**The basics of market basket analysis**

Market basket analysis uses lists of transactions to identify useful associations between items. Such associations can be written in the form of a rule that has an antecedent and a consequent. Let's assume a small grocery store has asked you to look at their transaction data. After some analysis, you find the rule given below.

***{cereal}***

***{milk}***

Which statement about this rule is correct?

A *{cereal}* is the antecedent, *{milk}* is the consequent, and both are items.

# Count the number of transactions with coffee and gum

coffee = transactions.count(['coffee', 'gum'])

# Count the number of transactions with cereal and gum

cereal = transactions.count(['cereal', 'gum'])

# Count the number of transactions with bread and gum

bread = transactions.count(['bread', 'gum'])

# Print the counts for each transaction.

print('coffee:', coffee)

print('cereal:', cereal)

print('bread:', bread)

# Import pandas under the alias pd

import pandas as pd

# Load transactions from pandas

groceries = pd.read\_csv(groceries\_path)

# Split transaction strings into lists

transactions = groceries['Transaction'].apply(lambda t: t.split(','))

# Convert DataFrame column into list of strings

transactions = list(transactions)

# Print the list of transactions

print(transactions)

# Import permutations from the itertools module

from itertools import permutations

# Define the set of groceries

flattened = [i for t in transactions for i in t]

groceries = list(set(flattened))

# Generate all possible rules from groceries list

rules = list(permutations(groceries, 2))

# Print the set of rules

print(rules)

# Print the number of rules

print(len(rules))

# Import the transaction encoder function from mlxtend

from mlxtend.preprocessing import TransactionEncoder

import pandas as pd

# Instantiate transaction encoder and identify unique items in transactions

encoder = TransactionEncoder().fit(transactions)

# One-hot encode transactions

onehot = encoder.transform(transactions)

# Convert one-hot encoded data to DataFrame

onehot = pd.DataFrame(onehot, columns = encoder.columns\_)

# Print the one-hot encoded transaction dataset

print(onehot)

# Compute the support

support = onehot.mean()

# Print the support

print(support)

# Compute support for Hunger and Potter

supportHP = np.logical\_and(books['Hunger'], books['Potter']).mean()

# Compute support for Hunger and Twilight

supportHT = np.logical\_and(books['Hunger'], books['Twilight']).mean()

# Compute support for Potter and Twilight

supportPT = np.logical\_and(books['Potter'], books['Twilight']).mean()

# Print support values

print("Hunger Games and Harry Potter: %.2f" % supportHP)

print("Hunger Games and Twilight: %.2f" % supportHT)

print("Harry Potter and Twilight: %.2f" % supportPT)

# Compute support for Potter and Twilight

supportPT = np.logical\_and(books['Potter'], books['Twilight']).mean()

# Compute support for Potter

supportP = books['Potter'].mean()

# Compute support for Twilight

supportT = books['Twilight'].mean()

# Compute confidence for both rules

confidencePT = supportPT / supportP

confidenceTP = supportPT / supportT

# Print results

print('{0:.2f}, {1:.2f}'.format(confidencePT, confidenceTP))

# Compute support for Potter and Twilight

supportPT = np.logical\_and(books['Potter'],books['Twilight']).mean()

# Compute support for Potter

supportP = books['Potter'].mean()

# Compute support for Twilight

supportT = books['Twilight'].mean()

# Compute lift

lift = supportPT / (supportP \* supportT)

# Print lift

print("Lift: %.2f" % lift)

# Compute support for Potter AND Hunger

supportPH = np.logical\_and(books['Potter'], books['Hunger']).mean()

# Compute support for Potter

supportP = books['Potter'].mean()

# Compute support for NOT Hunger

supportnH = 1.0 - books['Hunger'].mean()

# Compute support for Potter and NOT Hunger

supportPnH = supportP - supportPH

# Compute and print conviction for Potter -> Hunger

conviction = supportP \* supportnH / supportPnH

print("Conviction: %.2f" % conviction)

def conviction(antecedent, consequent):

    # Compute support for antecedent AND consequent

    supportAC = np.logical\_and(antecedent, consequent).mean()

    # Compute support for antecedent

    supportA = antecedent.mean()

    # Compute support for NOT consequent

    supportnC = 1.0 - consequent.mean()

    # Compute support for antecedent and NOT consequent

    supportAnC = supportA - supportAC

    # Return conviction

    return supportA \* supportnC / supportAnC

# Compute conviction for twilight -> potter and potter -> twilight

convictionTP = conviction(twilight, potter)

convictionPT = conviction(potter, twilight)

# Compute conviction for twilight -> hunger and hunger -> twilight

convictionTH = conviction(twilight, hunger)

convictionHT = conviction(hunger, twilight)

# Compute conviction for potter -> hunger and hunger -> potter

convictionPH = conviction(potter, hunger)

convictionHP = conviction(hunger,potter)

# Print results

print('Harry Potter -> Twilight: ', convictionHT)

print('Twilight -> Potter: ', convictionTP)

# Compute the support of Twilight and Harry Potter

supportT = books['Twilight'].mean()

supportP = books['Potter'].mean()

# Compute the support of both books

supportTP = np.logical\_and(books['Twilight'],books['Potter']).mean()

# Complete the expressions for the numerator and denominator

numerator = supportTP - supportT\*supportP

denominator = max(supportTP\*(1-supportT), supportT\*(supportP-supportTP))

# Compute and print Zhang's metric

zhang = numerator / denominator

print(zhang)

# Define a function to compute Zhang's metric

def zhang(antecedent, consequent):

    # Compute the support of each book

    supportA = antecedent.mean()

    supportC = consequent.mean()

    # Compute the support of both books

    supportAC = np.logical\_and(antecedent, consequent).mean()

    # Complete the expressions for the numerator and denominator

    numerator = supportAC - supportA\*supportC

    denominator = max(supportAC\*(1-supportA), supportA\*(supportC-supportAC))

    # Return Zhang's metric

    return numerator / denominator

# Define an empty list for Zhang's metric

zhangs\_metric = []

# Loop over lists in itemsets

for itemset in itemsets:

    # Extract the antecedent and consequent columns

    antecedent = books[itemset[0]]

    consequent = books[itemset[1]]

    # Complete Zhang's metric and append it to the list

    zhangs\_metric.append(zhang(antecedent, consequent))

# Print results

rules['zhang'] = zhangs\_metric

print(rules)

# Preview the rules DataFrame using the .head() method

print(rules.head())

# Select the subset of rules with antecedent support greater than 0.05

rules = rules[rules['antecedent support'] > 0.05]

# Select the subset of rules with a consequent support greater than 0.02

rules = rules[rules['consequent support']>0.02]

# Select the subset of rules with a conviction greater than 1.01

rules = rules[rules['conviction']>1.01]

# Print remaining rules

print(rules)

# Set the lift threshold to 1.5

rules = rules[rules['lift'] > 1.5]

# Set the conviction threshold to 1.0

rules = rules[rules['conviction'] > 1.0]

# Set the threshold for Zhang's rule to 0.65

rules = rules[rules['zhang']>0.65]

# Print rule

print(rules[['antecedents','consequents']])

# Select the column headers for sign items

sign\_headers = [i for i in onehot.columns if i.lower().find('sign')>=0]

# Select columns of sign items using sign\_headers

sign\_columns = onehot[sign\_headers]

# Perform aggregation of sign items into sign category

signs = sign\_columns.sum(axis = 1) >= 1.0

# Print support for signs

print('Share of Signs: %.2f' % signs.mean())

def aggregate(item):

    # Select the column headers for sign items in onehot

    item\_headers = [i for i in onehot.columns if i.lower().find(item)>=0]

    # Select columns of sign items

    item\_columns = onehot[item\_headers]

    # Return category of aggregated items

    return item\_columns.sum(axis = 1) >= 1.0

# Aggregate items for the bags, boxes, and candles categories

bags = aggregate('bag')

boxes = aggregate('boxes')

candles = aggregate('candles')

# Import apriori from mlxtend

from mlxtend.frequent\_patterns import apriori

# Compute frequent itemsets using the Apriori algorithm

frequent\_itemsets = apriori(onehot,

                            min\_support = 0.006,

                            max\_len = 3,

                            use\_colnames = True)

# Print a preview of the frequent itemsets

print(frequent\_itemsets.head())

# Import apriori from mlxtend

from mlxtend.frequent\_patterns  import apriori

# Compute frequent itemsets using a support of 0.003 and length of 3

frequent\_itemsets\_1 = apriori(onehot, min\_support = 0.003,

                            max\_len = 3, use\_colnames = True)

# Compute frequent itemsets using a support of 0.001 and length of 3

frequent\_itemsets\_2 = apriori(onehot, min\_support = 0.001,

                            max\_len = 3, use\_colnames = True)

# Print the number of freqeuent itemsets

print(len(frequent\_itemsets\_1), len(frequent\_itemsets\_2))

# Import the association rule function from mlxtend

from mlxtend.frequent\_patterns  import association\_rules

# Compute all association rules for frequent\_itemsets\_1

rules\_1 = association\_rules(frequent\_itemsets\_1,

                            metric = 'support',

                            min\_threshold = 0.0015)

# Compute all association rules for frequent\_itemsets\_2

rules\_2 = association\_rules(frequent\_itemsets\_2,

                            metric = 'support',

                            min\_threshold = 0.0015)

# Print the number of association rules generated

print(len(rules\_1), len(rules\_2))

# Import the association rules function

from mlxtend.frequent\_patterns  import association\_rules

# Compute frequent itemsets using the Apriori algorithm

frequent\_itemsets = apriori(onehot, min\_support = 0.001,

                            max\_len = 2, use\_colnames = True)

# Compute all association rules for frequent\_itemsets

rules = association\_rules(frequent\_itemsets,

                            metric = "lift",

                           min\_threshold = 1.0,)

# Print association rules

print(rules)

# Import the association rules function

from mlxtend.frequent\_patterns import association\_rules

# Compute frequent itemsets using the Apriori algorithm

frequent\_itemsets = apriori(onehot, min\_support=0.0015,

                            max\_len = 2, use\_colnames = True)

# Compute all association rules using confidence

rules = association\_rules(frequent\_itemsets,

                            metric = "confidence",

                           min\_threshold = 0.5)

# Print association rules

print(rules)

# Apply the apriori algorithm with a minimum support of 0.0001

frequent\_itemsets = apriori(aggregated, min\_support=0.0001, use\_colnames = True)

# Generate the initial set of rules using a minimum support of 0.0001

rules = association\_rules(frequent\_itemsets,

                          metric = "support", min\_threshold = 0.0001)

# Set minimum antecedent support to 0.35

rules = rules[rules['antecedent support'] > 0.35]

# Set maximum consequent support to 0.35

rules = rules[rules['consequent support'] < 0.35]

# Print the remaining rules

print(rules)

# Generate the initial set of rules using a minimum lift of 1.00

rules = association\_rules(frequent\_itemsets, metric = "lift", min\_threshold = 1.00)

# Set antecedent support to 0.005

rules = rules[rules['antecedent support'] > 0.005]

# Set consequent support to 0.005

rules = rules[rules['consequent support'] > 0.005]

# Compute Zhang's rule

rules['zhang'] = zhangs\_rule(rules)

# Set the lower bound for Zhang's rule to 0.98

rules = rules[rules['zhang'] > 0.98]

print(rules[['antecedents', 'consequents']])

# Apply the Apriori algorithm with a minimum support threshold of 0.001

frequent\_itemsets = apriori(onehot, min\_support = 0.001, use\_colnames = True)

# Recover association rules using a minium support threshold of 0.001

rules = association\_rules(frequent\_itemsets, metric = 'support', min\_threshold = 0.001)

# Apply a 0.002 antecedent support threshold, 0.60 confidence threshold, and 2.50 lift threshold

filtered\_rules = rules[(rules['antecedent support'] > 0.002) &

                        (rules['consequent support'] > 0.01) &

                        (rules['confidence'] > 0.60) &

                        (rules['lift'] > 2.50)]

# Print remaining rule

print(filtered\_rules[['antecedents','consequents']])

# Compute frequent itemsets using a minimum support of 0.07

frequent\_itemsets = apriori(onehot, min\_support = 0.07,

                            use\_colnames = True, max\_len = 2)

# Compute the association rules

rules = association\_rules(frequent\_itemsets, metric = 'support',

                          min\_threshold = 0.0)

# Import seaborn under its standard alias

import seaborn as sns

# Transform the DataFrame of rules into a matrix using the lift metric

pivot = rules.pivot(index = 'consequents',

                   columns = 'antecedents', values= 'lift')

# Generate a heatmap with annotations on and the colorbar off

sns.heatmap(pivot, annot = True, cbar = False)

plt.yticks(rotation=0)

plt.xticks(rotation=90)

plt.show()

# Interpreting heatmaps

In the previous exercise, you generated the heatmap shown below. Each cell of the heatmap shows the lift value for an association rule. Recall that your goal was to identify a narrow set of films that were all strongly associated according to the lift metric. These films would form the initial content library for a streaming service. Which of the following statements is true?

Fight Club, Braveheart, and Batman Begins would be the best initial content library.

# Import seaborn under its standard alias

import seaborn as sns

# Apply the Apriori algorithm with a support value of 0.0075

frequent\_itemsets = apriori(onehot, min\_support = 0.0075,

                            use\_colnames = True, max\_len = 2)

# Generate association rules without performing additional pruning

rules = association\_rules(frequent\_itemsets, metric = 'support',

                          min\_threshold = 0.0)

# Generate scatterplot using support and confidence

sns.scatterplot(x = "support", y = "confidence", data = rules)

plt.show()

# Import seaborn under its standard alias

import seaborn as sns

# Apply the Apriori algorithm with a support value of 0.0075

frequent\_itemsets = apriori(onehot, min\_support = 0.0075,

                         use\_colnames = True, max\_len = 2)

# Generate association rules without performing additional pruning

rules = association\_rules(frequent\_itemsets, metric = "support",

                          min\_threshold = 0.0)

# Generate scatterplot using support and confidence

sns.scatterplot(x = "support", y = "confidence",

                size = "lift", data = rules)

plt.show()

# Compute the frequent itemsets

frequent\_itemsets = apriori(onehot, min\_support = 0.05,

                         use\_colnames = True, max\_len = 2)

# Compute rules from the frequent itemsets with the confidence metric

rules = association\_rules(frequent\_itemsets, metric = 'confidence',

                          min\_threshold = 0.50)

# Convert rules into coordinates suitable for use in a parallel coordinates plot

coords = rules\_to\_coordinates(rules)

# Generate parallel coordinates plot

parallel\_coordinates(coords, 'rule')

plt.legend([])

plt.show()

# Import the parallel coordinates plot submodule

from pandas.plotting  import parallel\_coordinates

# Convert rules into coordinates suitable for use in a parallel coordinates plot

coords = rules\_to\_coordinates(rules)

# Generate parallel coordinates plot

parallel\_coordinates(coords, 'rule', colormap = 'ocean')

plt.legend([])

plt.show()