ShriDattaMadhira_FPGrowth_Assignment4

October 8, 2021

[1]:
```
# (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
import csv
import numpy as np
#from itertools import imap

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

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

    Returns: A list (database) of lists (transactions). Each element of a␣
 ↪transaction is
    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 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 Candidate
           Generation," 2000.

    """

    F = []
    support_data = {}
    for k,v in find_frequent_itemsets(dataset, min_support=min_support,␣
↪include_support=include_support, verbose=verbose):
        F.append(frozenset(k))
        support_data[frozenset(k)] = v

    # Create one array with subarrays that hold all transactions of equal␣
↪length.
    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.
↪values()

    F = bucket_list(F)

    return F, support_data
```

```
def find_frequent_itemsets(dataset, min_support, include_support=False,␣
↪verbose=False):
    """
    Find frequent itemsets in the given transactions using FP-growth. This
    function returns a generator instead of an eagerly-populated list of items.

    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, they
    # 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 frequent
                # itemsets within it.
                cond_tree = conditional_tree_from_paths(tree.prefix_paths(item),
                    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)], 3)))

    # 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., all
    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're
                # currently looking at.
                next_point = FPNode(self, item)
                point.add(next_point)

                # Update the route of nodes that contain this item to include
                # 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 first
        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 item.
        """
        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 the
    # 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, so
                        # add the sub-child's count to our child's count.
                        self._children[sub_child.item]._count += sub_child.count
                        sub_child.parent = None # it's an orphan now
                    except KeyError:
                        # Turns out we don't actually have a child, so just add
                        # 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 right"
        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,␣
↪verbose=False):
    """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_confidence, verbose)
    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_confidence, verbose)
        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_confidence, verbose)

def calc_confidence(freq_set, H, support_data, rules, min_confidence=0.5,␣
↪verbose=False):
    """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 as
    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 and
    associated support data. For each frequent itemset, we divide the support
    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␣
↪threshold
    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)]).
↪rstrip(', ') \
                    + "}" \
                    + " ---> " \
                    + "{" \
                    + "".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 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-2
    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 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_confidence, verbose)
            else:
```

```
                calc_confidence(freq_set, H1, support_data, rules,␣
↪min_confidence, verbose)

    return rules
```

[2]:
```
dataset = load_dataset('grocery.csv')
D = list(map(set, dataset))
```

```
/Users/shridatta/Downloads/ENTER/envs/cenv/lib/python3.9/site-
packages/numpy/core/_asarray.py:83: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant
to do this, you must specify 'dtype=object' when creating the ndarray
  return array(a, dtype, copy=False, order=order)
```

[3]:
```
f, fp_data = fpgrowth(dataset, min_support=0.02, verbose=True)
```

```
{citrus fruit}:  sup = 0.083
{whole milk, citrus fruit}:  sup = 0.031
{yogurt, citrus fruit}:  sup = 0.022
{other vegetables, citrus fruit}:  sup = 0.029
{margarine}:  sup = 0.059
{whole milk, margarine}:  sup = 0.024
{yogurt}:  sup = 0.14
{whole milk, yogurt}:  sup = 0.056
{soda, yogurt}:  sup = 0.027
{rolls/buns, yogurt}:  sup = 0.034
{other vegetables, yogurt}:  sup = 0.043
{whole milk, other vegetables, yogurt}:  sup = 0.022
{tropical fruit}:  sup = 0.105
{yogurt, tropical fruit}:  sup = 0.029
{other vegetables, tropical fruit}:  sup = 0.036
{whole milk, tropical fruit}:  sup = 0.042
{rolls/buns, tropical fruit}:  sup = 0.025
{root vegetables, tropical fruit}:  sup = 0.021
{soda, tropical fruit}:  sup = 0.021
{coffee}:  sup = 0.058
{whole milk}:  sup = 0.256
{pip fruit}:  sup = 0.076
{whole milk, pip fruit}:  sup = 0.03
{tropical fruit, pip fruit}:  sup = 0.02
{other vegetables, pip fruit}:  sup = 0.026
{cream cheese}:  sup = 0.04
{other vegetables}:  sup = 0.193
{whole milk, other vegetables}:  sup = 0.075
{long life bakery product}:  sup = 0.037
{butter}:  sup = 0.055
{whole milk, butter}:  sup = 0.028
```

```
{other vegetables, butter}:  sup = 0.02
{rolls/buns}:  sup = 0.184
{other vegetables, rolls/buns}:  sup = 0.043
{whole milk, rolls/buns}:  sup = 0.057
{bottled beer}:  sup = 0.081
{whole milk, bottled beer}:  sup = 0.02
{UHT-milk}:  sup = 0.033
{bottled water}:  sup = 0.111
{other vegetables, bottled water}:  sup = 0.025
{whole milk, bottled water}:  sup = 0.034
{yogurt, bottled water}:  sup = 0.023
{rolls/buns, bottled water}:  sup = 0.024
{soda, bottled water}:  sup = 0.029
{chocolate}:  sup = 0.05
{white bread}:  sup = 0.042
{curd}:  sup = 0.053
{whole milk, curd}:  sup = 0.026
{beef}:  sup = 0.052
{whole milk, beef}:  sup = 0.021
{soda}:  sup = 0.174
{rolls/buns, soda}:  sup = 0.038
{whole milk, soda}:  sup = 0.04
{other vegetables, soda}:  sup = 0.033
{frankfurter}:  sup = 0.059
{whole milk, frankfurter}:  sup = 0.021
{chicken}:  sup = 0.043
{newspapers}:  sup = 0.08
{whole milk, newspapers}:  sup = 0.027
{fruit/vegetable juice}:  sup = 0.072
{other vegetables, fruit/vegetable juice}:  sup = 0.021
{whole milk, fruit/vegetable juice}:  sup = 0.027
{sugar}:  sup = 0.034
{specialty bar}:  sup = 0.027
{pastry}:  sup = 0.089
{soda, pastry}:  sup = 0.021
{whole milk, pastry}:  sup = 0.033
{rolls/buns, pastry}:  sup = 0.021
{other vegetables, pastry}:  sup = 0.023
{butter milk}:  sup = 0.028
{root vegetables}:  sup = 0.109
{other vegetables, root vegetables}:  sup = 0.047
{whole milk, other vegetables, root vegetables}:  sup = 0.023
{rolls/buns, root vegetables}:  sup = 0.024
{whole milk, root vegetables}:  sup = 0.049
{yogurt, root vegetables}:  sup = 0.026
{waffles}:  sup = 0.038
{salty snack}:  sup = 0.038
{candy}:  sup = 0.03
```

```
{canned beer}:  sup = 0.078
{sausage}:  sup = 0.094
{rolls/buns, sausage}:  sup = 0.031
{soda, sausage}:  sup = 0.024
{whole milk, sausage}:  sup = 0.03
{other vegetables, sausage}:  sup = 0.027
{shopping bags}:  sup = 0.099
{soda, shopping bags}:  sup = 0.025
{whole milk, shopping bags}:  sup = 0.025
{other vegetables, shopping bags}:  sup = 0.023
{brown bread}:  sup = 0.065
{whole milk, brown bread}:  sup = 0.025
{beverages}:  sup = 0.026
{napkins}:  sup = 0.052
{hamburger meat}:  sup = 0.033
{hygiene articles}:  sup = 0.033
{whipped/sour cream}:  sup = 0.072
{whole milk, whipped/sour cream}:  sup = 0.032
{other vegetables, whipped/sour cream}:  sup = 0.029
{yogurt, whipped/sour cream}:  sup = 0.021
{pork}:  sup = 0.058
{whole milk, pork}:  sup = 0.022
{other vegetables, pork}:  sup = 0.022
{berries}:  sup = 0.033
{grapes}:  sup = 0.022
{dessert}:  sup = 0.037
{domestic eggs}:  sup = 0.063
{whole milk, domestic eggs}:  sup = 0.03
{other vegetables, domestic eggs}:  sup = 0.022
{misc. beverages}:  sup = 0.028
{hard cheese}:  sup = 0.025
{cat food}:  sup = 0.023
{ham}:  sup = 0.026
{oil}:  sup = 0.028
{chewing gum}:  sup = 0.021
{ice cream}:  sup = 0.025
{frozen vegetables}:  sup = 0.048
{whole milk, frozen vegetables}:  sup = 0.02
{specialty chocolate}:  sup = 0.03
{frozen meals}:  sup = 0.028
{onions}:  sup = 0.031
{sliced cheese}:  sup = 0.025
{meat}:  sup = 0.026
```

[5]:
```
fp = generate_rules(f, fp_data, min_confidence=0.3, verbose=True)
```

```
{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
{whole milk, other vegetables} ---> {root vegetables}:  conf = 0.31, sup = 0.023
{root vegetables, other vegetables} ---> {whole milk}:  conf = 0.489, sup =
0.023
{root vegetables, whole milk} ---> {other vegetables}:  conf = 0.474, sup =
0.023
```

1 Comments on the results of Fp-Growth Algorithm.

Using the FP-Growth algorithm we have generated some rules that have confidence more than a set minimum confidence. These association rules describe that these items occur together. In other words, if I were to shop other vegetables and yogurt, then there is a 51.3% (conf = 0.513) chance of me also buying whole milk. These types of association rules are used by Amazon, eBay, and other ecommerce websites to suggest what people bought with the item you are currently buying. There are also other interesting uses. For example, IKEA uses this consumer data to build new stores' layout in a way that arranges the items with strong confidence in association rules near each

other to instill the thought of buying in the customers. This has hugely increased the profits for the company and is now incorporated by every store around the world.