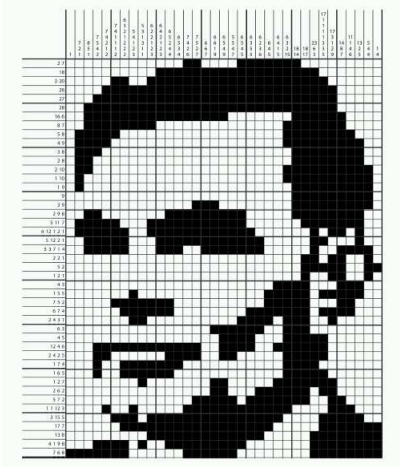
Data Mining (DM)

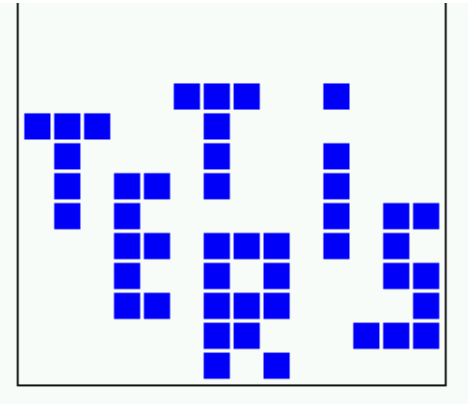
Walter Kosters, Universiteit Leiden

Tuesday 23 October 2012 — Rotterdam, ICT.Open, IPA

`http://www.liacs.nl/home/kosters/` `rdam.pdf`


VN Detective en Thrillergids
15261 titels
www.vnster.nl
snel zoeken
thuis
hulp
pdf/csv
uitgebreid zoeken op:
titel
auteur
bladeren
mobiel
contact
U bevindt zich hier: thuis — www.vnster.nl
Vrij Nederland
Detective en Thrillergids
21ste
Detective & Thrillergids
mobiel
contact





Data Mining

What is Data Mining?

1. Definition by algorithm

According to IEEE ICDM 2006, top 10 DM algorithms are:

Classification C4.5, CART, kNN, NaiveBayes

Statistical learning SVM, EM

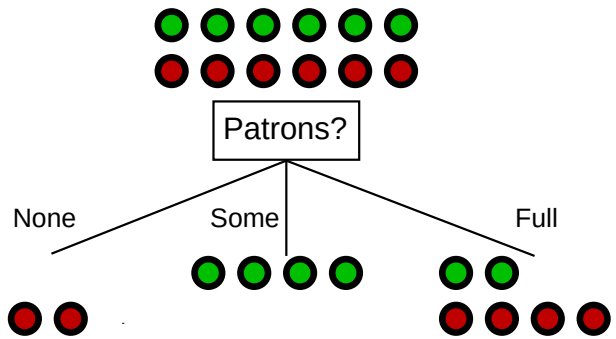
Link mining PageRank

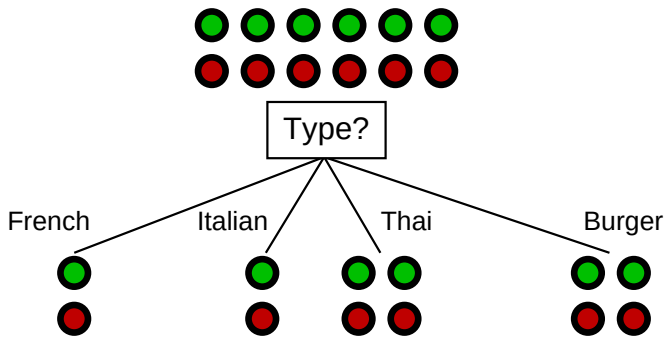
Association rules Apriori

Clustering k-Means

Bagging and boosting AdaBoost

In the 1980s J. Ross Quinlan developed C4.5, that builds decision trees based on entropy:

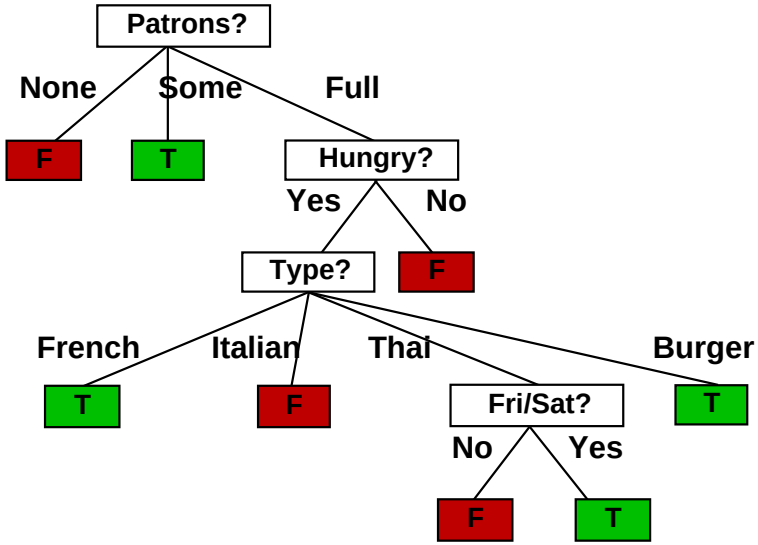




The Patrons question is "better" than the Type question.

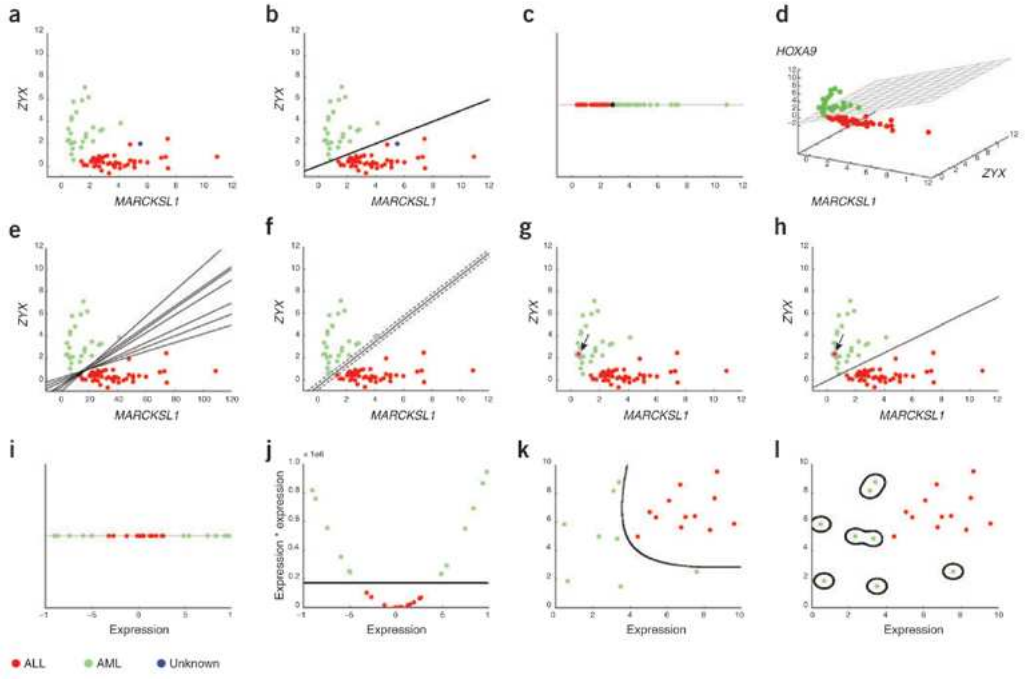
$$\sum_{i=1}^{\#\text{classes}} \frac{p_i + n_i}{p + n} \left\{ - \frac{p_i}{p_i + n_i} \log_2 \frac{p_i}{p_i + n_i} - \frac{n_i}{p_i + n_i} \log_2 \frac{n_i}{p_i + n_i} \right\}$$

C4.5 produces a decision tree like



From: S.J. Russell and P. Norvig, Artificial Intelligence, A Modern Approach, Prentice Hall, third edition, 2010.

A Support Vector Machine (SVM; Vapnik, 1990s) tries to embed input data into a high-dimensional feature space, in such a way that classes are linearly separable.



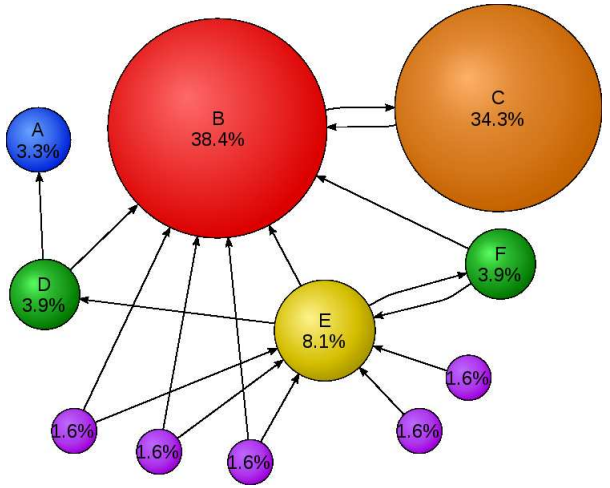
From: W.S. Noble, What is a Support Vector Machine?, Nature Biotechnology 24, 1565–1567 (2006).

Around 1998 Brin and Page published their ideas about PageRank, that "drives" Google.

The PageRank P_i of a page i satisfies

$$P_i = (1-d) + d \sum_{j:\, j \to i} P_j \,/\, O_j$$

Here d is the "damping factor", and O_j is the number of outgoing links of page j.



See books by Langville & Meyer.

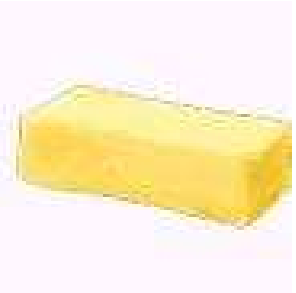
Consider this very small "market basket" dataset:

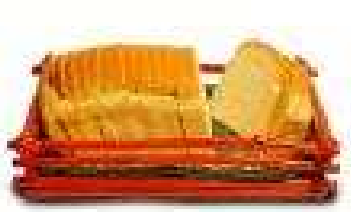
product = item customer = transaction	1	2	3	4	5	6	7	8	9
i	1	1	0	0	1	1	1	1	0
ii	1	0	1	0	0	0	0	1	1
iii	0	1	1	0	1	0	1	0	0
iv	1	0	1	0	1	1	0	1	1
v	0	0	0	0	1	0	0	0	0
vi	0	1	0	0	1	0	1	0	0

The support of an itemset is the number of customers that buy it. For example: the 2-itemset $\{1, 5\}$ has support 2: it is bought by customers i and iv.

An itemset with high support (above some support threshold) is called frequent. Suppose that from the α customers that buy set A, β buy set B too ($A \cap B = \emptyset$). Then we can say that the association rule $A \Rightarrow B$ has confidence β/α.

Now we are interested in association rules $A \Rightarrow B$ with both high confidence *and* high support, i.e., high support for the itemset $A \cup B$.





There is an extensive literature on association rules, in particular on the following aspects:

- efficient algorithms to find them . . . (see FIMI)
- how to select the interesting ones . . .
- how to deal with non-Boolean attributes . . .

For this last issue one can use fuzzy logic, where instead of 0/1 (not-buy vs. buy) intermediate values can occur. For example: age can be "young" to an extent of 0.35.

product = item customer = transaction	1	2	3	4	5	6	7	8	9
i	1	1	0	0	1	1	1	1	0
ii	1	0	1	0	0	0	0	1	1
iii	0	1	1	0	1	0	1	0	0
iv	1	0	1	0	1	1	0	1	1
v	0	0	0	0	1	0	0	0	0
vi	0	1	0	0	1	0	1	0	0

Even for this small dataset it is hard to see that the itemset $\{2, 5, 7\}$ is the only 3-itemset that is "bought" by at least 50% (i.e., 3, the support threshold) of the customers, and is therefore frequent.

Frequent itemsets naturally lead to association rules, like $\{2, 7\} \Rightarrow \{5\}$.

Around 1995 Agrawal et al. devised Apriori, relying on the following property:

A subset of a frequent set must be frequent too!

We have $A \subseteq B \Rightarrow \text{support}(A) \geq \text{support}(B)$, which is anti-monotone.

The algorithm proceeds this: small frequent sets are the building blocks for larger ones; first you join them to make candidates, and for these candidates you compute the support.

The Apriori algorithm works as follows:

count frequency of itemsets with 1 item
$L_1 \leftarrow$ frequent ones; $k \leftarrow 2$
while $L_{k-1} \neq \emptyset$ **do**
  $C_k \leftarrow$ candidates in $\{A \cup B \mid A, B \in L_{k-1}, |A \cup B| = k\}$
  compute their supports
  $L_k \leftarrow$ frequent sets from C_k; $k \leftarrow k + 1$
od
return $\bigcup_{\ell=1}^{k-2} L_\ell$

Example: $\{1, 2, 3, 6\}$ and $\{1, 2, 3, 8\}$ produce $\{1, 2, 3, 6, 8\}$ (if all its subsets are frequent).

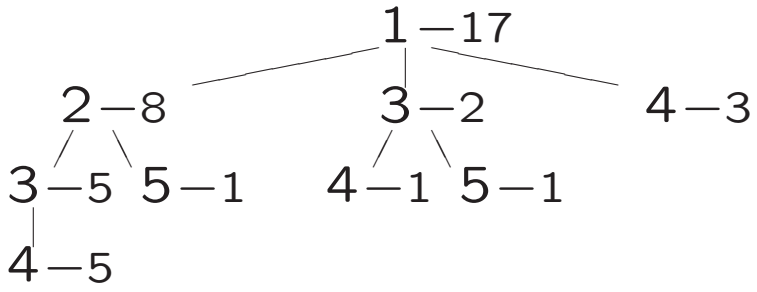
Example (continued)

product = item customer = transaction	1	2	3	4	5	6	7	8	9
i	1	1	0	0	1	1	1	1	0
ii	1	0	1	0	0	0	0	1	1
iii	0	1	1	0	1	0	1	0	0
iv	1	0	1	0	1	1	0	1	1
v	0	0	0	0	1	0	0	0	0
vi	0	1	0	0	1	0	1	0	0

$L_1 = \{\{1\},\{2\},\{3\},\{5\},\{7\},\{8\}\}$
$C_2 = \{\{1,2\},\{1,3\},\{1,5\},\ldots,\{7,8\}\}$; $|C_2| = 15 = \binom{6}{2} < \binom{9}{2}$
$L_2 = \{\{1,8\},\{2,5\},\{2,7\},\{5,7\}\}$
$C_3 = \{\{2,5,7\}\} = L_3$

Indeed $\{2,5,7\}$ is the only frequent (support ≥ 3) 3-itemset.

An FP-tree, that condenses a dataset, looks like this:



Paths represent itemsets, including the number of customers that "buy" them. The example tree shows (among other things) that there are $5 + 1 + 3 = 9$ customers that "buy" the 2-itemset $\{1, 4\}$ — so its support equals 9. Note that items are first sorted with respect to support.

The fastest algorithms use FP-trees (Han et al., 2000).

So far we have seen:

Classification C4.5

Statistical learning SVM

Link mining PageRank

Association rules Apriori

Apparently, there are many overlaps between Data Mining (DM) on the one hand and Artificial Intelligence (AI) and its subclass Machine Learning (ML) on the other hand. Furthermore, we have Databases. And Statistics!

DM tries to discover previously unknown knowledge, ML tries to predict based on known facts.

DM discovers hypotheses, Statistics tests them.

Today's definition:

"Data Mining discovers patterns"

+ surprising
+ large datasets
+ visualization
+ algorithms (IPA)

2. Some recent trends

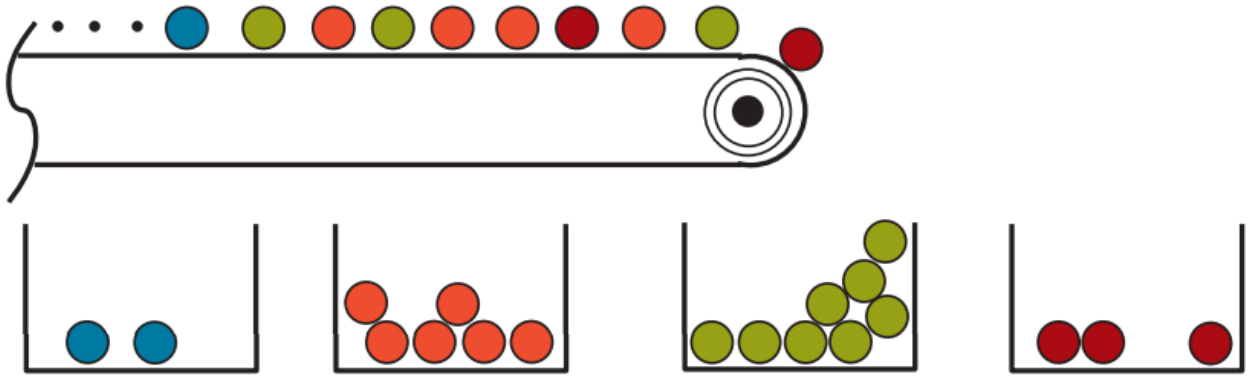
Streams — with time

In 1980 Boyer and Moore devised the following Majority algorithm for an array $a_1, \ldots, a_n$:

$x \leftarrow a_1;\ c \leftarrow 1;$
for $i \leftarrow 2, \ldots, n$ **do**
  if $a_i = x$ **then** $c \leftarrow c + 1;$
  else if $c = 0$ **then** $x \leftarrow a_i; c \leftarrow 1;$
  else $c \leftarrow c - 1;$ **fi fi**
od

If a has a majority element (occurs $> n/2$ times), it is x.

The middle two bins realize the threshold of 20% (1/5) of the "infinite" data stream:



From: G. Cormode and M. Hadjieleftheriou, Finding the Frequent Items in Streams of Data, Communications of the ACM 52, 97–105 (2009).

The "Frequent" algorithm (1982/2002) finds all items in sequence a whose frequency exceeds $1/k$ of the total count:

$T \leftarrow \emptyset$;
for $i \leftarrow 1, \ldots$ **do**
  if $a_i \in T$ **then** $c_{a_i} \leftarrow c_{a_i} + 1$;
  else if $|T| < k - 1$ **then** $T \leftarrow T \cup \{a_i\}$; $c_{a_i} \leftarrow 1$;
  else for $t \in T$ **do**
    $c_t \leftarrow c_t - 1$; **if** $c_t = 0$ **then** $T \leftarrow T \setminus \{t\}$; **fi**
  od fi fi
od

$\langle T, c \rangle$ acts as some sort of summary.

DM goes BIO — with algorithms

The Burrows-Wheeler transform (1994), used for compression (the transformed string allows for good runlength encoding), has applications in biological data mining.
The string "^BANANA$" is (efficiently!?) processed like this:

^BANANA$	ANANA$^B	B
$^BANANA	ANA$^BAN	N
A$^BANAN	A$^BANAN	N
NA$^BANA	BANANA$^	^
ANA$^BAN	NANA$^BA	A
NANA$^BA	NA$^BANA	A
ANANA$^B	^BANANA$	$
BANANA$^	$^BANANA	A

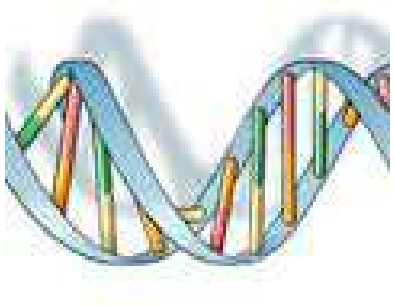
Sort the "rotations", and take the last column: BNN^AA$A.
The original string can be recovered from this.

The suffix array of a string is the *lexicographically sorted array of all its suffixes*. Usually we give the indexes where the suffixes begin.

Example: the string `example` has 7 (non-empty) suffixes:

`ample`, `e`, `example`, `le`, `mple`, `ple`, `xample`

So its suffix array is $[2, 6, 0, 5, 3, 4, 1]$.



The string $S =$ `example` has Burrows-Wheeler transform:

`example`	`ampleex`	`x`
`eexampl`	`eexampl`	`l`
`leexamp`	`example`	`e`
`pleexam`	`leexamp`	`p`
`mpleexa`	`mpleexa`	`a`
`ampleex`	`pleexam`	`m`
`xamplee`	`xamplee`	`e`

Note that BWT$[i] = S[$Suffix-Array$[i] - 1]$.
Normally you append a '$' to the string, with '$' $<$ 'a'.

The story begins in the 1990s, when finally Ukkonen came up with a linear time construction for suffix trees. Full details: Dan Gusfield's book, or Pekka Kilpeläinen's lecture notes:

`www.cs.uku.fi/~kilpelai/BSA07/index.shtml`

A depth first "lexical" suffix tree traversal easily gives the suffix array.

In 2003 three independent algorithms to directly construct suffix arrays (introduced by Myers and Manber) in linear time (sometimes together with the so-called lcp-array = lenghts of the longest common prefixes; together they are equivalent with suffix trees) were found: Kärkkäinen-Sanders, Ko-Aluru and Kim-Sim-Park-Park.

Suffix trees and suffix arrays are great when one wants to find, e.g., all overlaps in a large set of (DNA-)strings.

Often a special final character `$` is attached to the string at hand, to avoid a suffix that matches a prefix of another suffix: `xabxa`.

How to find an occurrence of a substring P of a string T? Perform a binary search on the suffix array SA: compare P to the middle element of SA, and so on. With help of the lcp-array, this can be done in $O(n + \log(m))$ time, where $n = |P|$ and $m = |T|$. (Don't forget the "preprocessing"; it works if you have many Ps.)

The Kärkkäinen-Sanders algorithm is the easiest (but perhaps not the very best) way to build the suffix array. It goes like this:

- recursively construct the suffix array of the suffixes starting at positions i that are not a multiple of 3: $1, 2, 4, 5, 7, 8, 10, 11, \ldots$
- construct the suffix array of the others using the result of the first step
- merge the two suffix arrays into one

`01234567890`
`mississippi`

- start with `ississippi` ($i = 1$), `issippi` ($i = 4$), `ippi` ($i = 7$), `i00` ($i = 10$, with extra `00`), `ssissippi` ($i = 2$), `ssippi` ($i = 5$), `ppi` ($i = 8$) — in this order
  we find $[3, 2, 1, 0, 6, 5, 4] \Rightarrow [10, 7, 4, 1, 8, 5, 2]$

- do `mississippi`, `pi0`, `sippi`, `sissippi`: $[0, 9, 6, 3]$

- merge the two suffix arrays: $[10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$

The lcp value for `issippi` and `ississippi` is $4 = lcp(2, 3)$.

How can the lcp-array help when searching for a substring?

Suppose we are looking for $P =$ `abcdemn`. Suppose that we do a binary search in $L =$ `abcdefg...`, ..., $M =$ `abcdefg...`, ...,$R =$ `abcdxyz...` (within the suffix array). P matches the first $\ell = 5$ characters of L, and the first $r = 4$ of R. Here $lcp(L, M) > \ell$.

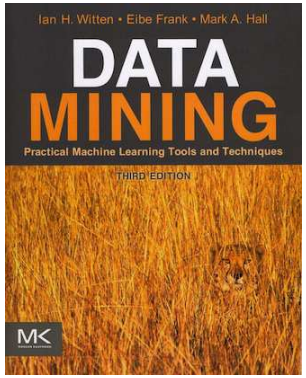
This helps ... also in general. We need the lcp value not just for neighbours!

For DNA (the human genome has 3,000,000,000 nucleotides A/C/G/T): its BWT requires 3GB, its suffix array maybe 12GB.

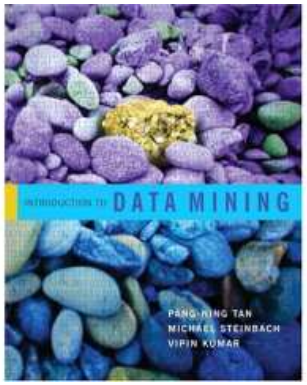
3. Privacy

Some good books:

I.H. Witten, E. Frank and M.A. Hall,
Data Mining: Practical Machine Learning Tools and Techniques,
third edition, 2011
(plus free WEKA software!)



P.-N. Tan, M. Steinbach and V. Kumar,
Introduction to Data Mining,
2006



Questions?