Lecture 10

Finding Patterns – Market Basket Analysis Using Association Rules

What are frequent patterns?

Frequent patterns are patterns (e.g., itemsets, subsequences, or substructures) that appear frequently in a data set. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a *frequent itemset*. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (*frequent*) *sequential pattern*. A *substructure* can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*. Finding frequent patterns plays an essential role in finding associations, correlations, and many other interesting relationships among data.

With massive amounts of data continuously being collected and stored, many industries are becoming interested in discovering such patterns from their databases. The discovery of interesting correlation relationships among huge amounts of business transaction records can help in many business decision-making processes such as catalog design, cross-marketing, and customer shopping behavior analysis.

Market basket analysis

A typical example of frequent itemset discovery is **market basket analysis**. This process analyzes customer buying habits by finding associations between the different items that customers place in their "shopping baskets". The discovery of these associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread on the same trip to the supermarket? This information can lead to increased sales by helping retailers do selective marketing and plan their shelf space, plan marketing or advertising strategies, or design a new catalog. For instance, market basket analysis may help retailers design different store layouts. In one strategy, items that are frequently purchased together can be placed in proximity to further encourage the combined sale of such items. If customers who purchase computers also tend to buy antivirus software at the same time, then placing the hardware display close to the software display may help increase the sales of both items.

Market basket analysis

In an alternative strategy, placing hardware and software at opposite ends of the store may entice customers who purchase such items to pick up other items along the way. For instance, after deciding on an expensive computer, a customer may observe security systems for sale while heading toward the software display to purchase antivirus software, and may decide to purchase a home security system as well. Market basket analysis can also help retailers plan which items to put on sale at reduced prices. If customers tend to purchase computers and printers together, then having a sale on printers may encourage the sale of printers *as well as* computers.

Association rules

Customers' purchasing patterns can be described using association rules. For example, the information that customers who purchase computers also tend to buy antivirus software at the same time is represented by the following association rule:

computer → antivirus software

Other association rules:

beer → nuts, chips

Rule **support** and **confidence** are two measures of rule interestingness of association rules. They respectively reflect the usefulness and certainty of discovered rules. A support of 2% for rule *computer → antivirus software* means that 2% of all the transactions show that computer and antivirus software are purchased together. A confidence of 60% means that 60% of the customers who purchased a computer also bought the software. Typically, association rules are considered interesting if they satisfy both a **minimum support threshold** and a **minimum confidence threshold**.

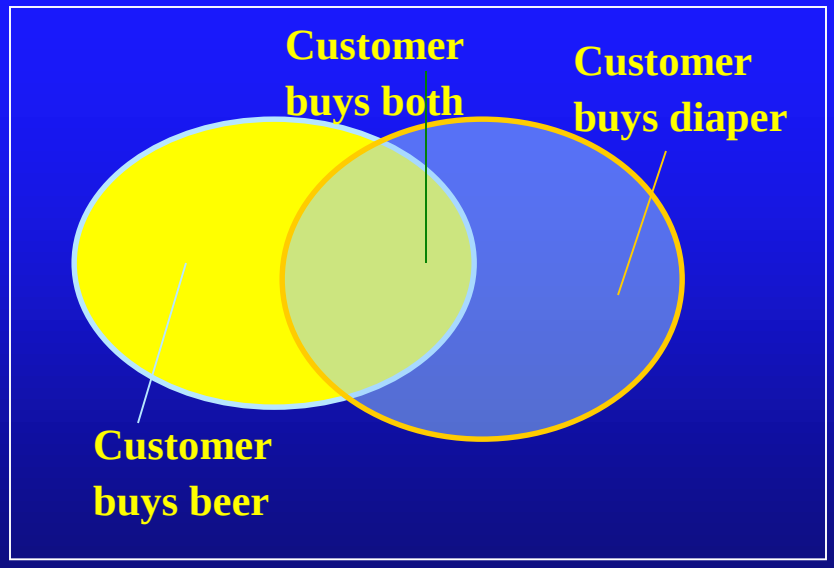
Transaction dataset

The set of all customers transactions is represented as a **transaction dataset**. It consists of separate transactions, each transactions being a set of items bought together in one transaction:

TID	Items bought
10	Beer, Nuts, Diaper
20	Beer, Coffee, Diaper
30	Beer, Diaper, Eggs
40	Nuts, Eggs, Milk
50	Nuts, Coffee, Diaper, Eggs, Milk

Basic Concepts: Frequent Patterns

Tid	Items bought
10	Beer, Nuts, Diaper
20	Beer, Coffee, Diaper
30	Beer, Diaper, Eggs
40	Nuts, Eggs, Milk
50	Nuts, Coffee, Diaper, Eggs, Milk



- itemset: A set of one or more items
- k-itemset X = $\{x_1, \ldots, x_k\}$
- *(absolute) support*, or, *support count* of X: Frequency or occurrence of an itemset X
- *(relative) support*, *s*, is the fraction of transactions that contains X
- An itemset X is *frequent* if X's support is no less than a *minsup* threshold
- Confidence:

$$(X \rightarrow Y) = \frac{count(X \cup Y)}{count(X)}$$

Basic Concepts: Frequent Patterns

For example, the absolute support for {beer, diaper} is 3 because the itemset occurs in 3 transactions. The relative support of {beer, diaper} is 3/5=60%, i.e., the itemset occurs in 60% in transactions.

The goal is to find all the rules $X \rightarrow Y$ with minimum support and confidence. Here, X and Y are itemsets. Another way to look at the support is to see it as the **probability** that a transaction contains $X \cup Y$.:

$$support(X \rightarrow Y) = P(X \cup Y)$$

Another way to look at the confidence*, is to see it as the* ***conditional probability*** that a transaction having X also contains *Y.*

$$confidence(X \rightarrow Y) = P(Y|X)$$

Let *minsup = 50%, minconf = 50%.* Some frequent itemsets are:

Beer:3, Nuts:3, Diaper:4, Eggs:3, {Beer, Diaper}:3, etc.

The rule beer $\rightarrow$ diapers has a support of 3 (60%) and a confidence of 100%. The association rule diaper $\rightarrow$ beer has a support of 60% and a confidence of 75%.

Lift

In the rule X → Y, X is called antecedent and Y is called consequent. A high value of confidence suggests a strong association rule. However, this can be deceptive because if the antecedent and/or the consequent have a high support, we can have a high value for confidence even when they are independent! A better measure to judge the strength of an association rule is to compare the confidence of the rule with the support of the consequent. This measure is called **lift**:

$$\text{lift}(X \rightarrow Y) = \frac{\text{confidence}(X \rightarrow Y)}{\text{support}(Y)}$$

In other words:

lift(X → Y) = $\frac{P(X \wedge Y)}{P(X)P(Y)}$, where $P(X \wedge Y)$ is the probability of X and Y occurring together. For example, lift(beer → diaper) = $\frac{confifence(beer \longrightarrow diaper)}{support(diaper)} = \frac{3/3}{4/5} = 1.25$

Lift greater than 1 indicates that the antecedent and the consequent are found together more often than one would expect by chance. The greater the lift, the stronger the association between X and Y.

The last formula shows that the lift is symmetric, i.e.

lift(X → Y) = lift(Y → X)

Apriori

Finding association rule mining can be viewed as a two-step process:

- **Find all frequent itemsets:** By definition, each of these itemsets will occur at least as frequently as a predetermined minimum support count.
- **Generate strong association rules from the frequent itemsets:** By definition, these rules must satisfy minimum support and minimum confidence.

One of the most popular algorithms for finding association rules is Apriori. It was proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses *prior knowledge* of frequent itemset properties. Apriori employs an iterative approach known as a *level-wise* search, where *k*-itemsets are used to explore k+1 itemsets. First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted by L_1. Next, L_1 is used to find L_2, the set of frequent 2-itemsets, which is used to find L_3, and so on, until no more frequent *k*-itemsets can be found. The finding of each L_k requires one full scan of the database.

Apriori

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the **Apriori property** is used to reduce the search space.

Apriori property: *All nonempty subsets of a frequent itemset must also be frequent.*

The Apriori property is based on the following observation. By definition, if an itemset *I* does not satisfy the minimum support threshold, then *I* is not frequent, that is, $P(I) < \min \sup$. If an item *A* is added to the itemset *I*, then the resulting itemset (i.e., $I \cup A$) cannot occur more frequently than *I*. Therefore, $I \cup A$ is not frequent either, that is, $P(I \cup A) < \min \sup$.

This property belongs to a special category of properties called **antimonotonicity** in the sense that *if a set cannot pass a test, all of its supersets will fail the same test as well*. It is called *antimonotonicity* because the property is monotonic in the context of failing a test.

To understand the algorithm, let us look at how L_{k-1} is used to find L_k for $k \geq 2$. A two-step process is followed, consisting of **join** and **prune** actions.

Apriori: the join step

The join step: To find L_k, a set of **candidate** k-itemsets is generated by joining L_{k-1} with itself (the join operation will be explained later). This set of candidates is denoted C_k. Let l_1 and l_2 be itemsets in L_{k-1}. The notation $l_i[j]$ refers to the j-th item in l_i. For efficient implementation, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the (k -1)-itemset, l_i , this means that the items are sorted such that $l_i[1] < l_i[2] < \ldots < l_i[k-1]$. The join, $L_{k-1} \bowtie 1\ L_{k-1}$, is performed, where members of L_{k-1} are joinable if their first k- 2 items are in common. That is, members l_1 and l_2 of L_{k-1} are joinable if:

$l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge l_1[k-2] = l_2[k-2] \wedge l_1[k-1] < l_2[k-1]$.

The condition $l_1[k-1] < l_2[k-1]$. simply ensures that no duplicates are generated. The resulting itemset formed by joining l_1 and l_2 is:

$l_1[1], l_1[2], : : : , l_1[k-2], l_1[k-1], l_2[k-1]$

Apriori: the prune step

The prune step: C_k is a superset of L_k, that is, its members may or may not be frequent, but all of the frequent k-itemsets are included in C_k. A database scan to determine the count of each candidate in C_k would result in the determination of L_k (i.e., all candidates having a count no less than the minimum support count are frequent by definition, and therefore belong to L_k). C_k, however, can be huge, and so this could involve heavy computation. To reduce the size of C_k, the Apriori property is used as follows. Any $(k-1)$-itemset that is not frequent cannot be a subset of a frequent k-itemset. Hence, if any $(k-1)$-subset of a candidate k-itemset is not in L_{k-1}, then the candidate cannot be frequent either and so can be removed from C_k. This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

Apriori: example

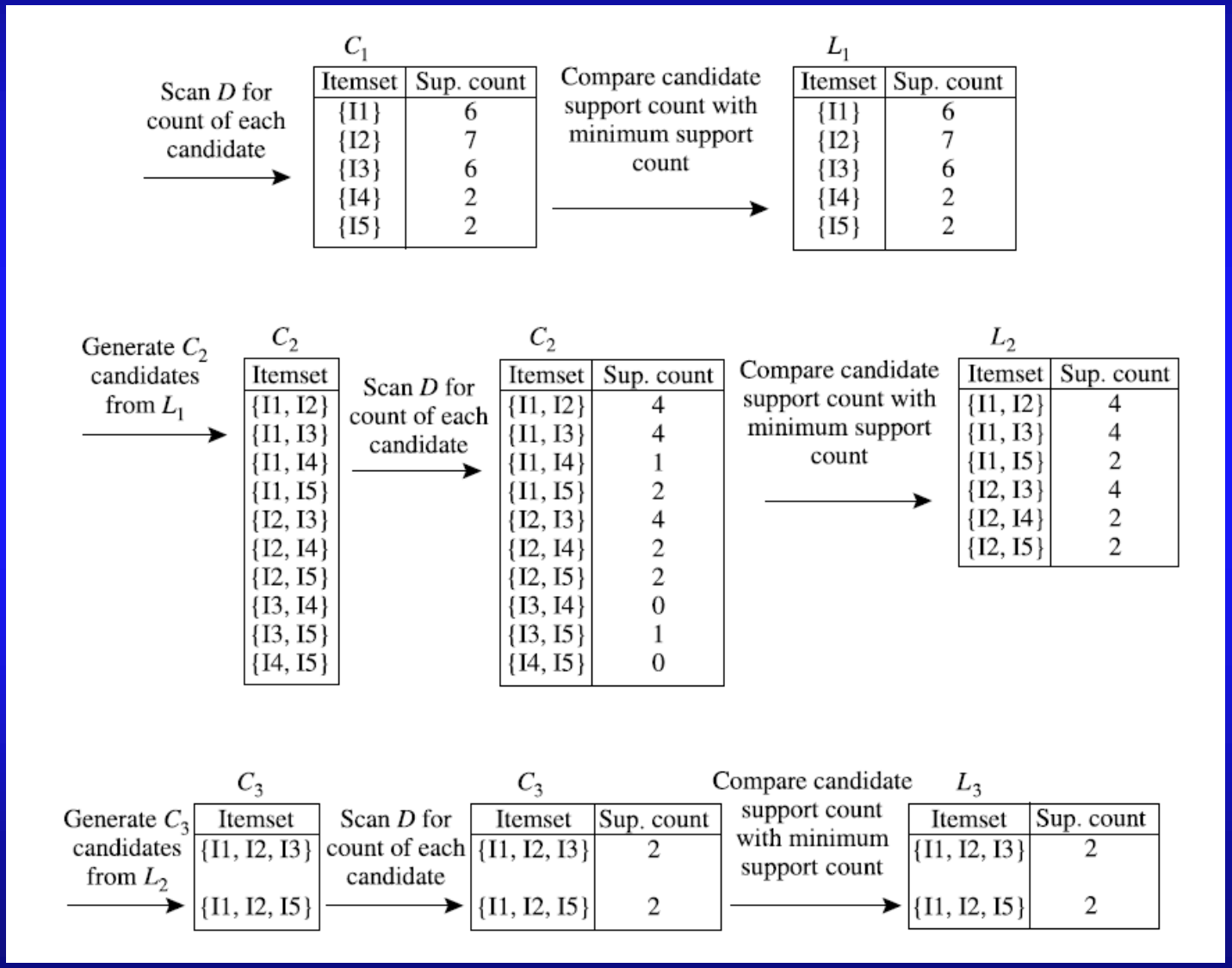
Let's consider the following transactional database:

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, C_1. The algorithm simply scans all of the transactions to count the number of occurrences of each item.

2. Suppose that the minimum support count required is 2, that is, *min sup* = 2. (Here, we are referring to *absolute* support because we are using a support count. The corresponding relative support is 2/9 = 22%.) The set of frequent 1-itemsets, L_1, can then be determined. It consists of the candidate 1-itemsets satisfying minimum support. In our example, all of the candidates in C_1 satisfy minimum support

Apriori: example



Apriori: example

3. To discover the set of frequent 2-itemsets, L_2, the algorithm uses the join $L_1 \bowtie L_1$ to generate a candidate set of 2-itemsets, C_2. C_2 consists of $\binom{|L_1|}{2}$ 2-itemsets. Note that no candidates are removed from C_2 during the prune step.

4. Next, the transactions are scanned and the support count of each candidate itemset in C_2 is accumulated.

5. The set of frequent 2-itemsets, L_2, is then determined, consisting of those candidate 2-itemsets in C_2 having minimum support.

6. The next step includes the generation of the set of the candidate 3-itemsets, C_3. From the join step, we first get $C_3 = L_2 \bowtie L_2 = \{$(I1, I2, I3), (I1, I2, I5), (I1, I3, I5), (I2, I3, I4), (I2, I3, I5), (I2, I4, I5)$\}$. Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that the four latter candidates cannot possibly be frequent. We therefore remove them from C_3, thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of the databse to determine L_3. Note that when given a candidate k-itemset, we only need to check if its $(k – 1)$-subsets are frequent since the Apriori algorithm uses a level-wise search strategy.

Apriori: example

7. The transactions in the database are scanned to determine L_3, consisting of those candidate 3-itemsets in C_3 having minimum support.

8. The algorithm uses $L_3 \bowtie L_3$ to generate a candidate set of 4-itemsets, C_4. Although the join results in {(I1, I2, I3, I5)}, itemset (I1, I2, I3, I5) is pruned because its subset (I2, I3, I5) is not frequent. Thus, $C_4 = \emptyset$, and the algorithm terminates, having found all of the frequent itemsets.

Generation and pruning of candidate 3-itemsets, *C*3, from *L*2

Join: $C_3 = L_2 \bowtie L_2$ ={(I1, I2), (I1, I3), (I1, I5), (I2, I3), (I2, I4), (I2, I5)} ⋈ {(I1, I2), (I1, I3), (I1, I5), (I2, I3), (I2, I4), (I2, I5)} = {(1, I2, I3), (I1, I2, I5), (I1, I3, I5), (I2, I3, I4), (I2, I3, I5), (I2, I4, I5)}.

Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?

The 2-item subsets of (I1, I2, I3) are (I1, I2), (I1, I3), and (I2, I3). All 2-item subsets of (I1, I2, I3) are members of L_2. Therefore, keep (I1, I2, I3) in C_3.

The 2-item subsets of (I1, I2, I5) are (I1, I2), (I1, I5), and (I2, I5). All 2-item subsets of (I1, I2, I5) are members of L_2. Therefore, keep (I1, I2, I5) in C_3.

The 2-item subsets of (I1, I3, I5) are (I1, I3), (I1, I5), and (I3, I5). (I3, I5) is not a member of L_2, and so it is not frequent. Therefore, remove (I1, I3, I5) from C_3.

Generation and pruning of candidate 3-itemsets, *C*3, from *L*2

The 2-item subsets of (I2, I3, I4) are (I2, I3), (I2, I4), and (I3, I4). (I3, I4) is not a member of L_2, and so it is not frequent. Therefore, remove (I2, I3, I4) from C_3.

The 2-item subsets of (I2, I3, I5) are (I2, I3), (I2, I5), and (I3, I5). (I3, I5) is not a member of L_2, and so it is not frequent. Therefore, remove (I2, I3, I5) from C_3.

The 2-item subsets of (I2, I4, I5) are (I2, I4), (I2, I5), and (I4, I5). (I4, I5) is not a member of L_2, and so it is not frequent. Therefore, remove (I2, I4, I5) from C_3.

Therefore, C_3 = {(I1, I2, I3), (I1, I2, I5)} after pruning.

Generating Association Rules from Frequent Itemsets

Once the frequent itemsets from transactions in a database *D* have been found, it is straightforward to generate strong association rules from them (where *strong* association rules satisfy both minimum support and minimum confidence). Association rules can be generated as follows:

- For each frequent itemset *l*, generate all nonempty subsets of *l*.
- For every nonempty subset *s* of *l*, output the rule "*s* → *l* - *s*" if

$$\frac{support\_count(l)}{support\_count(s)} > \min conf$$

where *min conf* is the minimum confidence threshold.

Because the rules are generated from frequent itemsets, each one automatically satisfies the minimum support. Frequent itemsets can be stored ahead of time in hash tables along with their counts so that they can be accessed quickly.

Example

Let's generate all association rules from the frequent itemset (I1, I2, I5). The nonempty subsets of the itemset are (I1, I2), (I1, I5), (I2, I5), (I1), (I2), and (I5). The resulting association rules are as shown below, each listed with its confidence:

(I1, I2) → I5, *confidence* = 2/4 = 50%

(I1, I5) → I2, *confidence* = 2/2 = 100%

(I2, I5) → I1, *confidence* = 2/2 = 100%

I1 → (I2, I5), *confidence* = 2/6 = 33%

I2 → (I1, I5), *confidence* = 2/7 = 29%

I5 → (I1, I2), *confidence* = 2/2 = 100%

Identifying frequently purchased groceries

As an example, we will perform a market basket analysis of transactional data from a grocery store. We will use the purchase data collected from one month of operation at a real-world grocery store. The data contains 9,835 transactions or about 327 transactions per day. You can download *groceries.csv* from Blackboard and install it in your working directory.

If you open the file with Excel with any txt editor, you will see that it looks like:

citrus fruit	semi-finished bread	margarine	ready soups	
tropical fruit	yogurt	coffee		
whole milk				
pip fruit	yogurt	cream cheese	meat spreads	
other vegetables	whole milk	condensed milk	long life bakery product	
whole milk	butter	yogurt	rice	abrasive cleaner
rolls/buns				
other vegetables	UHT-milk	rolls/buns	bottled beer	liquor (appetizer)
potted plants				
whole milk	cereals			

Reading data

The first question is “How to read the data file?” We can use the standard *read.csv()* function:

```
> data <- read.csv("groceries.csv")
> head(data)
      citrus.fruit semi.finished.bread     margarine                ready.soups
1   tropical fruit              yogurt        coffee
2       whole milk
3        pip fruit              yogurt   cream cheese            meat spreads
4 other vegetables          whole milk condensed milk long life bakery product
5       whole milk              butter        yogurt                      rice
6 abrasive cleaner
```

This does not work because R creates four columns to store the data. However, this could be a problem because the grocery purchases can contain more than four items; in the four column design such transactions will be broken across multiple rows in the matrix. R creates four variables because the first line has exactly four comma-separated values.

Creating a sparse matrix

The solution is to create a **sparse matrix**. It will have a column (that is, feature) for every item that could possibly appear in a transaction. Since there are 169 different items in our grocery store data, our sparse matrix will contain 169 columns. Most of the entries in the sparse matrix will be zeros. Since sparse matrixes are optimized for storing zeros (they actually do not store the full matrix in memory; instead they store the non zero values and their indexes), sparse matrixes are more efficient for storing transactional data than standard data frames. We can create sparse matrixes and association rules using the *arules* package:

```
> install.packages("arules")
> library(arules)
```

Then, we read the *groceries.csv* data into a sparse matrix named groceries:

```
> groceries <- read.transactions("groceries.csv", sep = ",")
```

The *summary()* function shows that there are 9835 transactions and a total of 169 items.

Creating a sparse matrix

```
> summary(groceries)
transactions as itemMatrix in sparse format with
 9835 rows (elements/itemsets/transactions) and
 169 columns (items) and a density of 0.02609146

most frequent items:
      whole milk other vegetables       rolls/buns             soda
            2513             1903             1809             1715
          yogurt          (Other)
            1372            34055

element (itemset/transaction) length distribution:
sizes
   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16
2159 1643 1299 1005  855  645  545  438  350  246  182  117   78   77   55   46
  17   18   19   20   21   22   23   24   26   27   28   29   32
  29   14   14    9   11    4    6    1    1    1    1    3    1

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   2.000   3.000   4.409   6.000  32.000
```

Creating a sparse matrix

A cell in the matrix is 1 if the item was purchased for the corresponding transaction, or 0 otherwise. The **density (**the proportion of nonzero matrix cells) of the matrix is 0.02609146, i.e., 2.6 %. The *summary()* function also lists the most common items and their counts. It also shows the counts for transactions of different sizes. For example, 2159 transactions included only one item. The first quartile and median purchase sizes are two and three items, respectively, showing that 25 percent of the transactions contained two or fewer items and the transactions were split in half between those with less than three items and those with more.

The contents of the first five transactions could be extracted using the *inspect()* function:

```
> inspect(groceries[1:5])
  items
1 {citrus fruit,
   margarine,
   ready soups,
   semi-finished bread}
```

Creating a sparse matrix

```
2 {coffee,
   tropical fruit,
   yogurt}
3 {whole milk}
4 {cream cheese,
   meat spreads,
   pip fruit,
   yogurt}
5 {condensed milk,
   long life bakery product,
   other vegetables,
   whole milk}
```

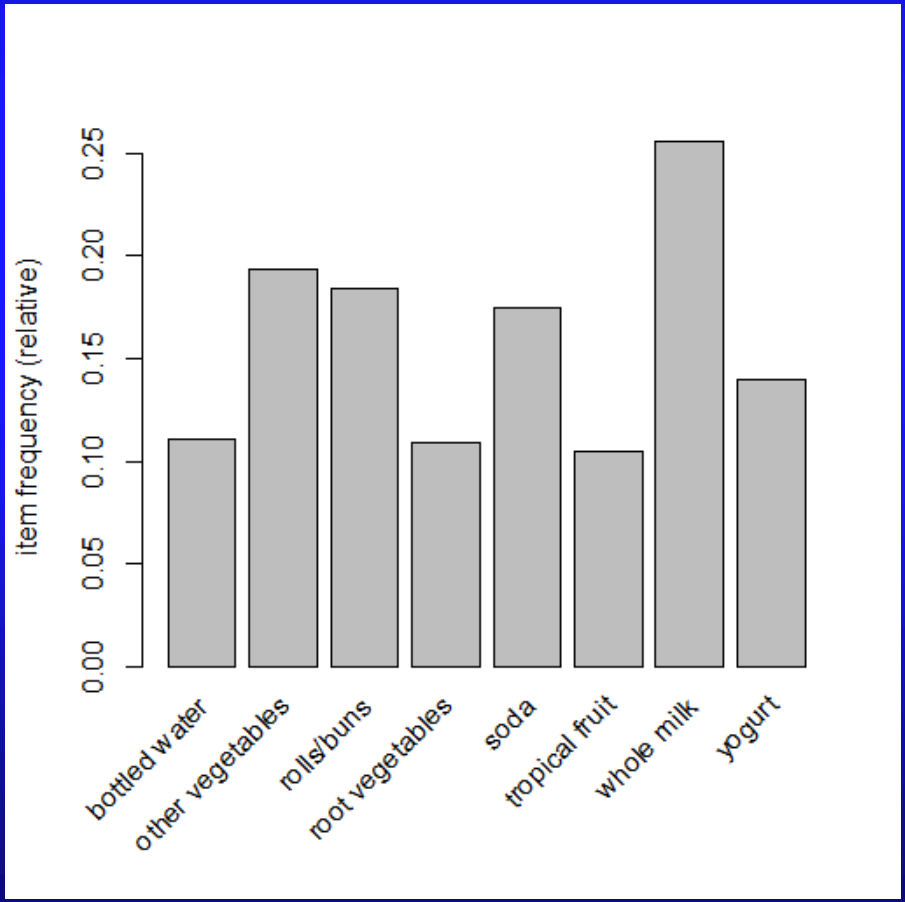
The *itemFrequency()* function shows the proportion of transactions that contain the item. For example, we can see the proportions of the transactions that contain the first three items:

```
> itemFrequency(groceries[, 1:3])
abrasive cleaner artif. sweetener   baby cosmetics
    0.0035587189     0.0032536858     0.0006100661
```

Item frequency plots

You can create a bar chart with the single items that meet the minimum support using the *itemFrequencyPlot()* function. The minimum support parameter is set to 10%:

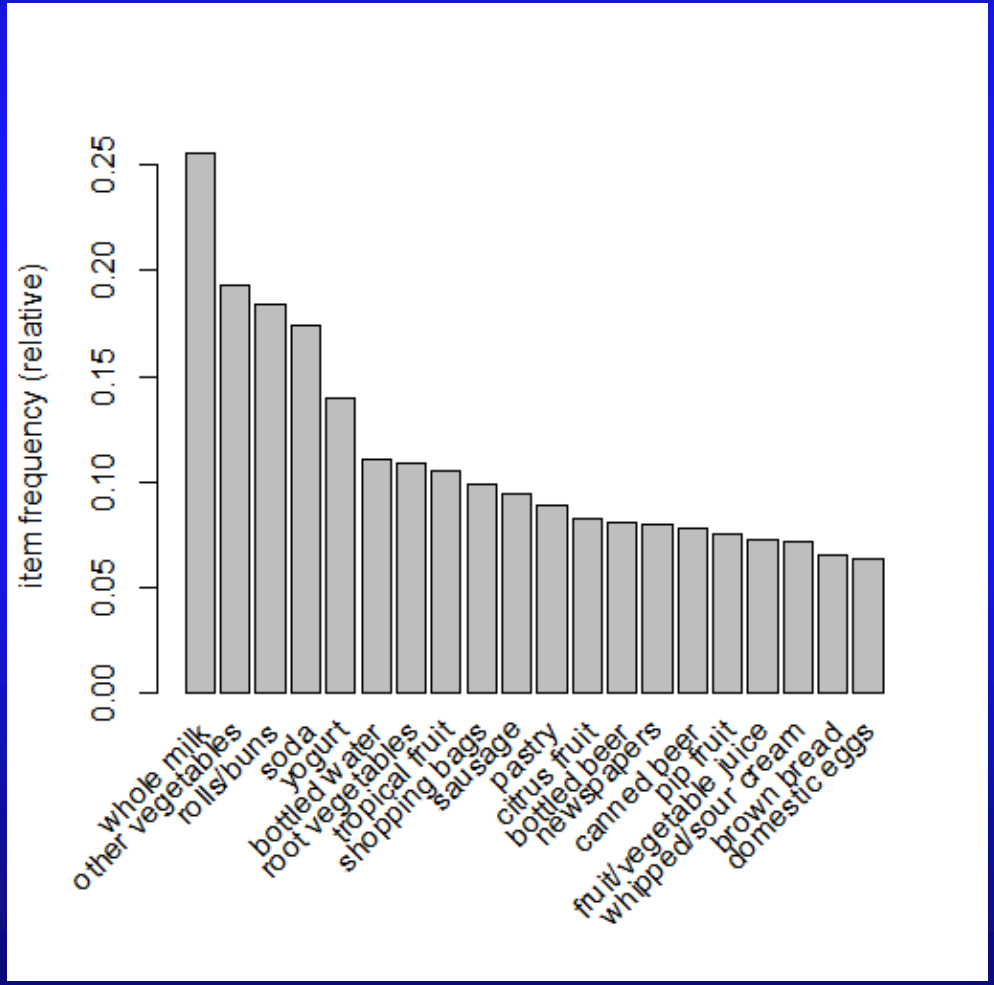
```
> itemFrequencyPlot(groceries, support = 0.1)
```



Item frequency plots

The following function call plots the top 20 most frequent items:

```
> itemFrequencyPlot(groceries, topN = 20)
```

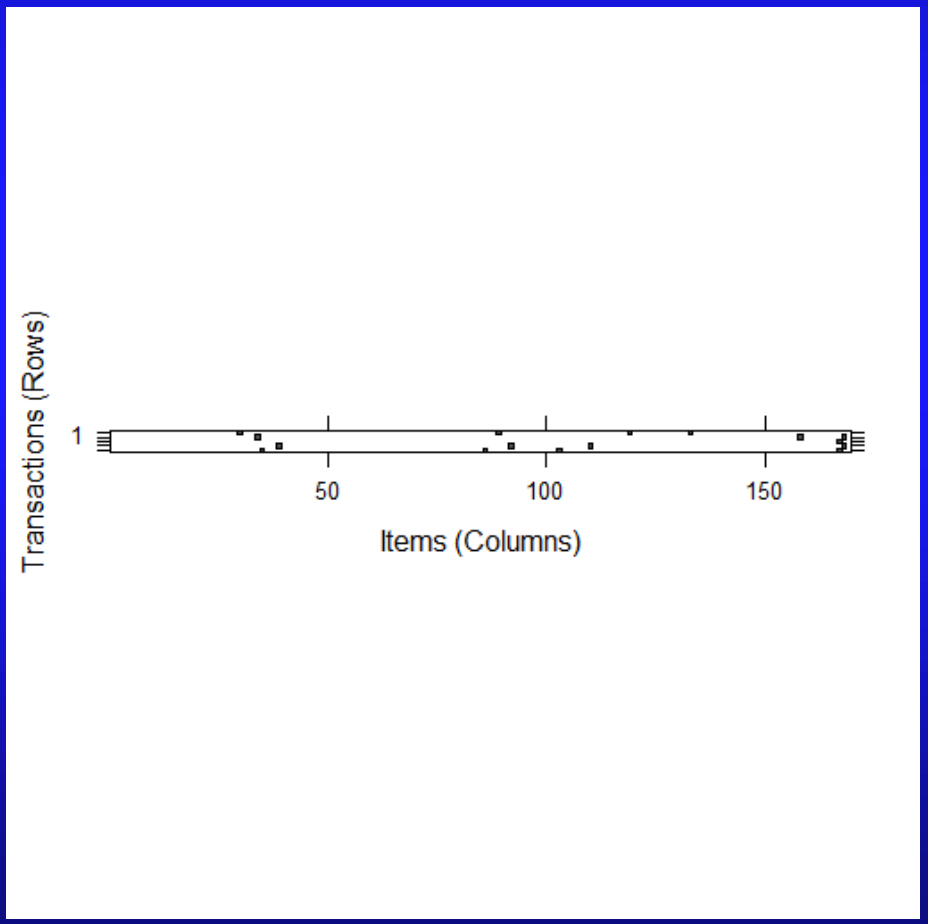


Plotting the sparse matrix

The *image()* function can be used to plot a segment of the matrix:

```
> image(groceries[1:5])
```

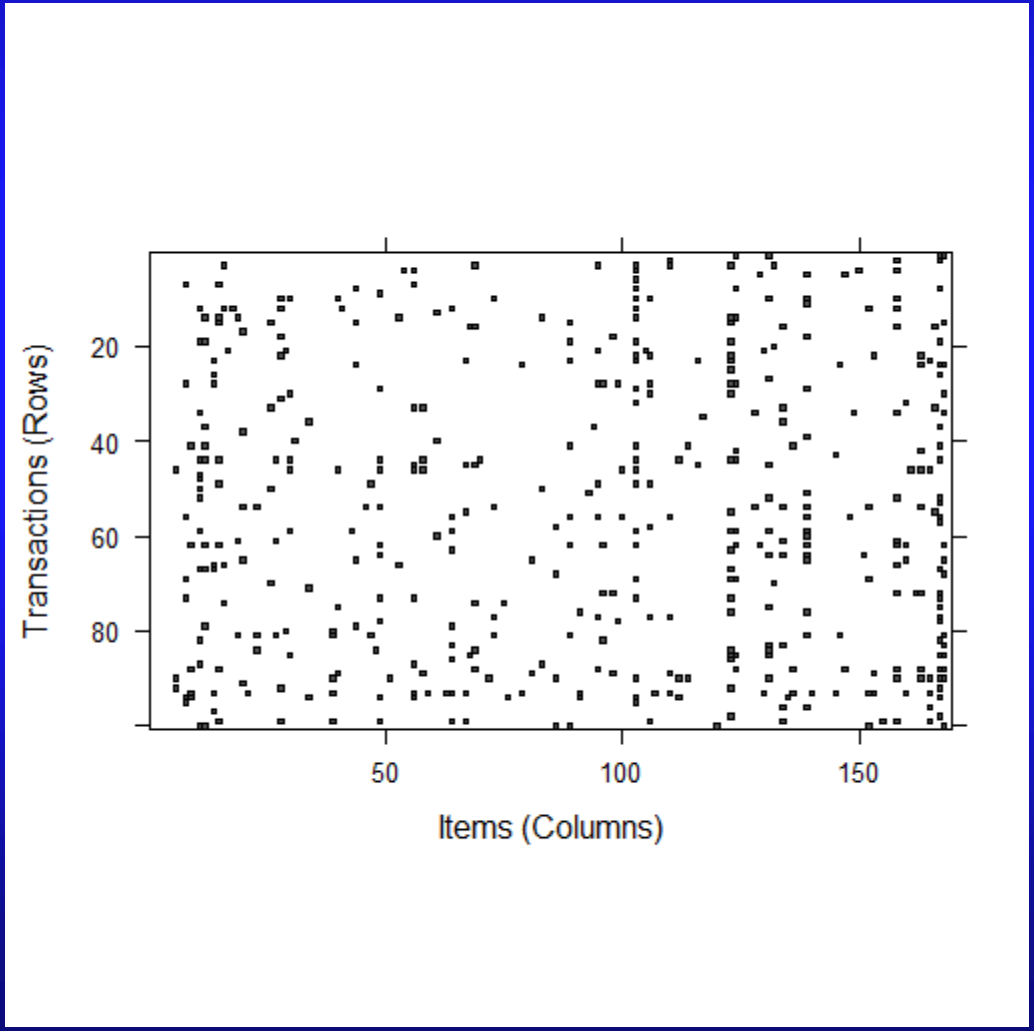
The matrix has 5 rows and 169 columns, corresponding to 5 transactions and 169 possible items. Cells in the matrix are filled with black for transactions (rows) where the item (column) was purchased.



Plotting the sparse matrix

The following command creates a matrix diagram with 100 randomly chosen rows and 169 columns:

```
> image(sample(groceries, 100))
```



Training a model on the data

The *apriori()* functions from the *arules* packages implements the Apriori algorithm. The minimum support and the minimum confidence must be fine tuned in order to produce meaningful results. If they are set too high, the algorithm might return no rules or rules that are too generic. On the other hand, if the minimum support and the minimum confidence are too low, the algorithm could produce spurious rules.

Association rule syntax

using the `apriori()` function in the `arules` package

Finding association rules:

```
myrules <- apriori(data = mydata, parameter =
  list(support = 0.1, confidence = 0.8, minlen = 1))
```

- `data` is a sparse item matrix holding transactional data
- `support` specifies the minimum required rule support
- `confidence` specifies the minimum required rule confidence
- `minlen` specifies the minimum required rule items

The function will return a rules object storing all rules that meet the minimum criteria.

Examining association rules:

```
inspect(myrules)
```

- `myrules` is a set of association rules from the `apriori()` function

This will output the association rules to the screen. Vector operators can be used on myrules to choose a specific rule or rules to view.

Example:

```
groceryrules <- apriori(groceries, parameter =
 list(support = 0.01, confidence = 0.25, minlen = 2))
inspect(groceryrules[1:3])
```

Training a model on the data

For example, if we run *apriori()* with the default settings, support = 0.1 and confidence = 0.8, we will end up with a set of zero rules:

```
 > apriori(groceries)
Apriori

Parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen target
        0.8    0.1    1 none FALSE            TRUE     0.1      1     10  rules
   ext
 FALSE

Algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

Absolute minimum support count: 983

set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].
sorting and recoding items ... [8 item(s)] done [0.00s].
creating transaction tree ... done [0.02s].
checking subsets of size 1 2 done [0.00s].
writing ... [0 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
set of 0 rules
```

Training a model on the data

This should not come as a surprise because a support of 0.1 is too high for our data. It corresponds to 0.1**9,385 = 938.5* transactions and there are only eight items that appeared in 938 transactions.

We will start with *support* of 0.006, *confidence* of 0.25, and *minlen* of .2. *Minlen* specifies the minimum required rule items. In other words, we will eliminate rules that contain fewer than two items.

```
> groceryrules <- apriori(groceries, parameter =
+ list(support = 0.006, confidence = 0.25, minlen = 2))
```

groceryrules is a rule object that contains 463 rules:

```
> groceryrules
set of 463 rules
To get more information about the rule object, you can use:
> summary(groceryrules)
```

The output is shown on the next slide.

Training a model on the data

```
set of 463 rules

rule length distribution (lhs + rhs):sizes
  2   3   4
150 297  16

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.000   3.000   2.711   3.000   4.000

summary of quality measures:
    support           confidence          lift
 Min.   :0.006101   Min.   :0.2500   Min.   :0.9932
 1st Qu.:0.007117   1st Qu.:0.2971   1st Qu.:1.6229
 Median :0.008744   Median :0.3554   Median :1.9332
 Mean   :0.011539   Mean   :0.3786   Mean   :2.0351
 3rd Qu.:0.012303   3rd Qu.:0.4495   3rd Qu.:2.3565
 Max.   :0.074835   Max.   :0.6600   Max.   :3.9565

mining info:
      data ntransactions support confidence
 groceries          9835   0.006       0.25
```

Evaluating model performance

The summary shows that 150 rules have only two items, 297 rules have three items, and 16 rules have four. The median rule length is 3 and the average rule length is 2.7. The size of the rule is calculated as the total of both the left-hand side (lhs) and right-hand side (rhs) of the rule. This means that a rule like *{pasta} → {whole milk}* is two items. The support range is from 0.006 to 0.7, which means that our minimum support of 0.006 is not too high.

One can extract specific rules using the *inspect()* function:

```
> inspect(groceryrules[1:5])
  lhs                    rhs                    support     confidence lift
1 {potted plants}     => {whole milk}           0.006914082 0.4000000  1.565460
2 {pasta}             => {whole milk}           0.006100661 0.4054054  1.586614
3 {herbs}             => {root vegetables}      0.007015760 0.4312500  3.956477
4 {herbs}             => {other vegetables}     0.007727504 0.4750000  2.454874
5 {herbs}             => {whole milk}           0.007727504 0.4750000  1.858983
```

Rule types

Association rules can be divided into three categories:

- Actionable
- Trivial
- Inexplicable

An **actionable** rule is one that provides a clear and useful insight

A **trivial rule** provides obvious information. For example, *{diapers}* → *{formula}* is a trivial rule.

A rules is **inexplicable** if the connection between the items is so unclear that figuring out how to use the information is impossible or nearly impossible. The rule may simply be a random pattern in the data. For example, *{pickles}* → *{chocolate ice cream}.*

Sorting association rules

In order to explore a set of association rules, one can use different functions. The *sort()* function can sort association rules by support, confidence, or lift. For example:

```
> groceryrules_sorted_lift<-sort(groceryrules, by = "lift")
> inspect(groceryrules_sorted_lift[1:5])
  lhs                     rhs                      support confidence     lift
1 {herbs}              => {root vegetables}    0.007015760  0.4312500 3.956477
2 {berries}            => {whipped/sour cream} 0.009049314  0.2721713 3.796886
3 {other vegetables,
   tropical fruit,
   whole milk}         => {root vegetables}    0.007015760  0.4107143 3.768074
4 {beef,
   other vegetables}   => {root vegetables}    0.007930859  0.4020619 3.688692
5 {other vegetables,
   tropical fruit}     => {pip fruit}          0.009456024  0.2634561 3.482649
```

The first rule, for example, shows that customers who buy herbs are almost four times more likely to buy root vegetables than the typical customer.

Taking subsets of association rules

Subsets of association rules can be extracted suing the *subset()* function. The matching criterion can include items, rhs or lhs. The operator *%in%* means that at least one list item must be in the rule. A complete match is specified by *%ain%* (i.e., all list items must be present in the rule). For example,

items %in% c(“whole milk”, "yogurt”)

matches the rules that have either “whole milk” or “yogurt”

items %ain% c(“whole milk”, "yogurt”).

matches the rules that have both “whole milk” and “yogurt”

The following command extracts the rules that include either “chicken” or “pork”, or both of them:

```
> meat_rules<-subset(groceryrules, items %in%
+ c("pork","chicken"))
```

Taking subsets of association rules

%pin% can be used to perform partial matches. For example, one can extract transactions the right hand sides of which contain “vegetables”, including “root vegetables and “other vegetables”:

```
> vegi<-subset(groceryrules, rhs %pin% "vegetables")
> inspect(vegi[1:5])
    lhs                     rhs                   support     confidence lift
3   {herbs}              => {root vegetables}   0.007015760 0.4312500  3.956477
4   {herbs}              => {other vegetables} 0.007727504 0.4750000  2.454874
9   {detergent}          => {other vegetables} 0.006405694 0.3333333  1.722719
11 {pickled vegetables} => {other vegetables} 0.006405694 0.3579545  1.849965
13 {baking powder}       => {other vegetables} 0.007320793 0.4137931  2.138547
```

Taking subsets of association rules

Logical operators such as and (&), or (|), and not (!) can be used to specify matching criterion. For example:

```
> soda<-subset(groceryrules, rhs %in% "soda" & lift>1.6)
> inspect(soda)
    lhs                           rhs    support     confidence lift
40  {candy}                    => {soda} 0.008642603 0.2891156  1.657990
332 {rolls/buns,shopping bags} => {soda} 0.006304016 0.3229167  1.851828
343 {rolls/buns,sausage}       => {soda} 0.009659380 0.3156146  1.809953
367 {bottled water,yogurt}     => {soda} 0.007422471 0.3230088  1.852357
369 {bottled water,rolls/buns} => {soda} 0.006812405 0.2815126  1.614389
```

Saving association rules to a file or data frame

Association rules can be saved into a CSV file with the *write()* function:

```
> write(groceryrules, file = "groceryrules.csv", sep = ",",
quote = TRUE, row.names = FALSE)
```

Association rules can be converted to a data frame using the *as()* function:

```
> groceryrules_df <- as(groceryrules, "data.frame")
> str(groceryrules_df)
'data.frame':   463 obs. of  4 variables:
 $ rules     : Factor w/ 463 levels "{baking powder} => {other
vegetables}",..: 340 302 207 206 208 341 402 21 139 140 ...
 $ support   : num  0.00691 0.0061 0.00702 0.00773 0.00773 ...
 $ confidence: num  0.4 0.405 0.431 0.475 0.475 ...
 $ lift      : num  1.57 1.59 3.96 2.45 1.86 ...
```

The same result can be obtained by reading the csv file:

```
> rules <- read.csv("groceryrules.csv",header=TRUE)
```