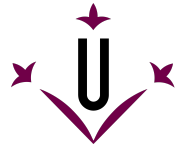
# Mining Frequent Itemsets (1)

Ramon Béjar

Data mining - Master on Computer Science



Universitat de Lleida

# Mining Frequent Itemsets

Consider a sequence/set of **transactions**:

$$T = \{T_1, T_2, \ldots, T_m\}$$

where each $T_i$ is a set of items that come from some catalog of possible items $I$. That is, $\forall T_i, T_i \subseteq I$.

Given a support threshold $\theta \in [0, 1]$, we say that a itemset $P \subseteq I$ is $\theta$—frequent if its support among $T$ is $\geq \theta$:

$$\frac{|\{T_i | P \subseteq T_i, T_i \in T\}|}{|T|} \geq \theta$$

# Marketing in Retail Stores

The context where this problem has been defined and studied extensively is the analysis of customer habits in traditional retail stores.

Discovering frequently bought together products (items) in different purchases (transactions) can be useful for making marketing campaigns directed towards capturing the interest of most customers of the store.

For example, suppose a supermarket finds that the pair of products:

$$\{beer, cocacola\}$$

is $0.76-$frequent among the transactions of the last month and that the pair of products:

$$\{beer, nachos\}$$

is $0.60-$frequent. Then, we may think that it may be good developing marketing strategies that promote buying such pairs of products.

Some straightforward marketing options:

- Put these products close to each other in some location of the store $\Rightarrow$ to remember to ALL customers that it may be good to buy them together
- Offer discounts if you buy them together $\Rightarrow$ To offer an small incentive towards buying them, in case customers think they do not need them together "today"

# What about on-line stores?

Observe that for the case of on-line shops, focusing only on the most frequent subsets may be of less relevance, as in a on-line store one usually is interested in making recommendations personalized for every costumer (as there is no real limitation of a single selling space shared by all the costumers). In other words, we can think about presenting a totally personalized shop for every costumer of an on-line shop.

This approach will be considered when we talk about recommender systems.

# The A-Priori Algorithm

This algorithm is based on exploiting the following basic principle, the monotonicity of itemsets:

```
If a set P of items is frequent, then so is every subset of P
```

More concretely, if $P$ is $\theta-$frequent, then every subset of $P$ will be *at least* $\theta$-frequent. The typical situation will be that for a fixed frequency $\theta$, there will be much less frequent sets of size $k$ than of size $k-1$.

In the particular domain of retail stores, the typical transaction size will be small, so when considering high frequency thresholds, we will not find many frequent item sets of big size.

So, the A-Priori algorithm is based on first finding smaller frequent-items sets, and then going to the next size but considering only sets such that any of their subsets has been found previously to be frequent. This suggests an iterative algorithm that first looks for frequent itemsets of size 1, then of size 2, and so on.

# High-level pseudo-code of the A-Priori algorithm

# A-Priori Implementation in map-reduce

To implement the algorithm with map-reduce, in principle we could perform in every iteration two main tasks:

1. Compute $C_k$ from $L_{k-1}$ : This can be done as a series of map transformations from $L_{k-1}$ and $I$
2. Compute $L_k$ from $C_k$ and $T$ : This can be done as a series of map transformations from $C_k$ and $T$ and a final filter transformation that uses the $\theta$ value

However, it is more efficient to avoid the explicit computation of the whole set $C_k$

That is, to compute for every transaction $T_i$, those subsets $P \subseteq T_i$ that satisfy the **condition** of belonging to $C_k$:

$$
\begin{aligned}
&f_k(t \in T, L_{k-1}):\\
&\quad l = [(P_i, 1) \mid P_i \in \binom{t}{k}, \forall S_j \subset P_i, |S_j| = k-1 \rightarrow S_j \in L_{k-1}]\\
&\quad return\ l
\end{aligned}
$$

Then perform $T.flatMap(\lambda\ t : f_k(t, L_{k-1}))$ to get the elements from $C_k$ that belong to **at least one** $t \in T$

So, in a next reduceByKey operation, we can count in how many transactions appears each such $P_i$

Observe that for the Spark framework, the previous function cannot be directly applied in a simple map transformation from an RDD with $T$ if the set $L_{k-1}$ **is also an RDD**. Options:

1. Transform the RDD of $L_{k-1}$ to a python list.
2. Transform the RDD of $L_{k-1}$ to a spark broadcast variable.

Even if the second option is more efficient, both solutions can only be applied if $L_{k-1}$ fits into the memory of any computation node.

$\Rightarrow$ we may have scaling issues if $L_{k-1}$ grows too much

Other options:

1. Store $L_{k-1}$ in a database that can be accessed by any node of the cluster
2. Perform a join between $P_k = \{(t, P_i) \mid t \in T, P_i \in \binom{t}{k}\}$ and $L_{k-1}$ where the condition for the join of $(t, P_i)$ with $S_j \in L_{k-1}$ is that:
$$S_j \subset P_i$$
Then, observe that for any $(t, P_i)$ such that all its $k-1$-subsets are in $L_{k-1}$, the final join will contain $k$ copies of $(t, P_i)$ (each one with a different subset $S_j$).

This kind of conditional join can be done in spark when working with spark dataframes.

# A simple implementation for k up to 2

Assume $L_1$ is much smaller than $T$ and so it can fit into the memory of any node. Then, we have a very simple implementation for the case of $k = 2$:

1. Use a map-reduceByKey-filter chain to compute $L_1$ from $T$, and to compute a version of $T$ with only items present in $L_1$ (call it $T_{L_1}$)
2. Use a map to compute candidate pairs for each $t \in T$ from $T_{L_1}$ $(C_2(T))$
3. Use a reduceByKey-filter chain to compute $L_2$ from $C_2(T)$

## Phase 1: Compute $L_1$ and $T_{L_1}$

For each t in rddT, this function maps $t = [it_1, \ldots, it_k]$ to $[(it_1, 1), \ldots, (it_k, 1)]$, and then with reduceByKey and filter detects those items that are $\theta$—frequent

In [3]:
```
# Compute the rdd with frequent singleton sets (L_1)
def computeL1 ( rddT, numtrans, theta ):
  rddtemp = rddT.flatMap( lambda t : [ (it,1) for it in t ] ).reduceByKey( lamb
da a,b : a+b  )
  return rddtemp.filter( lambda x : (float(x[1])/numtrans) >= theta )
```

Next, once we convert L1 to a python list back to the driver program, we compute the RDD of $T_{L_1}$ from $T$ and L1:

In [4]:
```
# Map any transaction to its version without elements not in L1
# L1 must be a python list, not a RDD
def computeTfilteredByL1( seqOfT, L1 ):
    for t in seqOfT:
       yield [ it for it in t if (it in L1) ]
```

Observe that we expect to call this function with mapPartitions, not map, so the first argument is a sequence of transactions, provided by a python iterable object, and the result will be also a python iterable object.

## Phase 2: Compute $C_2(T)$ from $T_{L_1}$

In [5]:
```
# For each t in seqofFilteredT (they come from T_{L_1}), compute pairs (a,b) fr
om t that belong to C_2
def generateC2( seqofFilteredT ):
    for t in seqofFilteredT:
      cpairslist = []
      for (a,b) in [ (a,b) for i,a in enumerate(t[:-1]) for b in t[i+1:] ]:
                cpairslist.append( ((a,b),1) if (a <= b) else ((b,a),1)  )
      yield cpairslist
```

When we apply flatMap to the resulting RDD of this function, each element $((a, b), 1)$ will appear as many times as the number of transactions where $(a, b)$ appears.

## Phase 3: Compute $L_2$ from $C_2(T)$

Here we assume that rddC2T is the *flattened* version of the RDD obtained with generateC2

In [6]:
```
def computeL2( rddC2T, numtrans, theta ):
    pairsCountedrdd = rddC2T.reduceByKey( lambda v1,v2 : v1+v2 )
    # Finally, filter out from the previous rdd those pairs with frequency belo
w theta
    return pairsCountedrdd.filter( lambda x : (float(x[1])/numtrans) >= theta )
```

What we do is first to count the number of transactions with a same pair $(a, b)$, and then we filter those pairs that are $\theta-$ frequent.

# A-Priori Execution Example

To illustrate the execution of the A-Priori algorithm, consider the following set of transactions and $\theta = 0.25$:

In [7]:
```
beeranddiapers1 = [['beer','diapers','cheese'],
                   ['beer','diapers','pizza'],
                   ['beer','diapers','pizza','yogurt'],
                   ['beer','diapers','milk'],
                   ['beer','diapers','milk','pizza'],
                   ['beer','diapers','toothpaste'],
                   ['beer','diapers','icecream'],
                   ['beer','diapers','pizza','yogurt']]
numtrans = len(beeranddiapers1)
theta=0.25
```

In [8]:
```
rddT = sc.parallelize(beeranddiapers1)
```

**Phase 1**: compute the RDD for $L_1$ and $T_{L_1}$ and convert it to a python list back to the driver (remember, this may not scale well depending on the size of $L_1$):

In [9]:
```
rddL1 = computeL1 ( rddT, numtrans, theta )
L1 = rddL1.keys().collect() # we need only the items (keys) in final L1
TL1 = rddT.mapPartitions( lambda seqOfT : computeTfilteredByL1( seqOfT, L1 )  )
```

In [10]:
```
print(" L1 items: ", L1)
print(" Transactions with only frequent elements: ", TL1.collect())
```

```
 L1 items:  ['pizza', 'milk', 'diapers', 'yogurt', 'beer']
 Transactions with only frequent elements:  [['beer', 'diapers'], ['beer', 'di
apers', 'pizza'], ['beer', 'diapers', 'pizza', 'yogurt'], ['beer', 'diapers',
'milk'], ['beer', 'diapers', 'milk', 'pizza'], ['beer', 'diapers'], ['beer',
'diapers'], ['beer', 'diapers', 'pizza', 'yogurt']]
```

**Phase 2**: Compute $C_2(T)$ from $T_{L_1}$

In [11]:
```
rddC2T = TL1.mapPartitions( lambda seqOfFilteredT : generateC2( seqOfFilteredT ) )
rddC2TFlat = rddC2T.flatMap( lambda x : x )
```

In [12]:
```
print( "flattened C2T: ", rddC2TFlat.collect() )
```

```
flattened C2T:  [(('beer', 'diapers'), 1), (('beer', 'diapers'), 1), (('beer',
'pizza'), 1), (('diapers', 'pizza'), 1), (('beer', 'diapers'), 1), (('beer',
'pizza'), 1), (('beer', 'yogurt'), 1), (('diapers', 'pizza'), 1), (('diapers',
'yogurt'), 1), (('pizza', 'yogurt'), 1), (('beer', 'diapers'), 1), (('beer',
'milk'), 1), (('diapers', 'milk'), 1), (('beer', 'diapers'), 1), (('beer', 'mi
lk'), 1), (('beer', 'pizza'), 1), (('diapers', 'milk'), 1), (('diapers', 'pizz
a'), 1), (('milk', 'pizza'), 1), (('beer', 'diapers'), 1), (('beer', 'diaper
s'), 1), (('beer', 'diapers'), 1), (('beer', 'pizza'), 1), (('beer', 'yogur
t'), 1), (('diapers', 'pizza'), 1), (('diapers', 'yogurt'), 1), (('pizza', 'yo
gurt'), 1)]
```

**Phase 3**: Compute $L2$ from $C2(T)$

In [13]:
```
rddL2 = computeL2( rddC2TFlat, numtrans, theta )
```

In [16]:
```
rddL2 = rddL2.sortBy(lambda a: -a[1])
for it in rddL2.toLocalIterator():
    print (it)
```

```
(('beer', 'diapers'), 8)
(('beer', 'pizza'), 4)
(('diapers', 'pizza'), 4)
(('diapers', 'yogurt'), 2)
(('pizza', 'yogurt'), 2)
(('beer', 'milk'), 2)
(('beer', 'yogurt'), 2)
(('diapers', 'milk'), 2)
```

As you can see, beer and diapers make the most frequent pair.

# Finding Association Rules

The original application domain of the frequent itemsets problem was finding good marketing strategies in traditional retail stores. For example, in a famous application of this problem, it was found that most people that bought diappers were also buying beer. This suggests a possible association rule:

$$diapers \rightarrow beer$$

To quantify how strong/realistic this association rule can be considered, we may compute two measures for such rule from our dataset, using the information computed by the frequent itemsets algorithm, the **confidence** and the **interest**.

## Confidence

The first such measure is the **confidence** for the rule. The confidence for the rule $diapers \rightarrow beer$ is the ratio:

$$\frac{support(diapers, beer)}{support(diapers)}$$

where support(set) is the number of transactions where the set is found.

Observe that this quantity will be high when the fraction of transactions with diapers that also contain beer is high.

## Interest

But observe that this alone does not mean that the rule {diapers} -> {beer} necessarily shows a *true relationship*. That is, it could happen that all the costumers buy beer, so the rule {diapers} -> {beer} does not really provide an useful information to discover when people buys **more beer**. To quantify this situation, we need also to quantify the **interest** of the rule {I} -> {j} as the difference between its confidence and the frequency of {j}:

$$confidence(diapers \rightarrow beer) - frequency(beer)$$

Observe that the interest can be positive, zero o negative:

- A negative interest indicates a negative effect, i.e. that when {I} is present {j} is less likely to be present than in general.
- A positive interest indicates a positive effect, i.e. that when {I} is present {j} is more likely to be present than in general.
- A zero interest indicates no significative effect of {I} to {j}: {I} being present does not affect the frequency of {j}

For the diapers-beer example mentioned, the interest was positive, meaning that when diapers was present the relative frequency of beer was higher than when looking for beer in all the transactions.

So, towards building a marketing recommendation system, it can be interesting to use such association rules, and select the most interesting ones. For example, given a set of frequent itemsets P, we can compute the confidence and interest of any association rule of the kind $P \setminus j \rightarrow j$, for each item j present in P.

For a given frequent itemset P, observe that we can build $\left|P\right|$ different association rules of such kind.

To consider the difference between the confidence and the interest of an association rule consider the following transaction set:

In [17]:
```
beeranddiapers1 = [['beer','diapers','cheese'],
                   ['beer','diapers','pizza'],
                   ['beer','diapers','pizza'],
                   ['beer','diapers','milk'],
                   ['beer','diapers','milk','pizza'],
                   ['beer','diapers','toothpaste'],
                   ['beer','diapers','icecream'],
                   ['beer','diapers','pizza','yogurt']]
```

We have that the confidence for the rule:

$$diapers -> beer$$

is:

$$\frac{support(\{beer, diapers\})}{support(\{diapers\})} = \frac{8}{8} = 1$$

So, the confidence of the rule is maximum. However, we cannot strongly suggest that diapers make more likely to have beer, because the frequency of beer is as high as the frequency of beer when diapers are present:

$$confidence(\{diapers\} \rightarrow \{beer\}) - frequency(\{beer\}) = 1 - 1 = 0$$

That is, the interest of the rule for this transaction set is zero

Consider this second example:

In [18]:
```
beeranddiapers2 = [['beer','diapers','cheese'],
                   ['beer','diapers','pizza'],
                   ['beer','diapers','pizza'],
                   ['beer','diapers','milk'],
                   ['yogurt','milk','pizza'],
                   ['pizza','toothpaste'],
                   ['yogurt','icecream'],
                   ['milk','icecream'],
                   ['pizza','icecream'],
                   ['cheese','icecream'],
                   ['beer','pizza','yogurt']]
```

This time, we have that the confidence for the rule:

$$diapers -> beer$$

is:

$$\frac{support(\{beer, diapers\})}{support(\{diapers\})} = \frac{4}{4} = 1$$

So, again the confidence of the rule is maximum. However, in this case the rule is really signalling a stronger relationship between both products, because the interest is:

$$confidence(\{diapers\} \rightarrow \{beer\}) - freq(\{beer\}) = 1 - (5/11) = 0.54$$

That is, in the second transaction set the presence of diapers makes more likely the presence of beer

# Exercise (will be graded)

Consider the following programming exercise. Given the information of the frequent singletons $(L_1)$ and frequent pairs $(L_2)$ we compute with our previous implementation of A-Priori for k=2, implement in spark functions to compute the **confidence** and **interest** of all the binary rules we can build from the set $L_2$. As the dataset to test your code, **compute** a set of transactions from this data set:

https://www.kaggle.com/datasets/heeraldedhia/groceries-dataset (https://www.kaggle.com/datasets/heeraldedhia/groceries-dataset)

Or this smaller one (that it already contains one transaction per line) if you are not able to work with the previous one:

https://www.kaggle.com/shazadudwadia/supermarket?select=GroceryStoreDataSet.csv (https://www.kaggle.com/shazadudwadia/supermarket?select=GroceryStoreDataSet.csv)

Try these values for $\theta$: 0.01, 0.1, 0.15

Once you have computed the sets T, L1, $T_{L1}$ and L2, your program should follow these steps:

1. Map, using mapPartitions, each frequent pair in the RDD with L2 to its list of binary association rules (two association rules per each different frequent pair). Use then the flattened version of this RDD.
2. Map each association rule of the previous resulting RDD, to a triple with (rule,confidence,interest). Observe that you will need to use the information in L1 and the number of transactions to compute these values. You can use the version of L1 stored as a python list in the driver (so it can be passed inside functions passed to spark tasks).
3. Finally, sort the association rules by their interest, and show back in the driver program the first 10 most interesting rules