VirginiaTech

CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #19: Data Mining and Warehousing

Just FYI, and not for exam!

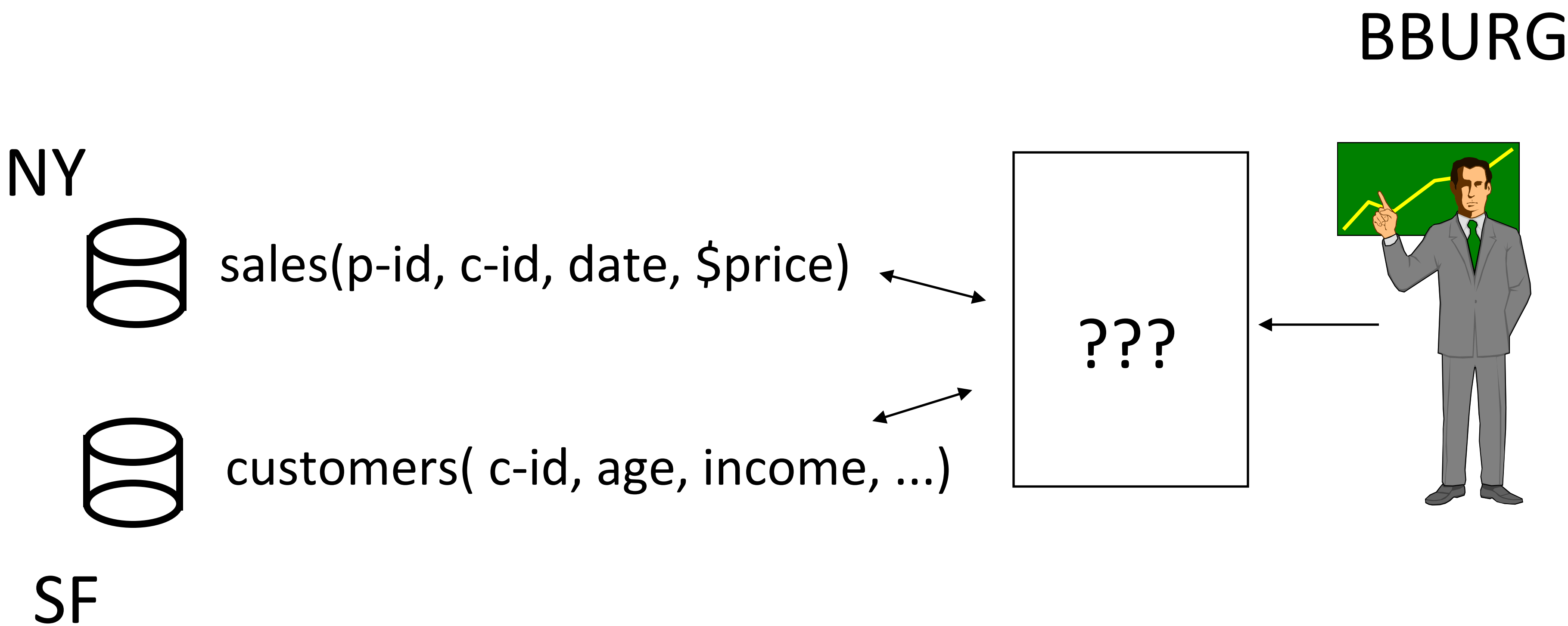
Overview

- Traditional database systems are tuned to many, small, simple queries.
- New applications use fewer, more time-consuming, *analytic* queries.
- New architectures have been developed to handle analytic queries efficiently.

Problem

Given: multiple data sources

Find: patterns (classifiers, rules, clusters, outliers...)



Data Ware-housing

First step: collect the data, in a single place (= Data Warehouse)

How?

How often?

How about discrepancies / non-homegeneities?

Data Ware-housing

First step: collect the data, in a single place (= Data Warehouse)

How? A: Triggers/Materialized views

How often? A: [Art!]

How about discrepancies / non-homegeneities? A: Wrappers/Mediators

Data Ware-housing

Step 2: collect counts. (DataCubes/OLAP)

The Data Warehouse

- The most common form of data integration.
  - Copy sources into a single DB (*warehouse*) and try to keep it up-to-date.
  - Usual method: periodic reconstruction of the warehouse, perhaps overnight.
  - Frequently essential for analytic queries.

OLTP

- Most database operations involve *On-Line Transaction Processing* (OTLP).
  - Short, simple, frequent queries and/or modifications, each involving a small number of tuples.
  - Examples: Answering queries from a Web interface, sales at cash registers, selling airline tickets.

OLAP

- *On-Line Application Processing* (OLAP, or "analytic") queries are, typically:
  - Few, but complex queries --- may run for hours.
  - Queries do not depend on having an absolutely up-to-date database.

OLAP Examples

1. Amazon analyzes purchases by its customers to come up with an individual screen with products of likely interest to the customer.
2. Analysts at Wal-Mart look for items with increasing sales in some region.
   - Use empty trucks to move merchandise between stores.

Common Architecture

- Databases at store branches handle OLTP.
- Local store databases copied to a central warehouse overnight.
- Analysts use the warehouse for OLAP.

Star Schemas

- A *star schema* is a common organization for data at a warehouse. It consists of:
  1. *Fact table* : a very large accumulation of facts such as sales.
     1. Often "insert-only."
  2. *Dimension tables* : smaller, generally static information about the entities involved in the facts.

Example: Star Schema

- Suppose we want to record in a warehouse information about every beer sale: the bar, the brand of beer, the drinker who bought the beer, the day, the time, and the price charged.
- The fact table is a relation:

Sales(bar, beer, drinker, day, time, price)

Example -- Continued

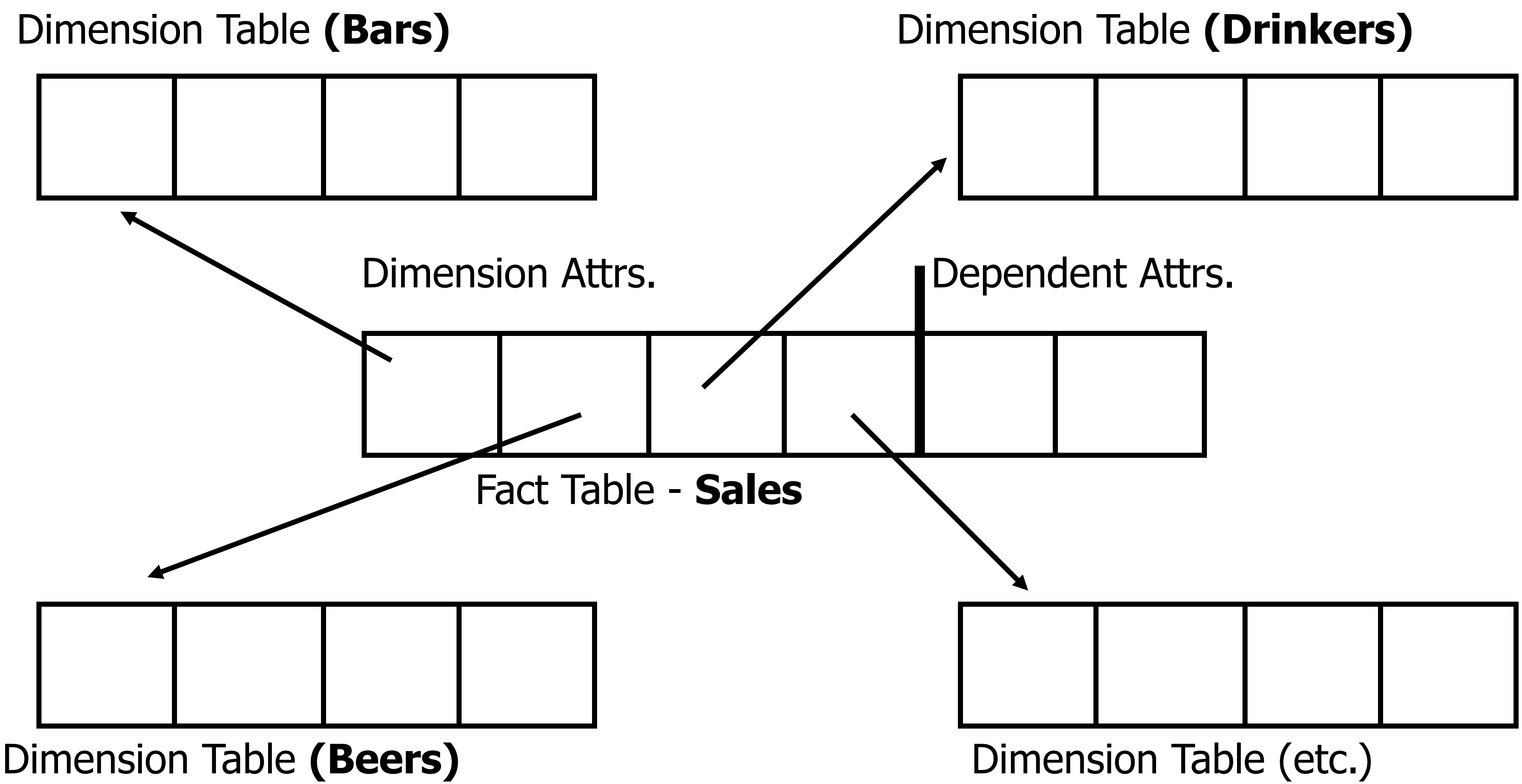
- The dimension tables include information about the bar, beer, and drinker "dimensions":

  Bars(bar, addr, license)

  Beers(beer, manf)

  Drinkers(drinker, addr, phone)

Visualization – Star Schema



Dimensions and Dependent Attributes

- Two classes of fact-table attributes:
  1. *Dimension attributes* : the key of a dimension table.
  2. *Dependent attributes* : a value determined by the dimension attributes of the tuple.

Example: Dependent Attribute

- price is the dependent attribute of our example Sales relation.
- It is determined by the combination of dimension attributes: bar, beer, drinker, and the time (combination of day and time-of-day attributes).

Approaches to Building Warehouses

1. *ROLAP* = "relational OLAP": Tune a relational DBMS to support star schemas.
2. *MOLAP* = "multidimensional OLAP": Use a specialized DBMS with a model such as the "data cube."

ROLAP Techniques

1. *Bitmap indexes* : For each key value of a dimension table (e.g., each beer for relation Beers) create a bit-vector telling which tuples of the fact table have that value.
2. *Materialized views* : Store the answers to several useful queries (views) in the warehouse itself.

Typical OLAP Queries

- Often, OLAP queries begin with a "star join": the natural join of the fact table with all or most of the dimension tables.
- Example:

```
SELECT *
FROM Sales, Bars, Beers, Drinkers
WHERE Sales.bar = Bars.bar AND
  Sales.beer = Beers.beer AND
  Sales.drinker = Drinkers.drinker;
```

Typical OLAP Queries --- (2)

- The typical OLAP query will:
  1. Start with a star join.
  2. Select for interesting tuples, based on dimension data.
  3. Group by one or more dimensions.
  4. Aggregate certain attributes of the result.

Example: OLAP Query

- For each bar in Blacksburg, find the total sale of each beer manufactured by Anheuser-Busch.
- Filter: addr = "Blacksburg" and manf = "Anheuser-Busch".
- Grouping: by bar and beer.
- Aggregation: Sum of price.

Example: In SQL

```
SELECT bar, beer, SUM(price)
FROM Sales NATURAL JOIN Bars
  NATURAL JOIN Beers
WHERE addr = 'Blacksburg' AND
  manf = 'Anheuser-Busch'
GROUP BY bar, beer;
```

Using Materialized Views

- A direct execution of this query from Sales and the dimension tables could take too long.
- If we create a materialized view that contains enough information, we may be able to answer our query much faster.

Example: Materialized View

- Which views could help with our query?
- Key issues:
  1. It must join Sales, Bars, and Beers, at least.
  2. It must group by at least bar and beer.
  3. It must not select out Blacksburg bars or Anheuser-Busch beers.
  4. It must not project out addr or manf.

Example --- Continued

- Here is a materialized view that could help:

```
CREATE VIEW BABMS(bar, addr,
   beer, manf, sales) AS
SELECT bar, addr, beer, manf,
   SUM(price) sales
FROM Sales NATURAL JOIN Bars
   NATURAL JOIN Beers
GROUP BY bar, addr, beer, manf;
```

Since bar -> addr and beer -> manf, there is no real grouping. We need addr and manf in the SELECT.

Example --- Concluded

- Here's our query using the materialized view BABMS:

```
SELECT bar, beer, sales
FROM BABMS
WHERE addr = 'Blacksburg' AND
   manf = 'Anheuser-Busch';
```

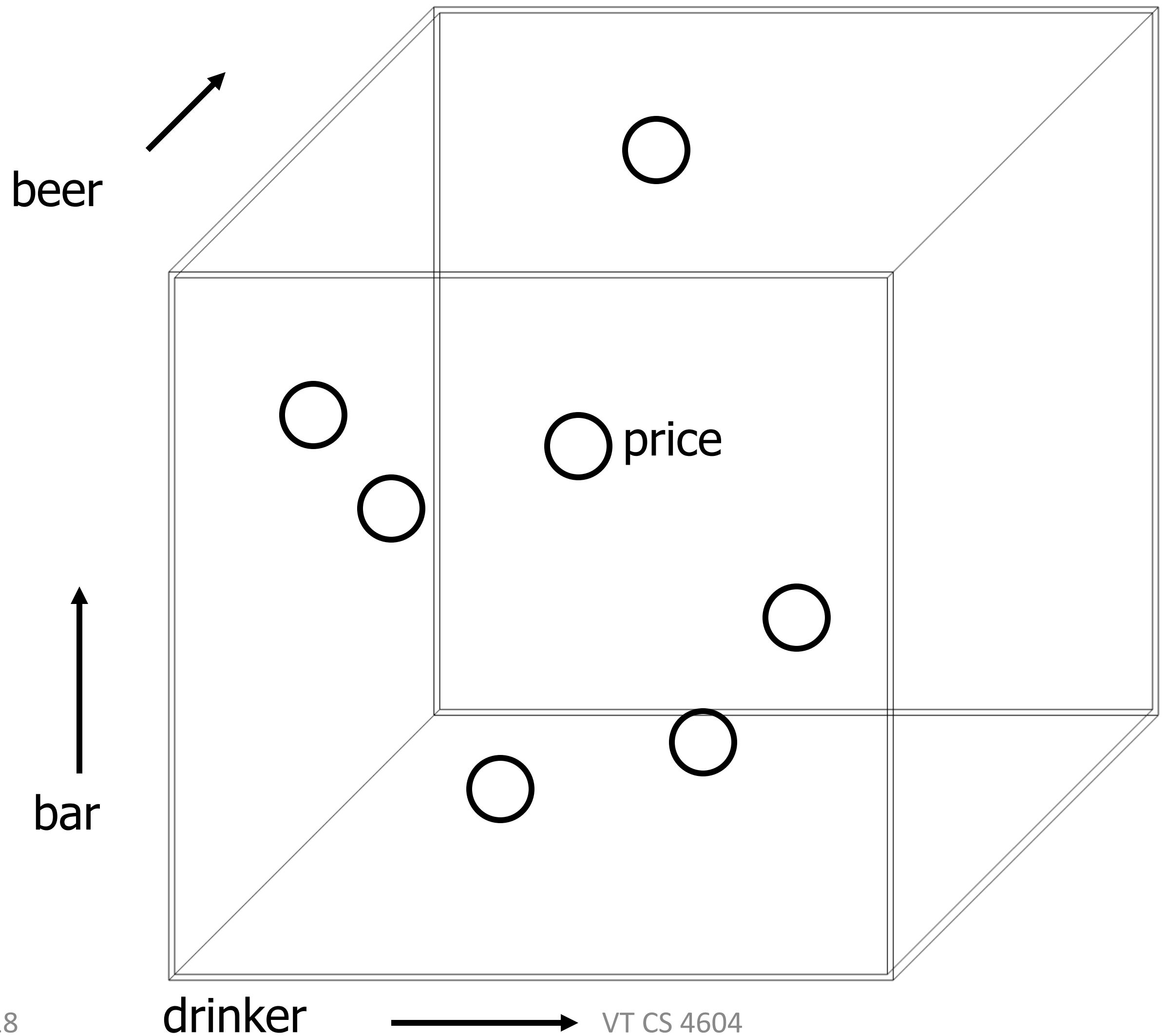
MOLAP and Data Cubes

- Keys of dimension tables are the dimensions of a hypercube.
- Example:

  Sales(bar, beer, drinker, time, price)

  – for the Sales data, the four dimensions are bar, beer, drinker, and time.
- Dependent attributes (e.g., price) appear at the points of the cube.

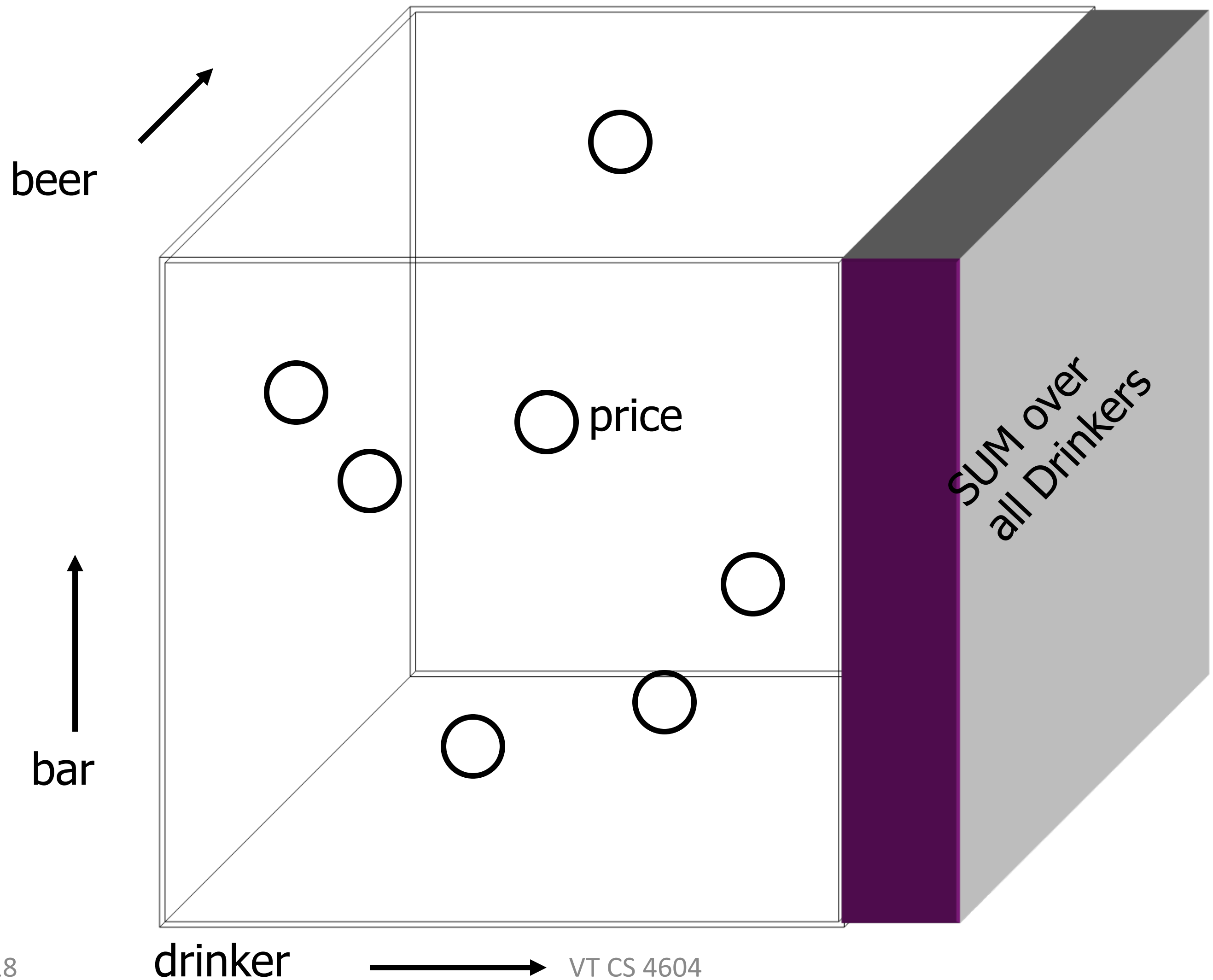
Visualization -- Data Cubes



Marginals

- The data cube also includes aggregation (typically SUM) along the margins of the cube.
- The *marginals* include aggregations over one dimension, two dimensions,…

Visualization --- Data Cube w/Aggregation



Example: Marginals

- Our 4-dimensional Sales cube includes the sum of price over each bar, each beer, each drinker, and each time unit (perhaps days).
- It would also have the sum of price over all bar-beer pairs, all bar-drinker-day triples,…

Marginals

- Think of each dimension as having an additional value *.
- A point with one or more *'s in its coordinates aggregates over the dimensions with the *'s.
- Example: ("Joe's Bar", "Bud", *, *) holds the sum, over all drinkers and all time, of the Bud consumed at Joe's.

Drill-Down

- *Drill-down* = “de-aggregate” = break an aggregate into its constituents.
- Example: having determined that Joe’s Bar sells very few Anheuser-Busch beers, break down his sales by particular A.-B. beer.

Roll-Up

- *Roll-up* = aggregate along one or more dimensions.
- Example: given a table of how much Bud each drinker consumes at each bar, roll it up into a table giving total amount of Bud consumed by each drinker.

Example: Roll Up and Drill Down

$ of Anheuser-Busch by drinker/bar

	Jim	Bob	Mary
Joe's Bar	45	33	30
Bull & Bones	50	36	42
Blue Chalk	38	31	40

Roll up by Bar

$ of A-B / drinker

Jim	Bob	Mary
133	100	112

Drill down by Beer

$ of A-B Beers / drinker

	Jim	Bob	Mary
Bud	40	29	40
M'lob	45	31	37
Bud Light	48	40	35

Structure of the Data Cube

- CUBE(F) of fact table F is *roughly* === the Fact table (F) + aggregations across all dimensions (i.e. marginals)
  - Note CUBE(F) is a relation itself!

CUBE in SQL: Example

- For our Sales example:

Sales(bar, beer, drinker, time, price)

```
CREATE MATERIALIZED VIEW SalesCube AS
 SELECT bar, beer, drinker, time,
  SUM(price)
 FROM Sales
 GROUP BY bar, beer, drinker, time WITH
  CUBE;
```

Tuples in SalesCube

- Tuples implied by the standard GROUP-BY:

```
(Joes, Bud, John, 4/19/13, 3.00)
```

- *And* those tuples of that are constructed by rolling-up the dimensions in GROUP-BY (== marginals, NULL == *). E.g:

```
(Joes, NULL, John, 4/19/13, 10.00)
(Joes, NULL, John, NULL, 200.00)
(Joes, NULL, NULL, NULL, 200000.00)
(NULL, NULL, NULL, NULL, 2000000.00)
```

Tuples in SalesCube

- Tuples implied by the standard GROUP-BY:

```
(Joes, Bud, John, 4/19/13, 3.00)
```

- *And* those tuples of that are constructed by rolling-up the dimensions in GROUP-BY (== marginals, NULL == *). E.g:

```
(Joes, NULL, John, 4/19/13, 10.00)
(Joes, NULL, John, NULL, 200.00)
(Joes, NULL, NULL, NULL, 200000.00)
(NULL, NULL, NULL, NULL, 2000000.00)
```

Total spent by John at Joes on Apr 19.

Tuples in SalesCube

- Tuples implied by the standard GROUP-BY:

```
(Joes, Bud, John, 4/19/13, 3.00)
```

- *And* those tuples of that are constructed by rolling-up the dimensions in GROUP-BY (== marginals, NULL == *). E.g:

```
(Joes, NULL, John, 4/19/13, 10.00)
(Joes, NULL, John, NULL, 200.00)
(Joes, NULL, NULL, NULL, 200000.00)
(NULL, NULL, NULL, NULL, 2000000.00)
```

Total spent by John at Joes ever.

Tuples in SalesCube

- Tuples implied by the standard GROUP-BY:

```
(Joes, Bud, John, 4/19/13, 3.00)
```

- *And* those tuples of that are constructed by rolling-up the dimensions in GROUP-BY (== marginals, NULL == *). E.g:

```
(Joes, NULL, John, 4/19/13, 10.00)
(Joes, NULL, John, NULL, 200.00)
(Joes, NULL, NULL, NULL, 200000.00)
(NULL, NULL, NULL, NULL, 2000000.00)
```

Total spent by everyone at Joes ever.

Tuples in SalesCube

- Tuples implied by the standard GROUP-BY:

```
(Joes, Bud, John, 4/19/13, 3.00)
```

- *And* those tuples of that are constructed by rolling-up the dimensions in GROUP-BY (== marginals, NULL == *). E.g:

```
(Joes, NULL, John, 4/19/13, 10.00)
(Joes, NULL, John, NULL, 200.00)
(Joes, NULL, NULL, NULL, 200000.00)
(NULL, NULL, NULL, NULL, 2000000.00)
```

Total spent by everyone at every bar ever.

Compare ROLAP vs MOLAP

ROLAP Solution

```
CREATE VIEW BABMS(bar, addr,
       beer, manf, sales) AS
SELECT bar, addr, beer, manf,
           SUM(price) sales
FROM Sales NATURAL JOIN Bars
       NATURAL JOIN Beers
GROUP BY bar, addr, beer, manf;
```

- A specific view for a specific type of query (note the join)

MOLAP (Data Cube) Solution

```
CREATE MATERIALIZED VIEW SalesCube
   AS
SELECT bar, beer, drinker, time,
   SUM(price)
FROM Sales
GROUP BY bar, beer, drinker, time
   WITH CUBE;
```

- A generalized view which stores marginals as well (no join)

VirginiaTech

How to answer queries using Cube?

- Essentially similar to ROLAP using materialized views, but now use the SalesCube

- How to do CUBE(F) efficiently?
  - ….Skip….

DATA MINING

VirginiaTech

Data Mining

- *Data mining* is a popular term for techniques to summarize big data sets in useful ways.
- Examples:
  1. Clustering all Web pages by topic.
  2. Finding characteristics of fraudulent credit-card use.

Supervised Learning: Decision Trees: Problem

Age	Chol-level	Gender	…	CLASS-ID
30	150	M		+
				…
				-

Has heart disease

Supervised Learning: Decision Trees: Problem

Age	Chol-level	Gender	…	CLASS-ID
30	150	M		+
				…
				-
15	90	F	…	??

What is the label for this new patient?

VirginiaTech

Supervised Learning: Decision Trees: Problem

Training Data Set

Age	Chol-level	Gender	…	CLASS-ID
30	150	M		+
				…
				-

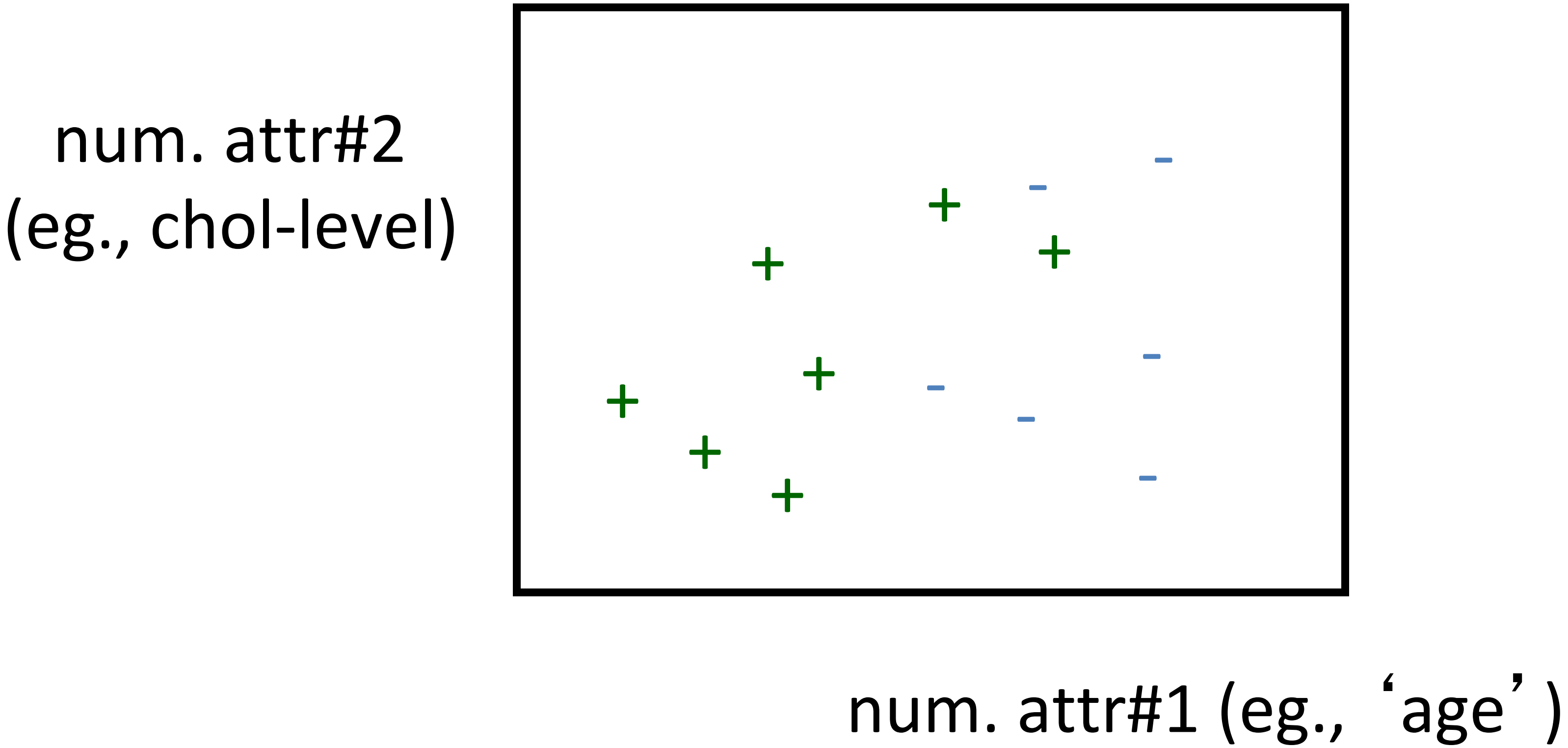
15	90	F	…

??

What is the label for this new patient?

Decision trees

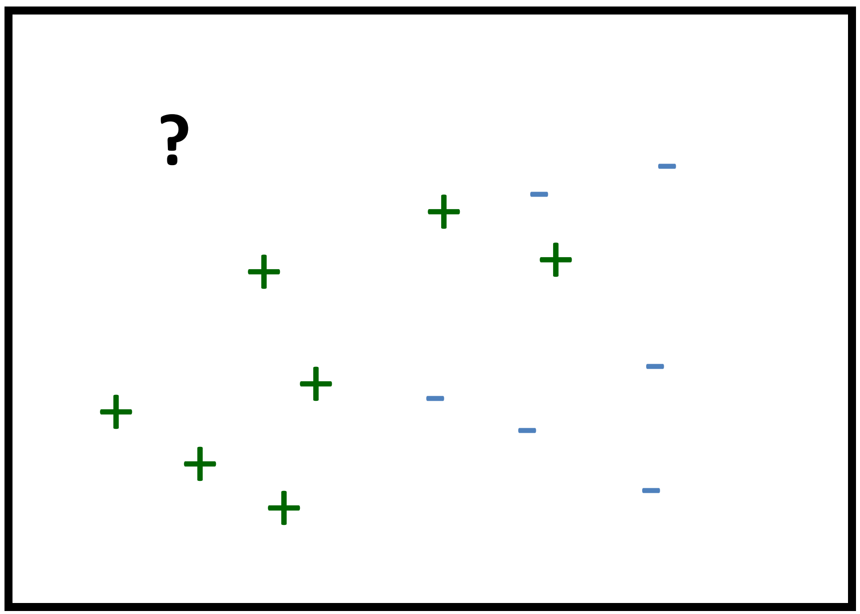
- Pictorially, we have



Decision trees

- and we want to label '?'

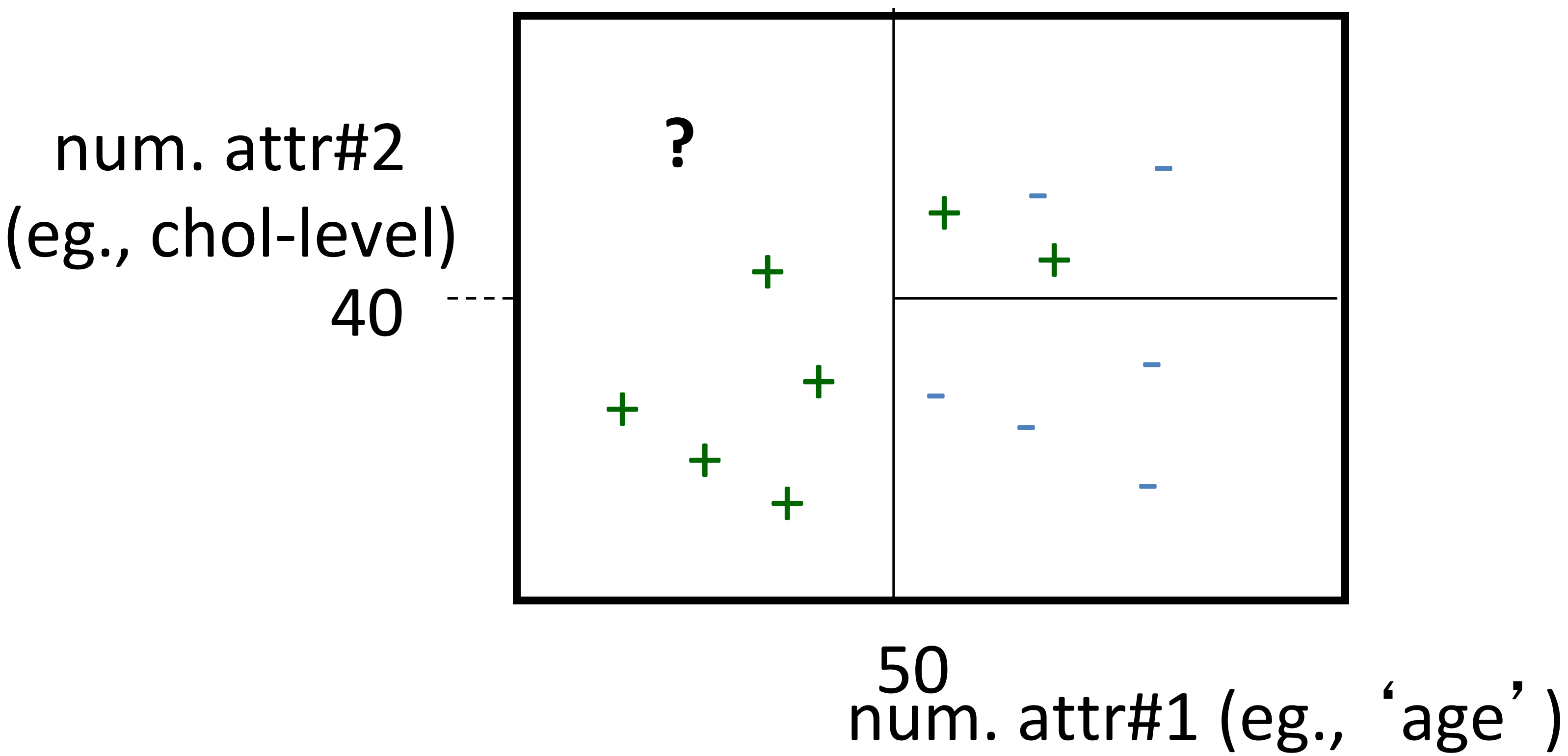
num. attr#2
(eg., chol-level)



num. attr#1 (eg., 'age')

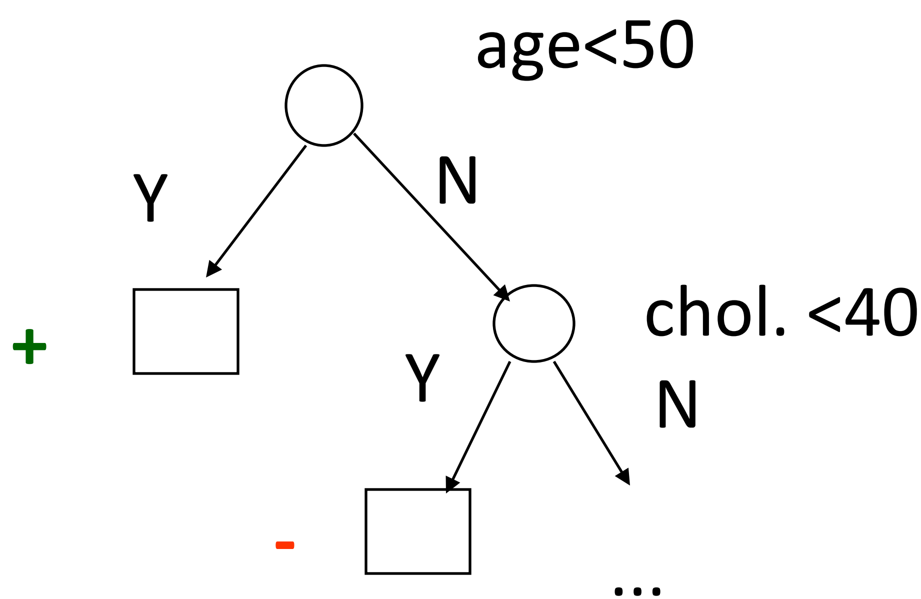
Decision trees

- so we build a decision tree:



Decision trees

- so we build a decision tree:



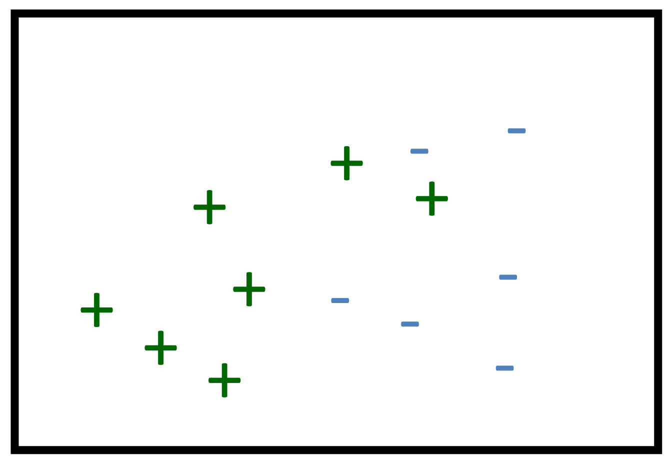
Decision trees: Approach

- Typically, two steps:
  - tree building
  - tree pruning (for over-training/over-fitting)

Tree building

- How?

num. attr#2
(eg., chol-level)



num. attr#1 (eg., ‘age’)

Tree building

- How?
- A: Partition, recursively - pseudocode:

  Partition (Dataset S)

  if all points in S have same label

  then return

  evaluate splits along each attribute A

  pick best split, to divide S into S1 and S2

  Partition(S1); Partition(S2)

Tree building

- **Q1: how to introduce splits along attribute A_i**
- Q2: how to evaluate a split?

Tree building

- Q1: how to introduce splits along attribute A_i
- A1:
  - for num. attributes:
    - binary split, or
    - multiple split
  - for categorical attributes:
    - compute all subsets (expensive!), or
    - use a greedy algo

Tree building

- Q1: how to introduce splits along attribute A_i
- **Q2: how to evaluate a split?**

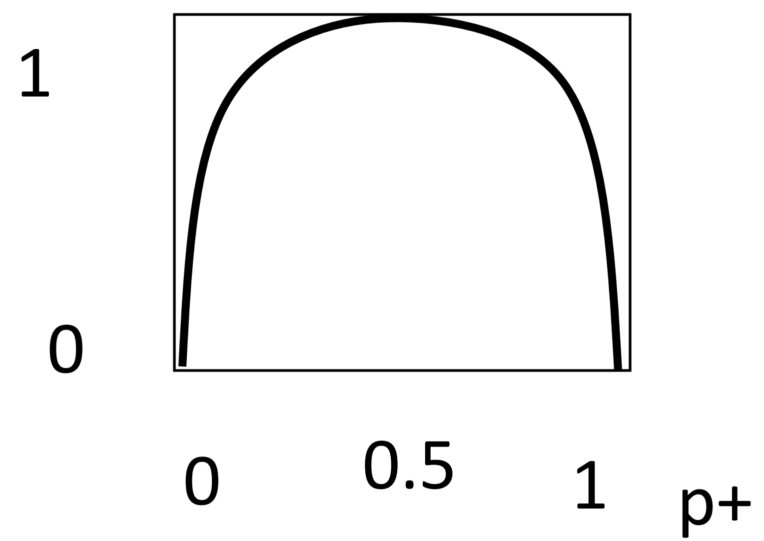
Tree building

- Q1: how to introduce splits along attribute A_i
- **Q2: how to evaluate a split?**
- A: by how close to uniform each subset is - ie., we need a measure of uniformity:

Tree building

entropy: $H(p_+, p_-)$

$$H = -p_+ \log(p_+) - p_- \log(p_-)$$

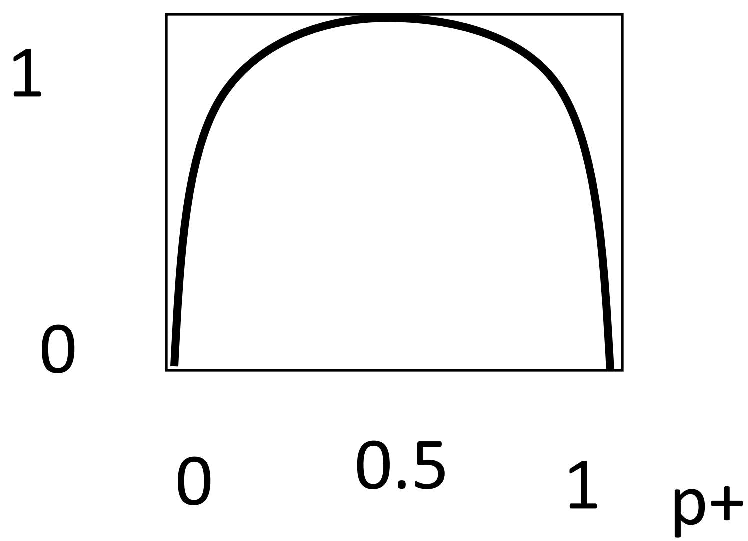


Any other measure?

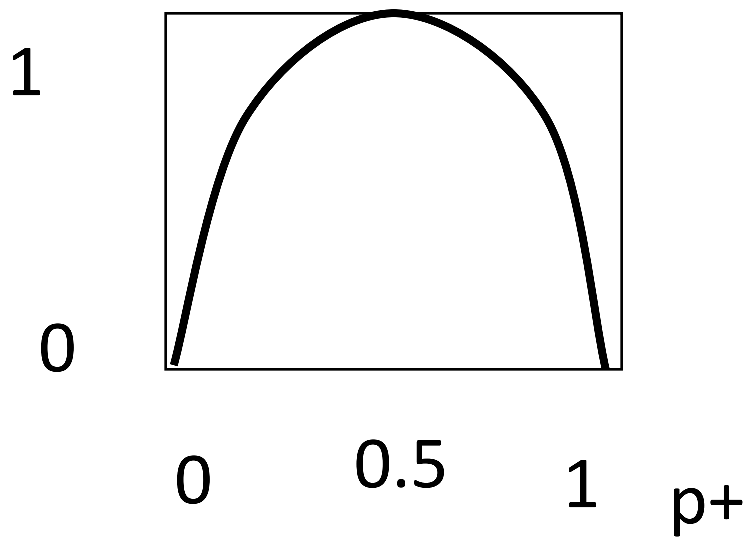
Tree building

entropy: $H(p_+, p_-)$

$$H = -p_+ \log(p_+) - p_- \log(p_-)$$



'gini' index: $1 - p_+^2 - p_-^2$



Tree building

entropy: $H(p_+, p_-)$

'gini' index: $1-p_+^2 - p_-^2$

(How about multiple labels?)

VirginiaTech

Tree building

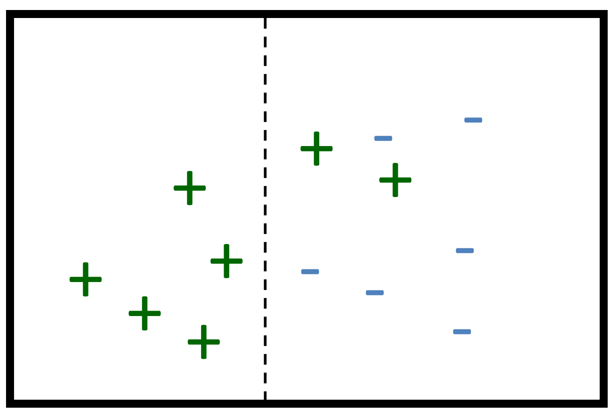
Intuition:

- entropy: #bits to encode the class label
- gini: classification error, if we randomly guess '+' with prob. p_+

Tree building

Thus, we choose the split that reduces entropy/ classification-error the most: Eg.:

num. attr#2 (eg., chol-level)



num. attr#1 (eg., ‘age’)

Tree building

- Before split: we need

  $(n_+ + n_-) * H(p_+, p_-) = (7+6) * H(7/13, 6/13)$

  bits total, to encode all the class labels

- After the split we need:

  0 bits for the first half and

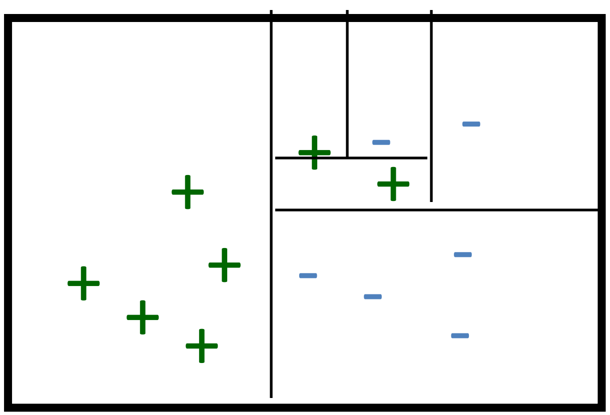
  $(2+6) * H(2/8, 6/8)$ bits for the second half

VirginiaTech

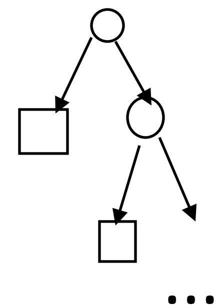
Tree pruning

- What for?

num. attr#2
(eg., chol-level)



num. attr#1 (eg., ‘age’)



Summary: classifiers

- Classification through trees
- Building phase - splitting policies
- Pruning phase (to avoid over-fitting)

Unsupervised Learning: Market-Basket Data

- An important form of mining from relational data involves *market baskets* = sets of "items" that are purchased together as a customer leaves a store.
- Summary of basket data is *frequent itemsets* = sets of items that often appear together in baskets.

Example: Market Baskets

- If people often buy hamburger and ketchup together, the store can:
  1. Put hamburger and ketchup near each other and put potato chips between.
  2. Run a sale on hamburger and raise the price of ketchup.

Finding Frequent Pairs

- The simplest case is when we only want to find "frequent pairs" of items.
- Assume data is in a relation Baskets(basket, item).
- The *support threshold* s is the minimum number of baskets in which a pair appears before we are interested.

Frequent Pairs in SQL

```
SELECT b1.item, b2.item
FROM Baskets b1, Baskets b2
WHERE b1.basket = b2.basket
  AND b1.item < b2.item
GROUP BY b1.item, b2.item
HAVING COUNT(*) >= s;
```

Look for two Basket tuples with the same basket and different items. First item must precede second, so we don't count the same pair twice.

Throw away pairs of items that do not appear at least *s* times.

Create a group for each pair of items that appears in at least one basket.

VirginiaTech

(Famous!) A-Priori Trick – (1)

- Straightforward implementation involves a join of a huge Baskets relation with itself.
- Anti-Monotonicity Property: The *a-priori algorithm* speeds the query by recognizing that a pair of items $\{i, j\}$ cannot have support s unless both $\{i\}$ and $\{j\}$ do.

R. Agrawal, T. Imielinski, A. Swami
'Mining Association Rules between Sets of Items in Large Databases', SIGMOD 1993.

A-Priori Trick – (2)

- Use a materialized view to hold only information about frequent items.

```
INSERT INTO Baskets1(basket, item)
SELECT * FROM Baskets
WHERE item IN (
  SELECT item FROM Baskets
  GROUP BY item
  HAVING COUNT(*) >= s
);
```

Items that appear in at least *s* baskets.

A-Priori Algorithm

1. Materialize the view Baskets1.
2. Run the obvious query, but on Baskets1 instead of Baskets.

- Computing Baskets1 is cheap, since it doesn't involve a join.
- Baskets1 *probably* has many fewer tuples than Baskets.
  – Running time shrinks with the *square* of the number of tuples involved in the join.

Example: A-Priori

- Suppose:
  1. A supermarket sells 10,000 items.
  2. The average basket has 10 items.
  3. The support threshold is 1% of the baskets.
- At most 1/10 of the items can be frequent.
- *Probably*, the minority of items in one basket are frequent -> factor 4 speedup.

Conclusions

- Data Mining: of high commercial interest (think BIG data)
- DM = DB + Machine Learning + Stats.

- Data Warehousing/OLAP: to get the data
- Tree Classifiers
- Association Rules

….. (like clustering etc.)