Machine Learning Group Project 2023

Ashkan Lotfollahzadeh - Michelle Lee Kruschke - Marco Mancuso Morini - Melissa Nieto Ridoutt- João Victor Pereira Dos Santos

Introduction

We chose the Kaggle competition https://www.kaggle.com/competitions/instacart-market-basket-analysis. Instacart is an American company that offers grocery shopping and delivery services via website and a moblie app. It connects users with a personal shopper in their area to shop and deliver groceries from local participating stores. Although the original objective of the competition was quite complex, we chose to evalute the problem lookiing at the whether a product would be reordered based on various features of past history, and additionally a Market Basket Analysis to determine which prodoucts could possibly be purchased together in the future. The datasets used in the evaluation were provided in 2017, by Instacart, in open sourced data taken from over 200,000 Instacart users.

In [1]:
```
#Importing libraries:

import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matri
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import f1_score, recall_score, precision_score
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.svm import SVC
from sklearn.model_selection import RandomizedSearchCV
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
from xgboost import XGBClassifier
from mlxtend.frequent_patterns import association_rules
from mlxtend.frequent_patterns import apriori
from sklearn.preprocessing import MinMaxScaler
```

In [2]:
```
#Creating relevant variables
rs = 42 #Random_state
np.random.seed(rs)
cv = 3
%matplotlib inline
```

In [3]:
```
aisles=pd.read_csv('aisles.csv')
departments=pd.read_csv('departments.csv')
products=pd.read_csv('products.csv')
orders=pd.read_csv('orders.csv')
order_products_prior = pd.read_csv('order_products__prior.csv')
order_products_train = pd.read_csv('order_products__train.csv')
```

0. Analysing the data

Aisles

In [4]:
```
aisles.head()
```

Out[4]:

	aisle_id	aisle
0	1	prepared soups salads
1	2	specialty cheeses
2	3	energy granola bars
3	4	instant foods
4	5	marinades meat preparation

In [5]:
```
aisles.shape, len(aisles['aisle_id'].unique())
```

Out[5]:
```
((134, 2), 134)
```

In [6]:
```
aisles.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 134 entries, 0 to 133
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   aisle_id  134 non-null    int64 
 1   aisle     134 non-null    object
dtypes: int64(1), object(1)
memory usage: 2.2+ KB
```

Departments

In [7]:
```
departments.head()
```

Out[7]:

	department_id	department
0	1	frozen
1	2	other
2	3	bakery
3	4	produce
4	5	alcohol

In [8]:
```
departments.shape, len(departments['department_id'].unique())
```
Out[8]:
```
((21, 2), 21)
```

In [9]:
```
departments.info()
```
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21 entries, 0 to 20
Data columns (total 2 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   department_id  21 non-null     int64 
 1   department     21 non-null     object
dtypes: int64(1), object(1)
memory usage: 464.0+ bytes
```

Products

In [10]:
```
products.head()
```
Out[10]:

	product_id	product_name	aisle_id	department_id
0	1	Chocolate Sandwich Cookies	61	19
1	2	All-Seasons Salt	104	13
2	3	Robust Golden Unsweetened Oolong Tea	94	7
3	4	Smart Ones Classic Favorites Mini Rigatoni Wit...	38	1
4	5	Green Chile Anytime Sauce	5	13

In [11]:
```
products.shape, len(products['product_id'].unique())
```
Out[11]:
```
((49688, 4), 49688)
```

In [12]:
```
products.info()
```
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 49688 entries, 0 to 49687
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   product_id     49688 non-null  int64 
 1   product_name   49688 non-null  object
 2   aisle_id       49688 non-null  int64 
 3   department_id  49688 non-null  int64 
dtypes: int64(3), object(1)
memory usage: 1.5+ MB
```

Orders

In [13]:
```
orders.head()
```

Out[13]:

	order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_orde
0	2539329	1	prior	1	2	8	NaN
1	2398795	1	prior	2	3	7	15.(
2	473747	1	prior	3	3	12	21.(
3	2254736	1	prior	4	4	7	29.(
4	431534	1	prior	5	4	15	28.(

In [14]:
```
orders.shape, len(orders['order_id'].unique())
```

Out[14]:
```
((3421083, 7), 3421083)
```

In [15]:
```
orders['eval_set'].value_counts()
```

Out[15]:
```
prior    3214874
train     131209
test       75000
Name: eval_set, dtype: int64
```

Order_Products_Prior

In [16]:
```
order_products_prior.head()
```

Out[16]:

	order_id	product_id	add_to_cart_order	reordered
0	2	33120	1	1
1	2	28985	2	1
2	2	9327	3	0
3	2	45918	4	1
4	2	30035	5	0

In [17]:
```
order_products_prior.shape, len(order_products_prior['order_id'].unique())
```

Out[17]:
```
((32434489, 4), 3214874)
```

In [18]:
```
order_products_prior.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32434489 entries, 0 to 32434488
Data columns (total 4 columns):
 #   Column             Dtype
---  ------             -----
 0   order_id           int64
 1   product_id         int64
 2   add_to_cart_order  int64
 3   reordered          int64
dtypes: int64(4)
memory usage: 989.8 MB
```

Order_Products_Train

In [19]:
```
order_products_train.head()
```

Out[19]:

	order_id	product_id	add_to_cart_order	reordered
0	1	49302	1	1
1	1	11109	2	1
2	1	10246	3	0
3	1	49683	4	0
4	1	43633	5	1

In [20]:
```
order_products_train.shape, len(order_products_train['order_id'].unique())
```

Out[20]:
```
((1384617, 4), 131209)
```

In [21]:
```
order_products_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1384617 entries, 0 to 1384616
Data columns (total 4 columns):
 #   Column             Non-Null Count    Dtype
---  ------             --------------    -----
 0   order_id           1384617 non-null  int64
 1   product_id         1384617 non-null  int64
 2   add_to_cart_order  1384617 non-null  int64
 3   reordered          1384617 non-null  int64
dtypes: int64(4)
memory usage: 42.3 MB
```

1. Merging the data

In [4]:
```
#Creating a unified product dataframe
products=pd.merge(products, aisles, on='aisle_id')
products=pd.merge(products, departments, on='department_id')
```

In [5]:
```
#Concating the orders_products train and prior df's:
order_prod_pt = pd.concat([order_products_prior, order_products_train])
```

In [6]:
```
#Merging the new df with the prior and train data with the order df:
full_orders_df=pd.merge(orders, order_prod_pt, on='order_id')
```

In [7]:

```
#Final dataframe with all datasets merged
df=pd.merge(full_orders_df, products, on='product_id')
```

In [26]:

```
df.head(25)
```

Out[26]:

	order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_ord
0	2539329	1	prior	1	2	8	Na
1	2398795	1	prior	2	3	7	15
2	473747	1	prior	3	3	12	21
3	2254736	1	prior	4	4	7	29
4	431534	1	prior	5	4	15	28
5	3367565	1	prior	6	2	7	19
6	550135	1	prior	7	1	9	20
7	3108588	1	prior	8	1	14	14
8	2295261	1	prior	9	1	16	0
9	2550362	1	prior	10	4	8	30
10	1187899	1	train	11	4	8	14
11	2968173	15	prior	15	1	9	7
12	1870022	15	prior	17	2	16	8
13	1911383	15	prior	18	2	11	7
14	2715276	15	prior	21	1	9	7
15	487368	15	prior	22	1	10	14
16	2293453	19	prior	2	5	14	6
17	1973799	19	prior	5	6	12	8
18	532817	19	prior	7	4	17	6
19	1573906	21	prior	10	3	10	6
20	1593000	31	prior	10	3	8	7
21	2231262	31	prior	17	3	11	8
22	2580647	43	prior	6	4	16	4

	order_id	user_id	eval_set	order_number	order_dow	order_hour_of_day	days_since_prior_ord
23	2187180	43	prior	9	4	12	3
24	2497897	52	prior	1	1	9	Na

In [27]:
```
df.describe()
```

Out[27]:

	order_id	user_id	order_number	order_dow	order_hour_of_day	days_since_pr
count	3.381911e+07	3.381911e+07	3.381911e+07	3.381911e+07	3.381911e+07	3.17
mean	1.710566e+06	1.029444e+05	1.713998e+01	2.737285e+00	1.343123e+01	1.13
std	9.874008e+05	5.946733e+04	1.749829e+01	2.093296e+00	4.246149e+00	8.94
min	1.000000e+00	1.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00	0.00
25%	8.554130e+05	5.143500e+04	5.000000e+00	1.000000e+00	1.000000e+01	5.00
50%	1.710660e+06	1.026260e+05	1.100000e+01	3.000000e+00	1.300000e+01	8.00
75%	2.565587e+06	1.544120e+05	2.400000e+01	5.000000e+00	1.600000e+01	1.50
max	3.421083e+06	2.062090e+05	1.000000e+02	6.000000e+00	2.300000e+01	3.00

2. Data Cleaning

In [28]:
```
#Seeing the information of the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 33819106 entries, 0 to 33819105
Data columns (total 15 columns):
 #   Column                  Dtype  
---  ------                  -----  
 0   order_id                int64  
 1   user_id                 int64  
 2   eval_set                object 
 3   order_number            int64  
 4   order_dow               int64  
 5   order_hour_of_day       int64  
 6   days_since_prior_order  float64
 7   product_id              int64  
 8   add_to_cart_order       int64  
 9   reordered               int64  
 10  product_name            object 
 11  aisle_id                int64  
 12  department_id           int64  
 13  aisle                   object 
 14  department              object 
dtypes: float64(1), int64(10), object(4)
memory usage: 4.0+ GB
```

In [29]:
```
#Seeing if we have null values:
df.isna().sum()
```

Out[29]:
```
order_id                     0
user_id                      0
eval_set                     0
order_number                 0
order_dow                    0
order_hour_of_day            0
days_since_prior_order 2078068
product_id                   0
add_to_cart_order            0
reordered                    0
product_name                 0
aisle_id                     0
department_id                0
aisle                        0
department                   0
dtype: int64
```

In [8]:
```
#filling nulls with the days_since_prior_order per product mean
df['days_since_prior_order']=df['days_since_prior_order'].fillna(df.groupby('produ
```

In [31]:
```
#verify the new mean and compare above. The raw indexed as 0 had null values in the
y=df.groupby('product_id', as_index=False)['days_since_prior_order'].mean()
print(y['days_since_prior_order'].loc[y['product_id']==196]==df['days_since_prior_
```

```
195    True
Name: days_since_prior_order, dtype: bool
```

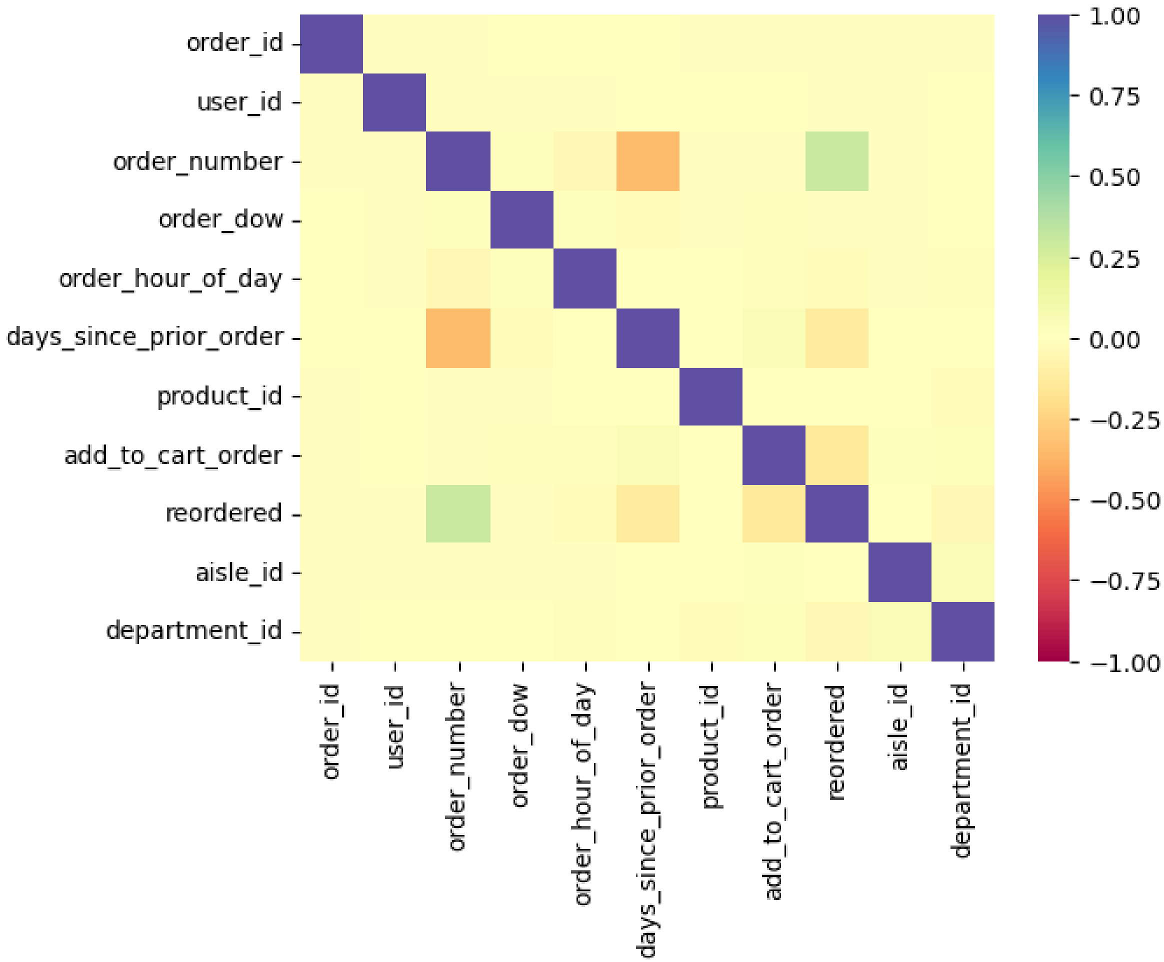
In [32]:
```
#Seeing if we have null values:
df.isna().sum()
```

Out[32]:
```
order_id                  0
user_id                   0
eval_set                  0
order_number              0
order_dow                 0
order_hour_of_day         0
days_since_prior_order   40
product_id                0
add_to_cart_order         0
reordered                 0
product_name              0
aisle_id                  0
department_id             0
aisle                     0
department                0
dtype: int64
```

In [9]:
```
#we are accepting 40 value=0, it is a particular product but it represents only 0.0
df['days_since_prior_order']=df['days_since_prior_order'].fillna(0)
df.isna().sum()
```

```
Out[9]: order_id                  0
user_id                   0
eval_set                  0
order_number              0
order_dow                 0
order_hour_of_day         0
days_since_prior_order    0
product_id                0
add_to_cart_order         0
reordered                 0
product_name              0
aisle_id                  0
department_id             0
aisle                     0
department                0
dtype: int64
```

```
In [34]: # Graph shows little or no correlation between features
sns.heatmap(df.corr(), annot=False, cmap='Spectral', vmin=-1, vmax=1)
plt.show()
```



3. Reduced Dataset

3.1. Getting a random sample of the dataframe

We need to sample the DataFrame before dividing it into train and validation to be sure that part of the train data is not randomly selected into the validation data.

In [11]:
```
# Creating the prior DF so we can divide it:
df_prior = df[df['eval_set'] == 'prior']
```

In [36]:
```
# ------------------------- Samplying the data by user' -------------------------

'''
Here it will sample X user_ids and the number of rows would be the sum of the rows
users selected
'''
user_ids_to_sample = list(df_prior['user_id'].sample(300, random_state=rs))

# filter the dataframe to only contain rows with the selected order_ids
prior_red = df_prior[df_prior['user_id'].isin(user_ids_to_sample)]
```

In [37]:
```
prior_red.shape, prior_red['user_id'].nunique()
```

Out[37]:
```
((134610, 15), 300)
```

In [38]:
```
#dropping columns that we fill that are not important as the others:
prior_red.drop(columns=['order_id', 'user_id','aisle_id', 'department_id','product_
```

3.2. Dividing the dataset into 3 (divide the dataset into 3 using the eval_set column:

- Prior --> Train
- Train --> Validation
- Test --> Test

In [39]:
```
#Dividing the new prior df reduced into train and validation:
train_data, val_data = train_test_split(prior_red, test_size=0.2)
```

In [40]:
```
#Test dataset:
df_test = df[df['eval_set'] == 'train'].iloc[0:13696]
```

Dropping the *eval_set* column

Makes no sense to maintain the eval column after the division, so we dropt it.

In [41]:
```
train_data.drop(['eval_set'], axis=1, inplace=True);
```

In [42]:
```
val_data.drop(['eval_set'], axis=1, inplace=True)
```

In [43]:
```
df_test.drop(columns=['eval_set', 'order_id', 'user_id','product_id', 'aisle', 'pr
```

In [44]:
```
train_data.shape, val_data.shape
```

Out[44]:
```
((107688, 6), (26922, 6))
```

In [45]:
```
df_test.shape
```

Out[45]:
```
(13696, 6)
```

Dealing with NANs

In [46]:
```
train_data.isna().sum()
```

Out[46]:
```
order_number              0
order_dow                 0
order_hour_of_day         0
days_since_prior_order    0
add_to_cart_order         0
reordered                 0
dtype: int64
```

In [47]:
```
val_data.isna().sum()
```

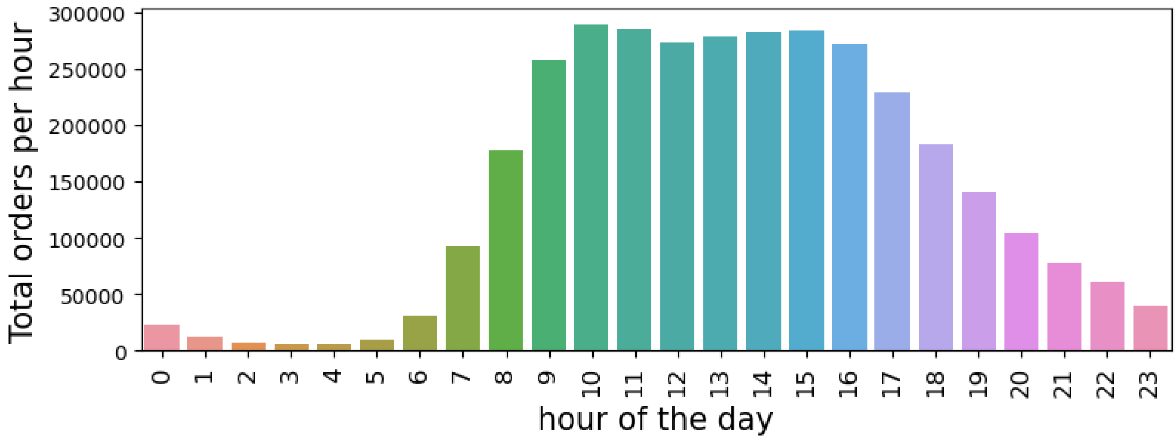
Out[47]:
```
order_number              0
order_dow                 0
order_hour_of_day         0
days_since_prior_order    0
add_to_cart_order         0
reordered                 0
dtype: int64
```

4. Baseline Model

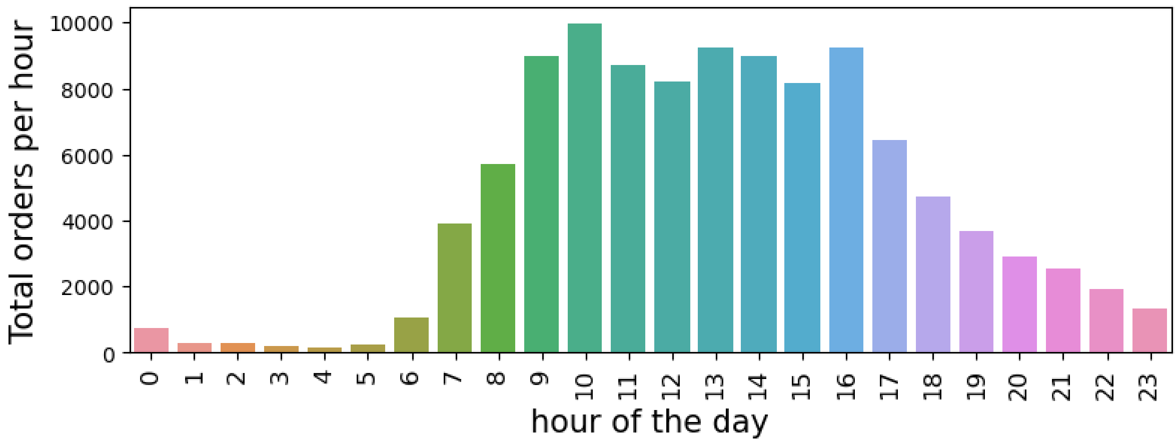
4.1. Data Visualization

Does our sampled data replicate the data distribution of the complete data?

In [48]:
```
#order_hour_of_day distribution from the original orders:
orders_hour = orders.groupby('order_hour_of_day')['order_number'].count().to_frame
plt.figure(figsize=(9,3))
sns.barplot(orders_hour.index, orders_hour.number_of_orders_each_hour, order=order:
plt.xlabel('hour of the day', size=15)
plt.ylabel('Total orders per hour', size=15)
plt.xticks(size=12, rotation=90)
plt.show();
```



In [49]:
```
#order_hour_of_day distribution from the train data (sampled):
train_hour = train_data.groupby('order_hour_of_day')['order_number'].count().to_fr
plt.figure(figsize=(9,3))
sns.barplot(train_hour.index, train_hour.number_of_orders_each_hour, order=train_h
plt.xlabel('hour of the day', size=15)
plt.ylabel('Total orders per hour', size=15)
plt.xticks(size=12, rotation=90)
plt.show();
```



As both graphs look very similar we concluded that our random sampling replicates pretty well the overall distribution and proportion of the orginal data.

4.2. Preprocessing

In [50]:
```
train_data['days_since_prior_order'] = train_data['days_since_prior_order'].astype
```

In [51]:
```
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 107688 entries, 25185383 to 30213723
Data columns (total 6 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   order_number            107688 non-null  int64
 1   order_dow               107688 non-null  int64
 2   order_hour_of_day       107688 non-null  int64
 3   days_since_prior_order  107688 non-null  int8
 4   add_to_cart_order       107688 non-null  int64
 5   reordered               107688 non-null  int64
dtypes: int64(5), int8(1)
memory usage: 5.0 MB
```

In [52]:
```
train_data.columns
```

Out[52]:
```
Index(['order_number', 'order_dow', 'order_hour_of_day',
       'days_since_prior_order', 'add_to_cart_order', 'reordered'],
      dtype='object')
```

4.3. Splitting the data into the predicting values X and the class y

In [53]:
```
#Spliting the train data:
X_train = train_data.drop('reordered', axis=1)
y_train = train_data['reordered']

#Splitting the validation data:
X_val = val_data.drop('reordered', axis=1)
y_val = val_data['reordered']

#Splitting the Test data:
X_test = df_test.drop('reordered', axis=1)
y_test = df_test['reordered']
```

In [54]:
```
#Preprocessing this Data:
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train, y_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

4.4. Running Models

4.4.1 Decision Tree Model

We know that a decision tree is too simple of a model, but we wanted to begin with the simplest one before trying random forest and XGBoost.

In [55]:
```
#Creating and fitting the Decision Tree model:
model_tree = DecisionTreeClassifier(criterion="entropy", random_state = rs)
model_tree.fit(X_train, y_train);
```

In [56]:
```
#Predicting:
y_tree = model_tree.predict(X_val)

#Seeing the accuracy:
accuracy_ho = accuracy_score(y_val, y_tree) * 100

#Understanding the max_depth and preparing a varible for a loop validation:
max_depth = model_tree.tree_.max_depth
parameter_values = range(1,max_depth+1)

print("The accuracy on test set is {0:.1f}%".format(accuracy_ho))
print("The maximum depth of the tree fitted on X_train is {}".format(max_depth))
```

```
The accuracy on test set is 72.9%
The maximum depth of the tree fitted on X_train is 38
```

```
    -------------------------------- Observation --------------------
    -------------

    We were surprised by the accuracy of the Decision Tree Classifier.
    Maybe this is not a complex problem as we thought. Maybe the other
    more complex models will tend to overfit. Still, we will try them
    anyway. But first let's try to improve de Decision Tree model a
    little bit more.

    -----------------------------------------------------------------
    -------------
```

In [57]:
```
avg_scores = []
for par in parameter_values:
    estimator = DecisionTreeClassifier(criterion="entropy"
                                          , max_depth = par
                                          , random_state = rs
                                          )
    scores = cross_val_score(estimator, X_train, y_train
                             , scoring='accuracy', cv = cv)
    # cross_val_score produces an array with one score for each fold
    avg_scores.append(np.mean(scores))
print(avg_scores)
```

```
[0.7499628556570834, 0.7499628556570834, 0.7520615110318699, 0.7559616670381102, 0.7568624173538371, 0.7575774459549812, 0.7574845850976896, 0.7568345590966495, 0.7580138919842508, 0.7575495876977936, 0.7576053042121685, 0.757577445954981, 0.7563888269816507, 0.7540115890349899, 0.7503342990862492, 0.7475577594532353, 0.7443819181338682, 0.7409739246712724, 0.7364516009211797, 0.732393581457544, 0.729552039224426, 0.7276391055642226, 0.7252897258747493, 0.7219931654409034, 0.7199130822375753, 0.7196159274942427, 0.7186780328355992, 0.7179630042344551, 0.7191701953792439, 0.7190494762647649, 0.7186223163212242, 0.7189844736646608, 0.7185851719783077, 0.7187987519500779, 0.7188173241215363, 0.7189751875789318, 0.7188916128073695, 0.7188916128073695]
```

In [58]:
```
top_par_cv = parameter_values[np.argmax(avg_scores)]
estimator_best = DecisionTreeClassifier(criterion="entropy", max_depth = top_par_c
estimator_best.fit(X_train,y_train);
y_predicted = estimator_best.predict(X_test)
accuracy_cv = accuracy_score(y_test, y_predicted) * 100
print("The accuracy on test set tuned with cross_validation is {:.1f}% with depth
```

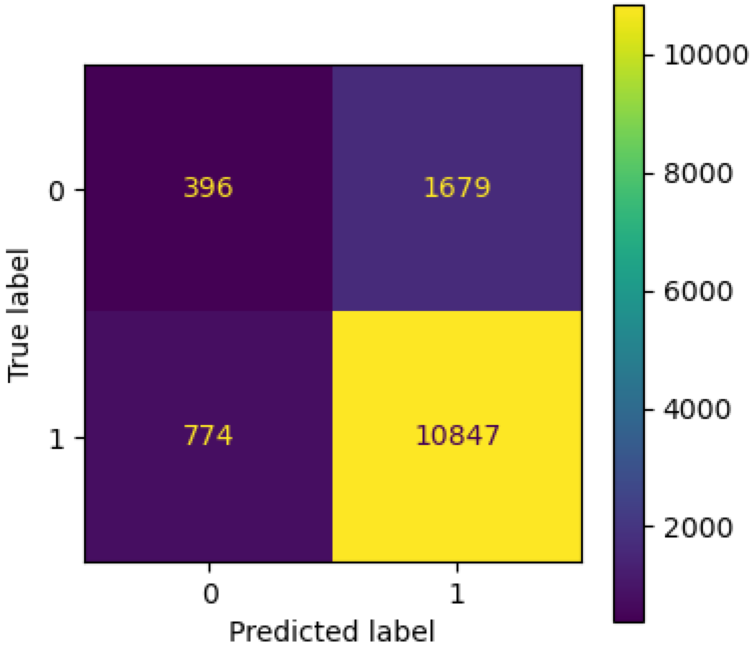
```
The accuracy on test set tuned with cross_validation is 82.1% with depth 9
```

In [59]:
```
#Classitication Report on the DT model:
print(classification_report(y_test, y_predicted))
```

```
              precision    recall  f1-score   support

           0       0.34      0.19      0.24      2075
           1       0.87      0.93      0.90     11621

    accuracy                           0.82     13696
   macro avg       0.60      0.56      0.57     13696
weighted avg       0.79      0.82      0.80     13696
```

In [60]:
```
#Confusion matrix in the DT model:
tree_matrix = confusion_matrix(y_test, y_predicted)
```

In [61]:
```
#Plotting the Confusion Matrix in a more vivid and clear way:
plt.rcParams['figure.figsize'] = [4, 4]
disp_tree = ConfusionMatrixDisplay(confusion_matrix=tree_matrix,
                 display_labels=estimator_best.classes_);
disp_tree.plot();
```



4.5. Fine Tunning

4.5.1. RandomSearchCV

We saw that our simple model was overffiting so we decided to limit the max_depth paramter between 10 to 21. ALso, we decided for the RandomSearchCV instead of the GridSearch because the RandomSearchCV oud impose less computational power.

In [62]:
```
#Definning our models and the params that we will try for each of them:
model_lbls = ['dt' # decision tree
             ,'xgb' # XGBoost
             ,'rf' # Random Forest
             ,'knn' #KNN classifier
            ]

models = {
    'dt': {'name': 'Decision Tree       ',
           'estimator': DecisionTreeClassifier(random_state=rs),
           'param': [{'criterion': ['gini', 'entropy']
                      , 'min_samples_split': [*range(1,10)]
                      , 'max_depth': [*range(1,max_depth+1)]
                      , 'class_weight':[None,'balanced']
                      , 'min_samples_leaf': [*range(1,10)]
          }]},

    'xgb':{'name': 'XGBoost           ',
           'estimator': XGBClassifier(random_state=rs),
           'param': [{'n_estimators':[10, 20, 30, 40 ],
                     'learning_rate':[0.001, 0.005, 0.01],
                     'n_jobs': [None, -1]
          }]},

    'rf': {'name': 'Random Forest       ',
           'estimator': RandomForestClassifier(random_state=rs),
           'param': [{'criterion': ['gini', 'entropy']
                     ,'max_depth': [*range(4,max_depth+1)]
                     ,'n_estimators':[*range(10,60,10)]
                     ,'min_samples_leaf': [*range(1,10)]
                     ,'min_samples_split': [*range(1,10)]
          }]},
    'knn':{'name': 'K Nearest Neighbor ',
           'estimator': KNeighborsClassifier(),
           'param': [{'n_neighbors': [2],
                      'weights' : ['uniform', 'distance'],
                      'metric': ['euclidean', 'manhattan', 'cityblock'],
           }]}
}
```

In [63]:
```
scorings = ['accuracy','precision_macro','recall_macro','f1_macro']
```

In [64]:
```
#Running a GridSearch for each model and each different combination of params:
clfs = []
my_models = pd.DataFrame(columns=['scoring','model','best_params','accuracy','prec
                                  'recall_macro','f1_macro'])
for model in model_lbls:
    for scoring in scorings:
        clf = RandomizedSearchCV(models[model]['estimator'], models[model]['param'
                                 scoring = scoring, n_jobs=-1, n_iter=200, verbose=2
                                 )
        #Appending the clfs list with each clf fit
```

```python
        clf.fit(X_train, y_train)
        clfs.append(clf)

        #creating the accuracy variable
        y_pred = clf.predict(X_val)
        accuracy = accuracy_score(y_val,y_pred)


        cr = classification_report(y_val,y_pred, output_dict=True)

        #creating the precision_macro variable
        precision_macro = precision_score(y_val, y_pred, average='macro')

        #creating the recall_macro variable
        recall_macro = recall_score(y_val, y_pred, average='macro')

        #creating the f1_macro variable
        f1_macro = f1_score(y_val, y_pred, average='macro')

        best_params = clf.best_params_

        #Appending the dataset row by row
        row = [scoring, model, best_params, accuracy, precision_macro, recall_macr
        my_models.loc[len(my_models)] = row
```

```
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 200 candidates, totalling 600 fits
Fitting 3 folds for each of 6 candidates, totalling 18 fits
Fitting 3 folds for each of 6 candidates, totalling 18 fits
Fitting 3 folds for each of 6 candidates, totalling 18 fits
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

In [65]:

```python
#Plotting our models dataframe:
my_models
```

Out[65]:

	scoring	model	best_params	accuracy	precision_macro	recall_macro	f1_macro
0	accuracy	dt	{'min_samples_split': 2, 'min_samples_leaf': 2...	0.758042	0.719503	0.645197	0.658558
1	precision_macro	dt	{'min_samples_split': 6, 'min_samples_leaf': 2...	0.756222	0.726250	0.630649	0.641953
2	recall_macro	dt	{'min_samples_split': 3, 'min_samples_leaf': 2...	0.704257	0.665862	0.688617	0.670509
3	f1_macro	dt	{'min_samples_split': 3, 'min_samples_leaf': 3...	0.751430	0.698781	0.677464	0.685452
4	accuracy	xgb	{'n_jobs': None, 'n_estimators': 40, 'learning...	0.757410	0.734108	0.627439	0.637984
5	precision_macro	xgb	{'n_jobs': None, 'n_estimators': 30, 'learning...	0.758116	0.736356	0.627757	0.638369
6	recall_macro	xgb	{'n_jobs': None, 'n_estimators': 40, 'learning...	0.757410	0.734108	0.627439	0.637984
7	f1_macro	xgb	{'n_jobs': None, 'n_estimators': 40, 'learning...	0.757410	0.734108	0.627439	0.637984
8	accuracy	rf	{'n_estimators': 40, 'min_samples_split': 8, '...	0.783857	0.758353	0.684369	0.702399
9	precision_macro	rf	{'n_estimators': 30, 'min_samples_split': 5, '...	0.752842	0.764583	0.600422	0.599991
10	recall_macro	rf	{'n_estimators': 30, 'min_samples_split': 2, '...	0.770299	0.725443	0.692570	0.703965
11	f1_macro	rf	{'n_estimators': 50, 'min_samples_split': 2, '...	0.775871	0.735938	0.691082	0.705052
12	accuracy	knn	{'weights': 'distance', 'n_neighbors': 2, 'met...	0.761422	0.714672	0.720672	0.717446
13	precision_macro	knn	{'weights': 'distance', 'n_neighbors': 2, 'met...	0.761422	0.714672	0.720672	0.717446
14	recall_macro	knn	{'weights': 'uniform', 'n_neighbors': 2, 'metr...	0.737241	0.709776	0.745475	0.714017

	scoring	model	best_params	accuracy	precision_macro	recall_macro	f1_macro
15	f1_macro	knn	{'weights': 'distance', 'n_neighbors': 2, 'met	0.761422	0.714672	0.720672	0.717446

In [66]:
```
#Seeing our best model for each score type:

print("Best Accuracy model is {} using this best parameters: {}.".format(my_models
                                                                      my_models[
print('-'*90)

print("Best Precision model is {} using this best parameters: {}.".format(my_model
                                                                      my_models[
print('-'*90)

print("Best Recall model is {} using this best parameters: {}.".format(my_models['
                                                                      my_models[
print('-'*90)

print("Best F1 model is {} using this best parameters: {}.".format(my_models['mode
                                                                      my_models[
```

```
Best Accuracy model is rf using this best parameters: {'n_estimators': 40, 'min_sa
mples_split': 8, 'min_samples_leaf': 1, 'max_depth': 32, 'criterion': 'entropy'}.
----------------------------------------------------------------------------------
--------
Best Precision model is rf using this best parameters: {'n_estimators': 30, 'min_s
amples_split': 5, 'min_samples_leaf': 6, 'max_depth': 4, 'criterion': 'entropy'}.
----------------------------------------------------------------------------------
--------
Best Recall model is knn using this best parameters: {'weights': 'uniform', 'n_nei
ghbors': 2, 'metric': 'manhattan'}.
----------------------------------------------------------------------------------
--------
Best F1 model is knn using this best parameters: {'weights': 'distance', 'n_neighb
ors': 2, 'metric': 'manhattan'}.
```

4.6. Testing the best model

Testing the best model to see if it can reproduce the same accuracy, precision, etc.

4.6.1. Random Forest (The Theoretically Best model)

In [67]:
```
best_model = RandomForestClassifier(criterion="entropy",
                                    random_state = rs,
                                    max_depth = 32,
                                    n_estimators=40,
                                    min_samples_split=8,
                                    min_samples_leaf=1)
best_model.fit(X_train, y_train)
```

Out[67]:
```
RandomForestClassifier(criterion='entropy', max_depth=32, min_samples_split=8,
                       n_estimators=40, random_state=42)
```

In [68]:
```
#Predicting:
y_best = best_model.predict(X_test)

#Seeing the accuracy:
accuracy_best = accuracy_score(y_test, y_best) * 100
```

```
#creating the precision_macro variable
precision_best = precision_score(y_test, y_best, average='macro') * 100

#creating the recall_macro variable
recall_best = recall_score(y_test, y_best, average='macro') * 100

#creating the f1_macro variable
f1_best = f1_score(y_test, y_best, average='macro')* 100

print("The accuracy on test set is {0:.1f}%".format(accuracy_best))
```

```
The accuracy on test set is 80.4%
```

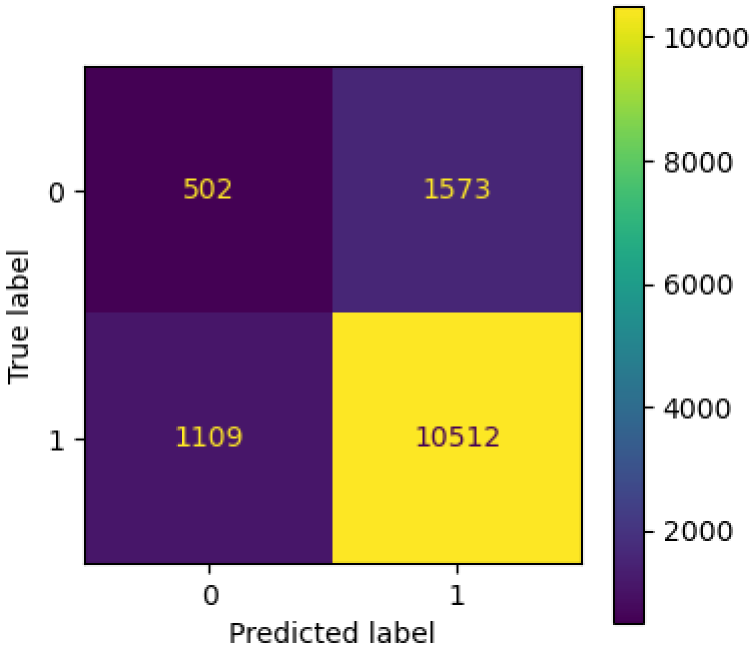
In [69]:
```
#Classitication Report on the RF model:
print(classification_report(y_test, y_best))
```

```
              precision    recall  f1-score   support

           0       0.31      0.24      0.27      2075
           1       0.87      0.90      0.89     11621

    accuracy                           0.80     13696
   macro avg       0.59      0.57      0.58     13696
weighted avg       0.79      0.80      0.79     13696
```

In [70]:
```
print(confusion_matrix(y_test, y_best))
```

```
[[  502  1573]
 [ 1109 10512]]
```

In [71]:
```
#Plotting the Confusion Matrix in a more vivid and clear way:
plt.rcParams['figure.figsize'] = [4, 4]
disp = ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_test, y_best),
                 display_labels=best_model.classes_);
disp.plot();
```



Market Basket Analysis Association and Column creation

We need to sample the dataframe again because otherwise our notebook wouldn't run the code below.

In [87]:
```
# -------------------------- Samplying the data by user' --------------------------
# create a list of order_ids to sample the data
'''
Here it will sample X user_ids and the number of rows would be the sum of the rows
users selected
'''
user_ids_to_sample2 = list(df_prior['user_id'].sample(15000, random_state=rs))

# filter the dataframe to only contain rows with the selected order_ids
filtered_df2 = df_prior[df_prior['user_id'].isin(user_ids_to_sample2)]

# group the filtered dataframe by order_id and apply the sample function to each gr
sampled_groups2 = [group.sample(n=len(group), replace=True) for _, group in filtere

# concatenate the sampled groups into a single dataframe
sampled_df2 = pd.concat(sampled_groups2)

# reset the index of the sampled dataframe
prior_red2 = sampled_df2.reset_index(drop=True)
```

In [88]:
```
prior_red2.shape
```

Out[88]:
```
(5380527, 15)
```

In [89]:
```
#Group the products by order_id and user_id to create transactions:
transactions = prior_red2.groupby(['order_id', 'user_id'])['product_id'].apply(lis
```

In [90]:
```
#Convert the products into binary indicators for each transaction:
from mlxtend.preprocessing import TransactionEncoder

te = TransactionEncoder()
te_ary = te.fit_transform(transactions['products'])
transactions = pd.DataFrame(te_ary, columns=te.columns_)
```

In [98]:
```
#Use the Apriori algorithm to find frequent itemsets:
from mlxtend.frequent_patterns import apriori

frequent_itemsets = apriori(transactions, min_support=0.01, use_colnames=True)
```

In [99]:
```
frequent_itemsets
```

Out[99]:

	support	itemsets
0	0.016407	(4605)
1	0.010004	(4799)
2	0.021629	(4920)
3	0.015997	(5077)
4	0.011712	(5450)
...	...	...
79	0.019091	(49235)
80	0.023270	(49683)
81	0.012566	(13176, 21137)
82	0.013100	(13176, 47209)
83	0.011019	(21137, 24852)

84 rows × 2 columns

In [100…

```
#Generate association rules:
from mlxtend.frequent_patterns import association_rules

rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1)
```

In [101…

```
rules
```

Out[101]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage
0	(13176)	(21137)	0.096375	0.075941	0.012566	0.130384	1.716914	0.005247
1	(21137)	(13176)	0.075941	0.096375	0.012566	0.165468	1.716914	0.005247
2	(13176)	(47209)	0.096375	0.060643	0.013100	0.135930	2.241465	0.007256
3	(47209)	(13176)	0.060643	0.096375	0.013100	0.216021	2.241465	0.007256
4	(21137)	(24852)	0.075941	0.116107	0.011019	0.145100	1.249705	0.002202
5	(24852)	(21137)	0.116107	0.075941	0.011019	0.094903	1.249705	0.002202

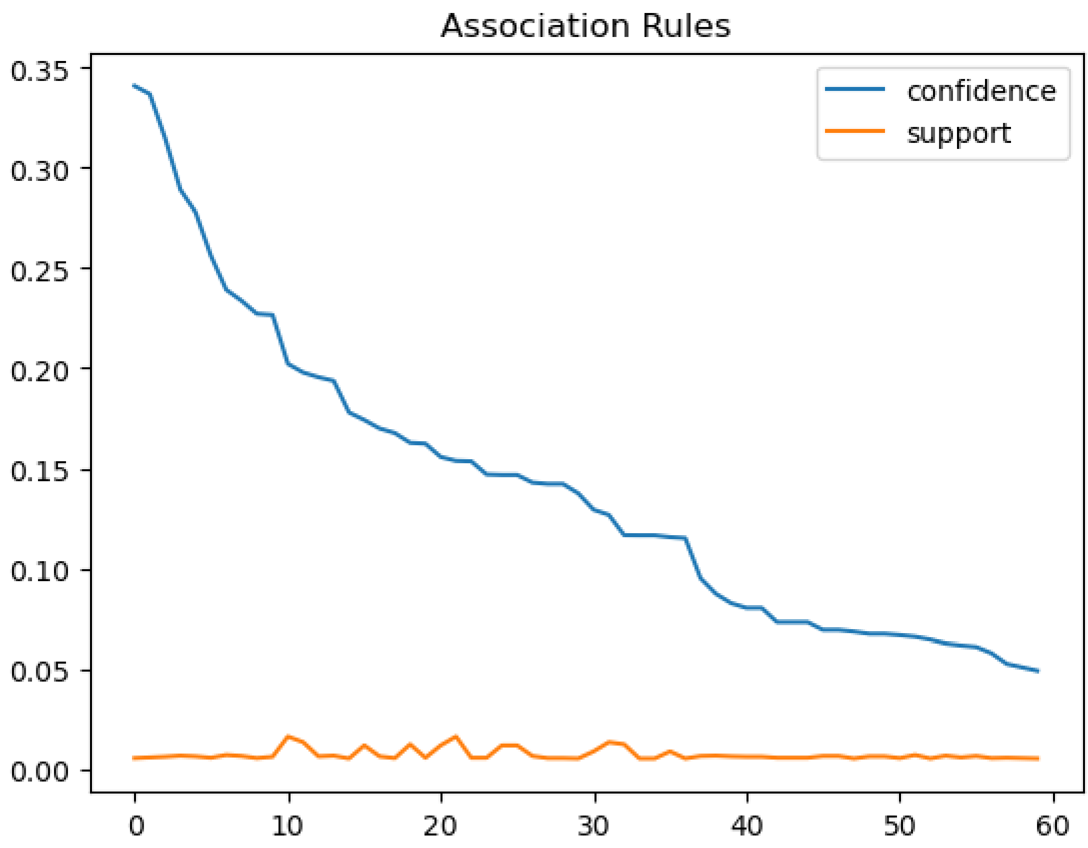
- **antecedent support**: the proportion of transactions that contain the antecedent set of items.
- **consequent support**: the proportion of transactions that contain the consequent set of items.
- **support**: the proportion of transactions that contain both the antecedent and consequent sets of items.
- **confidence**: the proportion of transactions that contain both the antecedent and consequent sets of items among the transactions that contain the antecedent set of

items. It measures how often the consequent items appear in transactions that contain the antecedent items.

- **lift**: the ratio of observed support to expected support if the antecedent and consequent were independent. A lift value greater than 1 indicates a positive correlation between the antecedent and consequent, while a value less than 1 indicates a negative correlation.
- **leverage**: the difference between observed support and expected support if the antecedent and consequent were independent. It measures the deviation of the observed frequency of both sets of items from what would be expected if there was no association.
- **conviction**: the ratio of the expected frequency of the antecedent to the frequency of the antecedent that are not followed by the consequent. It measures the degree of implication between the antecedent and consequent.
- **zhangs_metric**: a metric that takes into account the direction and strength of association between the antecedent and consequent. A value greater than 0 indicates a positive correlation, while a value less than 0 indicates a negative correlation.

In [63]:

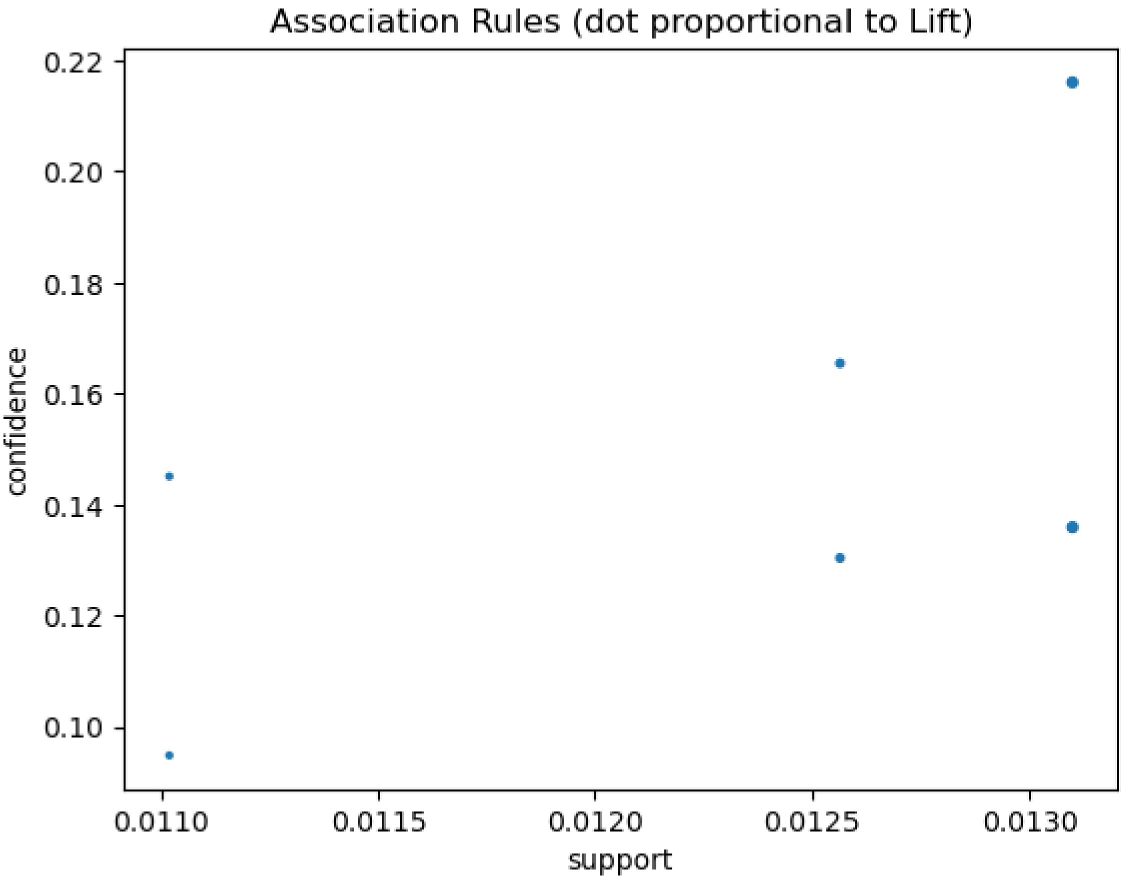
```
# Graph representing confidence and support

sorted_rules=rules.sort_values(by=['confidence','support'],ascending=False).reset_i
sorted_rules[['confidence','support']].plot(title='Association Rules');
```



In [105…

```
# 1.8 is chosen empirically to obtain the best graphical effect
s = [3**n for n in rules.lift]
rules.plot.scatter(x='support',
                   y='confidence',
```

```
                    title='Association Rules (dot proportional to Lift)',
                    s=s);
```



Conclusions

We are able to predict the reordering with a good performance on the sampled data. Unfortunately due to computational problems we were not able to make predictions on the whole dataset: it would be nice to be able to evaluate the performance on the whole dataset. In fact we have an F1 score of 89% and an accuracy of 87% on the test set. Which suggests that we are likely to be able to predict with good performance on the entire dataset as well. As far as the association rule is concerned, we have obtained interesting rules, but here too, the impossibility of being able to evaluate the perfoamance on the entire dataset leads us to limited conclusions.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: