Assignment 4 - Helper Functions

We begin by including the functions to generate frequent itemsets (via the Apriori algorithm) and resulting association rules:

In [ ]:
```
# (c) 2016 Everaldo Aguiar & Reid Johnson
#
# Modified from:
# Marcel Caraciolo (https://gist.github.com/marcelcaraciolo/1423287)
#
# Functions to compute and extract association rules from a given frequent i
# generated by the Apriori algorithm.
#
# The Apriori algorithm is defined by Agrawal and Srikant in:
# Fast algorithms for mining association rules
# Proc. 20th int. conf. very large data bases, VLDB. Vol. 1215. 1994
import csv
import numpy as np

def load_dataset(filename):
    '''Loads an example of market basket transactions from a provided csv fi

    Returns: A list (database) of lists (transactions). Each element of a tr
    an item.
    '''

    with open(filename,'r') as dest_f:
        data_iter = csv.reader(dest_f, delimiter = ',', quotechar = '"')
        data = [data for data in data_iter]
        data_array = np.asarray(data)

    return data_array

def apriori(dataset, min_support=0.5, verbose=False):
    """Implements the Apriori algorithm.

    The Apriori algorithm will iteratively generate new candidate
    k-itemsets using the frequent (k-1)-itemsets found in the previous
    iteration.

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate
        candidate itemsets.

    min_support : float
        The minimum support threshold. Defaults to 0.5.
```

```
    Returns
    -------
    F : list
        The list of frequent itemsets.

    support_data : dict
        The support data for all candidate itemsets.

    References
    ----------
    .. [1] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association
           Rules", 1994.

    """
    C1 = create_candidates(dataset)
    D = list(map(set, dataset))
    F1, support_data = support_prune(D, C1, min_support, verbose=False) # pr
    F = [F1] # list of frequent itemsets; initialized to frequent 1-itemsets
    k = 2 # the itemset cardinality
    while (len(F[k - 2]) > 0):
        Ck = apriori_gen(F[k-2], k) # generate candidate itemsets
        Fk, supK = support_prune(D, Ck, min_support) # prune candidate items
        support_data.update(supK) # update the support counts to reflect pru
        F.append(Fk) # add the pruned candidate itemsets to the list of freq
        k += 1

    if verbose:
        # Print a list of all the frequent itemsets.
        for kset in F:
            for item in kset:
                print("" \
                    + "{" \
                    + "".join(str(i) + ", " for i in iter(item)).rstrip(', '
                    + "}" \
                    + ":  sup = " + str(round(support_data[item], 3)))

    return F, support_data

def create_candidates(dataset, verbose=False):
    """Creates a list of candidate 1-itemsets from a list of transactions.

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate candidat
        itemsets.

    Returns
    -------
    The list of candidate itemsets (c1) passed as a frozenset (a set that is
    immutable and hashable).
    """
    c1 = [] # list of all items in the database of transactions
```

```
    for transaction in dataset:
        for item in transaction:
            if not [item] in c1:
                c1.append([item])
    c1.sort()

    if verbose:
        # Print a list of all the candidate items.
        print("" \
            + "{" \
            + "".join(str(i[0]) + ", " for i in iter(c1)).rstrip(', ') \
            + "}")

    # Map c1 to a frozenset because it will be the key of a dictionary.
    return list(map(frozenset, c1))

def support_prune(dataset, candidates, min_support, verbose=False):
    """Returns all candidate itemsets that meet a minimum support threshold.

    By the apriori principle, if an itemset is frequent, then all of its
    subsets must also be frequent. As a result, we can perform support-based
    pruning to systematically control the exponential growth of candidate
    itemsets. Thus, itemsets that do not meet the minimum support level are
    pruned from the input list of itemsets (dataset).

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate candidat
        itemsets.

    candidates : frozenset
        The list of candidate itemsets.

    min_support : float
        The minimum support threshold.

    Returns
    -------
    retlist : list
        The list of frequent itemsets.

    support_data : dict
        The support data for all candidate itemsets.
    """
    sscnt = {} # set for support counts
    for tid in dataset:
        for can in candidates:
            if can.issubset(tid):
                sscnt.setdefault(can, 0)
                sscnt[can] += 1

    num_items = float(len(dataset)) # total number of transactions in the da
```

```
    retlist = [] # array for unpruned itemsets
    support_data = {} # set for support data for corresponding itemsets
    for key in sscnt:
        # Calculate the support of itemset key.
        support = sscnt[key] / num_items
        if support >= min_support:
            retlist.insert(0, key)
        support_data[key] = support

    # Print a list of the pruned itemsets.
    if verbose:
        for kset in retlist:
            for item in kset:
                print("{" + str(item) + "}")
        print("")
        for key in sscnt:
            print("" \
                + "{" \
                + "".join([str(i) + ", " for i in iter(key)]).rstrip(', ') \
                + "}" \
                + ":  sup = " + str(support_data[key]))

    return retlist, support_data

def apriori_gen(freq_sets, k):
    """Generates candidate itemsets (via the F_k-1 x F_k-1 method).

    This operation generates new candidate k-itemsets based on the frequent
    (k-1)-itemsets found in the previous iteration. The candidate generation
    procedure merges a pair of frequent (k-1)-itemsets only if their first k
    items are identical.

    Parameters
    ----------
    freq_sets : list
        The list of frequent (k-1)-itemsets.

    k : integer
        The cardinality of the current itemsets being evaluated.

    Returns
    -------
    retlist : list
        The list of merged frequent itemsets.
    """
    retList = [] # list of merged frequent itemsets
    lenLk = len(freq_sets) # number of frequent itemsets
    for i in range(lenLk):
        for j in range(i+1, lenLk):
            a=list(freq_sets[i])
            b=list(freq_sets[j])
            a.sort()
            b.sort()
```

```
            F1 = a[:k-2] # first k-2 items of freq_sets[i]
            F2 = b[:k-2] # first k-2 items of freq_sets[j]

            if F1 == F2: # if the first k-2 items are identical
                # Merge the frequent itemsets.
                retList.append(freq_sets[i] | freq_sets[j])

    return retList

def rules_from_conseq(freq_set, H, support_data, rules, min_confidence=0.5,
    """Generates a set of candidate rules.

    Parameters
    ----------
    freq_set : frozenset
        The complete list of frequent itemsets.

    H : list
        A list of frequent itemsets (of a particular length).

    support_data : dict
        The support data for all candidate itemsets.

    rules : list
        A potentially incomplete set of candidate rules above the minimum
        confidence threshold.

    min_confidence : float
        The minimum confidence threshold. Defaults to 0.5.
    """
    m = len(H[0])
    if m == 1:
        Hmp1 = calc_confidence(freq_set, H, support_data, rules, min_confide
    if (len(freq_set) > (m+1)):
        Hmp1 = apriori_gen(H, m+1) # generate candidate itemsets
        Hmp1 = calc_confidence(freq_set, Hmp1, support_data, rules, min_conf
        if len(Hmp1) > 1:
            # If there are candidate rules above the minimum confidence
            # threshold, recurse on the list of these candidate rules.
            rules_from_conseq(freq_set, Hmp1, support_data, rules, min_confi

def calc_confidence(freq_set, H, support_data, rules, min_confidence=0.5, ve
    """Evaluates the generated rules.

    One measurement for quantifying the goodness of association rules is
    confidence. The confidence for a rule 'P implies H' (P -> H) is defined
    the support for P and H divided by the support for P
    (support (P|H) / support(P)), where the | symbol denotes the set union
    (thus P|H means all the items in set P or in set H).

    To calculate the confidence, we iterate through the frequent itemsets an
    associated support data. For each frequent itemset, we divide the suppor
    of the itemset by the support of the antecedent (left-hand-side of the
```

```
    rule).

    Parameters
    ----------
    freq_set : frozenset
        The complete list of frequent itemsets.

    H : list
        A list of frequent itemsets (of a particular length).

    min_support : float
        The minimum support threshold.

    rules : list
        A potentially incomplete set of candidate rules above the minimum
        confidence threshold.

    min_confidence : float
        The minimum confidence threshold. Defaults to 0.5.

    Returns
    -------
    pruned_H : list
        The list of candidate rules above the minimum confidence threshold.
    """
    pruned_H = [] # list of candidate rules above the minimum confidence thr
    for conseq in H: # iterate over the frequent itemsets
        conf = support_data[freq_set] / support_data[freq_set - conseq]
        if conf >= min_confidence:
            rules.append((freq_set - conseq, conseq, conf))
            pruned_H.append(conseq)

            if verbose:
                print("" \
                    + "{" \
                    + "".join([str(i) + ", " for i in iter(freq_set-conseq)]
                    + "}" \
                    + " ---> " \
                    + "{" \
                    + "".join([str(i) + ", " for i in iter(conseq)]).rstrip(
                    + "}" \
                    + ":  conf = " + str(round(conf, 3)) \
                    + ", sup = " + str(round(support_data[freq_set], 3)))

    return pruned_H

def generate_rules(F, support_data, min_confidence=0.5, verbose=True):
    """Generates a set of candidate rules from a list of frequent itemsets.

    For each frequent itemset, we calculate the confidence of using a
    particular item as the rule consequent (right-hand-side of the rule). By
    testing and merging the remaining rules, we recursively create a list of
    pruned rules.
```

```
    Parameters
    ----------
    F : list
        A list of frequent itemsets.

    support_data : dict
        The corresponding support data for the frequent itemsets (L).

    min_confidence : float
        The minimum confidence threshold. Defaults to 0.5.

    Returns
    -------
    rules : list
        The list of candidate rules above the minimum confidence threshold.
    """
    rules = []
    for i in range(1, len(F)):
        for freq_set in F[i]:
            H1 = [frozenset([itemset]) for itemset in freq_set]
            if (i > 1):
                rules_from_conseq(freq_set, H1, support_data, rules, min_con
            else:
                calc_confidence(freq_set, H1, support_data, rules, min_confi

    return rules
```

To load our dataset of grocery transactions, use the command below

In [ ]:
```
dataset = load_dataset('grocery.csv')
D = list(map(set, dataset))
```

```
/var/folders/b_/sw4nttp11zj3sjgrzkc3tg1c0000gn/T/ipykernel_38158/241484119.p
y:25: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequ
ences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differen
t lengths or shapes) is deprecated. If you meant to do this, you must specif
y 'dtype=object' when creating the ndarray.
  data_array = np.asarray(data)
```

dataset is now a ndarray containing each of the 9835 transactions

In [ ]:
```
type(dataset)
```

Out[ ]:
```
numpy.ndarray
```

In [ ]:
```
dataset.shape
```

Out[ ]:
```
(9835,)
```

In [ ]:
```
dataset[0]
```

Out[ ]:
```
['citrus fruit', 'semi-finished bread', 'margarine', 'ready soups']
```

In [ ]:
```
dataset[1]
```

Out[ ]:
```
['tropical fruit', 'yogurt', 'coffee']
```

D Contains that dataset in a set format (which excludes duplicated items and sorts them)

In [ ]:
```
type(D[0])
```

Out[ ]:
```
set
```

In [ ]:
```
D[0]
```

Out[ ]:
```
{'citrus fruit', 'margarine', 'ready soups', 'semi-finished bread'}
```

Complete the assignment below by making use of the provided funtions.

You may use the notebook file attached with lesson 3 as a reference

Assignment 4 starts here

Part 1

Task 1

First, let's set the minimum support to be 5% to explore and apply Apriori Algorithm to generate candidate itemsets and selete those that are frequent

In [ ]:
```
min_support = 0.05
frequent_itemsets, support_data = apriori(dataset, min_support=min_support,
```

```
{domestic eggs}:  sup = 0.063
{whipped/sour cream}:  sup = 0.072
{pork}:  sup = 0.058
{napkins}:  sup = 0.052
{shopping bags}:  sup = 0.099
{brown bread}:  sup = 0.065
{sausage}:  sup = 0.094
{canned beer}:  sup = 0.078
{root vegetables}:  sup = 0.109
{pastry}:  sup = 0.089
{newspapers}:  sup = 0.08
{fruit/vegetable juice}:  sup = 0.072
{soda}:  sup = 0.174
{frankfurter}:  sup = 0.059
{beef}:  sup = 0.052
{curd}:  sup = 0.053
{bottled water}:  sup = 0.111
{bottled beer}:  sup = 0.081
{rolls/buns}:  sup = 0.184
{butter}:  sup = 0.055
{other vegetables}:  sup = 0.193
{pip fruit}:  sup = 0.076
{whole milk}:  sup = 0.256
{yogurt}:  sup = 0.14
{tropical fruit}:  sup = 0.105
{coffee}:  sup = 0.058
{margarine}:  sup = 0.059
{citrus fruit}:  sup = 0.083
{whole milk, rolls/buns}:  sup = 0.057
{whole milk, yogurt}:  sup = 0.056
{whole milk, other vegetables}:  sup = 0.075
```

Knowing the frequent itemsets, we could now use generate_rules function to generate association rules from the requent itemsets. Here, we set the minimun coffidence = 5%.

In [ ]:
```
min_confidence = 0.05
rules = generate_rules(frequent_itemsets, support_data, min_confidence=min_c
```

```
{rolls/buns} ---> {whole milk}:  conf = 0.308, sup = 0.057
{whole milk} ---> {rolls/buns}:  conf = 0.222, sup = 0.057
{yogurt} ---> {whole milk}:  conf = 0.402, sup = 0.056
{whole milk} ---> {yogurt}:  conf = 0.219, sup = 0.056
{other vegetables} ---> {whole milk}:  conf = 0.387, sup = 0.075
{whole milk} ---> {other vegetables}:  conf = 0.293, sup = 0.075
```

Comments/Observations:

1. When rolls/buns are purchased, there is a 30.8% chance that whole milk is also purchased. The support of 5.7% suggests that both items are bought together in 5.6% of the transactions. Note that even though the support of the rule {whole milk} ---> {rolls/buns} is the same as {rolls/buns} ---> {whole milk}, the confidence for the second rule is lower than the first rule, showing that the association is not equally strong in both directions. This also align with what we see for the other pairs of the rules
2. From {yogurt} ---> {whole milk} with condience of 40.2%, we can reasonably conclude that if a customer likes yogurt, there is a relatively high likely that they also like whole milk

Task 2

In [ ]:
```
import matplotlib.pyplot as plt
```

In [ ]:
```
def find_rules_for_different_confidences_and_supports(dataset, support_value
    results = {} # Dictionary to hold the number of rules for each (support,

    for min_support in support_values:
        F, support_data = apriori(dataset, min_support=min_support, verbose=
        for min_confidence in confidence_values:
            rules = generate_rules(F, support_data, min_confidence=min_confi
            if min_support not in results:
                results[min_support] = []
            results[min_support].append(len(rules))

    return results
```

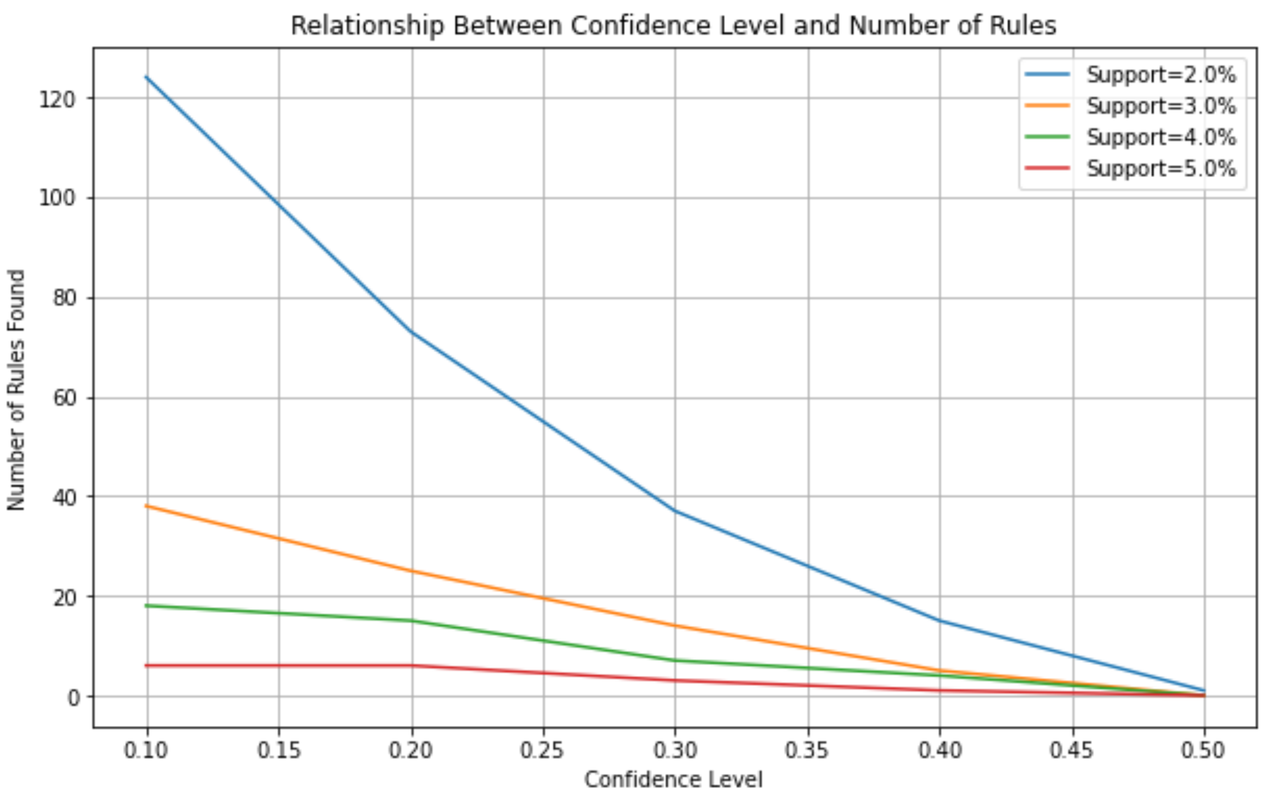
In [ ]:
```
support_values = [0.02, 0.03, 0.04, 0.05] # 2%, 3%, 4%, 5% support levels
confidence_values = [0.1, 0.2, 0.3, 0.4, 0.5] # 10%, 20%, 30%, 40%, 50% conf

results = find_rules_for_different_confidences_and_supports(dataset, support
print(results)

plt.figure(figsize=(10, 6))
for support, rule_counts in results.items():
    plt.plot(confidence_values, rule_counts, label=f'Support={support*100}%'

plt.xlabel('Confidence Level')
plt.ylabel('Number of Rules Found')
plt.title('Relationship Between Confidence Level and Number of Rules')
plt.legend()
plt.grid(True)
plt.show()
```

```
{0.02: [124, 73, 37, 15, 1], 0.03: [38, 25, 14, 5, 0], 0.04: [18, 15, 7, 4,
0], 0.05: [6, 6, 3, 1, 0]}
```



Part 2 - FPgrowth

The FP-Growth algorithm implementation used in the code was inspired by the work described in https://goo.gl/Rv8KAa.

In [ ]:
```
# (c) 2014 Reid Johnson
```

```
#
# Modified from:
# Eric Naeseth <eric@naeseth.com>
# (https://github.com/enaeseth/python-fp-growth/blob/master/fp_growth.py)
#
# A Python implementation of the FP-growth algorithm.

from collections import defaultdict, namedtuple
#from itertools import imap

__author__ = 'Eric Naeseth <eric@naeseth.com>'
__copyright__ = 'Copyright © 2009 Eric Naeseth'
__license__ = 'MIT License'

def fpgrowth(dataset, min_support=0.5, include_support=True, verbose=False):
    """Implements the FP-growth algorithm.

    The `dataset` parameter can be any iterable of iterables of items.
    `min_support` should be an integer specifying the minimum number of
    occurrences of an itemset for it to be accepted.

    Each item must be hashable (i.e., it must be valid as a member of a
    dictionary or a set).

    If `include_support` is true, yield (itemset, support) pairs instead of
    just the itemsets.

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate
        candidate itemsets.

    min_support : float
        The minimum support threshold. Defaults to 0.5.

    include_support : bool
        Include support in output (default=False).

    References
    ----------
    .. [1] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candida
           Generation," 2000.

    """

    F = []
    support_data = {}
    for k,v in find_frequent_itemsets(dataset, min_support=min_support, incl
        F.append(frozenset(k))
        support_data[frozenset(k)] = v

    # Create one array with subarrays that hold all transactions of equal le
```

```
    def bucket_list(nested_list, sort=True):
        bucket = defaultdict(list)
        for sublist in nested_list:
            bucket[len(sublist)].append(sublist)
        return [v for k,v in sorted(bucket.items())] if sort else bucket.val

    F = bucket_list(F)

    return F, support_data

def find_frequent_itemsets(dataset, min_support, include_support=False, verb
    """
    Find frequent itemsets in the given transactions using FP-growth. This
    function returns a generator instead of an eagerly-populated list of ite

    The `dataset` parameter can be any iterable of iterables of items.
    `min_support` should be an integer specifying the minimum number of
    occurrences of an itemset for it to be accepted.

    Each item must be hashable (i.e., it must be valid as a member of a
    dictionary or a set).

    If `include_support` is true, yield (itemset, support) pairs instead of
    just the itemsets.

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate
        candidate itemsets.

    min_support : float
        The minimum support threshold. Defaults to 0.5.

    include_support : bool
        Include support in output (default=False).

    """
    items = defaultdict(lambda: 0) # mapping from items to their supports
    processed_transactions = []

    # Load the passed-in transactions and count the support that individual
    # items have.
    for transaction in dataset:
        processed = []
        for item in transaction:
            items[item] += 1
            processed.append(item)
        processed_transactions.append(processed)

    # Remove infrequent items from the item support dictionary.
    items = dict((item, support) for item, support in items.items()
        if support >= min_support)
```

```
    # Build our FP-tree. Before any transactions can be added to the tree, t
    # must be stripped of infrequent items and their surviving items must be
    # sorted in decreasing order of frequency.
    def clean_transaction(transaction):
        #transaction = filter(lambda v: v in items, transaction)
        transaction.sort(key=lambda v: items[v], reverse=True)
        return transaction

    master = FPTree()
    for transaction in map(clean_transaction, processed_transactions):
        master.add(transaction)

    support_data = {}
    def find_with_suffix(tree, suffix):
        for item, nodes in tree.items():
            support = float(sum(n.count for n in nodes)) / len(dataset)
            if support >= min_support and item not in suffix:
                # New winner!
                found_set = [item] + suffix
                support_data[frozenset(found_set)] = support
                yield (found_set, support) if include_support else found_set

                # Build a conditional tree and recursively search for freque
                # itemsets within it.
                cond_tree = conditional_tree_from_paths(tree.prefix_paths(it
                    min_support)
                for s in find_with_suffix(cond_tree, found_set):
                    yield s # pass along the good news to our caller

    if verbose:
        # Print a list of all the frequent itemsets.
        for itemset, support in find_with_suffix(master, []):
            print("" \
                + "{" \
                + "".join(str(i) + ", " for i in iter(itemset)).rstrip(', ')
                + "}" \
                + ":  sup = " + str(round(support_data[frozenset(itemset)],

    # Search for frequent itemsets, and yield the results we find.
    for itemset in find_with_suffix(master, []):
        yield itemset

class FPTree(object):
    """
    An FP tree.

    This object may only store transaction items that are hashable (i.e., al
    items must be valid as dictionary keys or set members).
    """

    Route = namedtuple('Route', 'head tail')
```

```
    def __init__(self):
        # The root node of the tree.
        self._root = FPNode(self, None, None)

        # A dictionary mapping items to the head and tail of a path of
        # "neighbors" that will hit every node containing that item.
        self._routes = {}

    @property
    def root(self):
        """The root node of the tree."""
        return self._root

    def add(self, transaction):
        """
        Adds a transaction to the tree.
        """

        point = self._root

        for item in transaction:
            next_point = point.search(item)
            if next_point:
                # There is already a node in this tree for the current
                # transaction item; reuse it.
                next_point.increment()
            else:
                # Create a new point and add it as a child of the point we'r
                # currently looking at.
                next_point = FPNode(self, item)
                point.add(next_point)

                # Update the route of nodes that contain this item to incluc
                # our new node.
                self._update_route(next_point)

            point = next_point

    def _update_route(self, point):
        """Add the given node to the route through all nodes for its item.""
        assert self is point.tree

        try:
            route = self._routes[point.item]
            route[1].neighbor = point # route[1] is the tail
            self._routes[point.item] = self.Route(route[0], point)
        except KeyError:
            # First node for this item; start a new route.
            self._routes[point.item] = self.Route(point, point)

    def items(self):
        """
        Generate one 2-tuples for each item represented in the tree. The fir
```

```
        element of the tuple is the item itself, and the second element is a
        generator that will yield the nodes in the tree that belong to the i
        """
        for item in self._routes.keys():
            yield (item, self.nodes(item))


    def nodes(self, item):
        """
        Generates the sequence of nodes that contain the given item.
        """

        try:
            node = self._routes[item][0]
        except KeyError:
            return

        while node:
            yield node
            node = node.neighbor

    def prefix_paths(self, item):
        """Generates the prefix paths that end with the given item."""

        def collect_path(node):
            path = []
            while node and not node.root:
                path.append(node)
                node = node.parent
            path.reverse()
            return path

        return (collect_path(node) for node in self.nodes(item))

    def inspect(self):
        print("Tree:")
        self.root.inspect(1)

        print("")
        print("Routes:")
        for item, nodes in self.items():
            print("  %r" % item)
            for node in nodes:
                print("    %r" % node)

    def _removed(self, node):
        """Called when `node` is removed from the tree; performs cleanup."""

        head, tail = self._routes[node.item]
        if node is head:
            if node is tail or not node.neighbor:
                # It was the sole node.
                del self._routes[node.item]
```

```
            else:
                self._routes[node.item] = self.Route(node.neighbor, tail)
        else:
            for n in self.nodes(node.item):
                if n.neighbor is node:
                    n.neighbor = node.neighbor # skip over
                    if node is tail:
                        self._routes[node.item] = self.Route(head, n)
                    break

def conditional_tree_from_paths(paths, min_support):
    """Builds a conditional FP-tree from the given prefix paths."""
    tree = FPTree()
    condition_item = None
    items = set()

    # Import the nodes in the paths into the new tree. Only the counts of th
    # leaf notes matter; the remaining counts will be reconstructed from the
    # leaf counts.
    for path in paths:
        if condition_item is None:
            condition_item = path[-1].item

        point = tree.root
        for node in path:
            next_point = point.search(node.item)
            if not next_point:
                # Add a new node to the tree.
                items.add(node.item)
                count = node.count if node.item == condition_item else 0
                next_point = FPNode(tree, node.item, count)
                point.add(next_point)
                tree._update_route(next_point)
            point = next_point

    assert condition_item is not None

    # Calculate the counts of the non-leaf nodes.
    for path in tree.prefix_paths(condition_item):
        count = path[-1].count
        for node in reversed(path[:-1]):
            node._count += count

    # Eliminate the nodes for any items that are no longer frequent.
    for item in items:
        support = sum(n.count for n in tree.nodes(item))
        if support < min_support:
            # Doesn't make the cut anymore
            for node in tree.nodes(item):
                if node.parent is not None:
                    node.parent.remove(node)

    # Finally, remove the nodes corresponding to the item for which this
```

```
    # conditional tree was generated.
    for node in tree.nodes(condition_item):
        if node.parent is not None: # the node might already be an orphan
            node.parent.remove(node)

    return tree

class FPNode(object):
    """A node in an FP tree."""

    def __init__(self, tree, item, count=1):
        self._tree = tree
        self._item = item
        self._count = count
        self._parent = None
        self._children = {}
        self._neighbor = None

    def add(self, child):
        """Adds the given FPNode `child` as a child of this node."""

        if not isinstance(child, FPNode):
            raise TypeError("Can only add other FPNodes as children")

        if not child.item in self._children:
            self._children[child.item] = child
            child.parent = self

    def search(self, item):
        """
        Checks to see if this node contains a child node for the given item.
        If so, that node is returned; otherwise, `None` is returned.
        """

        try:
            return self._children[item]
        except KeyError:
            return None

    def remove(self, child):
        try:
            if self._children[child.item] is child:
                del self._children[child.item]
                child.parent = None
                self._tree._removed(child)
                for sub_child in child.children:
                    try:
                        # Merger case: we already have a child for that item
                        # add the sub-child's count to our child's count.
                        self._children[sub_child.item]._count += sub_child.c
                        sub_child.parent = None # it's an orphan now
                    except KeyError:
                        # Turns out we don't actually have a child, so just
```

```
                        # the sub-child as our own child.
                        self.add(sub_child)
                child._children = {}
            else:
                raise ValueError("that node is not a child of this node")
        except KeyError:
            raise ValueError("that node is not a child of this node")

    def __contains__(self, item):
        return item in self._children

    @property
    def tree(self):
        """The tree in which this node appears."""
        return self._tree

    @property
    def item(self):
        """The item contained in this node."""
        return self._item

    @property
    def count(self):
        """The count associated with this node's item."""
        return self._count

    def increment(self):
        """Increments the count associated with this node's item."""
        if self._count is None:
            raise ValueError("Root nodes have no associated count.")
        self._count += 1

    @property
    def root(self):
        """True if this node is the root of a tree; false if otherwise."""
        return self._item is None and self._count is None

    @property
    def leaf(self):
        """True if this node is a leaf in the tree; false if otherwise."""
        return len(self._children) == 0

    def parent():
        doc = "The node's parent."
        def fget(self):
            return self._parent
        def fset(self, value):
            if value is not None and not isinstance(value, FPNode):
                raise TypeError("A node must have an FPNode as a parent.")
            if value and value.tree is not self.tree:
                raise ValueError("Cannot have a parent from another tree.")
            self._parent = value
        return locals()
```

```
    parent = property(**parent())

    def neighbor():
        doc = """
        The node's neighbor; the one with the same value that is "to the rig
        of it in the tree.
        """
        def fget(self):
            return self._neighbor
        def fset(self, value):
            if value is not None and not isinstance(value, FPNode):
                raise TypeError("A node must have an FPNode as a neighbor.")
            if value and value.tree is not self.tree:
                raise ValueError("Cannot have a neighbor from another tree."
            self._neighbor = value
        return locals()
    neighbor = property(**neighbor())

    @property
    def children(self):
        """The nodes that are children of this node."""
        return tuple(self._children.values())

    def inspect(self, depth=0):
        print(('  ' * depth) + repr(self))
        for child in self.children:
            child.inspect(depth + 1)

    def __repr__(self):
        if self.root:
            return "<%s (root)>" % type(self).__name__
        return "<%s %r (%r)>" % (type(self).__name__, self.item, self.count)

def rules_from_conseq(freq_set, H, support_data, rules, min_confidence=0.5,
    """Generates a set of candidate rules.

    Parameters
    ----------
    freq_set : frozenset
        The complete list of frequent itemsets.

    H : list
        A list of frequent itemsets (of a particular length).

    support_data : dict
        The support data for all candidate itemsets.

    rules : list
        A potentially incomplete set of candidate rules above the minimum
        confidence threshold.

    min_confidence : float
        The minimum confidence threshold. Defaults to 0.5.
```

```
    """
    m = len(H[0])
    if m == 1:
        Hmp1 = calc_confidence(freq_set, H, support_data, rules, min_confide
    if (len(freq_set) > (m+1)):
        Hmp1 = apriori_gen(H, m+1) # generate candidate itemsets
        Hmp1 = calc_confidence(freq_set, Hmp1,  support_data, rules, min_con
        if len(Hmp1) > 1:
            # If there are candidate rules above the minimum confidence
            # threshold, recurse on the list of these candidate rules.
            rules_from_conseq(freq_set, Hmp1, support_data, rules, min_confi

def calc_confidence(freq_set, H, support_data, rules, min_confidence=0.5, ve
    """Evaluates the generated rules.

    One measurement for quantifying the goodness of association rules is
    confidence. The confidence for a rule 'P implies H' (P -> H) is defined
    the support for P and H divided by the support for P
    (support (P|H) / support(P)), where the | symbol denotes the set union
    (thus P|H means all the items in set P or in set H).

    To calculate the confidence, we iterate through the frequent itemsets an
    associated support data. For each frequent itemset, we divide the suppor
    of the itemset by the support of the antecedent (left-hand-side of the
    rule).

    Parameters
    ----------
    freq_set : frozenset
        The complete list of frequent itemsets.

    H : list
        A list of frequent itemsets (of a particular length).

    min_support : float
        The minimum support threshold.

    rules : list
        A potentially incomplete set of candidate rules above the minimum
        confidence threshold.

    min_confidence : float
        The minimum confidence threshold. Defaults to 0.5.

    Returns
    -------
    pruned_H : list
        The list of candidate rules above the minimum confidence threshold.
    """
    pruned_H = [] # list of candidate rules above the minimum confidence thr
    for conseq in H: # iterate over the frequent itemsets
        conf = support_data[freq_set] / support_data[freq_set - conseq]
        if conf >= min_confidence:
```

```
            rules.append((freq_set - conseq, conseq, conf))
            pruned_H.append(conseq)

            if verbose:
                print("" \
                    + "{" \
                    + "".join([str(i) + ", " for i in iter(freq_set-conseq)]
                    + "}" \
                    + " ---> " \
                    + "{" \
                    + "".join([str(i) + ", " for i in iter(conseq)]).rstrip(
                    + "}" \
                    + ":  conf = " + str(round(conf, 3)) \
                    + ", sup = " + str(round(support_data[freq_set], 3)))

    return pruned_H

def generate_rules(F, support_data, min_confidence=0.5, verbose=True):
    """Generates a set of candidate rules from a list of frequent itemsets.

    For each frequent itemset, we calculate the confidence of using a
    particular item as the rule consequent (right-hand-side of the rule). By
    testing and merging the remaining rules, we recursively create a list of
    pruned rules.

    Parameters
    ----------
    F : list
        A list of frequent itemsets.

    support_data : dict
        The corresponding support data for the frequent itemsets (L).

    min_confidence : float
        The minimum confidence threshold. Defaults to 0.5.

    Returns
    -------
    rules : list
        The list of candidate rules above the minimum confidence threshold.
    """
    rules = []
    for i in range(1, len(F)):
        for freq_set in F[i]:
            H1 = [frozenset([item]) for item in freq_set]
            if (i > 1):
                rules_from_conseq(freq_set, H1, support_data, rules, min_con
            else:
                calc_confidence(freq_set, H1, support_data, rules, min_confi

    return rules
```

Task 1

With the provided functions, now we could use fpgrowth function to find the frquent itemsets. Let's set the minimum support = 0.05, similar to what we had in Apriori

In [ ]:
```
min_support = 0.05
F, support_data = fpgrowth(dataset, min_support=min_support, include_support

for itemset_group in F:
    for itemset in itemset_group:
        support = support_data[itemset]
        print(f"Frequent Itemset: {set(itemset)}, Support: {support}")
```

```
Frequent Itemset: {'citrus fruit'}, Support: 0.08276563294356888
Frequent Itemset: {'margarine'}, Support: 0.05856634468734113
Frequent Itemset: {'yogurt'}, Support: 0.13950177935943062
Frequent Itemset: {'tropical fruit'}, Support: 0.10493136756481952
Frequent Itemset: {'coffee'}, Support: 0.05805795627859685
Frequent Itemset: {'whole milk'}, Support: 0.25551601423487547
Frequent Itemset: {'pip fruit'}, Support: 0.07564819522114896
Frequent Itemset: {'other vegetables'}, Support: 0.1934926283680732
Frequent Itemset: {'butter'}, Support: 0.05541433655312659
Frequent Itemset: {'rolls/buns'}, Support: 0.18393492628368074
Frequent Itemset: {'bottled beer'}, Support: 0.08052872394509406
Frequent Itemset: {'bottled water'}, Support: 0.11052364006100661
Frequent Itemset: {'curd'}, Support: 0.05327910523640061
Frequent Itemset: {'beef'}, Support: 0.05246568378240976
Frequent Itemset: {'soda'}, Support: 0.17437722419928825
Frequent Itemset: {'frankfurter'}, Support: 0.058973055414336555
Frequent Itemset: {'newspapers'}, Support: 0.07981698017285206
Frequent Itemset: {'fruit/vegetable juice'}, Support: 0.0722928317234367
Frequent Itemset: {'pastry'}, Support: 0.08896797153024912
Frequent Itemset: {'root vegetables'}, Support: 0.10899847483477376
Frequent Itemset: {'canned beer'}, Support: 0.07768174885612608
Frequent Itemset: {'sausage'}, Support: 0.09395017793594305
Frequent Itemset: {'shopping bags'}, Support: 0.09852567361464158
Frequent Itemset: {'brown bread'}, Support: 0.06487036095577021
Frequent Itemset: {'napkins'}, Support: 0.05236400610066091
Frequent Itemset: {'whipped/sour cream'}, Support: 0.07168276563294357
Frequent Itemset: {'pork'}, Support: 0.05765124555160142
Frequent Itemset: {'domestic eggs'}, Support: 0.06344687341128623
Frequent Itemset: {'whole milk', 'yogurt'}, Support: 0.05602440264361973
Frequent Itemset: {'whole milk', 'other vegetables'}, Support: 0.07483477376
715811
Frequent Itemset: {'whole milk', 'rolls/buns'}, Support: 0.05663446873411286
```

With the frequent itemsets, we could now gerenate association rules using minimun confidence = 0.05, similar to what we used for Apriori

In [ ]:
```
min_confidence = 0.05
rules = generate_rules(F, support_data, min_confidence=min_confidence, verbo
```

```
{yogurt} ---> {whole milk}:  conf = 0.402, sup = 0.056
{whole milk} ---> {yogurt}:  conf = 0.219, sup = 0.056
{other vegetables} ---> {whole milk}:  conf = 0.387, sup = 0.075
{whole milk} ---> {other vegetables}:  conf = 0.293, sup = 0.075
{rolls/buns} ---> {whole milk}:  conf = 0.308, sup = 0.057
{whole milk} ---> {rolls/buns}:  conf = 0.222, sup = 0.057
```

Define a helper function similar to before

Task 2

In [ ]:
```
def find_rules_for_different_confidences_and_supports(dataset, support_value
    results = {}  # Dictionary to hold the number of rules for each (support

    for min_support in support_values:
        F, support_data = fpgrowth(dataset, min_support=min_support, include
        for min_confidence in confidence_values:
            rules = generate_rules(F, support_data, min_confidence=min_confi
            if min_support not in results:
                results[min_support] = []
            results[min_support].append(len(rules))

    return results
```

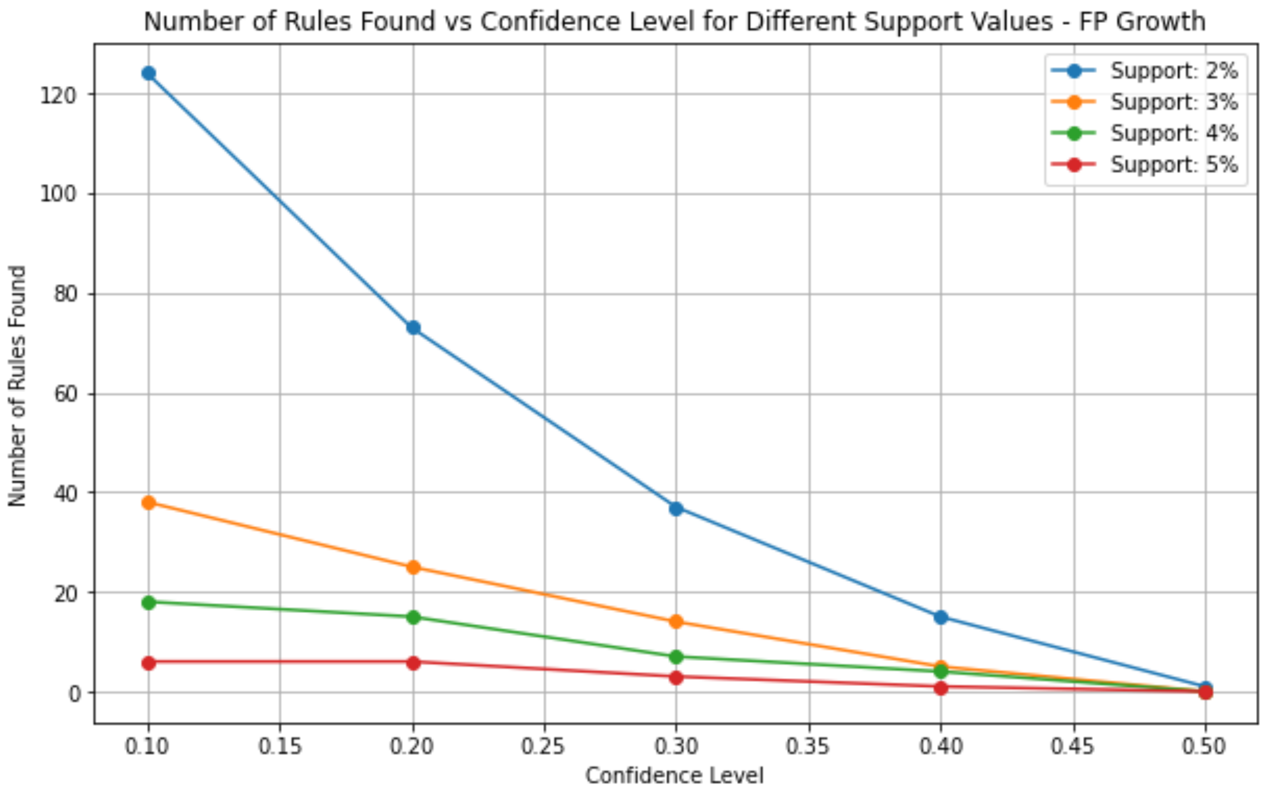
In [ ]:
```
support_values = [0.02, 0.03, 0.04, 0.05]
confidence_values = [0.1, 0.2, 0.3, 0.4, 0.5]

# Get the number of rules for each (support, confidence) pair
results = find_rules_for_different_confidences_and_supports(dataset, support

# Plot the results
plt.figure(figsize=(10, 6))

for support in support_values:
    plt.plot(confidence_values, results[support], marker='o', label=f'Suppor

plt.xlabel('Confidence Level')
plt.ylabel('Number of Rules Found')
plt.title('Number of Rules Found vs Confidence Level for Different Support V
plt.legend()
plt.grid(True)
plt.show()
```



Part 3 - Interest Factor

Use min_support = 0.02 and min_confidence = 0.3 to generate rules using FPGrowth as Part 2

In [ ]:
```
min_support = 0.02  # 2% support
min_confidence = 0.3  # 30% confidence

F, support_data = fpgrowth(dataset, min_support=min_support, include_support
rules = generate_rules(F, support_data, min_confidence=min_confidence, verbc
```

```
{citrus fruit} ---> {whole milk}:  conf = 0.369, sup = 0.031
{citrus fruit} ---> {other vegetables}:  conf = 0.349, sup = 0.029
{margarine} ---> {whole milk}:  conf = 0.413, sup = 0.024
{yogurt} ---> {whole milk}:  conf = 0.402, sup = 0.056
{yogurt} ---> {other vegetables}:  conf = 0.311, sup = 0.043
{tropical fruit} ---> {other vegetables}:  conf = 0.342, sup = 0.036
{tropical fruit} ---> {whole milk}:  conf = 0.403, sup = 0.042
{pip fruit} ---> {whole milk}:  conf = 0.398, sup = 0.03
{pip fruit} ---> {other vegetables}:  conf = 0.345, sup = 0.026
{other vegetables} ---> {whole milk}:  conf = 0.387, sup = 0.075
{butter} ---> {whole milk}:  conf = 0.497, sup = 0.028
{butter} ---> {other vegetables}:  conf = 0.361, sup = 0.02
{rolls/buns} ---> {whole milk}:  conf = 0.308, sup = 0.057
{bottled water} ---> {whole milk}:  conf = 0.311, sup = 0.034
{curd} ---> {whole milk}:  conf = 0.49, sup = 0.026
{beef} ---> {whole milk}:  conf = 0.405, sup = 0.021
{frankfurter} ---> {whole milk}:  conf = 0.348, sup = 0.021
{newspapers} ---> {whole milk}:  conf = 0.343, sup = 0.027
{fruit/vegetable juice} ---> {whole milk}:  conf = 0.368, sup = 0.027
{pastry} ---> {whole milk}:  conf = 0.374, sup = 0.033
{root vegetables} ---> {other vegetables}:  conf = 0.435, sup = 0.047
{root vegetables} ---> {whole milk}:  conf = 0.449, sup = 0.049
{sausage} ---> {rolls/buns}:  conf = 0.326, sup = 0.031
{sausage} ---> {whole milk}:  conf = 0.318, sup = 0.03
{brown bread} ---> {whole milk}:  conf = 0.389, sup = 0.025
{whipped/sour cream} ---> {whole milk}:  conf = 0.45, sup = 0.032
{whipped/sour cream} ---> {other vegetables}:  conf = 0.403, sup = 0.029
{pork} ---> {whole milk}:  conf = 0.384, sup = 0.022
{pork} ---> {other vegetables}:  conf = 0.376, sup = 0.022
{domestic eggs} ---> {whole milk}:  conf = 0.473, sup = 0.03
{domestic eggs} ---> {other vegetables}:  conf = 0.351, sup = 0.022
{frozen vegetables} ---> {whole milk}:  conf = 0.425, sup = 0.02
{other vegetables, yogurt} ---> {whole milk}:  conf = 0.513, sup = 0.022
{whole milk, yogurt} ---> {other vegetables}:  conf = 0.397, sup = 0.022
{root vegetables, other vegetables} ---> {whole milk}:  conf = 0.489, sup = 
0.023
{whole milk, root vegetables} ---> {other vegetables}:  conf = 0.474, sup = 
0.023
{whole milk, other vegetables} ---> {root vegetables}:  conf = 0.31, sup = 0
.023
```

Add a calc_interest_factor function for calculating lift. Recall that the lift = Support (A U B) / Support(A) * Support(B). Then, we will modify the calc_confidence function, which is called by generate_rules, such that for each rule there is a interest factor calculated.

In [ ]:
```
def calculate_interest_factor(antecedent, consequent, support, support_data)
    support_antecedent = support_data[antecedent]
    support_consequent = support_data[consequent]
    return support / (support_antecedent * support_consequent)

# For each rule in rules, calculate its interest factor
processed_rules = []
for rule in rules:
    antecedent, consequent, confidence = rule
    support = support_data[antecedent | consequent]
    interest_factor = calculate_interest_factor(antecedent, consequent, supp
    processed_rules.append((antecedent, consequent, support, confidence, int
```

Now, prepare three sorted sets by support, confidence, and interest factor, respectively

In [ ]:
```
rules_by_support = sorted(processed_rules, key=lambda x: x[2], reverse=True)
rules_by_confidence = sorted(processed_rules, key=lambda x: x[3], reverse=Tr
rules_by_interest = sorted(processed_rules, key=lambda x: x[4], reverse=True
```

The top 5 rules by support:

In [ ]:
```
for i, rule in enumerate(rules_by_support[:5], 1):
        antecedent, consequent, support, confidence, interest = rule
        print(f"{i}. {set(antecedent)} -> {set(consequent)}")
        print(f"   Support: {support:.4f}, Confidence: {confidence:.4f}, Int
```

```
1. {'other vegetables'} -> {'whole milk'}
   Support: 0.0748, Confidence: 0.3868, Interest Factor: 1.5136
2. {'rolls/buns'} -> {'whole milk'}
   Support: 0.0566, Confidence: 0.3079, Interest Factor: 1.2050
3. {'yogurt'} -> {'whole milk'}
   Support: 0.0560, Confidence: 0.4016, Interest Factor: 1.5717
4. {'root vegetables'} -> {'whole milk'}
   Support: 0.0489, Confidence: 0.4487, Interest Factor: 1.7560
5. {'root vegetables'} -> {'other vegetables'}
   Support: 0.0474, Confidence: 0.4347, Interest Factor: 2.2466
```

The top 5 rules by confidence:

In [ ]:
```
for i, rule in enumerate(rules_by_confidence[:5], 1):
        antecedent, consequent, support, confidence, interest = rule
        print(f"{i}. {set(antecedent)} -> {set(consequent)}")
        print(f"   Support: {support:.4f}, Confidence: {confidence:.4f}, Int
```

```
1. {'other vegetables', 'yogurt'} -> {'whole milk'}
   Support: 0.0223, Confidence: 0.5129, Interest Factor: 2.0072
2. {'butter'} -> {'whole milk'}
   Support: 0.0276, Confidence: 0.4972, Interest Factor: 1.9461
3. {'curd'} -> {'whole milk'}
   Support: 0.0261, Confidence: 0.4905, Interest Factor: 1.9195
4. {'root vegetables', 'other vegetables'} -> {'whole milk'}
   Support: 0.0232, Confidence: 0.4893, Interest Factor: 1.9148
5. {'whole milk', 'root vegetables'} -> {'other vegetables'}
   Support: 0.0232, Confidence: 0.4740, Interest Factor: 2.4498
```

The top 5 rules by interest factor:

In [ ]:

```
for i, rule in enumerate(rules_by_interest[:5], 1):
        antecedent, consequent, support, confidence, interest = rule
        print(f"{i}. {set(antecedent)} -> {set(consequent)}")
        print(f"   Support: {support:.4f}, Confidence: {confidence:.4f}, Int
```

```
1. {'whole milk', 'other vegetables'} -> {'root vegetables'}
   Support: 0.0232, Confidence: 0.3098, Interest Factor: 2.8421
2. {'whole milk', 'root vegetables'} -> {'other vegetables'}
   Support: 0.0232, Confidence: 0.4740, Interest Factor: 2.4498
3. {'root vegetables'} -> {'other vegetables'}
   Support: 0.0474, Confidence: 0.4347, Interest Factor: 2.2466
4. {'whipped/sour cream'} -> {'other vegetables'}
   Support: 0.0289, Confidence: 0.4028, Interest Factor: 2.0819
5. {'whole milk', 'yogurt'} -> {'other vegetables'}
   Support: 0.0223, Confidence: 0.3975, Interest Factor: 2.0541
```

Now we could find the common rules for the three sorted sets. From the results below we can conclude:

1. there is no common rules when we compare the top 5 rules between support and confidence.
2. There is one rule between support and interest factor, which is {'root vegetables'} -> {'other vegetables'}.This implies that Root vegetables and other vegetables are frequently bought together (high support), and maybe these two categories are good candidates to put together for bundle sales.
3. There is one rule between confidence and interest factor, which is {'whole milk', 'root vegetables'} -> {'other vegetables'}. When customers buy whole milk and root vegetables, they are very likely to also buy other vegetables (high confidence). The high interest factor also suggests that this combination occurs more frequently than would be expected by chance.

In [ ]:
```
def rule_to_tuple(rule):
    return (frozenset(rule[0]), frozenset(rule[1]))

# Create sets of top 5 rules for each criterion
top_support = set(map(rule_to_tuple, rules_by_support[:5]))
top_confidence = set(map(rule_to_tuple, rules_by_confidence[:5]))
top_interest = set(map(rule_to_tuple, rules_by_interest[:5]))

# Find common rules across all criteria
common_rules = top_support & top_confidence & top_interest

print("\nCommon rules across all three criteria:")
if common_rules:
    for i, rule in enumerate(common_rules, 1):
        print(f"{i}. {set(rule[0])} -> {set(rule[1])}")
else:
    print("No common rules found across all three criteria.")

# Compare pairs of criteria
pairs = [
    (top_support, top_confidence, "support", "confidence"),
    (top_support, top_interest, "support", "interest factor"),
    (top_confidence, top_interest, "confidence", "interest factor")
]

for set1, set2, name1, name2 in pairs:
    common = set1 & set2
    print(f"\nCommon rules between {name1} and {name2}: {len(common)}")
    if common:
        print("Common rules:")
        for i, rule in enumerate(common, 1):
            print(f"{i}. {set(rule[0])} -> {set(rule[1])}")
```

```
Common rules across all three criteria:
No common rules found across all three criteria.

Common rules between support and confidence: 0

Common rules between support and interest factor: 1
Common rules:
1. {'root vegetables'} -> {'other vegetables'}

Common rules between confidence and interest factor: 1
Common rules:
1. {'whole milk', 'root vegetables'} -> {'other vegetables'}
```