Data Mining: Mining Association Rules

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: $S \subset I$.

A **market basket dataset** or a set of **market basket transactions** is a collection $T = \{t_1, \ldots, t_n\}$ of *market baskets*.

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

$$X \longrightarrow Y,$$

where $X \subset I$, $Y \subset I$ and $X \cap Y = \emptyset$.

We refer to X and Y as **itemsets**.

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 $R : X \longrightarrow Y$ be an association rule.

The **support** of R in the dataset T is the percentage of market baskets $t_i \in T$ which contain $X \cup Y$:

$$support_T(X \longrightarrow Y) = \frac{|\{t_i \in T | X \cup Y \subseteq t_i\}|}{n}.$$

The **confidence** of R in the dataset T is the percentage of market baskets $t_i \in T$ that contain X, which *also* contain Y:

$$confidence_T(X \longrightarrow Y) = \frac{|\{t_i \in T | X \cup Y \subseteq t_i\}|}{|\{t_j \in T | X \subseteq t_j\}|}.$$

Also, given an itemset X, its **support** is the percentage of market baskets $t_i \in T$ that contain it:

$$support_T(X) = \frac{|\{t_i \in T | X \subseteq t_i\}|}{n}.$$

- **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.*

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:

```
Algrithm 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], we estimate the algorithmic complexity of ARM_BRUTE_FORCE as

$$O(2^{1.5m} \cdot P(|T|+|I|+|minSup|+|minConf|) = O(2^{|T|+|I|+|minSup|+|minConf|}),$$

i.e., the running time of the brute force algorithm is exponential in the size of the input.

[1] As we will see below, they, indeed, do.

```
Algorithm Apriori(T, I, minSup)
begin
 F_1 := {{i}|i ∈ I; support_T({i}) ≥ minSup}; //first pass
 k := 2;
 repeat  //main loop
   C_k = candidateGen(F_{k-1}, k − 1);   //candidate frequent itemsets
   foreach c ∈ C_k do count[c] := 0; //initialize counts
   foreach t ∈ T do
     foreach c ∈ C_k do
       if c ⊆ t then count[c] + +;
     endfor
   endfor
   F_k = {c ∈ C_k| count[c]/n ≥ minSup};
   k := k + 1;
 until F_{k-1} = ∅;
 return F := ∪_{i=1}^{k-1} F_i;
end
```

Figure 1: Apriori Algorithm for mining 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 minSup, an itemset $X \subseteq I$ is a **frequent itemset** in T, **iff** $support_T(X) >$ 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(F,k)
begin
 C := ∅;
  foreach f1, f2 ∈ F s.t. |f1| = |f2| = k do
    if |f1 ∪ f2| == |f1| + 1 then
      c := f1 ∪ f2;                    // join step
      flag := true;
      foreach s ⊆ c s.t. |s| = |c| − 1 do   // pruning step
        if s ∉ F then flag := false;
      endfor
      if flag == true then C := C ∪ c;
    endif
  endfor
  return C;
end
```

Figure 2: Generation of candidate frequent itemsets.

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 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.

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^m 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*.

[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.

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.

(*Note to CSC 468 students: if data for Apriori is paginated and database-style buffer management is used to bring it from persistent storage into main memory, then the memory footprint of the algorithm is O(1), as a single disk buffer is sufficient to support the scan operation.*)

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 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

$$(f - \alpha) \longrightarrow \alpha \text{ for all } \alpha \subset f.$$

For each such rule, we will compute $confidence_T((f - \alpha) \longrightarrow \alpha)$ 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/transations** $T = \{t_1, \ldots, t_n\}$,

$$|I| >> |t_i|.$$

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

```
Algorithm genRules(F, minConf) // F - frequent itemsets
begin
  foreach f ∈ F s.t. |f| = k ≥ 2 do
    H_1 = ∅;
    foreach s ∈ f do
      if confidence_T(f − {s} ⟶ {s}) ≥ minConf then
        H_1 := H_1 ∪ {f − {s} ⟶ {s}};
    endfor
    apGenRules(f,H_1);
  endfor
end
```

```
Procedure apGenRules(f, H_m)
begin
  if (k > m + 1) AND H ≠ ∅ then
    H_{m+1} := candidateGen(H_m, m);
    foreach h ∈ H_{m+1} do
      confidence := count(f) / count(f−h);
      if confidence ≥ minConf then
        output (f − h) ⟶ h;   //new rule found
      else
        H_{m+1} := H_{m+1} − {h}
    endfor
  apGenRules(f,H_{m+1})
  endif
end
```

Figure 3: Generation of association rules from frequent itemsets.

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

```
0 1 0 0 0 1 0 1
0 0 0 1 1 1 0 1
0 1 0 1 1 1 0 1
1 1 0 0 0 1 0 1
0 1 0 0 0 0 0 1
1 1 0 0 0 0 1 0
```

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:

```
2,6,8
4,5,6,8
2,4,5,6,8
1,2,6,8
2,8
1,2,7
```

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.

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 $I = \{CSC365, CSC366, CSC480, CSC437, CSC408, CSC466, CSC481, CSC409\}$, 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:

Itemset	Binary Vector
$\{CSC365, CSC366, CSC480\}$	1,1,1,0,0,0,0,0
$\{CSC408, CSC409\}$	0,0,0,0,1,0,0,1
$\{CSC365, CSC366, CSC408, CSC409\}$	1,1,0,0,1,0,0,1
$\{CSC480, CSC437, CSC481\}$	0,0,1,1,0,0,1,0
$\{CSC480, CSC481\}$	0,0,1,0,0,0,1,0
$\{CSC365, CSC480, CSC481\}$	1,0,1,0,0,0,1,0

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:

Student	CSC365	CSC366	CSC480	CSC437	CSC408	CSC466	CSC481	CSC409
1	A	B	B	NULL	NULL	NULL	NULL	NULL
2	NULL	NULL	NULL	NULL	A	NULL	NULL	A
3	C	C	NULL	NULL	B	NULL	NULL	B
4	NULL	NULL	B	C	NULL	NULL	C	NULL
5	NULL	NULL	A	NULL	NULL	NULL	A	NULL
6	C	NULL	B	NULL	NULL	NULL	B	NULL

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 $dom(A_i) = \{a_{i1}, \ldots, a_{il}\}$. 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 $\hat{t} = (x_{11}, \ldots, x_{1l}, x_{21}, \ldots, x_{2l}, \ldots, x_{s1}, \ldots x_{sl})$, 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:

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

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

```
 365       366       480       437       408       466       481       409
1,0,0,0, 0,1,0,0, 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, 1,0,0,0, 0,0,0,0, 0,0,0,0, 1,0,0,0
0,0,1,0, 0,0,1,0, 0,0,0,0, 0,0,0,0, 0,1,0,0, 0,0,0,0, 0,0,0,0, 0,1,0,0
0,0,0,0, 0,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,0, 0,0,0,0, 0,0,1,0, 0,0,0,0
0,0,0,0, 0,0,0,0, 1,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 1,0,0,0, 0,0,0,0
0,0,1,0, 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
```

- 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,

$$R = (YearsWithCompany, Salary, Position, Department),$$

which specifies four attributes for employees of some company. Suppose that $YearsWithCompany$ ranges from 0 to 30, and $Salary$ ranges from \$20,000 to \$110,000. Also, let's assume that the domain of $Position$ is {Assistant, Associate, Manager, Senior Manager, Head} and the domain of $Department$ 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*:

YearsWithComapny Range	Discretized Value
0 — 3	newbie
4 — 10	average
11 — 20	veteran
20 — 30	dedicated

Salary Range	Discretized Value
20,000 — 39,999	low
40,000 — 64,999	medium-low
65,000 — 84,999	medium-high
85,000 — 110,000	high

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

[1] Agrawal R, Imielinski T, Swami AN. "Mining Association Rules between Sets of Items in Large Databases." *in Proc. ACM SIGMOD*. June 1993, 22(2):207-16.