Definitions

Market Baskets. Consider a set \( I = \{i_1, \ldots, i_m\} \). We call the elements of \( I \), items.
A market basket is any subset \( S \) of \( I \): \( S \subseteq I \).
A market basket dataset or a set of market basket transactions (or transactions for short) is a collection \( T = \{t_1, \ldots, t_n\} \) of market baskets.
A subset \( T' = \{t'_1, \ldots, t'_k\} \subseteq T \) of a market basket dataset is called a tidset. Without loss of generality, we allow the notation \( t_i \in T \) describing a single market basket in the dataset \( T \) to be viewed both as a unique identifier of that market basket (a tid, or - essentially - a receipt Id), and the market basket itself, i.e., the items that comprise it.

Association Rules. Let \( I \) be a set of items. An association rule is an implication of the form

\[ X \rightarrow Y, \]

where \( X \subseteq I, Y \subseteq I \) and \( X \cap Y = \emptyset \).
We refer to \( X \) and \( Y \) as itemsets.

Meaning of Association Rules. The association rule

\[ X \rightarrow Y \]

is a stand-in for an English statement

*Whenever a market basket contains all items from the itemset \( X \), it tends to contain items from the itemset \( Y \).*

The statement above is *not categorical* — the relationship between the left-hand side itemset \( X \) and the right-hand side itemset \( Y \) is ”tends to”.
Our first goal is to determine the degree of the strength of the relationship between \( X \) and \( Y \) in an association rule \( X \rightarrow Y \).
Turns out, this degree can be measured.

Support and confidence for association rules. Let \( I \) be a set of items and \( T = \{t_1, \ldots, t_n\} \) be a market basket dataset. Let \( R : X \rightarrow Y \) be an association rule.
The support of an itemset \( X \), is the number of market baskets \( t_i \in T \) that contain it:

\[ \text{support}_T(X) = |\{t_i \in T | X \subseteq t_i\}|. \]

The relative support of an itemset \( X \), is the percentage of market baskets \( t_i \in T \) that contain it:

\[ \text{rsupport}_T(X) = \frac{|\{t_i \in T | X \subseteq t_i\}|}{n}. \]

The support of an association rule \( R \) in the dataset \( T \) is the number of market baskets \( t_i \in T \) which contain \( X \cup Y \):
support}_T(X \rightarrow Y) = |\{t_i \in T | X \cup Y \subseteq t_i\}|.

The relative support of an association rule _R_ in the dataset _T_ is the percentage of market baskets _t_i \in T_ which contain _X \cup Y_:

\[ \text{rsupport}_T(X \rightarrow Y) = \frac{|\{t_i \in T | X \cup Y \subseteq t_i\}|}{n}. \]

The confidence of an association rule _R_ in the dataset _T_ is the percentage of market baskets _t_i \in T_ that contain _X_, which also contain _Y_:

\[ \text{confidence}_T(X \rightarrow Y) = \frac{|\{t_i \in T | X \cup Y \subseteq t_i\}|}{|\{t_j \in T | X \subseteq t_j\}|}. \]

- **Support** of an association rule determines its coverage: how many market baskets (or what percentage of all market baskets) the rule affects. We want to find association rules with high support, because such rules will be about transactions/market baskets that commonly occur.

- **Confidence** of an association rule determines its predictability, i.e., how often it occurs among the affected market baskets. We want to find association rules with high confidence, because such rules represent strong relationships between items.

**Assertion of association rules.** Given an association rule _X \rightarrow Y_, the strength of this association can be determined by support(_X \rightarrow Y_) and confidence(_X \rightarrow Y_).

If we are satisfied with the values of the support and confidence for _X \rightarrow Y_, we say that we can assert the association rule _X \rightarrow Y_.

**Association Rules mining problem.** Given a set of items _I_, a market basket dataset _T_ and two numbers: minSup and minConf, find all association rules that occur with the support of at least minSup and confidence of at least minConf for _T_.

Note: The problem of mining association rules requires the return of all association rules found, i.e., it is complete. There are variations on the theme, which allow for return of a subset of all discovered association rules.

**Brute-force solution for association rules mining problem.** There is a simple brute-force algorithm for mining association rules:

```
Algorithm ARM_BRUTE_FORCE(T, I, minSup, minConf)
    for each X such that X is a subset of I
        for each Y such that Y is a subset of I
            if Y and X are disjoint then
                compute s := support(T, X->Y);
                compute c := confidence(T, X-> Y);
                if (s > minSup) AND (c > minConf) then output(X->Y);
            fi
        end // for
    end // for
```

_I_ has _m_ elements, hence the outer loop has _2^m_ iterations. Given _X \subset I_, there are _2^m-|X|_ choices to select _Y_, which, in average, gives us _2^{\frac{m}{2}}_ iterations of the inner loop. Assuming that computations of support() and confidence() functions require polynomial time\(^1\) in the size of the input (which is _|T| + |I| = n + m_), we estimate the algorithmic complexity of ARM_BRUTE_FORCE as

\[ O(2^{1.5m} \cdot P(n + m)), \]

\(^1\)As we will see below, they, indeed, do.
where \( P(x) \) is a polynomial function.

This running time has two properties:

- **It is exponential in the number of unique items.** Typical Association Rule Mining problems have large inventories, i.e., sets of available items (think a grocery store!). This makes the brute force algorithm an unacceptable choice in production.

- **It is polynomial in the number of transactions.** When the number of transactions is large enough so that all transactions cannot fit main memory, \( T \) is stored on disk, and the body of the inner loop entails a full scan of the set of transactions \( T \) from disk. This is a costly operation, and the brute force algorithm as described above implies \( O(2^m) \) full scans - which is also unacceptable.

However, it is not very difficult to **significantly improve** the process of finding association rules.

### Apriori Algorithm

**Apriori Algorithm** [1] was the first efficient algorithm for mining association rules. **Apriori Algorithm** is an algorithm for discovery of **frequent itemsets** in a dataset.

**Frequent itemsets.** Let \( I \) be a set of items and \( T \) be a market basket dataset. Given a minimum support number \( \text{minSup} \), an itemset \( X \subseteq I \) is a **frequent itemset** in \( T \), iff \( \text{support}_T(X) > \text{minSup} \).

**The Apriori Principle.** (also known as **Downward Closure Property**). This principle establishes the main driving force behind the **Apriori algorithm**.

If \( X \) is a frequent itemset in \( T \), then its every non-empty subset is also a frequent itemset in \( T \).

**Why is this useful?** Any frequent itemset discovery algorithm is essentially a specialized search algorithm over the space of all itemsets. **Apriori principle** allows us to **prune** potentially a lot of itemsets from consideration: if a set \( X \) is known to NOT be a frequent itemset, then any superset of \( X \) will not be frequent!

**Idea behind algorithm. Level-wise search:** search for frequent itemsets by the itemset size: first find all frequent itemsets of size 1, then — all frequent itemsets of size 2, then — all frequent itemsets of size 3, and so on.

**The algorithm.** The **Apriori algorithm** consists of two parts. Figure 1 shows the pseudocode for the algorithm itself. The algorithm, on each step calls **candidateGen()** function. The pseudocode of this function is shown in Figure 2.

**Function candidateGen().** On step \( i \) of its execution, the **Apriori algorithm** discovers frequent itemsets of size \( i \). On each step starting with step 2, function **candidateGen()** is called. On step \( i \) it takes as input the list of frequent itemsets of size \( i - 1 \) computed on previous step and outputs the list of candidate frequent itemsets of size \( i \). The **Apriori algorithm** then checks whether the support for each itemset returned by **candidateGen()** exceeds \( \text{minSup} \).

**candidateGen()** function works as follows. On step \( k \), it receives as input a list \( F_{k-1} \) of frequent itemsets of size \( k - 1 \). It considers all itemsets of size \( k \) which can be constructed as unions of pairs of itemsets from \( F_{k-1} \) (join step). candidateGen() function then checks if all subsets of size \( i - 1 \) of such unions belong to \( F_{k-1} \) (pruning step). Itemsets that pass this check are added to the list of candidate frequent itemsets that is eventually returned.
Algorithm Apriori($T$, $I$, $\text{minSup}$)
begin
$F_1 := \{\{i\} | i \in I; \text{support}_T(\{i\}) \geq \text{minSup}\}$; //first pass
$k := 2$; //main loop
repeat
$C_k = \text{candidateGen}(F_{k-1}, k - 1)$; //candidate frequent itemsets
foreach $c \in C_k$ do $\text{count}[c] := 0$; //initialize counts
foreach $t \in T$ do
  foreach $c \in C_k$ do
    if $c \subseteq t$ then $\text{count}[c] += 1$;
  endfor
endforeach
$F_k = \{c \in C_k | \frac{\text{count}[c]}{n} \geq \text{minSup}\}$; //filter out all itemsets with insufficient support
$k := k + 1$;
until $F_{k-1} = \emptyset$;
return $F := \bigcup_{i=1}^{k-1} F_i$;
end

Figure 1: Apriori Algorithm for mining association rules.

Function candidateGen($F, k$)
begin
$C := \emptyset$;
foreach $f_1, f_2 \in F$ s.t. $|f_1| = |f_2| = k$ do
  if $|f_1 \cup f_2| = |f_1| + 1$ then
    $c := f_1 \cup f_2$; // join step
    $\text{flag} := \text{true}$;
    foreach $s \subseteq c$ s.t. $|s| = |c| - 1$ do // pruning step
      if $s \notin F$ then $\text{flag} := \text{false}$;
    endfor
    if $\text{flag} == \text{true}$ then $C := C \cup c$;
  endif
endfor
return $C$;
end

Figure 2: Generation of candidate frequent itemsets.
Properties of Apriori Algorithm

Worst-case complexity. Apriori algorithm has \(O(2^N)\) (where \(N\) is the size of the input) algorithmic complexity. This is because in the worst case scenario, all \(2^N\) possible itemsets are frequent and have to be explored.

The heuristic efficiency of the Apriori algorithm comes from the fact that typically observed market basket data is sparse. This means that, in practice, relatively few itemsets, especially large itemsets, will be frequent.

Data Complexity. What makes Apriori an excellent data mining algorithm is its data complexity\(^2\). The algorithm uses \(\min(K + 1, m)\) scans of the input dataset, where \(K\) is the size of the largest frequent itemset.

Memory Footprint. Another important property of Apriori is its small memory footprint. Each market basket \(t \in T\) is analyzed independently from others, so, only a small number of market baskets needs to be kept in main memory at each moment of time.

Formally, the data complexity of the Apriori algorithm is \(O(1)\).

Level-wise search. Each iteration of the Apriori produces frequent itemsets of specific size. If larger frequent itemsets are not needed, the algorithm can stop after any iteration.

Finding Association Rules

Apriori Algorithm discovers frequent itemsets in the market basket data. A collection of frequent itemsets can be extended to a collection of association rules using Algorithm GenRules described in Figure 3.

Idea. Let \(f\) be a frequent itemset of size greater than 1. Given \(f\), we will consider all possible association rules of the form

\[(f - \alpha) \rightarrow \alpha \quad \text{for all} \quad \alpha \subset f.\]

For each such rule, we will compute \(\text{confidence}_T((f - \alpha) \rightarrow \alpha)\) and compare it to \(\minConf\) number given to us as input.

Algorithm genRules. This algorithm proceeds similarly to the Apriori algorithm. For each frequent itemset, first, genRules generates all rules with a single item on the right, and finds among them those, that have confidence higher than \(\minConf\).

After that, it uses candidateGen function to generate candidate rules more items on the right. For each candidate rule returned by candidateGen, the algorithm computes its confidence and determines if the rule should be reported.

Data Formats for Mining Association Rules

Market Baskets as Sparse Vectors

In a typical scenario, given a list of items \(I\) and a list of market baskets/transactions \(T = \{t_1, \ldots, t_n\}\),

\[|I| >> |t_i|.\]

\(^2\)Data complexity of a problem is the number of Input/Output operations necessary to complete solve the problem. This way of measuring performance of algorithms comes from database systems, where data complexity, rather than algorithmic complexity is used to estimate the quality of query processing algorithms.
Algorithm genRules($F$, $\text{minConf}$) // $F$ - frequent itemsets
begin
   foreach $f \in F$ s.t. $|f| = k \geq 2$ do
       $H_1 = \emptyset$;
       foreach $s \in f$ do
           if $\text{confidence}_T(f - \{s\} \rightarrow \{s\}) \geq \text{minConf}$ then
               $H_1 := H_1 \cup \{f - \{s\} \rightarrow \{s\}\}$;
           endfor
       apGenRules($f, H_1$);
   endfor
end

Procedure apGenRules($f, H_m$)
begin
   if ($k > m + 1$) AND $H \neq \emptyset$ then
      $H_{m+1} := \text{candidateGen}(H_m, m)$;
      foreach $h \in H_{m+1}$ do
         $\text{confidence} := \frac{\text{count}(f)}{\text{count}(f - h)}$;
         if $\text{confidence} \geq \text{minConf}$ then
            output $(f - h) \rightarrow h$; // new rule found
         else
            $H_{m+1} := H_{m+1} - \{h\}$
         endif
      endfor
   endif
   apGenRules($f, H_{m+1}$)
end

Figure 3: Generation of association rules from frequent itemsets.

That is, individual market baskets are relatively small, when compared to the set of all possible items.

Dense Vector representation. If $I$ is not large, $T$ can be represented as a set of dense binary vectors:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

In the example above, $|I| = 8$. Each element $t_i \in T$ is represented as a binary vector of size 8. E.g., the first vector indicates a market basket $\{i_2, i_6, i_8\}$.

Advantages:
- Regular representation;
- Suitable for relational databases.

Disadvantages:
- Inefficient use of space.
Sparse Vector representation. If \( I \) is large, dense vectors will contain way too many zeroes and, will require significant overhead when read and processed. **Sparse vector representation** is used in this case. E.g. the dataset above can be represented as follows (first row is the tid):

\[ \begin{align*}
\text{t1, 2, 6, 8} \\
\text{t2, 4, 5, 6, 8} \\
\text{t3, 2, 4, 5, 6, 8} \\
\text{t4, 1, 2, 6, 8} \\
\text{t5, 2, 8} \\
\text{t6, 1, 2, 7}
\end{align*} \]

Here, each vector contains information about the *non-empty* columns in it.

**Advantages:**

- Efficient use of space.
- Universality.
- Relatively straightforward algorithms for simple vector operations.

**Disadvantages:**

- Not very suitable for relational databases.
- Variable-length records.

Vertical Representation of Market Basket Data. In some computation one may take advantage of an *inverted index* style of representing the market basket data. The "forward index" is the sparse vector representation above: it associates a collection of column identifiers (item IDs) with each row Id (transaction/market basket id).

An *inverted index* does the opposite: with each column Id (item Id) it associates the list of transaction IDs that contain that item. For market basket collections this is often referred to as a *vertical representation*.

The dataset above can be represented "vertically" as follows:

\[ \begin{align*}
1, \text{t4, t6} \\
2, \text{t1, t3, t4, t5, t6} \\
3, \\
4, \text{t2, t3} \\
5, \text{t2, t3} \\
6, \text{t1, t2, t3, t4} \\
7, \text{t6} \\
8, \text{t1, t2, t3, t4, t5}
\end{align*} \]

**Relational Data as Market Baskets**

**Market Baskets** are binary vectors. A lot of data that could use association rules mining is relational in nature, i.e., each "item" can have more than one value.
Example. Let \( I = \{ \text{CSC365, CSC366, CSC480, CSC437, CSC408, CSC466, CSC481, CSC409} \} \), a list of eight Computer Science/Software Engineering electives at Cal Poly.

In a simple market basket dataset, each market basket is a student record indicating which electives the student took. Consider, for example, the following six student records:

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Binary Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>{CSC365, CSC366, CSC480}</td>
<td>1,1,1,0,0,0,0,0</td>
</tr>
<tr>
<td>{CSC408, CSC409}</td>
<td>0,0,0,0,1,0,0,1</td>
</tr>
<tr>
<td>{CSC365, CSC366, CSC408, CSC409}</td>
<td>1,1,0,0,1,0,0,1</td>
</tr>
<tr>
<td>{CSC480, CSC437, CSC481}</td>
<td>0,0,1,1,0,0,1,0</td>
</tr>
<tr>
<td>{CSC480, CSC481}</td>
<td>0,0,1,0,0,0,0,1</td>
</tr>
<tr>
<td>{CSC365, CSC480, CSC481}</td>
<td>1,0,1,0,0,0,1,0</td>
</tr>
</tbody>
</table>

Using this dataset, we can find patterns in classes students choose to take. However, we won’t find any patterns concerning student performance in the classes.

This dataset, however, can be expanded to specify student grades in each course they take. Assume for a moment, that a student can have one of the following grades: A, B, C, F in the class. We can then consider the following relational database snapshot of the data above:

<table>
<thead>
<tr>
<th>Student</th>
<th>CSC365</th>
<th>CSC366</th>
<th>CSC480</th>
<th>CSC437</th>
<th>CSC408</th>
<th>CSC466</th>
<th>CSC481</th>
<th>CSC409</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>A</td>
<td>NULL</td>
<td>NULL</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>C</td>
<td>NULL</td>
<td>NULL</td>
<td>B</td>
<td>NULL</td>
<td>NULL</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td>NULL</td>
<td>B</td>
<td>C</td>
<td>NULL</td>
<td>NULL</td>
<td>C</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>NULL</td>
<td>NULL</td>
<td>A</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>A</td>
<td>NULL</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>NULL</td>
<td>B</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>B</td>
<td>NULL</td>
</tr>
</tbody>
</table>

We may be interested in finding association rules of the sort:

Students with a C in CSC365 tended to take CSC 408 and earn B in it.

Converting Relational Datasets into Market Basket datasets. Let \( R = (A_1, \ldots A_s) \) be the relational schema (w.o. the primary key). For simplicity, let \( \text{dom}(A_i) = \{ a_{i1}, \ldots, a_{il} \} \). Given \( R \) and a set of tuples \( T = \{ t_1, \ldots, t_n \} \) over schema \( R \), we construct a set of items \( I_R \) and a market basket dataset \( \hat{T} = \{ \hat{t}_1, \ldots, \hat{t}_n \} \) as follows:

- The set of items \( I_R = \{ (A_1, a_{11}), \ldots, (A_1, a_{1l}), (A_2, a_{21}), \ldots, (A_2, a_{2l}), \ldots, (A_s, a_{sl}) \} \).
  That is, each item in the set of items \( I_R \) a name-value pair from the relational schema \( R \).

- A tuple \( t = (b_1, b_2, \ldots, b_s) \) is converted into a binary vector
  \( \hat{t} = (x_{11}, \ldots, x_{11}, x_{21}, \ldots, x_{21}, \ldots, x_{s1}, \ldots, x_{sl}) \),
  where, \( x_{1b_1} = x_{2b_2} = \ldots = x_{sb_s} = 1 \) and all other \( x_{ij} = 0 \).

Apriori Algorithm for Relational Datasets. Once we convert relational data to market basket data, we can apply a modified version of the Apriori algorithm to find frequent itemsets. The following modification needs to be made to the candidateGen() function:

- When creating the list of candidate frequent itemsets for the join stage of the Apriori Algorithm, generate only the itemsets that have no more than one column for each original attribute \( A_1, \ldots, A_s \) of the relational dataset.

Example. The dataset of student transcripts described above can be transformed to a market basket dataset as follows.
• The set \( I \) of items is:

\[
I = \{ \text{CSC365A, CSC365B, CSC365C, CSC365F}, \\
\text{CSC366A, CSC366B, CSC366C, CSC366F}, \\
\text{CSC480A, CSC480B, CSC480C, CSC480F}, \\
\text{CSC437A, CSC437B, CSC437C, CSC437F}, \\
\text{CSC408A, CSC408B, CSC408C, CSC408F}, \\
\text{CSC466A, CSC466B, CSC466C, CSC466F}, \\
\text{CSC481A, CSC481B, CSC481C, CSC481F}, \\
\text{CSC409A, CSC409B, CSC409C, CSC409F} \} 
\]

• The six transcript fragments described above are transformed into the following six binary vectors (for simplicity, we group columns for the same course):

<table>
<thead>
<tr>
<th></th>
<th>365</th>
<th>366</th>
<th>480</th>
<th>437</th>
<th>408</th>
<th>466</th>
<th>481</th>
<th>409</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector 1</td>
<td>1,0,0,0, 0,1,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector 2</td>
<td>0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 1,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 1,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector 3</td>
<td>0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector 4</td>
<td>0,0,0,0, 0,0,0,0, 1,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 1,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector 5</td>
<td>0,0,1,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector 6</td>
<td>0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• In the sparse vector notation, the transformed dataset is represented as follows:

1, 6, 10
17, 29
3, 6, 18, 30
10, 15, 27
9, 25
3, 10, 26

(note, that now the sparse dataset representation really takes significantly less space)

**Dealing with numeric parameters.** The transformation above applies to the situations when all attributes in the relational dataset are **categorical**. When some attributes are **numerical**, and come with **large domains** (or are continuous), these domains need to be **discretized**:

• If the domain of an attribute \( A \) is continuous (or a large discrete numeric), the **discretization** process involves selection of a **small number** of **value ranges** and replacement of the attribute \( A \) in the dataset with a new attribute \( A_d \), whose value is the discretized version of \( A \).

**Example.** Consider, for example a relational domain,

\[
R = (\text{YearsWithCompany, Salary, Position, Department}),
\]

which specifies four attributes for employees of some company. Suppose that \( \text{YearsWithCompany} \) ranges from 0 to 30, and \( \text{Salary} \) ranges from $20,000 to $110,000. Also, let’s assume that the domain of \( \text{Position} \) is \{Assistant, Associate, Manager, Senior Manager, Head\} and the domain of \( \text{Department} \) is \{Sales, Production, HR, Analytics\}. Consider the following small dataset:
YearsWithCompany | Salary | Position    | Department
---|---|---|---
5  | 70,000 | Manager | Sales
23 | 105,000 | Head | HR
2  | 36,000 | Assistant | Production
3  | 60,000 | Associate | Analytics
16 | 85,000 | Senior Manager | Production

Before converting it into a **market basket dataset**, we first, **discretize** YearsWithCompany and Salary:

<table>
<thead>
<tr>
<th>YearsWithCompany Range</th>
<th>Discretized Value</th>
<th>Salary Range</th>
<th>Discretized Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 — 3</td>
<td>newbie</td>
<td>20,000 — 39,999</td>
<td>low</td>
</tr>
<tr>
<td>4 — 10</td>
<td>average</td>
<td>40,000 — 64,999</td>
<td>medium-low</td>
</tr>
<tr>
<td>11 — 20</td>
<td>veteran</td>
<td>65,000 — 84,999</td>
<td>medium-high</td>
</tr>
<tr>
<td>20 — 30</td>
<td>dedicated</td>
<td>85,000 — 110,000</td>
<td>high</td>
</tr>
</tbody>
</table>

We can now, replace these two attributes in the dataset with YWCDiscr and SalaryDiscr:

| YWCDiscr | SalaryDiscr | Position    | Department
---|---|---|---
| average | medium-high | Manager | Sales
| dedicated | high | Head | HR
| newbie | low | Assistant | Production
| newbie | medium-low | Associate | Analytics
| veteran | high | Senior Manager | Production

This dataset contains four categorical attributes and can be transformed into a **market basket dataset** as described above.

**Discretizing categorical attributes.** Analysts may choose to discretize certain categorical attributes to provide better/simpler views of their data.

For example, we could choose to merge A and B grades into a single attribute for each course. This would reduce the size of the dataset (going from 32 columns to 24) and would potentially uncover new association rules.

**References**