Part 1 - Apriori

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

```
import numpy as np
import pandas as pd
import csv

# (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 itemset
# 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 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 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) # prune candidate 1-itemsets
    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
itemsets
        support_data.update(supK) # update the support counts to
reflect pruning
        F.append(Fk) # add the pruned candidate itemsets to the list
of frequent itemsets
        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
candidate
        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
candidate
        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 dataset
    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-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 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 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_confidence, verbose)
            else:
                calc_confidence(freq_set, H1, support_data, rules,
min_confidence, verbose)

    return rules
```

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

```
dataset = load_dataset('/content/grocery.csv')
D = list(map(set, dataset))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:25:
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.
```

dataset **is now a ndarray containing each of the 9835 transactions**

```
type(dataset)
```

```
numpy.ndarray
```

```
dataset.shape
```

```
(9835,)
```

```
dataset[0]
```

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

```
dataset[1]
```

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

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

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

```
set
```

```
D[0]
```

```
{'citrus fruit', 'margarine', 'ready soups', 'semi-finished bread'}
```

TASK 1

```
# Generate candidate itemsets.
C1 = create_candidates(dataset, verbose=True) # candidate 1-itemsets
```

```
{Instant food products, UHT-milk, abrasive cleaner, artif. sweetener,
baby cosmetics, baby food, bags, baking powder, bathroom cleaner,
beef, berries, beverages, bottled beer, bottled water, brandy, brown
```

```
bread, butter, butter milk, cake bar, candles, candy, canned beer,
canned fish, canned fruit, canned vegetables, cat food, cereals,
chewing gum, chicken, chocolate, chocolate marshmallow, citrus fruit,
cleaner, cling film/bags, cocoa drinks, coffee, condensed milk,
cooking chocolate, cookware, cream, cream cheese , curd, curd cheese,
decalcifier, dental care, dessert, detergent, dish cleaner, dishes,
dog food, domestic eggs, female sanitary products, finished products,
fish, flour, flower (seeds), flower soil/fertilizer, frankfurter,
frozen chicken, frozen dessert, frozen fish, frozen fruits, frozen
meals, frozen potato products, frozen vegetables, fruit/vegetable
juice, grapes, hair spray, ham, hamburger meat, hard cheese, herbs,
honey, house keeping products, hygiene articles, ice cream, instant
coffee, jam, ketchup, kitchen towels, kitchen utensil, light bulbs,
liqueur, liquor, liquor (appetizer), liver loaf, long life bakery
product, make up remover, male cosmetics, margarine, mayonnaise, meat,
meat spreads, misc. beverages, mustard, napkins, newspapers, nut
snack, nuts/prunes, oil, onions, organic products, organic sausage,
other vegetables, packaged fruit/vegetables, pasta, pastry, pet care,
photo/film, pickled vegetables, pip fruit, popcorn, pork, pot plants,
potato products, preservation products, processed cheese, prosecco,
pudding powder, ready soups, red/blush wine, rice, roll products ,
rolls/buns, root vegetables, rubbing alcohol, rum, salad dressing,
salt, salty snack, sauces, sausage, seasonal products, semi-finished
bread, shopping bags, skin care, sliced cheese, snack products, soap,
soda, soft cheese, softener, sound storage medium, soups, sparkling
wine, specialty bar, specialty cheese, specialty chocolate, specialty
fat, specialty vegetables, spices, spread cheese, sugar, sweet
spreads, syrup, tea, tidbits, toilet cleaner, tropical fruit, turkey,
vinegar, waffles, whipped/sour cream, whisky, white bread, white wine,
whole milk, yogurt, zwieback}
```

```python
# Prune candidate 1-itemsets via support-based pruning to generate
frequent 1-itemsets.
F1, support_data = support_prune(dataset, C1, 0.1, verbose=True)
```

```
{root vegetables}
{soda}
{bottled water}
{rolls/buns}
{other vegetables}
{whole milk}
{yogurt}
{tropical fruit}

{citrus fruit}:  sup = 0.08276563294356888
{margarine}:  sup = 0.05856634468734113
{ready soups}:  sup = 0.0018301982714794102
{semi-finished bread}:  sup = 0.017691916624300967
{coffee}:  sup = 0.05805795627859685
{tropical fruit}:  sup = 0.10493136756481952
```

```
{yogurt}:  sup = 0.13950177935943062
{whole milk}:  sup = 0.25551601423487547
{cream cheese}:  sup = 0.03965429588205389
{meat spreads}:  sup = 0.004270462633451958
{pip fruit}:  sup = 0.07564819522114896
{condensed milk}:  sup = 0.010269445856634469
{long life bakery product}:  sup = 0.037417386883579054
{other vegetables}:  sup = 0.1934926283680732
{abrasive cleaner}:  sup = 0.0035587188612099642
{butter}:  sup = 0.05541433655312659
{rice}:  sup = 0.007625826131164209
{rolls/buns}:  sup = 0.18393492628368074
{UHT-milk}:  sup = 0.03345195729537367
{bottled beer}:  sup = 0.08052872394509406
{liquor (appetizer)}:  sup = 0.007930859176410779
{pot plants}:  sup = 0.01728520589730554
{cereals}:  sup = 0.0056939501779359435
{bottled water}:  sup = 0.11052364006100661
{chocolate}:  sup = 0.04961870869344179
{white bread}:  sup = 0.042094560244026434
{curd}:  sup = 0.05327910523640061
{dishes}:  sup = 0.01759023894255211
{flour}:  sup = 0.017386883579054397
{beef}:  sup = 0.05246568378240976
{frankfurter}:  sup = 0.058973055414336555
{soda}:  sup = 0.17437722419928825
{chicken}:  sup = 0.04290798169801729
{fruit/vegetable juice}:  sup = 0.0722928317234367
{newspapers}:  sup = 0.07981698017285206
{sugar}:  sup = 0.03385866802236909
{packaged fruit/vegetables}:  sup = 0.013014743263853584
{specialty bar}:  sup = 0.027351296390442297
{butter milk}:  sup = 0.027961362480935434
{pastry}:  sup = 0.08896797153024912
{detergent}:  sup = 0.019217081850533807
{processed cheese}:  sup = 0.016573462125063547
{bathroom cleaner}:  sup = 0.0027452974072191155
{candy}:  sup = 0.0298932384341637
{frozen dessert}:  sup = 0.010777834265378749
{root vegetables}:  sup = 0.10899847483477376
{salty snack}:  sup = 0.03782409761057448
{sweet spreads}:  sup = 0.009049313675648195
{waffles}:  sup = 0.038434163701067614
{canned beer}:  sup = 0.07768174885612608
{sausage}:  sup = 0.09395017793594305
{brown bread}:  sup = 0.06487036095577021
{shopping bags}:  sup = 0.09852567361464158
{beverages}:  sup = 0.026029486527707167
{hamburger meat}:  sup = 0.033248601931875954
{hygiene articles}:  sup = 0.03294356888662939
```

```
{napkins}:  sup = 0.05236400610066091
{spices}:  sup = 0.005185561769191663
{artif. sweetener}:  sup = 0.003253685815963396
{berries}:  sup = 0.033248601931875954
{pork}:  sup = 0.05765124555160142
{whipped/sour cream}:  sup = 0.07168276563294357
{grapes}:  sup = 0.022369089984748347
{dessert}:  sup = 0.03711235383833249
{zwieback}:  sup = 0.006914082358922217
{domestic eggs}:  sup = 0.06344687341128623
{spread cheese}:  sup = 0.011184544992374174
{misc. beverages}:  sup = 0.02836807320793086
{hard cheese}:  sup = 0.024504321301474327
{cat food}:  sup = 0.023284189120488054
{ham}:  sup = 0.026029486527707167
{baking powder}:  sup = 0.017691916624300967
{turkey}:  sup = 0.00813421453990849
{pickled vegetables}:  sup = 0.017895271987798677
{chewing gum}:  sup = 0.021047280122013217
{chocolate marshmallow}:  sup = 0.009049313675648195
{oil}:  sup = 0.02806304016268429
{ice cream}:  sup = 0.025012709710218607
{canned fish}:  sup = 0.015048296898830707
{frozen vegetables}:  sup = 0.04809354346720895
{seasonal products}:  sup = 0.014234875444839857
{curd cheese}:  sup = 0.005083884087442806
{red/blush wine}:  sup = 0.019217081850533807
{frozen potato products}:  sup = 0.008439247585155059
{candles}:  sup = 0.008947635993899338
{flower (seeds)}:  sup = 0.010371123538383325
{specialty chocolate}:  sup = 0.030401626842907982
{specialty fat}:  sup = 0.0036603965429588205
{sparkling wine}:  sup = 0.005592272496187087
{salt}:  sup = 0.010777834265378749
{frozen meals}:  sup = 0.02836807320793086
{canned vegetables}:  sup = 0.010777834265378749
{onions}:  sup = 0.031011692933401117
{herbs}:  sup = 0.01626842907981698
{white wine}:  sup = 0.019013726487036097
{brandy}:  sup = 0.004168784951703101
{photo/film}:  sup = 0.009252669039145907
{sliced cheese}:  sup = 0.024504321301474327
{pasta}:  sup = 0.015048296898830707
{softener}:  sup = 0.005490594814438231
{cling film/bags}:  sup = 0.011387900355871887
{fish}:  sup = 0.0029486527707168276
{male cosmetics}:  sup = 0.004575495678698526
{canned fruit}:  sup = 0.003253685815963396
{Instant food products}:  sup = 0.008032536858159633
{soft cheese}:  sup = 0.01708185053380783
```

```
{honey}:  sup = 0.001525165226232842
{dental care}:  sup = 0.005795627859684799
{popcorn}:  sup = 0.007219115404168785
{cake bar}:  sup = 0.013218098627351297
{snack products}:  sup = 0.003050330452465684
{flower soil/fertilizer}:  sup = 0.0019318759532282665
{specialty cheese}:  sup = 0.008540925266903915
{finished products}:  sup = 0.006507371631926792
{cocoa drinks}:  sup = 0.0022369089984748346
{dog food}:  sup = 0.008540925266903915
{prosecco}:  sup = 0.0020335536349771225
{frozen fish}:  sup = 0.011692933401118455
{make up remover}:  sup = 0.0008134214539908490
{cleaner}:  sup = 0.005083884087442806
{female sanitary products}:  sup = 0.006100660904931368
{cookware}:  sup = 0.0027452974072191155
{dish cleaner}:  sup = 0.010472801220132180
{meat}:  sup = 0.025826131164209457
{tea}:  sup = 0.003863751906456533
{mustard}:  sup = 0.011997966446365024
{house keeping products}:  sup = 0.008337569903406202
{skin care}:  sup = 0.0035587188612099642
{potato products}:  sup = 0.0028469750889679717
{liquor}:  sup = 0.011082867310625319
{pet care}:  sup = 0.009456024402643620
{soups}:  sup = 0.006812404677173360
{rum}:  sup = 0.004473817996949669
{salad dressing}:  sup = 0.000813421453990849
{sauces}:  sup = 0.005490594814438231
{vinegar}:  sup = 0.006507371631926792
{soap}:  sup = 0.0026436197254702592
{hair spray}:  sup = 0.0011184544992374173
{instant coffee}:  sup = 0.007422470767666497
{roll products}:  sup = 0.010269445856634469
{mayonnaise}:  sup = 0.009150991357397052
{rubbing alcohol}:  sup = 0.0010167768174885613
{syrup}:  sup = 0.003253685815963396
{liver loaf}:  sup = 0.005083884087442806
{baby cosmetics}:  sup = 0.0006100660904931368
{organic products}:  sup = 0.0016268429079816980
{nut snack}:  sup = 0.003152008134214540
{kitchen towels}:  sup = 0.005998983223182512
{frozen chicken}:  sup = 0.0006100660904931368
{light bulbs}:  sup = 0.004168784951703101
{ketchup}:  sup = 0.004270462633451958
{jam}:  sup = 0.005388917132689374
{decalcifier}:  sup = 0.001525165226232842
{nuts/prunes}:  sup = 0.003355363497712252
{liqueur}:  sup = 0.0009150991357397051
{organic sausage}:  sup = 0.0022369089984748346
```

```
{cream}:  sup = 0.0013218098627351296
{toilet cleaner}:  sup = 0.0007117437722419929
{specialty vegetables}:  sup = 0.0017285205897305542
{baby food}:  sup = 0.00010167768174885612
{pudding powder}:  sup = 0.002338586680223691
{tidbits}:  sup = 0.002338586680223691
{whisky}:  sup = 0.000813421453990849
{frozen fruits}:  sup = 0.0012201321809862736
{bags}:  sup = 0.0004067107269954245
{cooking chocolate}:  sup = 0.002541942043721403
{sound storage medium}:  sup = 0.00010167768174885612
{kitchen utensil}:  sup = 0.0004067107269954245
{preservation products}:  sup = 0.00020335536349771224
```

```
# Generate all the frequent itemsets using the Apriori algorithm.
F, support_dataa = apriori(dataset, min_support=0.02, verbose=True)
```

```
{meat}:  sup = 0.026
{sliced cheese}:  sup = 0.025
{onions}:  sup = 0.031
{frozen meals}:  sup = 0.028
{specialty chocolate}:  sup = 0.03
{frozen vegetables}:  sup = 0.048
{ice cream}:  sup = 0.025
{oil}:  sup = 0.028
{chewing gum}:  sup = 0.021
{ham}:  sup = 0.026
{cat food}:  sup = 0.023
{hard cheese}:  sup = 0.025
{misc. beverages}:  sup = 0.028
{domestic eggs}:  sup = 0.063
{dessert}:  sup = 0.037
{grapes}:  sup = 0.022
{whipped/sour cream}:  sup = 0.072
{pork}:  sup = 0.058
{berries}:  sup = 0.033
{napkins}:  sup = 0.052
{hygiene articles}:  sup = 0.033
{hamburger meat}:  sup = 0.033
{beverages}:  sup = 0.026
{shopping bags}:  sup = 0.099
{brown bread}:  sup = 0.065
{sausage}:  sup = 0.094
{canned beer}:  sup = 0.078
{waffles}:  sup = 0.038
{salty snack}:  sup = 0.038
{root vegetables}:  sup = 0.109
{candy}:  sup = 0.03
{pastry}:  sup = 0.089
{butter milk}:  sup = 0.028
{specialty bar}:  sup = 0.027
```

```
{sugar}:  sup = 0.034
{newspapers}:  sup = 0.08
{fruit/vegetable juice}:  sup = 0.072
{chicken}:  sup = 0.043
{soda}:  sup = 0.174
{frankfurter}:  sup = 0.059
{beef}:  sup = 0.052
{curd}:  sup = 0.053
{white bread}:  sup = 0.042
{chocolate}:  sup = 0.05
{bottled water}:  sup = 0.111
{bottled beer}:  sup = 0.081
{UHT-milk}:  sup = 0.033
{rolls/buns}:  sup = 0.184
{butter}:  sup = 0.055
{other vegetables}:  sup = 0.193
{long life bakery product}:  sup = 0.037
{pip fruit}:  sup = 0.076
{cream cheese}:  sup = 0.04
{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
{whipped/sour cream, yogurt}:  sup = 0.021
{yogurt, other vegetables}:  sup = 0.043
{pip fruit, other vegetables}:  sup = 0.026
{other vegetables, pastry}:  sup = 0.023
{shopping bags, other vegetables}:  sup = 0.023
{other vegetables, sausage}:  sup = 0.027
{whole milk, bottled beer}:  sup = 0.02
{shopping bags, whole milk}:  sup = 0.025
{citrus fruit, other vegetables}:  sup = 0.029
{fruit/vegetable juice, whole milk}:  sup = 0.027
{whole milk, frankfurter}:  sup = 0.021
{whole milk, newspapers}:  sup = 0.027
{whole milk, margarine}:  sup = 0.024
{pip fruit, tropical fruit}:  sup = 0.02
{whole milk, pip fruit}:  sup = 0.03
{whole milk, rolls/buns}:  sup = 0.057
{whole milk, beef}:  sup = 0.021
{whole milk, sausage}:  sup = 0.03
{whole milk, frozen vegetables}:  sup = 0.02
{rolls/buns, pastry}:  sup = 0.021
{fruit/vegetable juice, other vegetables}:  sup = 0.021
{domestic eggs, other vegetables}:  sup = 0.022
{butter, other vegetables}:  sup = 0.02
{yogurt, rolls/buns}:  sup = 0.034
{bottled water, soda}:  sup = 0.029
```

```
{tropical fruit, soda}:  sup = 0.021
{yogurt, soda}:  sup = 0.027
{whole milk, pastry}:  sup = 0.033
{yogurt, root vegetables}:  sup = 0.026
{whole milk, brown bread}:  sup = 0.025
{whole milk, domestic eggs}:  sup = 0.03
{pastry, soda}:  sup = 0.021
{whole milk, soda}:  sup = 0.04
{other vegetables, soda}:  sup = 0.033
{whole milk, pork}:  sup = 0.022
{pork, other vegetables}:  sup = 0.022
{whole milk, whipped/sour cream}:  sup = 0.032
{whipped/sour cream, other vegetables}:  sup = 0.029
{whole milk, root vegetables}:  sup = 0.049
{rolls/buns, bottled water}:  sup = 0.024
{shopping bags, soda}:  sup = 0.025
{rolls/buns, sausage}:  sup = 0.031
{sausage, soda}:  sup = 0.024
{rolls/buns, tropical fruit}:  sup = 0.025
{root vegetables, tropical fruit}:  sup = 0.021
{other vegetables, root vegetables}:  sup = 0.047
{rolls/buns, root vegetables}:  sup = 0.024
{rolls/buns, soda}:  sup = 0.038
{yogurt, citrus fruit}:  sup = 0.022
{whole milk, citrus fruit}:  sup = 0.031
{whole milk, tropical fruit}:  sup = 0.042
{yogurt, bottled water}:  sup = 0.023
{whole milk, bottled water}:  sup = 0.034
{whole milk, curd}:  sup = 0.026
{other vegetables, tropical fruit}:  sup = 0.036
{other vegetables, bottled water}:  sup = 0.025
{other vegetables, rolls/buns}:  sup = 0.043
{whole milk, yogurt}:  sup = 0.056
{whole milk, butter}:  sup = 0.028
{whole milk, other vegetables}:  sup = 0.075
{yogurt, tropical fruit}:  sup = 0.029
{whole milk, yogurt, other vegetables}:  sup = 0.022
{whole milk, other vegetables, root vegetables}:  sup = 0.023
```

```
# Generate the association rules from a list of frequent itemsets.
H = generate_rules(F, support_dataa, min_confidence=0.1, verbose=True)
```

```
{yogurt} ---> {whipped/sour cream}:  conf = 0.149, sup = 0.021
{whipped/sour cream} ---> {yogurt}:  conf = 0.289, sup = 0.021
{other vegetables} ---> {yogurt}:  conf = 0.224, sup = 0.043
{yogurt} ---> {other vegetables}:  conf = 0.311, sup = 0.043
{other vegetables} ---> {pip fruit}:  conf = 0.135, sup = 0.026
{pip fruit} ---> {other vegetables}:  conf = 0.345, sup = 0.026
{pastry} ---> {other vegetables}:  conf = 0.254, sup = 0.023
{other vegetables} ---> {pastry}:  conf = 0.117, sup = 0.023
{other vegetables} ---> {shopping bags}:  conf = 0.12, sup = 0.023
```

```
{shopping bags} ---> {other vegetables}:  conf = 0.235, sup = 0.023
{sausage} ---> {other vegetables}:  conf = 0.287, sup = 0.027
{other vegetables} ---> {sausage}:  conf = 0.139, sup = 0.027
{bottled beer} ---> {whole milk}:  conf = 0.254, sup = 0.02
{shopping bags} ---> {whole milk}:  conf = 0.249, sup = 0.025
{other vegetables} ---> {citrus fruit}:  conf = 0.149, sup = 0.029
{citrus fruit} ---> {other vegetables}:  conf = 0.349, sup = 0.029
{whole milk} ---> {fruit/vegetable juice}:  conf = 0.104, sup = 0.027
{fruit/vegetable juice} ---> {whole milk}:  conf = 0.368, sup = 0.027
{frankfurter} ---> {whole milk}:  conf = 0.348, sup = 0.021
{newspapers} ---> {whole milk}:  conf = 0.343, sup = 0.027
{whole milk} ---> {newspapers}:  conf = 0.107, sup = 0.027
{margarine} ---> {whole milk}:  conf = 0.413, sup = 0.024
{tropical fruit} ---> {pip fruit}:  conf = 0.195, sup = 0.02
{pip fruit} ---> {tropical fruit}:  conf = 0.27, sup = 0.02
{pip fruit} ---> {whole milk}:  conf = 0.398, sup = 0.03
{whole milk} ---> {pip fruit}:  conf = 0.118, sup = 0.03
{rolls/buns} ---> {whole milk}:  conf = 0.308, sup = 0.057
{whole milk} ---> {rolls/buns}:  conf = 0.222, sup = 0.057
{beef} ---> {whole milk}:  conf = 0.405, sup = 0.021
{sausage} ---> {whole milk}:  conf = 0.318, sup = 0.03
{whole milk} ---> {sausage}:  conf = 0.117, sup = 0.03
{frozen vegetables} ---> {whole milk}:  conf = 0.425, sup = 0.02
{pastry} ---> {rolls/buns}:  conf = 0.235, sup = 0.021
{rolls/buns} ---> {pastry}:  conf = 0.114, sup = 0.021
{other vegetables} ---> {fruit/vegetable juice}:  conf = 0.109, sup =
0.021
{fruit/vegetable juice} ---> {other vegetables}:  conf = 0.291, sup =
0.021
{other vegetables} ---> {domestic eggs}:  conf = 0.115, sup = 0.022
{domestic eggs} ---> {other vegetables}:  conf = 0.351, sup = 0.022
{other vegetables} ---> {butter}:  conf = 0.104, sup = 0.02
{butter} ---> {other vegetables}:  conf = 0.361, sup = 0.02
{rolls/buns} ---> {yogurt}:  conf = 0.187, sup = 0.034
{yogurt} ---> {rolls/buns}:  conf = 0.246, sup = 0.034
{soda} ---> {bottled water}:  conf = 0.166, sup = 0.029
{bottled water} ---> {soda}:  conf = 0.262, sup = 0.029
{soda} ---> {tropical fruit}:  conf = 0.12, sup = 0.021
{tropical fruit} ---> {soda}:  conf = 0.199, sup = 0.021
{soda} ---> {yogurt}:  conf = 0.157, sup = 0.027
{yogurt} ---> {soda}:  conf = 0.196, sup = 0.027
{pastry} ---> {whole milk}:  conf = 0.374, sup = 0.033
{whole milk} ---> {pastry}:  conf = 0.13, sup = 0.033
{root vegetables} ---> {yogurt}:  conf = 0.237, sup = 0.026
{yogurt} ---> {root vegetables}:  conf = 0.185, sup = 0.026
{brown bread} ---> {whole milk}:  conf = 0.389, sup = 0.025
{domestic eggs} ---> {whole milk}:  conf = 0.473, sup = 0.03
{whole milk} ---> {domestic eggs}:  conf = 0.117, sup = 0.03
{soda} ---> {pastry}:  conf = 0.121, sup = 0.021
{pastry} ---> {soda}:  conf = 0.237, sup = 0.021
```

```
{soda} ---> {whole milk}:  conf = 0.23, sup = 0.04
{whole milk} ---> {soda}:  conf = 0.157, sup = 0.04
{soda} ---> {other vegetables}:  conf = 0.188, sup = 0.033
{other vegetables} ---> {soda}:  conf = 0.169, sup = 0.033
{pork} ---> {whole milk}:  conf = 0.384, sup = 0.022
{other vegetables} ---> {pork}:  conf = 0.112, sup = 0.022
{pork} ---> {other vegetables}:  conf = 0.376, sup = 0.022
{whipped/sour cream} ---> {whole milk}:  conf = 0.45, sup = 0.032
{whole milk} ---> {whipped/sour cream}:  conf = 0.126, sup = 0.032
{other vegetables} ---> {whipped/sour cream}:  conf = 0.149, sup =
0.029
{whipped/sour cream} ---> {other vegetables}:  conf = 0.403, sup =
0.029
{root vegetables} ---> {whole milk}:  conf = 0.449, sup = 0.049
{whole milk} ---> {root vegetables}:  conf = 0.191, sup = 0.049
{bottled water} ---> {rolls/buns}:  conf = 0.219, sup = 0.024
{rolls/buns} ---> {bottled water}:  conf = 0.132, sup = 0.024
{soda} ---> {shopping bags}:  conf = 0.141, sup = 0.025
{shopping bags} ---> {soda}:  conf = 0.25, sup = 0.025
{sausage} ---> {rolls/buns}:  conf = 0.326, sup = 0.031
{rolls/buns} ---> {sausage}:  conf = 0.166, sup = 0.031
{soda} ---> {sausage}:  conf = 0.139, sup = 0.024
{sausage} ---> {soda}:  conf = 0.259, sup = 0.024
{tropical fruit} ---> {rolls/buns}:  conf = 0.234, sup = 0.025
{rolls/buns} ---> {tropical fruit}:  conf = 0.134, sup = 0.025
{tropical fruit} ---> {root vegetables}:  conf = 0.201, sup = 0.021
{root vegetables} ---> {tropical fruit}:  conf = 0.193, sup = 0.021
{root vegetables} ---> {other vegetables}:  conf = 0.435, sup = 0.047
{other vegetables} ---> {root vegetables}:  conf = 0.245, sup = 0.047
{root vegetables} ---> {rolls/buns}:  conf = 0.223, sup = 0.024
{rolls/buns} ---> {root vegetables}:  conf = 0.132, sup = 0.024
{soda} ---> {rolls/buns}:  conf = 0.22, sup = 0.038
{rolls/buns} ---> {soda}:  conf = 0.208, sup = 0.038
{citrus fruit} ---> {yogurt}:  conf = 0.262, sup = 0.022
{yogurt} ---> {citrus fruit}:  conf = 0.155, sup = 0.022
{citrus fruit} ---> {whole milk}:  conf = 0.369, sup = 0.031
{whole milk} ---> {citrus fruit}:  conf = 0.119, sup = 0.031
{tropical fruit} ---> {whole milk}:  conf = 0.403, sup = 0.042
{whole milk} ---> {tropical fruit}:  conf = 0.166, sup = 0.042
{bottled water} ---> {yogurt}:  conf = 0.208, sup = 0.023
{yogurt} ---> {bottled water}:  conf = 0.165, sup = 0.023
{bottled water} ---> {whole milk}:  conf = 0.311, sup = 0.034
{whole milk} ---> {bottled water}:  conf = 0.135, sup = 0.034
{curd} ---> {whole milk}:  conf = 0.49, sup = 0.026
{whole milk} ---> {curd}:  conf = 0.102, sup = 0.026
{tropical fruit} ---> {other vegetables}:  conf = 0.342, sup = 0.036
{other vegetables} ---> {tropical fruit}:  conf = 0.185, sup = 0.036
{bottled water} ---> {other vegetables}:  conf = 0.224, sup = 0.025
{other vegetables} ---> {bottled water}:  conf = 0.128, sup = 0.025
{rolls/buns} ---> {other vegetables}:  conf = 0.232, sup = 0.043
```

```
{other vegetables} ---> {rolls/buns}:  conf = 0.22, sup = 0.043
{yogurt} ---> {whole milk}:  conf = 0.402, sup = 0.056
{whole milk} ---> {yogurt}:  conf = 0.219, sup = 0.056
{butter} ---> {whole milk}:  conf = 0.497, sup = 0.028
{whole milk} ---> {butter}:  conf = 0.108, sup = 0.028
{other vegetables} ---> {whole milk}:  conf = 0.387, sup = 0.075
{whole milk} ---> {other vegetables}:  conf = 0.293, sup = 0.075
{tropical fruit} ---> {yogurt}:  conf = 0.279, sup = 0.029
{yogurt} ---> {tropical fruit}:  conf = 0.21, sup = 0.029
{yogurt, other vegetables} ---> {whole milk}:  conf = 0.513, sup =
0.022
{whole milk, other vegetables} ---> {yogurt}:  conf = 0.298, sup =
0.022
{whole milk, yogurt} ---> {other vegetables}:  conf = 0.397, sup =
0.022
{other vegetables} ---> {whole milk, yogurt}:  conf = 0.115, sup =
0.022
{yogurt} ---> {whole milk, other vegetables}:  conf = 0.16, sup =
0.022
{other vegetables, root 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
{root vegetables} ---> {whole milk, other vegetables}:  conf = 0.213,
sup = 0.023
{other vegetables} ---> {whole milk, root vegetables}:  conf = 0.12,
sup = 0.023
```

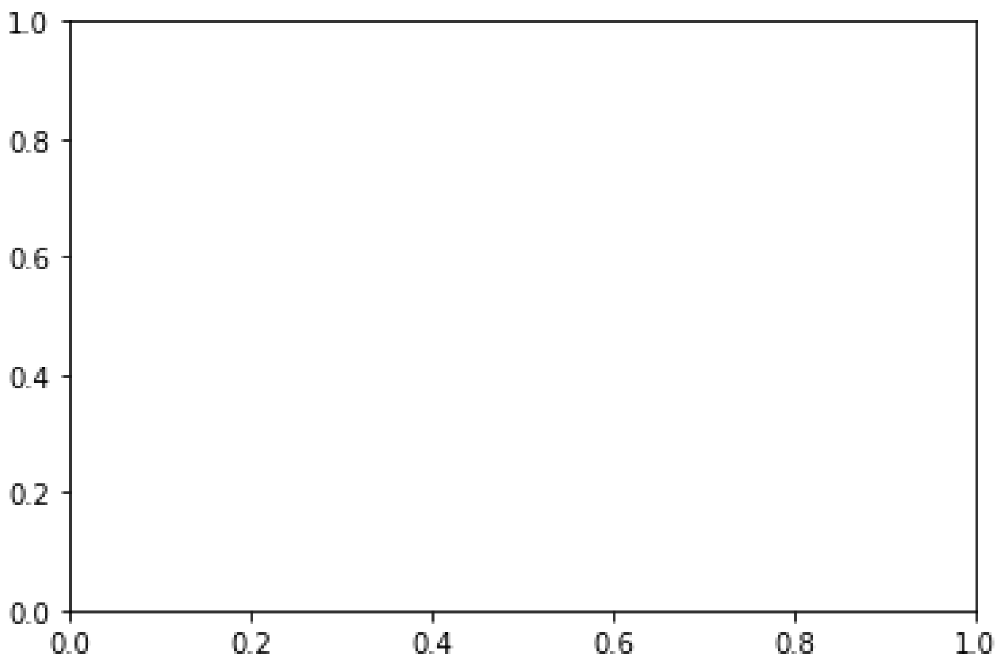
Task 2

```
import matplotlib.pyplot as plt

xpoints = H

plt.plot(xpoints)
plt.show()
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-31-02fb6ccb8f5f> in <module>()
      3 xpoints = H
      4
----> 5 plt.plot(xpoints)
      6 plt.show()

/usr/local/lib/python3.7/dist-packages/matplotlib/pyplot.py in
plot(scalex, scaley, data, *args, **kwargs)
```

```
   2761     return gca().plot(
   2762         *args, scalex=scalex, scaley=scaley, **({"data": data}
if data
-> 2763         is not None else {}), **kwargs)
   2764 
   2765 

/usr/local/lib/python3.7/dist-packages/matplotlib/axes/_axes.py in
plot(self, scalex, scaley, data, *args, **kwargs)
   1645         """
   1646         kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
-> 1647         lines = [*self._get_lines(*args, data=data, **kwargs)]
   1648         for line in lines:
   1649             self.add_line(line)

/usr/local/lib/python3.7/dist-packages/matplotlib/axes/_base.py in
__call__(self, *args, **kwargs)
    214                 this += args[0],
    215                 args = args[1:]
--> 216             yield from self._plot_args(this, kwargs)
    217 
    218     def get_next_color(self):

/usr/local/lib/python3.7/dist-packages/matplotlib/axes/_base.py in
_plot_args(self, tup, kwargs)
    337             self.axes.xaxis.update_units(x)
    338         if self.axes.yaxis is not None:
--> 339             self.axes.yaxis.update_units(y)
    340 
    341         if x.shape[0] != y.shape[0]:

/usr/local/lib/python3.7/dist-packages/matplotlib/axis.py in
update_units(self, data)
   1514         neednew = self.converter != converter
   1515         self.converter = converter
-> 1516         default = self.converter.default_units(data, self)
   1517         if default is not None and self.units is None:
   1518             self.set_units(default)

/usr/local/lib/python3.7/dist-packages/matplotlib/category.py in
default_units(data, axis)
    105         # the conversion call stack is default_units ->
axis_info -> convert
    106         if axis.units is None:
--> 107             axis.set_units(UnitData(data))
    108         else:
    109             axis.units.update(data)

/usr/local/lib/python3.7/dist-packages/matplotlib/category.py in
__init__(self, data)
```

```
    173         self._counter = itertools.count()
    174         if data is not None:
--> 175             self.update(data)
    176 
    177     @staticmethod

/usr/local/lib/python3.7/dist-packages/matplotlib/category.py in
update(self, data)
    208         # check if convertible to number:
    209         convertible = True
--> 210         for val in OrderedDict.fromkeys(data):
    211             # OrderedDict just iterates over unique values in
data.
    212             cbook._check_isinstance((str, bytes), value=val)

TypeError: unhashable type: 'numpy.ndarray'
```



Part 2 - FP Growth

FP-growth ("frequent pattern growth") is an algorithm for frequent item set mining and association rule learning over transactional databases.

In the first pass, the algorithm counts occurrence of items (attribute-value pairs) in the dataset, and stores them to 'header table'. In the second pass, it builds the FP-tree structure by inserting instances. Items in each instance have to be sorted by descending order of their frequency in the dataset, so that the tree can be processed quickly. Items in each instance that do not meet minimum coverage threshold are discarded. If many instances share most frequent items, FP-tree provides high compression close to tree root.

Recursive processing of this compressed version of main dataset grows large item sets directly, instead of generating candidate items and testing them against the entire database. Growth starts from the bottom of the header table (having longest branches), by finding all instances matching given condition. New tree is created, with counts projected from the original tree corresponding to the set of instances that are conditional on the attribute, with each node getting sum of its children counts. Recursive growth ends when no individual items conditional on the attribute meet minimum support threshold, and processing continues on the remaining header items of the original FP-tree.

```
# (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
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
```

```python
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 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
```

First, we load an example market basket transactions dataset (a list of lists), map it to a 'set' datatype (for programmatic reasons), and print the transactions. We import and use pprint to format the output.

```
import pprint

def load_dataset():
    """Loads an example of market basket transactions for testing
purposes.

    Returns
    -------
    A list (database) of lists (transactions). Each element of a
transaction
    is an item.
    """
    return [['Bread', 'Milk'],
            ['Bread', 'Diapers', 'Beer', 'Eggs'],
            ['Milk', 'Diapers', 'Beer', 'Coke'],
            ['Bread', 'Milk', 'Diapers', 'Beer'],
            ['Bread', 'Milk', 'Diapers', 'Coke']]

dataset = load_dataset() # list of transactions; each transaction is a
list of items
D = map(set, dataset) # set of transactions; each transaction is a
list of items

pprint.pprint(dataset)
```

```
[['Bread', 'Milk'],
 ['Bread', 'Diapers', 'Beer', 'Eggs'],
 ['Milk', 'Diapers', 'Beer', 'Coke'],
 ['Bread', 'Milk', 'Diapers', 'Beer'],
 ['Bread', 'Milk', 'Diapers', 'Coke']]
```

Now we input the initial dataset into the 'fpgrowth' function (along with a minimum support threshold) and it will return a list of all the frequent itemsets:

```
# Generate all the frequent itemsets using the FP-growth algorithm.
F, support_data = fpgrowth(dataset, min_support=0.6, verbose=True)
```

```
{Bread}:  sup = 0.8
{Milk}:  sup = 0.8
{Bread, Milk}:  sup = 0.6
{Diapers}:  sup = 0.8
{Bread, Diapers}:  sup = 0.6
{Milk, Diapers}:  sup = 0.6
{Beer}:  sup = 0.6
{Diapers, Beer}:  sup = 0.6
```

Given a frequent itemset (here, extracted by the FP-growth algorithm), we can generate the association rules with high support and confidence (via the generate_rules function):

```
# Generate the association rules from a list of frequent itemsets.
H = generate_rules(F, support_data, min_confidence=0.8, verbose=True)
```

```
{Beer} ---> {Diapers}:  conf = 1.0, sup = 0.6
```

Part 2 - Interest Factor

```
F_fp , sd = fpgrowth(dataset, min_support = 0.04, verbose=True)
H_fp = generate_rules(F_fp, sd, min_confidence= 0.3, verbose=True)
```

```
{Bread}:  sup = 0.8
{Milk}:  sup = 0.8
{Bread, Milk}:  sup = 0.6
{Diapers}:  sup = 0.8
{Bread, Diapers}:  sup = 0.6
{Milk, Diapers}:  sup = 0.6
{Bread, Milk, Diapers}:  sup = 0.4
{Beer}:  sup = 0.6
{Bread, Beer}:  sup = 0.4
{Diapers, Beer}:  sup = 0.6
{Bread, Diapers, Beer}:  sup = 0.4
{Milk, Diapers, Beer}:  sup = 0.4
{Bread, Milk, Diapers, Beer}:  sup = 0.2
{Milk, Beer}:  sup = 0.4
{Bread, Milk, Beer}:  sup = 0.2
{Eggs}:  sup = 0.2
{Bread, Eggs}:  sup = 0.2
{Diapers, Eggs}:  sup = 0.2
{Bread, Diapers, Eggs}:  sup = 0.2
{Beer, Eggs}:  sup = 0.2
{Bread, Beer, Eggs}:  sup = 0.2
{Diapers, Beer, Eggs}:  sup = 0.2
{Bread, Diapers, Beer, Eggs}:  sup = 0.2
{Coke}:  sup = 0.4
{Milk, Coke}:  sup = 0.4
{Bread, Milk, Coke}:  sup = 0.2
{Diapers, Coke}:  sup = 0.4
{Milk, Diapers, Coke}:  sup = 0.4
{Bread, Milk, Diapers, Coke}:  sup = 0.2
{Bread, Diapers, Coke}:  sup = 0.2
{Beer, Coke}:  sup = 0.2
{Milk, Beer, Coke}:  sup = 0.2
{Diapers, Beer, Coke}:  sup = 0.2
{Milk, Diapers, Beer, Coke}:  sup = 0.2
{Bread, Coke}:  sup = 0.2
{Bread} ---> {Milk}:  conf = 0.75, sup = 0.6
{Milk} ---> {Bread}:  conf = 0.75, sup = 0.6
{Diapers} ---> {Bread}:  conf = 0.75, sup = 0.6
{Bread} ---> {Diapers}:  conf = 0.75, sup = 0.6
{Diapers} ---> {Milk}:  conf = 0.75, sup = 0.6
{Milk} ---> {Diapers}:  conf = 0.75, sup = 0.6
{Bread} ---> {Beer}:  conf = 0.5, sup = 0.4
{Beer} ---> {Bread}:  conf = 0.667, sup = 0.4
```

```
{Diapers} ---> {Beer}:  conf = 0.75, sup = 0.6
{Beer} ---> {Diapers}:  conf = 1.0, sup = 0.6
{Beer} ---> {Milk}:  conf = 0.667, sup = 0.4
{Milk} ---> {Beer}:  conf = 0.5, sup = 0.4
{Eggs} ---> {Bread}:  conf = 1.0, sup = 0.2
{Eggs} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Beer} ---> {Eggs}:  conf = 0.333, sup = 0.2
{Eggs} ---> {Beer}:  conf = 1.0, sup = 0.2
{Coke} ---> {Milk}:  conf = 1.0, sup = 0.4
{Milk} ---> {Coke}:  conf = 0.5, sup = 0.4
{Diapers} ---> {Coke}:  conf = 0.5, sup = 0.4
{Coke} ---> {Diapers}:  conf = 1.0, sup = 0.4
{Coke} ---> {Beer}:  conf = 0.5, sup = 0.2
{Beer} ---> {Coke}:  conf = 0.333, sup = 0.2
{Coke} ---> {Bread}:  conf = 0.5, sup = 0.2
{Bread, Diapers} ---> {Milk}:  conf = 0.667, sup = 0.4
{Milk, Diapers} ---> {Bread}:  conf = 0.667, sup = 0.4
{Milk, Bread} ---> {Diapers}:  conf = 0.667, sup = 0.4
{Diapers} ---> {Milk, Bread}:  conf = 0.5, sup = 0.4
{Bread} ---> {Milk, Diapers}:  conf = 0.5, sup = 0.4
{Milk} ---> {Bread, Diapers}:  conf = 0.5, sup = 0.4
{Bread, Diapers} ---> {Beer}:  conf = 0.667, sup = 0.4
{Beer, Diapers} ---> {Bread}:  conf = 0.667, sup = 0.4
{Beer, Bread} ---> {Diapers}:  conf = 1.0, sup = 0.4
{Diapers} ---> {Beer, Bread}:  conf = 0.5, sup = 0.4
{Bread} ---> {Beer, Diapers}:  conf = 0.5, sup = 0.4
{Beer} ---> {Bread, Diapers}:  conf = 0.667, sup = 0.4
{Beer, Diapers} ---> {Milk}:  conf = 0.667, sup = 0.4
{Milk, Diapers} ---> {Beer}:  conf = 0.667, sup = 0.4
{Milk, Beer} ---> {Diapers}:  conf = 1.0, sup = 0.4
{Diapers} ---> {Milk, Beer}:  conf = 0.5, sup = 0.4
{Beer} ---> {Milk, Diapers}:  conf = 0.667, sup = 0.4
{Milk} ---> {Beer, Diapers}:  conf = 0.5, sup = 0.4
{Beer, Bread} ---> {Milk}:  conf = 0.5, sup = 0.2
{Milk, Bread} ---> {Beer}:  conf = 0.333, sup = 0.2
{Milk, Beer} ---> {Bread}:  conf = 0.5, sup = 0.2
{Beer} ---> {Milk, Bread}:  conf = 0.333, sup = 0.2
{Bread, Diapers} ---> {Eggs}:  conf = 0.333, sup = 0.2
{Eggs, Diapers} ---> {Bread}:  conf = 1.0, sup = 0.2
{Eggs, Bread} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Eggs} ---> {Bread, Diapers}:  conf = 1.0, sup = 0.2
{Beer, Bread} ---> {Eggs}:  conf = 0.5, sup = 0.2
{Eggs, Bread} ---> {Beer}:  conf = 1.0, sup = 0.2
{Eggs, Beer} ---> {Bread}:  conf = 1.0, sup = 0.2
{Beer} ---> {Eggs, Bread}:  conf = 0.333, sup = 0.2
{Eggs} ---> {Beer, Bread}:  conf = 1.0, sup = 0.2
{Beer, Diapers} ---> {Eggs}:  conf = 0.333, sup = 0.2
{Eggs, Diapers} ---> {Beer}:  conf = 1.0, sup = 0.2
{Eggs, Beer} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Beer} ---> {Eggs, Diapers}:  conf = 0.333, sup = 0.2
```

```
{Eggs} ---> {Beer, Diapers}:  conf = 1.0, sup = 0.2
{Coke, Bread} ---> {Milk}:  conf = 1.0, sup = 0.2
{Milk, Bread} ---> {Coke}:  conf = 0.333, sup = 0.2
{Milk, Coke} ---> {Bread}:  conf = 0.5, sup = 0.2
{Coke} ---> {Milk, Bread}:  conf = 0.5, sup = 0.2
{Coke, Diapers} ---> {Milk}:  conf = 1.0, sup = 0.4
{Milk, Diapers} ---> {Coke}:  conf = 0.667, sup = 0.4
{Milk, Coke} ---> {Diapers}:  conf = 1.0, sup = 0.4
{Diapers} ---> {Milk, Coke}:  conf = 0.5, sup = 0.4
{Coke} ---> {Milk, Diapers}:  conf = 1.0, sup = 0.4
{Milk} ---> {Coke, Diapers}:  conf = 0.5, sup = 0.4
{Bread, Diapers} ---> {Coke}:  conf = 0.333, sup = 0.2
{Coke, Diapers} ---> {Bread}:  conf = 0.5, sup = 0.2
{Coke, Bread} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Coke} ---> {Bread, Diapers}:  conf = 0.5, sup = 0.2
{Beer, Coke} ---> {Milk}:  conf = 1.0, sup = 0.2
{Milk, Coke} ---> {Beer}:  conf = 0.5, sup = 0.2
{Milk, Beer} ---> {Coke}:  conf = 0.5, sup = 0.2
{Coke} ---> {Milk, Beer}:  conf = 0.5, sup = 0.2
{Beer} ---> {Milk, Coke}:  conf = 0.333, sup = 0.2
{Coke, Diapers} ---> {Beer}:  conf = 0.5, sup = 0.2
{Beer, Diapers} ---> {Coke}:  conf = 0.333, sup = 0.2
{Beer, Coke} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Coke} ---> {Beer, Diapers}:  conf = 0.5, sup = 0.2
{Beer} ---> {Coke, Diapers}:  conf = 0.333, sup = 0.2
{Beer, Bread, Diapers} ---> {Milk}:  conf = 0.5, sup = 0.2
{Milk, Bread, Diapers} ---> {Beer}:  conf = 0.5, sup = 0.2
{Milk, Beer, Diapers} ---> {Bread}:  conf = 0.5, sup = 0.2
{Milk, Beer, Bread} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Bread, Diapers} ---> {Milk, Beer}:  conf = 0.333, sup = 0.2
{Beer, Diapers} ---> {Milk, Bread}:  conf = 0.333, sup = 0.2
{Beer, Bread} ---> {Milk, Diapers}:  conf = 0.5, sup = 0.2
{Milk, Diapers} ---> {Beer, Bread}:  conf = 0.333, sup = 0.2
{Milk, Bread} ---> {Beer, Diapers}:  conf = 0.333, sup = 0.2
{Milk, Beer} ---> {Bread, Diapers}:  conf = 0.5, sup = 0.2
{Beer} ---> {Milk, Bread, Diapers}:  conf = 0.333, sup = 0.2
{Beer, Bread, Diapers} ---> {Eggs}:  conf = 0.5, sup = 0.2
{Eggs, Bread, Diapers} ---> {Beer}:  conf = 1.0, sup = 0.2
{Eggs, Beer, Diapers} ---> {Bread}:  conf = 1.0, sup = 0.2
{Eggs, Beer, Bread} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Bread, Diapers} ---> {Eggs, Beer}:  conf = 0.333, sup = 0.2
{Beer, Diapers} ---> {Eggs, Bread}:  conf = 0.333, sup = 0.2
{Beer, Bread} ---> {Eggs, Diapers}:  conf = 0.5, sup = 0.2
{Eggs, Diapers} ---> {Beer, Bread}:  conf = 1.0, sup = 0.2
{Eggs, Bread} ---> {Beer, Diapers}:  conf = 1.0, sup = 0.2
{Eggs, Beer} ---> {Bread, Diapers}:  conf = 1.0, sup = 0.2
{Beer} ---> {Eggs, Bread, Diapers}:  conf = 0.333, sup = 0.2
{Eggs} ---> {Beer, Bread, Diapers}:  conf = 1.0, sup = 0.2
{Coke, Bread, Diapers} ---> {Milk}:  conf = 1.0, sup = 0.2
{Milk, Bread, Diapers} ---> {Coke}:  conf = 0.5, sup = 0.2
```

```
{Milk, Coke, Diapers} ---> {Bread}:  conf = 0.5, sup = 0.2
{Milk, Coke, Bread} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Bread, Diapers} ---> {Milk, Coke}:  conf = 0.333, sup = 0.2
{Coke, Diapers} ---> {Milk, Bread}:  conf = 0.5, sup = 0.2
{Coke, Bread} ---> {Milk, Diapers}:  conf = 1.0, sup = 0.2
{Milk, Diapers} ---> {Coke, Bread}:  conf = 0.333, sup = 0.2
{Milk, Bread} ---> {Coke, Diapers}:  conf = 0.333, sup = 0.2
{Milk, Coke} ---> {Bread, Diapers}:  conf = 0.5, sup = 0.2
{Coke} ---> {Milk, Bread, Diapers}:  conf = 0.5, sup = 0.2
{Beer, Coke, Diapers} ---> {Milk}:  conf = 1.0, sup = 0.2
{Milk, Coke, Diapers} ---> {Beer}:  conf = 0.5, sup = 0.2
{Milk, Beer, Diapers} ---> {Coke}:  conf = 0.5, sup = 0.2
{Milk, Beer, Coke} ---> {Diapers}:  conf = 1.0, sup = 0.2
{Coke, Diapers} ---> {Milk, Beer}:  conf = 0.5, sup = 0.2
{Beer, Diapers} ---> {Milk, Coke}:  conf = 0.333, sup = 0.2
{Beer, Coke} ---> {Milk, Diapers}:  conf = 1.0, sup = 0.2
{Milk, Diapers} ---> {Beer, Coke}:  conf = 0.333, sup = 0.2
{Milk, Coke} ---> {Beer, Diapers}:  conf = 0.5, sup = 0.2
{Milk, Beer} ---> {Coke, Diapers}:  conf = 0.5, sup = 0.2
{Coke} ---> {Milk, Beer, Diapers}:  conf = 0.5, sup = 0.2
{Beer} ---> {Milk, Coke, Diapers}:  conf = 0.333, sup = 0.2
```

```python
#IF
data = []
rules = []
support = []
i_f = {}

for i in range(len(H_fp)):
    sA=sd[H_fp[i][0]]
    sB=sd[H_fp[i][1]]
    temp1,temp2="",""
    for item1 in H_fp[i][0]:
        temp1 = item1
    for item2 in H_fp[i][1]:
        temp2 = item2
    t=frozenset([temp1,temp2])
    sAB=sd[t]
    inf=sAB/(sA*sB)
    item=temp1+"-->"+temp2
    rules.append([item,sAB,H_fp[i][2],inf])
    i_f[item] = inf
print(i_f)
```

```
{'Bread-->Milk': 3.7499999999999999, 'Milk-->Bread':
0.9374999999999998, 'Diapers-->Bread': 5.0, 'Bread-->Diapers': 2.5,
'Diapers-->Milk': 3.7499999999999999, 'Milk-->Diapers':
1.8749999999999996, 'Bread-->Beer': 3.3333333333333335, 'Beer--
>Bread': 3.3333333333333335, 'Diapers-->Beer': 3.7499999999999999,
'Beer-->Diapers': 2.5, 'Beer-->Milk': 0.8333333333333334, 'Milk--
```

```
>Beer': 0.8333333333333334, 'Eggs-->Bread': 2.4999999999999996,
'Eggs-->Diapers': 2.4999999999999996, 'Beer-->Eggs':
1.6666666666666667, 'Eggs-->Beer': 1.6666666666666667, 'Coke-->Milk':
2.4999999999999996, 'Milk-->Coke': 1.2499999999999998, 'Diapers--
>Coke': 3.3333333333333335, 'Coke-->Diapers': 2.4999999999999996,
'Coke-->Beer': 1.2499999999999998, 'Beer-->Coke': 0.8333333333333334,
'Coke-->Bread': 0.8333333333333334, 'Diapers-->Eggs':
2.4999999999999996, 'Bread-->Eggs': 2.4999999999999996, 'Bread--
>Coke': 0.8333333333333334}
```

```
names=["ITEM","SUPPORT","CONFIDENCE","IF"]
df = pd.DataFrame(rules,columns=names)
sup=df.sort_values(by=['SUPPORT'],ascending=True).head(5)
print(sup)
```

```
               ITEM  SUPPORT  CONFIDENCE        IF
105  Eggs-->Diapers      0.2    1.000000  2.500000
62     Coke-->Bread      0.2    0.500000  0.833333
22     Coke-->Bread      0.2    0.500000  0.625000
21      Beer-->Coke      0.2    0.333333  0.833333
20      Coke-->Beer      0.2    0.500000  0.833333
```

```
conf=df.sort_values(by=['CONFIDENCE'],ascending=True).head(5)
print(conf)
```

```
               ITEM  SUPPORT  CONFIDENCE        IF
128  Beer-->Diapers      0.6    0.333333  2.500000
21      Beer-->Coke      0.2    0.333333  0.833333
54   Diapers-->Eggs      0.2    0.333333  1.666667
104  Beer-->Diapers      0.6    0.333333  5.000000
82   Beer-->Diapers      0.6    0.333333  2.500000
```

```
inf=df.sort_values(by=['IF'],ascending=True).head(5)
print(inf)
```

```
            ITEM  SUPPORT  CONFIDENCE        IF
22  Coke-->Bread      0.2    0.500000  0.625000
61  Coke-->Bread      0.2    0.500000  0.625000
20   Coke-->Beer      0.2    0.500000  0.833333
74   Coke-->Beer      0.2    0.500000  0.833333
77   Beer-->Coke      0.2    0.333333  0.833333
```