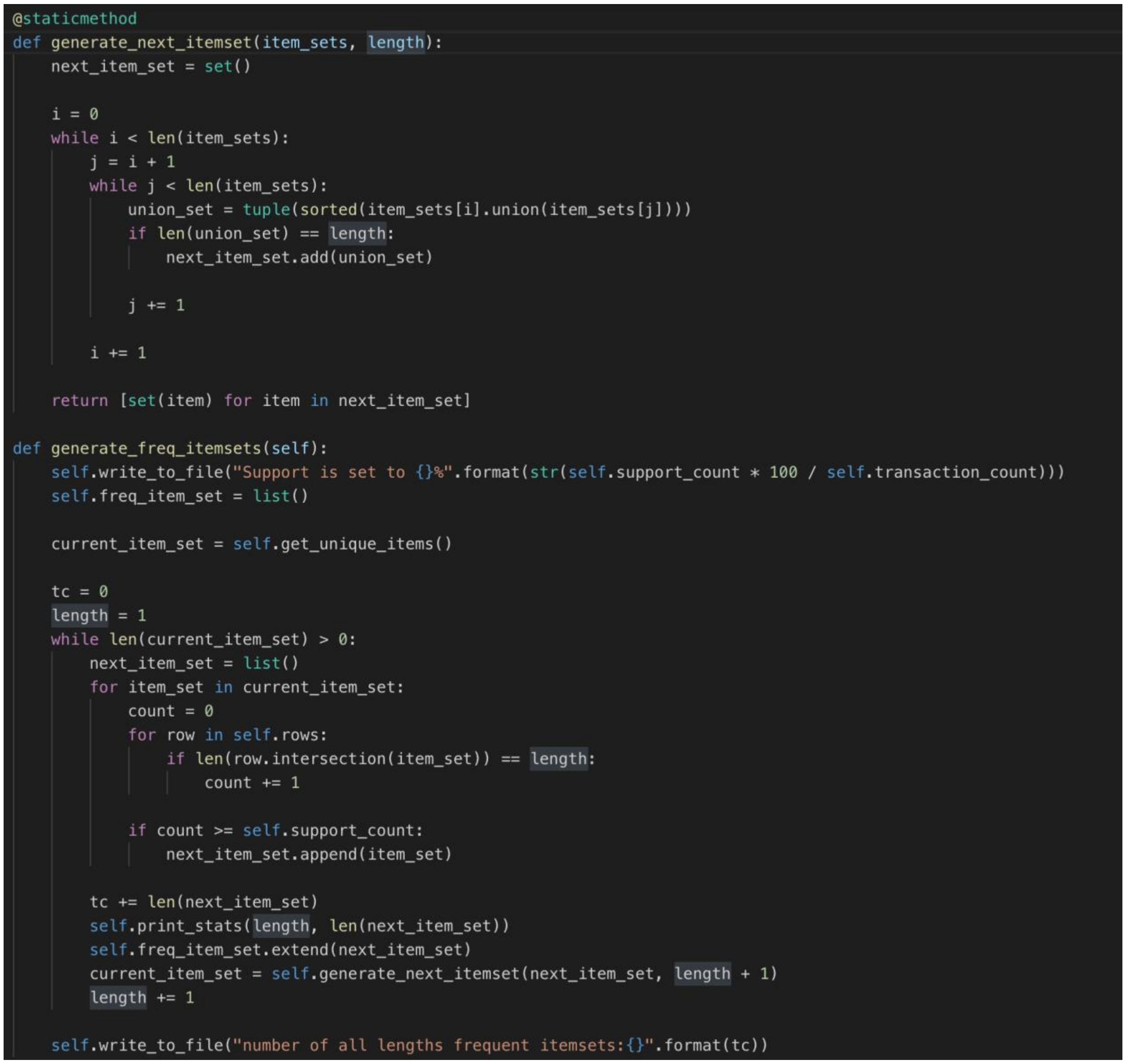
```python
def read_file(self):
    with open(self.path, 'r') as f:
        for line in f:
            line = line.strip()
            self.rows.append(self.transform_row(re.split("\t+", line)))

@staticmethod
def transform_row(row):
    transformed_row = set()
    i = 0
    while i < len(row) - 1:
        transformed_row.add("G{}_{}".format(i, row[i]))
        i += 1

    transformed_row.add(row[i])

    return transformed_row
```

- Generate all the unique itemsets from the data obtained above.



```python
def get_unique_items(self):
    unique_itemset = set()

    for row in self.rows:
        unique_itemset = unique_itemset.union(row)

    return [{item} for item in unique_itemset]
```

- Now create combinations of each itemset level by level and eliminate those whose support count falls below the set support threshold. (Pruning the infrequent itemsets makes the algorithm is lot faster)



```python
@staticmethod
def generate_next_itemset(item_sets, length):
    next_item_set = set()

    i = 0
    while i < len(item_sets):
        j = i + 1
        while j < len(item_sets):
            union_set = tuple(sorted(item_sets[i].union(item_sets[j])))
            if len(union_set) == length:
                next_item_set.add(union_set)

            j += 1

        i += 1

    return [set(item) for item in next_item_set]

def generate_freq_itemsets(self):
    self.write_to_file("Support is set to {}%".format(str(self.support_count * 100 / self.transaction_count)))
    self.freq_item_set = list()

    current_item_set = self.get_unique_items()

    tc = 0
    length = 1
    while len(current_item_set) > 0:
        next_item_set = list()
        for item_set in current_item_set:
            count = 0
            for row in self.rows:
                if len(row.intersection(item_set)) == length:
                    count += 1

            if count >= self.support_count:
                next_item_set.append(item_set)

        tc += len(next_item_set)
        self.print_stats(length, len(next_item_set))
        self.freq_item_set.extend(next_item_set)
        current_item_set = self.generate_next_itemset(next_item_set, length + 1)
        length += 1

    self.write_to_file("number of all lengths frequent itemsets:{}".format(tc))
```

- Write the results to an output file in “Output” directory.

Association Rule Generation:

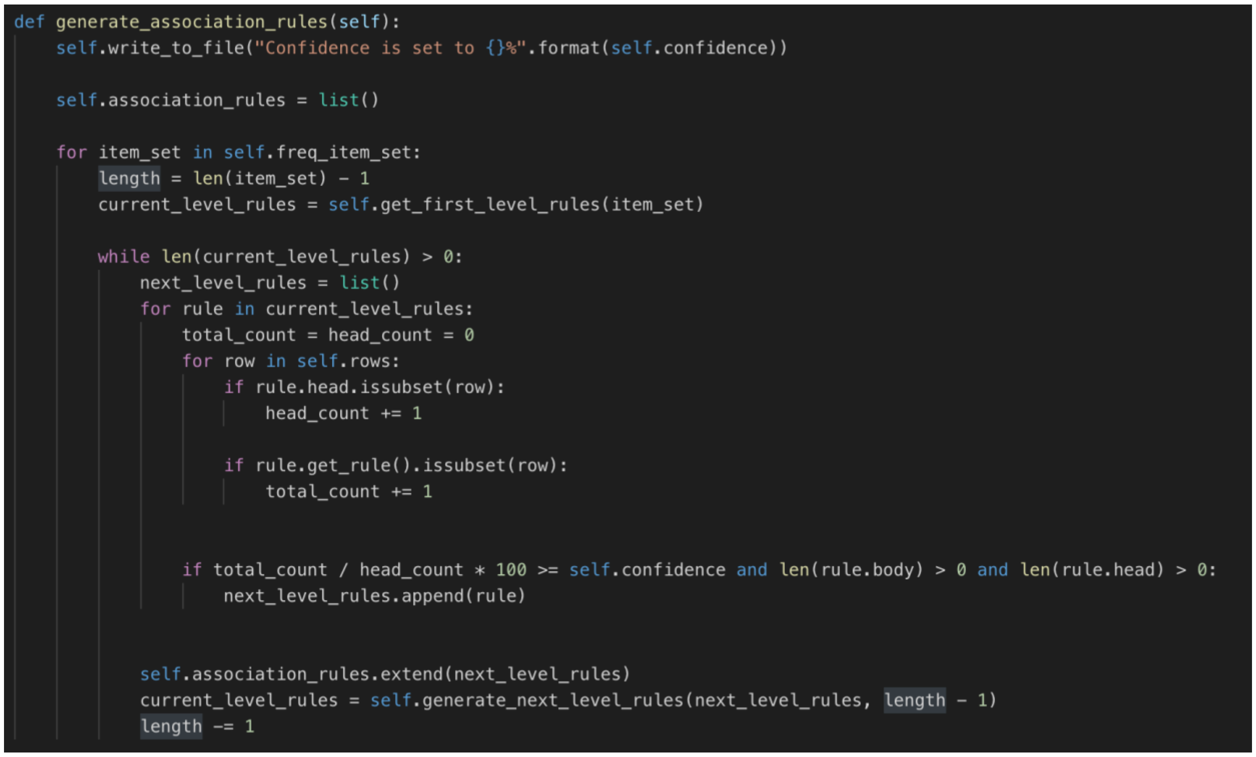
Rules are formed by binary partition of each itemset. Just like the anti-monotone property of support, confidence of rules generated from the same itemset also follows an anti-monotone property. It is anti-monotone with respect to the number of elements in consequent. In apriori algorithm, we try to prune the rules that do not satisfy the minimum confidence requirement. For a given rule X -> Y, Confidence of the rule is calculated as follows:

Confidence = No. of rows that contain X + Y / No. of rows that contain X

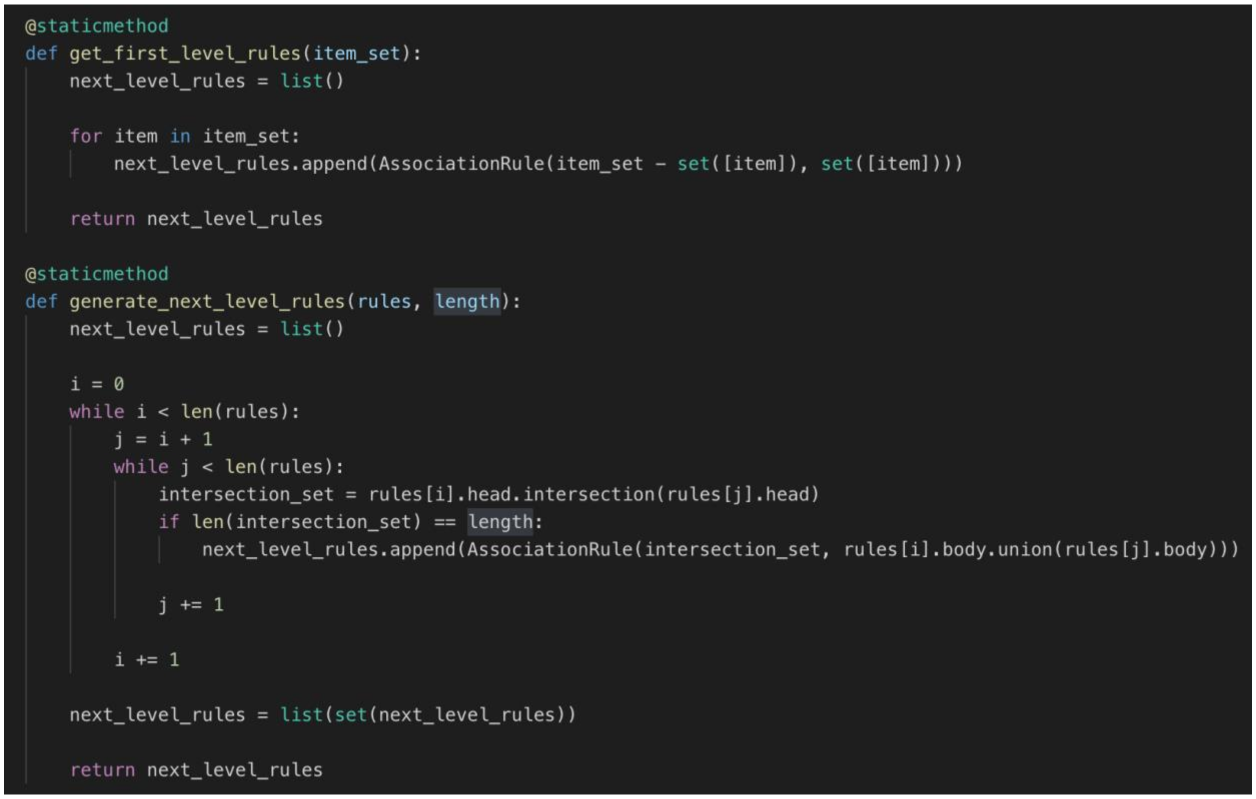
Or simply put, measures how often items in Y appears in transactions that contain X

Steps:

- From the frequent itemsets, we create the association rule lattice level by level.

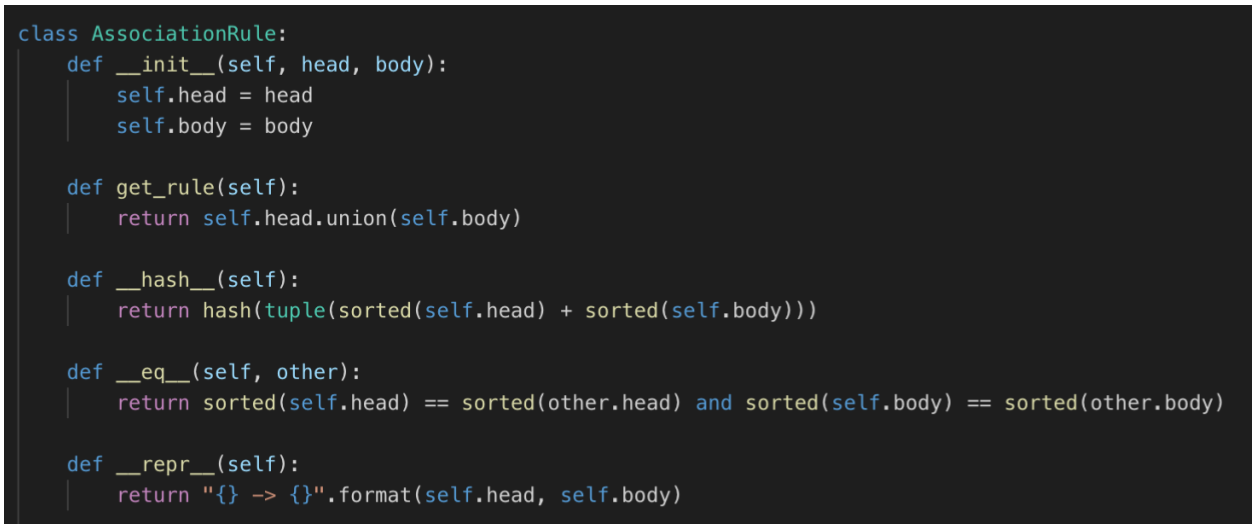


```python
def generate_association_rules(self):
    self.write_to_file("Confidence is set to {}%".format(self.confidence))

    self.association_rules = list()

    for item_set in self.freq_item_set:
        length = len(item_set) - 1
        current_level_rules = self.get_first_level_rules(item_set)

        while len(current_level_rules) > 0:
            next_level_rules = list()
            for rule in current_level_rules:
                total_count = head_count = 0
                for row in self.rows:
                    if rule.head.issubset(row):
                        head_count += 1

                    if rule.get_rule().issubset(row):
                        total_count += 1


                if total_count / head_count * 100 >= self.confidence and len(rule.body) > 0 and len(rule.head) > 0:
                    next_level_rules.append(rule)


            self.association_rules.extend(next_level_rules)
            current_level_rules = self.generate_next_level_rules(next_level_rules, length - 1)
            length -= 1
```



```python
@staticmethod
def get_first_level_rules(item_set):
    next_level_rules = list()

    for item in item_set:
        next_level_rules.append(AssociationRule(item_set - set([item]), set([item])))

    return next_level_rules

@staticmethod
def generate_next_level_rules(rules, length):
    next_level_rules = list()

    i = 0
    while i < len(rules):
        j = i + 1
        while j < len(rules):
            intersection_set = rules[i].head.intersection(rules[j].head)
            if len(intersection_set) == length:
                next_level_rules.append(AssociationRule(intersection_set, rules[i].body.union(rules[j].body)))

            j += 1

        i += 1

    next_level_rules = list(set(next_level_rules))

    return next_level_rules
```

- For each rule generated in a level, we prune the rules that have lower confidence than the minimum confidence threshold.
- Association rules are stored as instances of the class "AssociationRule" to help simply query processing.



```python
class AssociationRule:
    def __init__(self, head, body):
        self.head = head
        self.body = body

    def get_rule(self):
        return self.head.union(self.body)

    def __hash__(self):
        return hash(tuple(sorted(self.head) + sorted(self.body)))

    def __eq__(self, other):
        return sorted(self.head) == sorted(other.head) and sorted(self.body) == sorted(other.body)

    def __repr__(self):
        return "{} -> {}".format(self.head, self.body)
```

Query processing:

From the high **confidence** rules generated, we are now supposed to answer some queries based on three template types.

- For the template 1, we can perform the following set of operations on either of RULE, BODY, or HEAD.
  Template Query: {RULE|HEAD|BODY} HAS ({ANY|NUMBER|NONE}) OF (ITEM1, ITEM2, ..., ITEMn)

  ANY: Fetch all the rules which has atleast one of the itemset from the given list

  NONE: Fetch all the rules which has none of the itemset from the given list

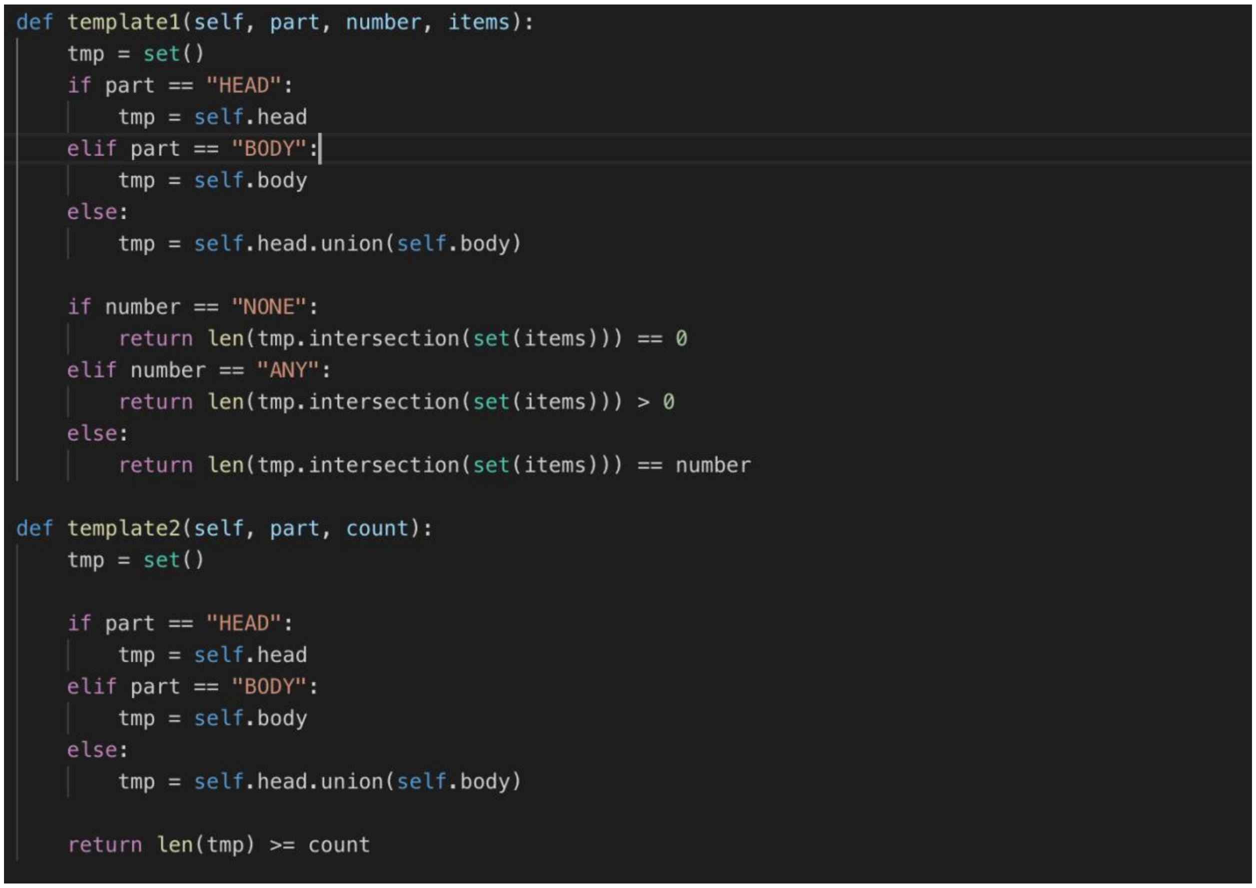
  NUMBER: Fetch all the rules which have exactly "Number" of items from the given list.

- For the template 2, we can perform the following set of operations on either RULE, BODY or HEAD
  Template Query: SizeOf({HEAD|BODY|RULE}) $\geq$ NUMBER.
  Fetch all the rules which have total number of items in head, body or rule >= item_count.

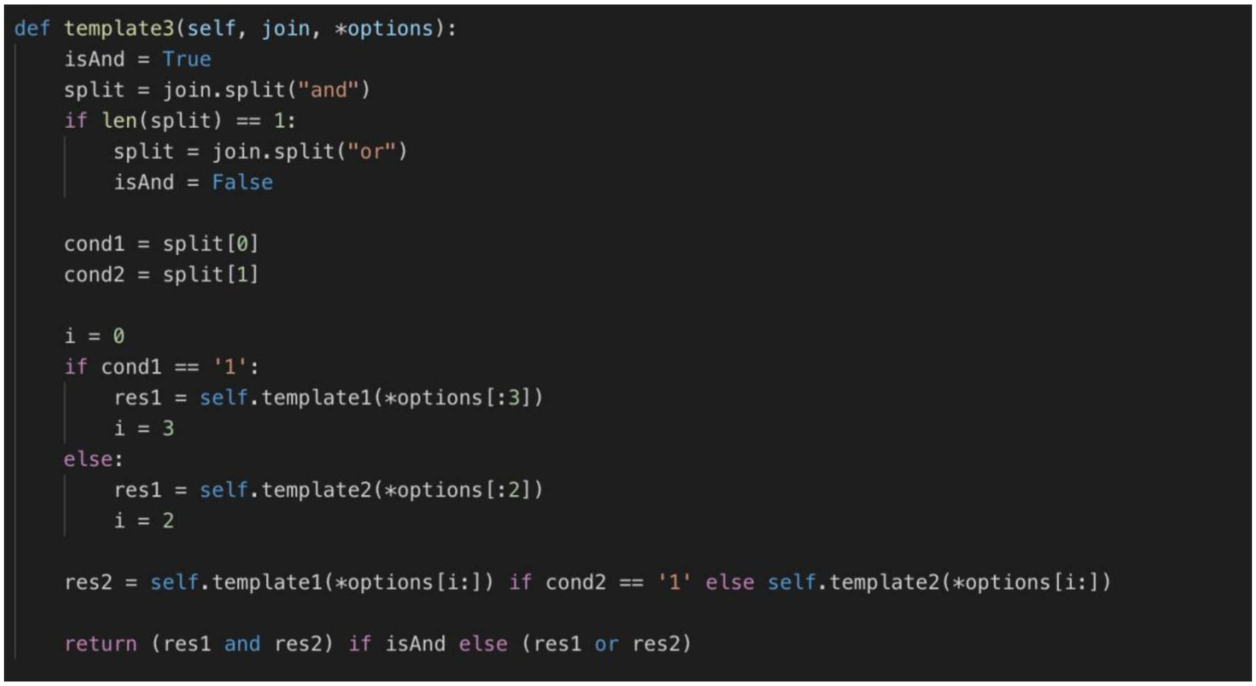
- For template 3, this is the combination of rules from template1 and template 2
  Template Queries: template3("1or2", "BODY", "ANY", ['G10_Down'], "HEAD", 2)
  There will be two sets of operations for the template 3 queries. Perform either of 'AND' or 'OR' operations on the results of those two parts according to the given query.

Implementation:

The AssociationRule class has the helper methods template1, template2 and template3 methods that filter out the given rule based on the query asked.



```python
def template1(self, part, number, items):
    tmp = set()
    if part == "HEAD":
        tmp = self.head
    elif part == "BODY":
        tmp = self.body
    else:
        tmp = self.head.union(self.body)

    if number == "NONE":
        return len(tmp.intersection(set(items))) == 0
    elif number == "ANY":
        return len(tmp.intersection(set(items))) > 0
    else:
        return len(tmp.intersection(set(items))) == number

def template2(self, part, count):
    tmp = set()

    if part == "HEAD":
        tmp = self.head
    elif part == "BODY":
        tmp = self.body
    else:
        tmp = self.head.union(self.body)

    return len(tmp) >= count
```



```python
def template3(self, join, *options):
    isAnd = True
    split = join.split("and")
    if len(split) == 1:
        split = join.split("or")
        isAnd = False

    cond1 = split[0]
    cond2 = split[1]

    i = 0
    if cond1 == '1':
        res1 = self.template1(*options[:3])
        i = 3
    else:
        res1 = self.template2(*options[:2])
        i = 2

    res2 = self.template1(*options[i:]) if cond2 == '1' else self.template2(*options[i:])

    return (res1 and res2) if isAnd else (res1 or res2)
```

Results/Outputs:

Frequent itemset counts

```
Support is set to 30.0%

number of length-1 frequent itemsets:196

number of length-2 frequent itemsets:5340

number of length-3 frequent itemsets:5287

number of length-4 frequent itemsets:1518

number of length-5 frequent itemsets:438

number of length-6 frequent itemsets:88

number of length-7 frequent itemsets:11

number of length-8 frequent itemsets:1

number of all lengths frequent itemsets:12879


Support is set to 40.0%

number of length-1 frequent itemsets:167

number of length-2 frequent itemsets:753

number of length-3 frequent itemsets:149

number of length-4 frequent itemsets:7

number of length-5 frequent itemsets:1

number of all lengths frequent itemsets:1077


Support is set to 50.0%

number of length-1 frequent itemsets:109

number of length-2 frequent itemsets:63

number of length-3 frequent itemsets:2

number of length-4 frequent itemsets:0

number of all lengths frequent itemsets:174
```

Support is set to 60.0%

number of length-1 frequent itemsets:34

number of length-2 frequent itemsets:2

number of length-3 frequent itemsets:0

number of all lengths frequent itemsets:36

Support is set to 70.0%

number of length-1 frequent itemsets:7

number of length-2 frequent itemsets:0

number of all lengths frequent itemsets:7

Query Results:

Support is set to 50.0%

Confidence is set to 50%

number of association rules:138

Query Number	No. of Association Rules
11	36
12	102
13	48
14	18
15	120
16	26
17	18
18	120
19	26
21	12
22	6
23	138
31	25
32	1
33	14
34	0
35	138
36	6

Support is set to 50.0%

Confidence is set to 70%

number of association rules:117

Query Number	No. of Association Rules
11	26
12	91
13	39
14	9
15	108
16	17
17	17
18	100
19	24
21	9
22	6
23	117
31	24
32	1
33	11
34	0
35	117
36	3