1 What Is Data Mining?

Originally, "data mining" was a statistician's term for overusing data to draw invalid inferences.

- Bonferroni's theorem warns us that if there are too many possible conclusions to draw, some will be true for purely statistical reasons, with no physical validity.
- Famous example: David Rhine, a "parapsychologist" at Duke in the 1950's tested students for "extrasensory perception" by asking them to guess 10 cards — red or black. He found about 1/1000 of them guessed all 10, and instead of realizing that that is what you'd expect from random guessing, declared them to have ESP. When he retested them, he found they did no better than average. His conclusion: telling people they have ESP causes them to lose it!

Our definition: "discovery of useful summaries of data."

1.1 Applications

Some examples of "successes":

1. Decision trees constructed from bank-loan histories to produce algorithms to decide whether to grant a loan.
2. Patterns of traveler behavior mined to manage the sale of discounted seats on planes, rooms in hotels, etc.
3. "Diapers and beer." Observation that customers who buy diapers are more likely to by beer than average allowed supermarkets to place beer and diapers nearby, knowing many customers would walk between them. Placing potato chips between increased sales of all three items.
4. Skycat and Sloan Sky Survey: clustering sky objects by their radiation levels in different bands allowed astromomers to distinguish between galaxies, nearby stars, and many other kinds of celestial objects.
5. Comparison of the genotype of people with/without a condition allowed the discovery of a set of genes that together account for many cases of diabetes. This sort of mining will become much more important as the human genome is constructed.

1.2 The Data-Mining Communities

As data-mining has become recognized as a powerful tool, several different communities have laid claim to the subject:

1. Statistics.
2. AI, where it is called "machine learning."
3. Researchers in clustering algorithms.
4. Visualization researchers.
5. Databases. We'll be taking this approach, of course, concentrating on the challenges that appear when the data is large and the computations complex. In a sense, data mining can be thought of as algorithms for executing very complex queries on non-main-memory data.

1.3 Stages of the Data-Mining Process

1. *Data gathering*, e.g., data warehousing, Web crawling.
2. *Data cleansing*: eliminate errors and/or bogus data, e.g., patient fever = 125.
3. *Feature extraction*: obtaining only the interesting attributes of the data, e.g., "date acquired" is probably not useful for clustering celestial objects, as in Skycat.
4. *Pattern extraction and discovery.* This is the stage that is often thought of as "data mining," and is where we shall concentrate our effort.
5. Visualization of the data.
6. Evaluation of results; not every discovered fact is useful, or even true! Judgement is necessary before following your software's conclusions.

2 Association Rules and Frequent Itemsets

The *market-basket problem* assumes we have some large number of *items*, e.g., "bread," "milk." Customers fill their market baskets with some subset of the items, and we get to know what items people buy together, even if we don't know who they are. Marketers use this information to position items, and control the way a typical customer traverses the store.
In addition to the marketing application, the same sort of question has the following uses:

1. Baskets = documents; items = words. Words appearing frequently together in documents may represent phrases or linked concepts. Can be used for intelligence gathering.

2. Baskets = sentences, items = documents. Two documents with many of the same sentences could represent plagiarism or mirror sites on the Web.

2.1 Goals for Market-Basket Mining

1. *Association rules* are statements of the form $\{X_1, X_2, \ldots, X_n\} \Rightarrow Y$, meaning that if we find all of $X_1, X_2, \ldots, X_n$ in the market basket, then we have a good chance of finding Y. The probability of finding Y for us to accept this rule is called the *confidence* of the rule. We normally would search only for rules that had confidence above a certain threshold. We may also ask that the confidence be significantly higher than it would be if items were placed at random into baskets. For example, we might find a rule like $\{milk, butter\} \Rightarrow bread$ simply because a lot of people buy bread. However, the beer/diapers story asserts that the rule $\{diapers\} \Rightarrow beer$ holds with confidence siginificantly greater than the fraction of baskets that contain beer.

2. *Causality.* Ideally, we would like to know that in an association rule the presence of $X_1, \ldots, X_n$ actually "causes" Y to be bought. However, "causality" is an elusive concept. nevertheless, for market-basket data, the following test suggests what causality means. If we lower the price of diapers and raise the price of beer, we can lure diaper buyers, who are more likely to pick up beer while in the store, thus covering our losses on the diapers. That strategy works because "diapers causes beer." However, working it the other way round, running a sale on beer and raising the price of diapers, will not result in beer buyers buying diapers in any great numbers, and we lose money.

3. *Frequent itemsets.* In many (but not all) situations, we only care about association rules or causalities involving sets of items that appear frequently in baskets. For example, we cannot run a good marketing strategy involving items that no one buys anyway. Thus, much data mining starts with the assumption that we only care about sets of items with high *support*; i.e., they appear together in many baskets. We then find association rules or causalities only involving a high-support set of items (i.e., $\{X_1, \ldots, X_n, Y\}$ must appear in at least a certain percent of the baskets, called the *support threshold*.

2.2 Framework for Frequent Itemset Mining

We use the term *frequent itemset* for "a set S that appears in at least fraction s of the baskets," where s is some chosen constant, typically 0.01 or 1%.
We assume data is too large to fit in main memory. Either it is stored in a RDB, say as a relation $Baskets(BID, item)$ or as a flat file of records of the form $(BID, item1, item2, \ldots, itemn)$. When evaluating the running time of algorithms we:

- Count the number of passes through the data. Since the principal cost is often the time it takes to read data from disk, the number of times we need to read each datum is often the best measure of running time of the algorithm.

There is a key principle, called *monotonicity* or the *a-priori trick* that helps us find frequent itemsets:

- If a set of items S is frequent (i.e., appears in at least fraction s of the baskets), then every subset of S is also frequent.

To find frequent itemsets, we can:

1. Proceed levelwise, finding first the frequent items (sets of size 1), then the frequent pairs, the frequent triples, etc. In our discussion, we concentrate on finding frequent pairs because:
   (a) Often, pairs are enough.
   (b) In many data sets, the hardest part is finding the pairs; proceeding to higher levels takes less time than finding frequent pairs.

   Levelwise algorithms use one pass per level.

2. Find all *maximal frequent itemsets* (i.e., sets S such that no proper superset of S is frequent) in one pass or a few passes.

2.3 The A-Priori Algorithm

This algorithm proceeds levelwise.

1. Given support threshold s, in the first pass we find the items that appear in at least fraction s of the baskets. This set is called L_1, the frequent items. Presumably there is enough main memory to count occurrences of each item, since a typical store sells no more than 100,000 different items.

2. Pairs of items in L_1 become the *candidate pairs* C_2 for the second pass. We hope that the size of C_2 is not so large that there is not room for an integer count per candidate pair. The pairs in C_2 whose count reaches s are the frequent pairs, L_2.

3. The candidate triples, C_3 are those sets $\{A, B, C\}$ such that all of $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$ are in L_2. On the third pass, count the occurrences of triples in C_3; those with a count of at least s are the frequent triples, L_3.

4. Proceed as far as you like (or the sets become empty). L_i is the frequent sets of size i; C_{i+1} is the set of sets of size $i+1$ such that each subset of size i is in L_i.

2.4 Why A-Priori Helps

Consider the following SQL on a $Baskets(BID, item)$ relation with 10^8 tuples involving 10^7 baskets of 10 items each; assume 100,000 different items (typical of Wal-Mart, e.g.).

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

Note: s is the support threshold, and the second term of the `WHERE` clause is to prevent pairs of items that are really one item, and to prevent pairs from appearing twice.
In the join $Baskets \bowtie Baskets$, each basket contributes $\binom{10}{2} = 45$ pairs, so the join has 4.5×10^8 tuples.
A-priori "pushes the `HAVING` down the expression tree," causing us first to replace $Baskets$ by the result of

```
SELECT *
FROM Baskets
GROUP by item
HAVING COUNT(*) >= s;
```

If $s = 0.01$, then at most 1000 items' groups can pass the `HAVING` condition. Reason: there are 10^8 item occurrences, and an item needs $0.01 \times 10^7 = 10^5$ of those to appear in 1% of the baskets.

- Although 99% of the items are thrown away by a-priori, we should not assume the resulting $Baskets$ relation has only 10^6 tuples. In fact, *all* the tuples may be for the high-support items. However, in real situations, the shrinkage in $Baskets$ is substantial, and the size of the join shrinks in proportion to the *square* of the shrinkage in $Baskets$.

2.5 Improvements to A-Priori

Two types:

1. Cut down the size of the candidate sets C_i for $i \geq 2$. This option is important, even for finding frequent pairs, since the number of candidates must be sufficiently small that a count for each can fit in main memory.

2. Merge the attempts to find $L_1, L_2, L_3, \ldots$ into one or two passes, rather than a pass per level.

2.6 PCY Algorithm

Park, Chen, and Yu proposed using a hash table to determine on the first pass (while L_1 is being determined) that many pairs are not possibly frequent. Takes advantage of the fact that main memory is usualy *much* bigger than the number of items. During the two passes to find L_2, the main memory is laid out as in Fig. 1.

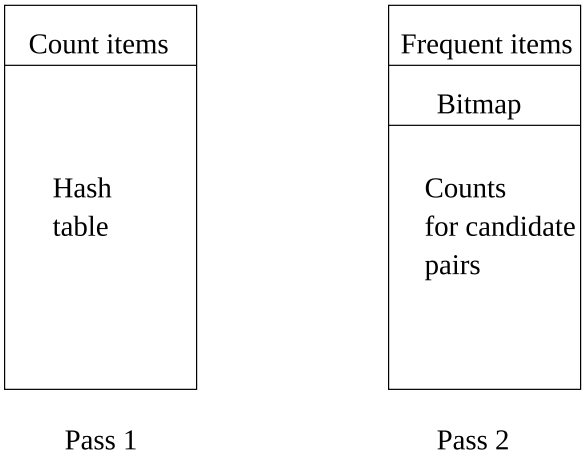


Figure 1: Two passes of the PCY algorithm

Assume that data is stored as a flat file, with records consisting of a basket ID and a list of its items.

1. Pass 1:

   (a) Count occurrences of all items.

   (b) For each bucket, consisting of items $\{i_1, \ldots, i_k\}$, hash all pairs to a bucket of the hash table, and increment the count of the bucket by 1.

   (c) At the end of the pass, determine L_1, the items with counts at least s.

   (d) Also at the end, determine those buckets with counts at least s.

      - Key point: a pair (i, j) cannot be frequent unless it hashes to a frequent bucket, so pairs that hash to other buckets need not be candidates in C_2.

      Replace the hash table by a *bitmap*, with one bit per bucket: 1 if the bucket was frequent, 0 if not.

2. Pass 2:

   (a) Main memory holds a list of all the frequent items, i.e. L_1.

   (b) Main memory also holds the bitmap summarizing the results of the hashing from pass 1.

      - Key point: The buckets must use 16 or 32 bits for a count, but these are compressed to 1 bit. Thus, even if the hash table occupied almost the entire main memory on pass 1, its bitmap ocupies no more than 1/16 of main memory on pass 2.

(c) Finally, main memory also holds a table with all the candidate pairs and their counts. A pair (i, j) can be a candidate in C_2 only if *all* of the following are true:

   i. i is in L_1.

   ii. j is in L_1.

   iii. (i, j) hashes to a frequent bucket.

   It is the last condition that distinguishes PCY from straight a-priori and reduces the requirements for memory in pass 2.

(d) During pass 2, we consider each basket, and each pair of its items, making the test outlined above. If a pair meets all three conditions, add to its count in memory, or create an entry for it if one does not yet exist.

- When does PCY beat a-priori? When there are too many pairs of items from L_1 to fit a table of candidate pairs and their counts in main memory, yet the number of frequent buckets in the PCY algorithm is sufficiently small that it reduces the size of C_2 below what can fit in memory (even with 1/16 of it given over to the bitmap).

- When will most of the buckets be infrequent in PCY? When there are a few frequent pairs, but most pairs are so infrequent that even when the counts of all the pairs that hash to a given bucket are added, they still are unlikely to sum to s or more.

2.7 The "Iceberg" Extensions to PCY

1. *Multiple hash tables*: share memory between two or more hash tables on pass 1, as in Fig. 2. On pass 2, a bitmap is stored for each hash table; note that the space needed for all these bitmaps is exactly the same as what is needed for the one bitmap in PCY, since the total number of buckets represented is the same. In order to be a candidate in C_2, a pair must:

   (a) Consist of items from L_1, and

   (b) Hash to a frequent bucket in *every* hash table.

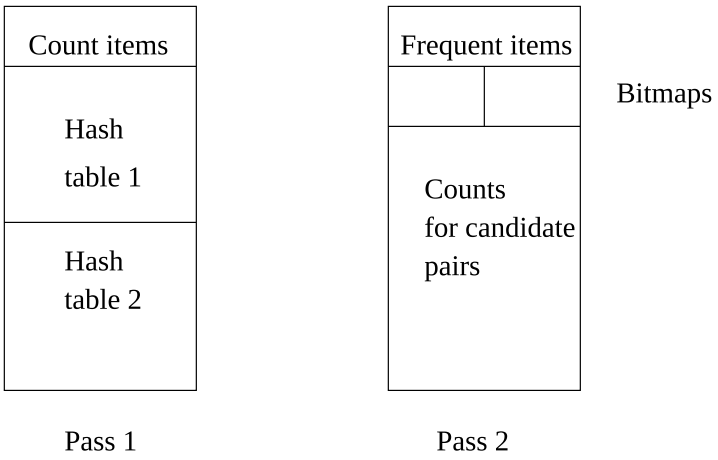


Figure 2: Multiple hash tables memory utilization

2. Iterated hash tables *Multistage*: Instead of checking candidates in pass 2, we run another hash table (different hash function!) in pass 2, but we only hash those pairs that meet the test of PCY; i.e., they are both from L_1 and hashed to a frequent bucket on pass 1. On the third pass, we keep bitmaps from both hash tables, and treat a pair as a candidate in C_2 only if:

   (a) Both items are in L_1.

(b) The pair hashed to a frequent bucket on pass 1.

(c) The pair also was hashed to a frequent bucket on pass 2.

Figure 3 suggests the use of memory. This scheme could be extended to more passes, but there is a limit, because eventually the memory becomes full of bitmaps, and we can't count any candidates.

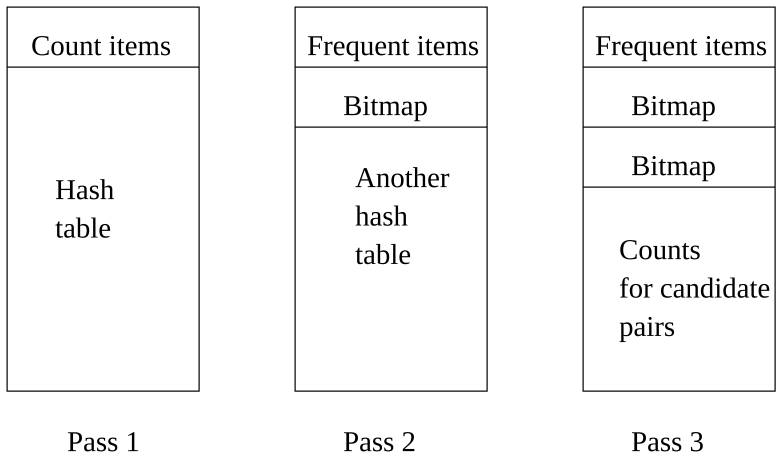


Figure 3: Multistage hash tables memory utilization

- When does multiple hash tables help? When most buckets on the first pass of PCY have counts *way* below the threshold s. Then, we can double the counts in buckets and still have most buckets below threshold.
- When does multistage help? When the number of frequent buckets on the first pass is high (e.g., 50%), but not all buckets. Then, a second hashing with some of the pairs ignored may reduce the number of frequent buckets significantly.

2.8 All Frequent Itemsets in Two Passes

The methods above are best when you only want frequent pairs, a common case. If we want all maximal frequent itemsets, including large sets, too many passes may be needed. There are several approaches to getting all frequent itemsets in two passes or less. They each rely on randomness of data in some way.

1. *Simple approach*: Taka a main-memory-sized sample of the data. Run a levelwise algorithm in main memory (so you don't have to pay for disk I/O), and hope that the sample will give you the truly frequent sets.
   - Note that you must scale the threshold s back; e.g., if your sample is 1% of the data, use $s/100$ as your support threshold.
   - You can make a complete pass through the data to verify that the frequent itemsets of the sample are truly frequent, but you will miss a set that is frequent in the whole data but not in the sample.
   - To minimize false negatives, you can lower the threshold a bit in the sample, thus finding more candidates for the full pass through the data. Risk: you will have too many candidates to fit in main memory.
2. *SON95* (Savasere, Omiecinski, and Navathe from 1995 VLDB; referenced by Toivonen). Read subsets of the data into main memory, and apply the "simple approach" to discover candidate sets. Every basket is part of one such main-memory subset. On the second pass, a set is a candidate if it was identified as a candidate in any one or more of the subsets.

- Key point: A set cannot be frequent in the entire data unless it is frequent in at least one subset.

3. *Toivonen's Algorithm*:

   (a) Take a sample that fits in main memory. Run the simple approach on this data, but with a threshold lowered so that we are unlikely to miss any truly frequent itemsets (e.g., if sample is 1% of the data, use $s/125$ as the support threshold).

   (b) Add to the candidates of the sample the *negative border*: those sets of items S such that S is *not* identified as frequent in the sample, but *every* immediate subset of S is. For example, if $ABCD$ is not frequent in the sample, but all of ABC, ABD, ACD, and BCD are frequent in the sample, then $ABCD$ is in the negative border.

   (c) Make a pass over the data, counting all the candidate itemsets and the negative border. If no member of the negative border is frequent in the full data, then the frequent itemsets are exactly those candidates that are above threshold.

   (d) Unfortunately, if there is a member of the negative border that turns out to be frequent, then we don't know whether some of its supersets are also frequent, so the whole process needs to be repeated (or we accept what we have and don't worry about a few false negatives).

3 Low-Support, High-Correlation Mining

We continue to assume a "market-basket" model for data, and we visualize the data as a boolean matrix, where rows = baskets and columns = items. Key assumptions:

1. Matrix is very sparse; almost all 0's.
2. The number of columns (items) is sufficiently small that we can store something per column in main memory, but sufficiently large that we cannot store something per pair of items in main memory (same assumption we've made in all association-rule work so far).
3. The number of rows is so large that we cannot store the entire matrix in memory, even if we take advantage of sparseness and compress (again, sames assumption as always).
4. We are not interested in high-support pairs or sets of columns; rather we want highly correlated pairs of columns.

3.1 Applications

While marketing applications generally care only about high support (it doesn't pay to try to market things that nobody buys anyway), there are several applications that meet the model above, especially the point about pairs of columns/items with low support but high correlation being interesting:

1. Rows and columns are Web pages; $(r, c) = 1$ means that the page of row r links to the page of column c. Similar columns may be pages about the same topic.
2. Same as (1), but the page of column c links to the page of row r. Now, similar columns may represent mirror pages.
3. Rows = Web pages or documents; columns = words. Similar columns are words that appear almost always together, e.g., "phrases."
4. Same as (3), but rows are sentences. Similar columns may indicate mirror pages or plagiarisms.

3.2 Similarity

Think of a column as the set of rows in which the column has a 1. Then the *similarity* of two columns C_1 and C_2 is $Sim(C_1, C_2) = |C_1 \cap C_2|/|C_1 \cup C_2|$.

Example 3.1 :

$$\begin{array}{cc} 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \end{array} \quad = 2/5 = 40\% \text{ similar}$$

□

3.3 Signatures

Key idea: map ("hash") each column C to a small amount of data [the *signature*, $Sig(C)$] such that:

1. $Sig(C)$ is small enough that a signature for each column can be fit in main memory.
2. Columns C_1 and C_2 are highly similar if and only if $Sig(C_1)$ and $Sig(C_2)$ are highly similar. (But note that we need to define "similarity" for signatures.)

An idea that doesn't work: Pick 100 rows at random, and make that string of 100 bits be the signature for each column. The reason is that the matrix is assumed sparse, so many columns will have an all-0 signature even if they are quite dissimilar.
Useful convention: given two columns C_1 and C_2, we'll refer to rows as being of four types — a, b, c, d — depending on their bits in these columns, as follows:

Type	C_1	C_2
a	1	1
b	1	0
c	0	1
d	0	0

We'll also use a as "the number of rows of type a," and so on.

- Note, $Sim(C_1, C_2) = a/(a + b + c)$.
- But since most rows are of type d, a selection of, say, 100 random rows will be all of type d, so the similarity of the columns in these 100 rows is not even defined.

3.4 Min Hashing

Imagine the rows permuted randomly in order. "Hash" each column C to $h(C)$, the number of the first row in which column C has a 1.

- The probability that $h(C_1) = h(C_2)$ is $a/(a + b + c)$, since the hash values agree if the first row with a 1 in either column is of type a, and they disagree if the first such row is of type b or c. Note this probability is the same as $Sim(C_1, C_2)$.
- If we repeat the experiment, with a new permutation of rows a large number of times, say 100, we get a signature consisting of 100 row numbers for each column. The "similarity" of these lists (fraction of positions in which they agree) will be very close to the similarity of the columns.
- Important trick: we don't actually permute the rows, which would take many passes over the entire data. Rather, we read the rows in whatever order, and hash each row using (say) 100 different hash functions. For each column we maintain the lowest hash value of a row in which that column has a 1, independently for each of the 100 hash functions. After considering all rows, we shall have for each column the first rows in which the column has 1, if the rows had been permuted in the orders given by each of the 100 hash functions.

3.5 Locality-Sensitive Hashing

Problem: we've got signatures for all the columns in main memory, and similar signatures mean similar columns, with high probability, but there still may be so many columns that doing anything that is quadratic in the number of columns, even in main memory, is prohibitive. *Locality-sensitive hashing* (LSH) is a technique to be used in main memory for approximating the set of similar column-pairs with a lot less than quadratic work.
The goal: in time proportional to the number of columns, eliminate as possible similar pairs the vast majority of the column pairs.

1. Think of the signatures as columns of integers.
2. Partition the rows of the signatures into *bands*, say l bands of r rows each.
3. Hash the columns in each band into buckets. A pair of columns is a candidate-pair if they hash to the same bucket in any band.
4. After identifying candidates, verify each candidate-pair (C_i, C_j) by examining $Sig(C_i)$ and $Sig(C_j)$ for similarity.

Example 3.2 : To see the effect of LSH, consider data with 100,000 columns, and signatures consisting of 100 integers each. The signatures take 40Mb of memory, not too much by today's standards. Suppose we want pairs that are 80% similar. We'll look at the signatures, rather than the columns, so we are really identifying columns whose *signatures* are 80% similar — not quite the same thing.

- If two columns are 80% similar, then the probability that they are identical in any one band of 5 integers is $(0.8)^5 = 0.328$. The probability that they are *not* similar in *any* of the 20 bands is $(1 - .328)^{20} = .00035$. Thus, all but about 1/3000 of the pairs with 80%-similar signatures will be identified as candidates.
- Now, suppose two columns are only 40% similar. Then the probability that they are identical in one band is $(0.4)^5 = .01$, and the probability that they are similar in at least one of the 20 bands is no more than 0.2. Thus, we can skip at least 4/5 of the pairs that will turn out not to be candidates, if 40% is the typical similarity of columns.
- In fact, most pairs of columns will be a *lot less* than 40% similar, so we really eliminate a huge fraction of the dissimilar columns.

□

3.6 k-Min Hashing

Min hashing requires that we hash each row number k times, if we want a signature of k integers. With *k-min hashing*. In *k-min hashing* we instead hash each row once, and for each column, we take the k lowest-numbered rows in which that column has a 1 as the signature.
To see why the similarity of these signatures is almost the same as the similarity of the columns from which they are derived, examine Fig. refkmin-fig. This figure represents the signatures Sig_1 and Sig_2 for columns C_1 and C_2, respectively, as if the rows were permuted in the order of their hash values, and rows of type d (neither column has 1) are omitted. Thus, we see only rows of types a, b, and c, and we indicate that a row is in the signature by a 1.

$$\begin{array}{ccc} 100\ 1\text{'s} & \left(\begin{array}{cc} \text{Sig1} & \text{Sig2} \\ 1 & 1 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & 1 \\ \hline 1 & 0 \\ 1 & 1 \end{array}\right) & 100\ 1\text{'s} \end{array}$$

Figure 4: Example of the signatures of two columns using k-min hashing

Let us assume $c \geq b$, so the typical situation (assuming $k = 100$) is as shown in Fig. 4: the top 100 rows in the first column includes some rows that are not among the top 100 rows for the second column. Then an estimate of the similarity of Sig_1 and Sig_2 can be computed as follows:

$$|Sig_1 \cap Sig_2| = \frac{100a}{a+c}$$

because on average, the fraction of the 100 top rows of C_2 that are also rows of C_1 is $a/(a+c)$. Also:

$$|Sig_1 \cup Sig_2| = 100 + \frac{100c}{a+c}$$

The argument is that all 100 rows of Sig_1 are in the union. In addition, those rows of Sig_2 that are not rows of Sig_1 are in the union, and the latter set of rows is on average $100c/(a+c)$ rows. Thus, the similarity of Sig_1 and Sig_2 is:

$$\frac{|Sig_1 \cap Sig_2|}{|Sig_1 \cup Sig_2|} = \frac{\frac{100a}{a+c}}{100 + \frac{100c}{a+c}} = \frac{a}{a+2c}$$

Note that if c is close to b, then the similarity of the signatures is close to the similarity of the columns, which is $a/(a+b+c)$. In fact, if the columns are very similar, then b and c are both small compared to a, and the similarities of the signatures and columns *must* be close.

3.7 Amplification of 1's (Hamming LSH)

If columns are not sparse, but have about 50% 1's, then we don't need min-hashing; a random collection of rows serves as a signature. *Hamming LSH* constructs a series of matrices, each with half as many rows as the previous, by OR-ing together two consecutive rows from the previous, as in Fig. 5.

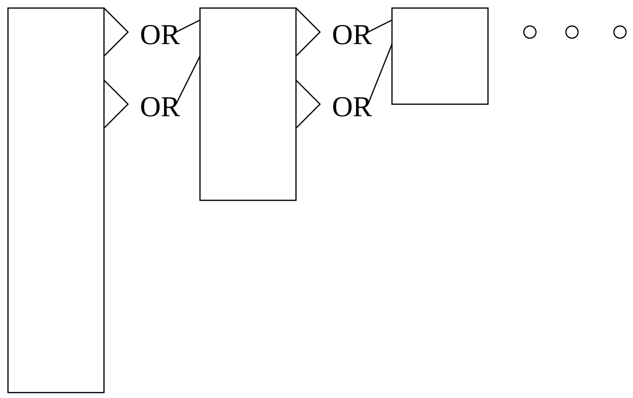


Figure 5: Construction of a series of exponentially smaller, denser matrices

- There are no more than $\log n$ matrices if n is the number of rows. The total number of rows in all matrices is $2n$, and they can all be computed with one pass through the original matrix, storing the large ones on disk.
- In each matrix, produce as *candidate pairs* those columns that:
  1. Have a medium density of 1's, say between 20% and 80%, and
  2. Are likely to be similar, based on an LSH test.
- Note that the density range 20–80% guarantees that any two columns that are at least 50% similar will be considered together in at least one matrix, unless by bad luck their relative densities change due to the OR operation combining two 1's into one.
- A second pass through the original data confirms which of the candidates are really similar.
- This method exploits an idea that can be useful elsewhere: similar columns have similar numbers of 1's, so there is no point ever comparing columns whose numbers of 1's are very different.

4 Query Flocks

Goal: apply a-priori trick and other association-rule tricks to a more general class of complex queries.

4.1 Query Flock Notation

A *query flock* is a generate-and-test system consisting of:

1. A query with parameters; we write the query in Datalog to simplify certain optimizations later.
2. A filter condition that says when the values of the parameters yields a query result that we accept.

- Note that the query flock is really a single query about its parameters; the parametrized-query component is *not* the real query.

Example 4.1: Frequent item pairs in a relation $Baskets(BID, item)$ can be written as the query flock:

```
Answer(b) <- Baskets(b,$1) AND Baskets(b,$2)

COUNT(Answer) >= s
```

If we replace parameters \$1 and \$2 by values, e.g., "diapers" and "beer," respectively, then the query is asking for the set of basket ID's such that the basket contains both diapers and beer. The condition on the answer says that there must be at least s such baskets, where s is the support threshold. Thus, this query flock asks the usual question about the parameters \$1 and \$2: "which pairs of items appear in at least s baskets?" □

Example 4.2: Here is a less usual example. It supposes relations:

1. $Cust(name, attr, value)$. Tuple (n, a, v) means the customer with name n has value v for attribute a. For instance, $(Sue, age, 45)$ means that Sue is of age 45.
2. $Buys(name, prod)$ tells what products each customer buys.
3. $Type(prod, type)$ tells the type of each product, e.g., product "Coke" is of type "soft drink."

Here is the query flock that asks for values of some attribute that occur at least s times among buyers of a certain type of product:

```
Answer(n) <- Cust(n,$a,$v) AND Buys(n,p) AND Type(p,$t)

COUNT(Answer) >= s
```

□

4.2 Execution Strategies

The analog of a-priori is the observation that if we delete one or more subgoals from a Datalog query, the size of the set of answers can only increase. Our hope is that by computing some temporary relations using a subset of the subgoals, we can filter the sets of values for one or more parameters, using computations that are much less expensive than computing the entire query about the full set of parameters.
We can describe the intermediate steps, as well as the final computation of the parameter-values that pass the test by a sequence of steps of the form

```
<Relation> := FILTER(<parameters>, <query>, <condition>)
```

- The query is the flock query, with zero or more subgoals eliminated. A requirement is that this query be *safe*; i.e., every variable appearing in the head appears in a nonnegated subgoal involving a relation (i.e., not a subgoal involving an arithmetic comparison like $a < b$).

- The parameters are those appearing in the query.
- The condition is the same as the condition of the flock itself.

Example 4.3: The flock of Example 4.1 might be solved by using the first subgoal to filter \$1 and the second subgoal to filter \$2.

```
OK1($1) := FILTER({$1}, Answer(b) <- Baskets(b,$1), COUNT(Answer) >= s)
OK2($2) := FILTER({$2}, Answer(b) <- Baskets(b,$2), COUNT(Answer) >= s)
OK($1,$2) := FILTER({$1,$2}, Answer(b) <- Baskets(b,$1) AND
                             Baskets(b,$2) AND OK1($1) AND OK2($2),
                     COUNT(Answer) >= s)
```

- Of course a clever flocks compiler recognizes that these two filtering steps are really the same and only computes one of OK1 and OK2.
- The reason a-priori often saves a lot of time is because the join of four relations at the last step [computation of $OK(\$1,\$2)$] can be carried out in an order that reduces the size of intermediate relations, when compared with just joining *Baskets* with itself, as suggested by the ordering of Fig. 6.

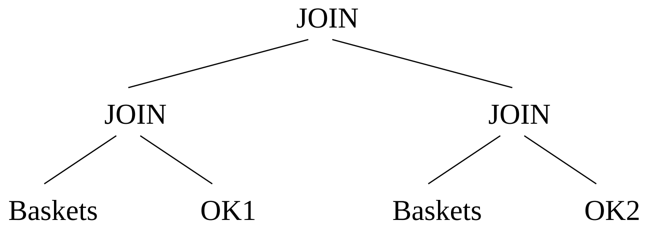


Figure 6: Preferred order for join in market-basket flock

- Notice that the ordering in Fig. 6 is not a left-deep ordering, which suggests that the typical commercial DBMS would *not* find this order, and a query-flocks compiler needs to feed simpler queries to the DBMS so the right order of join is used by the DBMS.

□

Example 4.4: Now let us consider how we might use filter steps to improve the running time of the final join in Example 4.2. Using just the *Cust* subgoal is a filter on $\{\$a, \$v\}$, but there is no useful filter for just one of these parameters. We cannot use:

```
Answer(n) <- Type(p,$t)
```

to filter \$t, because the query is not safe (n appears in the head but not the body). However,

```
Answer(n) <- Buys(n,p) AND Type(p,$t)
```

is safe and may be used. A possible plan for optimizing this query flock is in Fig. 7. Figure 8 shows the preferred join order for the final step. □

```
OK1($a,$v) := FILTER({$a,$v}, Answer(n) <- Cust(n,$a,$v),
                     COUNT(Answer) >= s)
OK2($t) := FILTER({$t}, Answer(n) <- Buys(n,p) AND Type(p,$t),
                     COUNT(Answer) >= s)
OK($a,$v,$t) := FILTER({$a,$v,$t}, Answer(n) <- Cust(n,$a,$v), AND
                                   Buys(n,p) AND Type(p,$t) AND
                                   OK1($a,$v) AND OK2($t),
                     COUNT(Answer) >= s)
```

Figure 7: Query-flock plan for Example 4.2

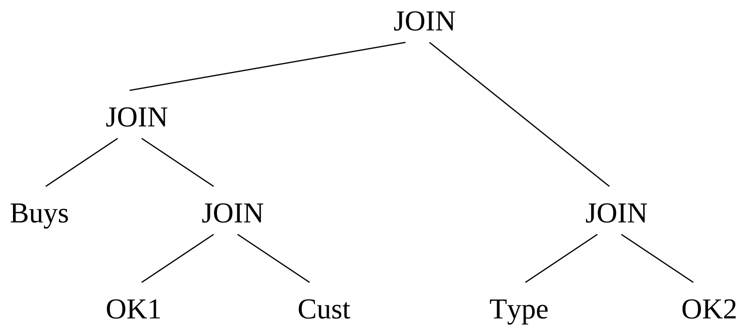


Figure 8: Join order for final step in Fig. 7

5 Web Search

Outline:

1. *Page rank*, for discovering the most "important" pages on the Web, as used in Google.
2. *Hubs and authorities*, a more detailed evaluation of the importance of Web pages using a variant of the eigenvector calculation used for Page rank.

5.1 Page Rank

Intuitively, we solve the recursive definition of "importance": a page is important if important pages link to it.
Create a stochastic matrix of the Web; that is:

1. Each page i corresponds to row i and column i of the matrix.
2. If page j has n successors (links), then the ijth entry is $1/n$ if page i is one of these n successors of page j, and 0 otherwise.

The intuition behind this matrix is:

- Imagine that initially each page has one unit of importance. At each round, each page shares whatever importance it has among its successors, and receives new importance from its predecessors.
- Eventually, the importance of each page reaches a limit, which happens to be its component in the principal eigenvector of this matrix.
- That importance is also the probability that a Web surfer, starting at a random page, and following random links from each page will be at the page in question after a long series of links.

Example 5.1: In 1839, the Web consisted on only three pages — Netscape, Microsoft, and Amazon. The links among these pages were as shown in Fig. 9.

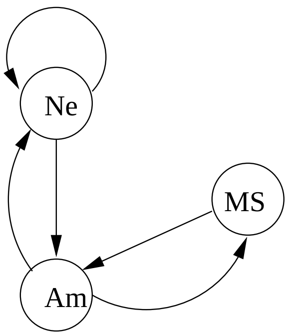


Figure 9: The Web in 1839

Let $[n, m, a]$ be the vector of importances for the three pages: Netscape, Microsoft, Amazon, in that order. Then the equation describing the asymptotic values of these three variables is:

$$\begin{bmatrix} n \\ m \\ a \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 \\ 1/2 & 1 & 0 \end{bmatrix} \begin{bmatrix} n \\ m \\ a \end{bmatrix}$$

For example, the first column of the matrix reflects the fact that Netscape divides its importance between itself and Amazon. The second column indicates that Microsoft gives all its importance to Amazon.
We can solve equations like this one by starting with the assumption $n = m = a = 1$, and applying the matrix to the current estimate of these values repeatedly. The first four iterations give the following estimates:

$$\begin{array}{lcllll} n & = & 1 & 1 & 5/4 & 9/8 & 5/4 \\ m & = & 1 & 1/2 & 3/4 & 1/2 & 11/16 \\ a & = & 1 & 3/2 & 1 & 11/8 & 17/16 \end{array}$$

In the limit, the solution is $n = a = 6/5$; $m = 3/5$. That is, Netscape and Amazon each have the same importance, and twice the importance of Microsoft (well this was 1839). □

- Note that we can never get absolute values of n, m, and a, just their ratios, since the initial assumption that they were each 1 was arbitrary.
- Since the matrix is *stochastic* (sum of each column is 1), the above *relaxation* process converges to the principal eigenvector.

5.2 Problems With Real Web Graphs

1. *Dead ends*: a page that has no successors has nowhere to send its importance. Eventually, all importance will "leak out of" the Web.
2. *Spider traps*: a group of one or more pages that have no links out of the group will eventually accumulate all the importance of the Web.

Example 5.2: Suppose Microsoft tries to duck charges that it is a monopoly by removing all links from its site. The new Web is as shown in Fig. 10, and the matrix describing transitions is:

$$\begin{bmatrix} n \\ m \\ a \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 \\ 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} n \\ m \\ a \end{bmatrix}$$

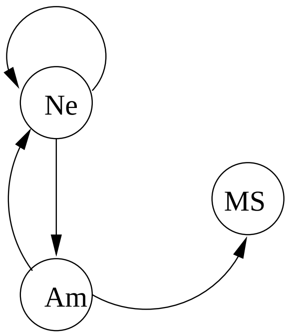


Figure 10: Microsoft becomes a dead end

The first four steps of the iterative solution are:

$$\begin{array}{lcllll} n & = & 1 & 1 & 3/4 & 5/8 & 1/2 \\ m & = & 1 & 1/2 & 1/4 & 1/4 & 3/16 \\ a & = & 1 & 1/2 & 1/2 & 3/8 & 5/16 \end{array}$$

Eventually, each of n, m, and a become 0; i.e., all the importance leaked out. □

Example 5.3: Angered by the decision, Microsoft decides it will link only to itself from now on. Now, Microsoft has become a spider trap. The new Web is in Fig. 11, and the equation to solve is:

$$\begin{bmatrix} n \\ m \\ a \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 1 & 1/2 \\ 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} n \\ m \\ a \end{bmatrix}$$

The first steps of the solution are:

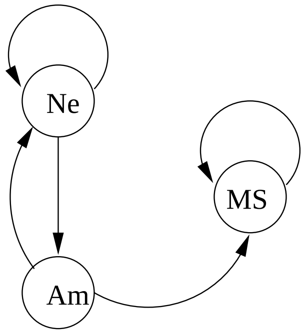


Figure 11: Microsoft becomes a spider trap

$$
\begin{array}{lcllllll}
n & = & 1 & 1 & 3/4 & 5/8 & 1/2 \\
m & = & 1 & 3/2 & 7/4 & 2 & 35/16 \\
a & = & 1 & 1/2 & 1/2 & 3/8 & 5/16
\end{array}
$$

Now, m converges to 3, and $n = a = 0$. □

5.3 Google Solution to Dead Ends and Spider Traps

Instead of applying the matrix directly, "tax" each page some fraction of its current importance, and distribute the taxed importance equally among all pages.

Example 5.4: If we use a 20% tax, the equation of Example 5.3 becomes:

$$
\begin{bmatrix} n \\ m \\ a \end{bmatrix} = 0.8 \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 1 & 1/2 \\ 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} n \\ m \\ a \end{bmatrix} + \begin{bmatrix} 0.2 \\ 0.2 \\ 0.2 \end{bmatrix}
$$

The solution to this equation is $n = 7/11$; $m = 21/11$; $a = 5/11$.

- Note that the sum of the three values is not 3, but there is a more reasonable distribution of importance than in Example 5.3.

□

5.4 Google Anti-Spam Devices

"Spamming" is the attempt by many Web sites to appear to be about a subject that will attract surfers, without truly being about that subject.

- Google, like other search engines, tries to match the words in your query to the words on the Web pages. However, Google, unlike other engines tends to believe what others say about you in their anchor text, making it harder for you to *appear* to be about something you are not.
- The use of Page rank to measure importance, rather than the more naive "number of links into the page" also protects against spammers. The naive measure can be fooled by the spammer who creates 1000 pages that mutually link to one another, while Page rank recognizes that none of the pages have any real importance.

5.5 Hubs and Authorities

Intuitively, we define "hub" and "authority" in a mutually recursive way: a hub links to many authorities, and an authority is linked to by many hubs.

- Authorities turn out to be pages that offer information about a topic, e.g., the Quest home page about the IBM data-mining project.
- Hubs are pages that don't provide the information, but tell you where to find the information, e.g., the CS345 home page.
- Uses a matrix formulation similar to that of Page rank, but without the stochastic restriction. We count each link as 1, regardless of how many successors or predecessors a page has.
- Repeated application of the matrix leads to divergence, but we can introduce scaling factors and keep the computed values of "authority" and "hubbiness" for each page within finite bounds.

Define a matrix A whose rows and columns correspond to Web pages, with entry $A_{ij} = 1$ if page i links to page j, and 0 if not.

- Notice that A^T, the transpose of A, looks like the matrix used for computing Page rank, but A^T has 1's where the Page-rank matrix has fractions.

Let $\vec{a}$ and $\vec{h}$ be vectors, whose ith component corresponds to the degrees of authority and hubbiness of the ith page. let λ and μ be suitable scaling factors to be determined later. Then we can state:

1. $\vec{h} = \lambda A\vec{a}$. That is, the hubbiness of each page is the sum of the authorities of all the pages it links to, scaled by λ.
2. $\vec{a} = \mu A^T\vec{h}$. That is, the authority of each page is the sum of the hubbiness of all the pages that link to it, scaled by μ.

We can derive from (1) and (2), using simple substitution, two equations that relate vectors $\vec{a}$ and $\vec{h}$ only to themselves:

$$\vec{a} = \lambda\mu A^T A\vec{a};\ \vec{h} = \lambda\mu AA^T\vec{h}$$

As a result, we can compute $\vec{h}$ and $\vec{a}$ by relaxation, giving us the principal eigenvectors of the matrices AA^T and A^TA, respectively.

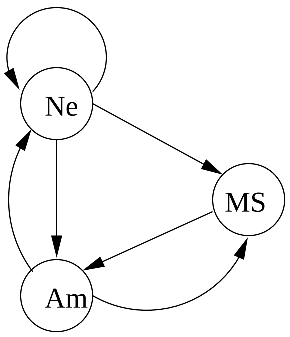


Figure 12: Web for Example 5.5

Example 5.5: Consider the Web of Fig. 12. The relevant matrices are:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad AA^T = \begin{bmatrix} 3 & 1 & 2 \\ 1 & 1 & 0 \\ 2 & 0 & 2 \end{bmatrix} \quad A^TA = \begin{bmatrix} 2 & 2 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

If we use $\lambda = \mu = 1$ and assume that the vectors $\vec{h} = [h_n, h_m, h_a]$ and $\vec{a} = [a_n, a_m, a_a]$ are each initially $[1, 1, 1]$, the first three iterations of the equations for $\vec{a}$ and $\vec{h}$ are:

$$
\begin{array}{lllll}
a_n & = & 1 & 5 & 24 & 114 \\
a_m & = & 1 & 5 & 24 & 114 \\
a_a & = & 1 & 4 & 18 & 84 \\
\\
h_n & = & 1 & 6 & 28 & 132 \\
h_m & = & 1 & 2 & 8 & 36 \\
h_a & = & 1 & 4 & 20 & 96
\end{array}
$$

For instance, the vector $\vec{a}$, properly scaled, will converge to a vector where $a_n = a_m$, and each of these is greater than a_a in the ratio $1 + \sqrt{3}\ :\ 2$, or about 1.36. $\square$

6 Mining the Web

Outline:

1. *Dynamic itemset counting*: Searching for *interesting* sets of items in a space too large ever to consider even each pair of items.

2. *"Books and authors"*: Sergey Brin's intriguing experiment to mine the Web for relational data.

6.1 Finding Unusual Itemsets

The problem is to find sets of words that appear together "unusually often" on the Web, e.g., "New" and "York" or {"Dutchess", "of", "York"}.

- "Unusually often" can be defined in various ways, in order to capture the idea that the number of Web documents containing the set of words is much greater than what one would expect if words were sprinkled at random, each word with its own probability of occurrence in a document.

- One appropriate way is *entropy per word in the set.* Formally, the *interest* of a set of words S is

$$\frac{\log_2\left(\frac{prob(S)}{\prod_{w\ in\ S} prob(w)}\right)}{|S|}$$

  Note that we divide by the size of S to avoid the "Bonferroni effect," where there are so many sets of a given size that some, by chance alone, will appear to be correlated.

- Example: If words a, b, and c each appear in 1% of all documents, and $S = \{a, b, c\}$ appears in 0.1% of documents, then the interestingness of S is $\big(\log_2(.001/(.01 \times .01 \times .01))\big)/3 = \log_2(1000)/3$ or about 3.3.

- Technical problem: interest is not monotone, or "downwards closed," the way high support is. That is, we can have a set S with a high value of interest, yet some, or even all, of its immediate proper subsets are not interesting. In contrast, if S has high support, then all of its subsets have support at least as high.

- Technical problem: With more than 10^8 different words appearing in the Web, it is not possible even to consider all pairs of words.

6.2 The DICE Engine

DICE (dynamic itemset counting engine) repeatedly visits the pages of the Web, in a round-robin fashion. At all times, it is counting occurrences of certain sets of words, and of the individual words in that set. The number of sets being counted is small enough that the counts fit in main memory.
From time to time, say every 5000 pages, DICE reconsiders the sets that it is counting. It throws away those sets that have the lowest interest, and replaces them with other sets.
The choice of new sets is based on the *heavy edge property*, which is an experimentally justified observation that those words that appear in a high-interest set are more likely than others to appear in other high-interest sets. Thus, when selecting new sets to start counting, DICE is biased in favor of words that already appear in high-interest sets. However, it does not rely on those words exclusively, or else it could never find high-interests sets composed of the many words it has never looked at. Some (but not all) of the constructions that DICE uses to create new sets are:

1. Two random words. This is the only rule that is independent of the heavy edge assumption, and helps new words get into the pool.

2. A word in one of the interesting sets and one random word.

3. Two words from two different interesting pairs.

4. The union of two interesting sets whose intersection is of size 2 or more.

5. $\{a, b, c\}$ if all of $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$ are found to be interesting.

Of course, there are generally too many options to do all of the above in all possible ways, so a random selection among options, giving some choices to each of the rules, is used.

6.3 Books and Authors

The general idea is to search the Web for facts of a given type, typically what might form the tuples of a relation such as $Books(title, author)$. The computation is suggested by Fig. 13.

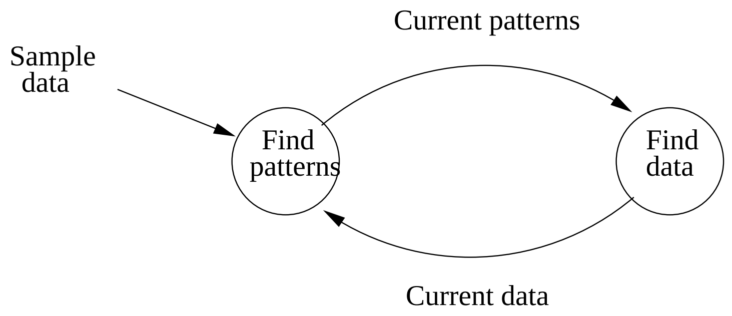


Figure 13: Extracting relations from the Web

1. Start with a sample of the tuples one would like to find. In the example discussed in the Brin paper, five examples of book titles and their authors were used.

2. Given a set of known examples, find where that data appears on the Web. If a pattern is found that identifies several examples of known tuples, and is sufficiently specific that it is unlikely to identify too much, then accept this pattern.

3. Given a set of accepted patterns, find the data that appears in these patterns, add it to the set of known data.

4. Repeat steps (2) and (3) several times. In the example cited, four rounds were used, leading to 15,000 tuples; about 95% were true title-author pairs.

6.4 What is a Pattern?

The notion suggested consists of five elements:

1. The *order*; i.e., whether the title appears prior to the author in hte text, or vice-versa. In a more general case, where tuples have more than 2 components, the order would be the permutation of components.

2. The *URL prefix*.

3. The *prefix* of text, just prior to the first of the title or author.

4. The *middle*: text appearing between the two data elements.

5. The *suffix* of text following the second of the two data elements. Both the prefix and suffix were limited to 10 characters.

Example 6.1 : A possible pattern might consist of the following:

1. Order: title then author.
2. URL prefix: `www.stanford.edu/class/`
3. Prefix, middle, and suffix of the following form:

   ```
   <LI><I>title</I> by author<P>
   ```

   Here the prefix is `<LI><I>`, the middle is `</I> by ` (including the blank after "by"), and the suffix is `<P>`. The title is whatever appears between the prefix and middle; the author is whatever appears between the middle and suffix.

The intuition behind why this pattern might be good is that there are probably lots of reading lists among the class pages at Stanford. □

To focus on patterns that are likely to be accurate, Brin used several constraints on patterns, as follows:

- Let the *specificity* of a pattern be the product of the lengths of the prefix, middle, suffix, and URL prefix. Roughly, the specificity measures how likely we are to find the pattern; the higher the specificity, the fewer occurrences we expect.
- Then a pattern must meet two conditions to be accepted:
  1. There must be at least 2 known data items that appear in this pattern.
  2. The product of the specificity of the pattern and the number of occurrences of data items in the pattern must exceed a certain threshold T (not specified).

6.5 Data Occurrences

An occurrence of a tuple is associated with a pattern in which it occurs; i.e., the same title and author might appear in several different patterns. Thus, a data occurrence consists of:

1. The particular title and author.
2. The complete URL, not just the prefix as for a pattern.
3. The order, prefix, middle, and suffix of the pattern in which the title and author occurred.

6.6 Finding Data Occurrences Given Data

If we have some known title-author pairs, our first step in finding new patterns is to search the Web to see where these titles and authors occur. We assume that there is an index of the Web, so given a word, we can find (pointers to) all the pages containing that word. The method used is essentially a-priori:

1. Find (pointers to) all those pages containing any known author. Since author names generally consist of 2 words, use the index for each first name and last name, and check that the occurrences are consecutive in the document.
2. Find (pointers to) all those pages containing any known title. Start by finding pages with each word of a title, and then checking that the words appear in order on the page.
3. Intersect the sets of pages that have an author and a title on them. Only these pages need to be searched to find the patterns in which a known title-author pair is found. For the prefix and suffix, take the 10 surrounding characters, or fewer if there are not as many as 10.

6.7 Building Patterns from Data Occurrences

1. Group the data occurrences according to their order and middle. For example, one group in the "group-by" might correspond to the order "title-then-author" and the middle "`</I> by` .
2. For each group, find the longest common prefix, suffix, and URL prefix.
3. If the specificity test for this pattern is met, then accept the pattern.
4. If the specificity test is *not* met, then try to split the group into two by extending the length of the URL prefix by one character, and repeat from step (2). If it is impossible to split the group (because there is only one URL) then we fail to produce a pattern from the group.

Example 6.2: Suppose our group contains the three URL's:

```
www.stanford.edu/class/cs345/index.html
www.stanford.edu/class/cs145/intro.html
www.stanford.edu/class/cs140/readings.html
```

The common prefix is `www.stanford.edu/class/cs` . If we have to split the group, then the next character, 3 versus 1, breaks the group into two, with those data occurrences in the first page (there could be many such occurrences) going into one group, and those occurrences on the other two pages going into another. □

6.8 Finding Occurrences Given Patterns

1. Find all URL's that match the URL prefix in at least one pattern.
2. For each of those pages, scan the text using a regular expression built from the pattern's prefix, middle, and suffix.
3. Extract from each match the title and author, according the order specified in the pattern.

7 Clustering

Given points in some space — often a high-dimensional space — group the points into a small number of *clusters*, each cluster consisting of points that are "near" in some sense. Some applications:

1. Many years ago, during a cholera outbreak in London, a physician plotted the location of cases on a map, getting a plot that looked like Fig. 14. Properly visualized, the data indicated that cases clustered around certain intersections, where there were polluted wells, not only exposing the cause of cholera, but indicating what to do about the problem. Alas, not all data mining is this easy, often because the clusters are in so many dimensions that visualization is very hard.

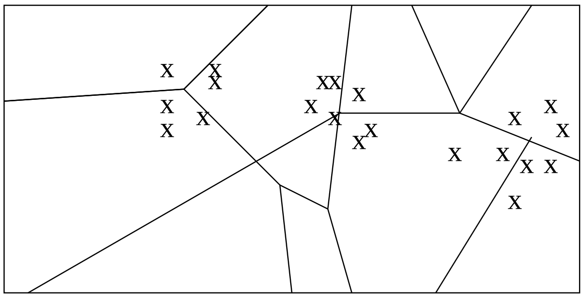


Figure 14: Clusters of cholera cases indicated where the polluted wells were

2. *Skycat* clustered 2×10^9 sky objects into stars, galaxies, quasars, etc. Each object was a point in a space of 7 dimensions, with each dimension representing radiation in one band of the spectrum. The Sloan Sky Survey is a more ambitious attempt to catalog and cluster the entire visible universe.

3. Documents may be thought of as points in a high-dimensional space, where each dimension corresponds to one possible word. The position of a document in a dimension is the number of times the word occurs in the document (or just 1 if it occurs, 0 if not). Clusters of documents in this space often correspond to groups of documents on the same topic.

7.1 Distance Measures

To discuss whether a set of points is close enough to be considered a cluster, we need a *distance measure* $D(x, y)$ that tells how far points x and y are. The usual axioms for a distance measure D are:

1. $D(x, x) = 0$. A point is distance 0 from itself.

2. $D(x, y) = D(y, x)$. Distance is symmetric.

3. $D(x, y) \le D(x, z) + D(z, y)$. The *triangle inequality*.

Often, our points may be thought to live in a k-dimensional Euclidean space, and the distance between any two points, say $x = [x_1, x_2, \ldots, x_k]$ and $y = [y_1, y_2, \ldots, y_k]$ is given in one of the usual manners:

1. Common distance ("L_2 norm"): $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$.

2. *Manhattan distance* ("L_1 norm"): $\sum_{i=1}^{k} |x_i - y_i|$.

3. Max of dimensions ("L_∞ norm"): $\max_{i=1}^{k} |x_i - y_i|$.

When there is no Euclidean space in which to place the points, clustering becomes more difficult. Here are some examples where a distance measure without a Euclidean space makes sense.

Example 7.1: We can think of Web pages as points in the roughly 10^8-dimensional space where each dimension corresponds to one word. However, computation of distances would be prohibitive in a space this large. A better approach is to base the distance $D(x, y)$ on the dot product of vectors corresponding to x and y, since we then have to deal only with the words actually present in both x and y.
We need to compute the lengths of the vectors involved, which is the square root of the sum of the squares of the numbers of occurrences of each word. The sum of the product of the occurrences of each word in each document is divided by each of the lengths to get a normalized dot product. We subtract this quantity from 1 to get the distance between x and y.
For instance, suppose there were only 4 words of interest, and the vectors $x = [2, 0, 3, 1]$ and $y = [5, 3, 2, 0]$ represented the numbers of occurrences of these words in the two documents. The dot product $x.y$ is $2 \times 5 + 0 \times 3 + 3 \times 2 + 1 \times 0 = 16$, the length of the first vector is $\sqrt{2^2 + 0^2 + 3^2 + 1^2} = \sqrt{14}$, and the length of the second is $\sqrt{5^2 + 3^2 + 2^2 + 0^2} = \sqrt{38}$. Thus,

$$D(x, y) = 1 - \frac{16}{\sqrt{14}\sqrt{38}} = 0.304$$

As another example, suppose document x has word-vector $[a_1, a_2, \ldots]$, and y is two copies of x; i.e., $y = [2a_1, 2a_2, \ldots]$. Then

$$D(x, y) = 1 - \frac{\sum_i 2a_i^2}{\sqrt{\sum_i a_i^2}\sqrt{\sum_i (2a_i)^2}} = 0$$

That is, x and y are essentially the same document; surely they are on the same topic, and deserve to be clustered together. □

Example 7.2: Character strings, such as DNA sequences may be similar even though there are some insertions and deletions as well as changes in some characters. For instance, *abcde* and *bcdxye* are rather similar, even though they don't have any positions in common, and don't even have the same length. Thus, instead of trying to construct a Euclidean space with one dimension for each position, we can define the distance function $D(x, y) = |x| + |y| - 2|LCS(x, y)|$, where LCS stands for the longest common subsequence of x and y. In our example, $LCS(abcde, bcdxye)$ is $bcde$, of length 4, so $D(abcde, bcdxye) = 5 + 6 - 2 \times 4 = 3$; i.e., the strings are fairly close. □

7.2 The Curse of Dimensionality

An unintuitive consequence of working in a high-dimensional space is that almost all pairs of points are about as far away as average.

Example 7.3: Suppose we throw points at random into a k-dimensional unit cube. If $k = 2$, we expect that the points will spread out in the plane, with some very nearby points and some pairs at almost the maximum possible distance, $\sqrt{2}$.
However, suppose k is large, say 100, or 100,000. Regardless of which norm we use, L_2, L_1, or L_∞, we know that $D(x, y) \geq \max_i |x_i - y_i|$, if $x = [x_1, x_2, \ldots]$ and $y = [y_1, y_2, \ldots]$. For large k, it is very likely that there will be some dimension i such that x_i and y_i are almost as different as possible, even if x and y are very close in other dimensions. Thus, $D(x, y)$ is going to be very close to 1. □

Another interesting consequence of high-dimensionality is that all vectors, such as $x = [x_1, x_2, \ldots]$ and $y = [y_1, y_2, \ldots]$ are almost orthogonal. The reason is that if we project x and y onto any of the $\binom{k}{2}$ planes formed by two of the k axes, there is going to be one in which the projected vectors are almost orthogonal.

7.3 Approaches to Clustering

At a high level, we can divide clustering algorithms into two broad classes:

1. *Centroid* approaches. We guess the centroids or central point in each cluster, and assign points to the cluster of their nearest centroid.

2. *Hierarchical* approaches. We begin assuming that each point is a cluster by itself. We repeatedly merge nearby clusters, using some measure of how close two clusters are (e.g., distance between their centroids), or how good a cluster the resulting group would be (e.g., the average distance of points in the cluster from the resulting centroid).

7.4 Outline

We shall consider the following algorithms for clustering; they differ in whether or not they assume a Euclidean distance, and in whether they use a centroid or hierarchical approach.

1. BFR: Centroid based; assumes Euclidean measure, with clusters formed by a Gaussian process in each dimension around the centroid.

2. Fastmap: Not really a clustering algorithm, but a way to construct a low-dimensional Euclidean space from an arbitrary distance measure.

3. GRGPF: Centroid-based, but uses only a distance measure, not a Euclidean space.

4. CURE: Hierarchical and Euclidean, this algorithm deals with odd-shaped clusters.

Note that all the algorithms we consider are oriented toward clustering large amounts of data, in particular, data so large it does not fit in main memory. Thus, we are especially concerned with the number of passes through the data that must be made, preferably one pass.

7.5 The k-Means Algorithm

This algorithm is a popular main-memory algorithm, on which the BFR algorithm is based. k-means picks k cluster centroids and assigns points to the clusters by picking the closest centroid to the point in question. As points are assigned to clusters, the centroid of the cluster may migrate.

Example 7.4: For a very simple example of five points in two dimensions, observe Fig. 15. Suppose we assign the points 1, 2, 3, 4, and 5 in that order, with $k = 2$. Then the points 1 and 2 are assigned to the two clusters, and become their centroids for the moment.

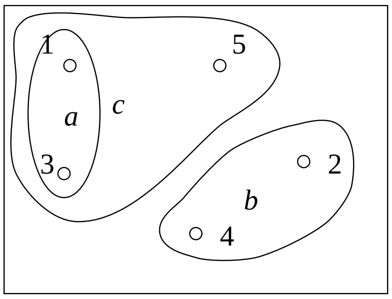


Figure 15: An example of the k-means algorithm

When we consider point 3, suppose it is closer to 1, so 3 joins the cluster of 1, whose centroid moves to the point indicated as a. Suppose that when we assign 4, we find that 4 is closer to 2 than to a, so 4 joins 2 in its cluster, whose center thus moves to b. Finally, 5 is closer to a than b, so it joins the cluster $\{1, 3\}$, whose centroid moves to c. □

- We can initialize the k centroids by picking points sufficiently far away from any other centroid, until we have k.

- As computation progresses, we can decide to split one cluster and merge two, to keep the total at k. A test for whether to do so might be to ask whether so doing reduces the average distance from points to their centroids.

- Having located the centroids of the k clusters, we can reassign all points, since some points that were assigned early may actually wind up closer to another centroid, as the centroids move about.
- If we are not sure of k, we can try different values of k until we find the smallest k such that increasing k does not much decrease the average distance of points to their centroids. Example 7.5 illustrates this point.

Example 7.5: Consider the data suggested by Fig. 16. Clearly, $k = 3$ is the right number of clusters, but suppose we first try $k = 1$. Then all points are in one cluster, and the average distance to the centroid will be high.

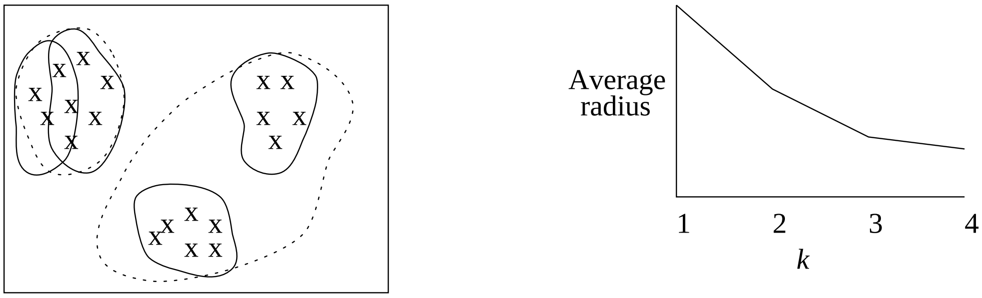


Figure 16: Discovering that $k = 3$ is the right number of clusters

Suppose we then try $k = 2$. One of the three clusters will be by itself and the other two will be forced into one cluster, as suggested by the dotted lines. The average distance of points to the centroid will thus shrink considerably.
If $k = 3$, then each of the apparent clusters should be a cluster by itself, and the average distance from points to their centroids shrinks again, as indicated by the graph in Fig. 16. However, if we increase k to 4, then one of the true clusters will be artificially partitioned into two nearby clusters, as suggested by the solid lines. The average distance to centroid will drop a bit, but not much. It is this failure to drop further that tips us off that $k = 3$ is right, even if the data is in so many dimensions that we cannot visualize the clusters. □

7.6 The BFR Algorithm

Based on k-means, this algorithm reads its data once, consuming a main-memory-full at a time. The algorithm works best if the clusters are normally distributed around a central point, perhaps with a different standard deviation in each dimension. Figure 17 suggests what the data belonging to a typical cluster in two-dimensions might look like. A centroid, marked by +, has points scattered around, with the standard deviation σ in the horizontal dimension being twice what it is in the vertical dimension. About 70% of the points will lie within the 1σ ellipse; 95% will lie within 2σ, 99.9% within 3σ, and 99.9999% within 4σ.

7.7 Representation of Clusters in BFR

Having worked on Skycat, Usama Fayyad (the "F" in BFR) probably thought of clusters as "galaxies," as suggested in Fig. 18. A cluster consists of:

1. A central core, the *Discard set* (DS). This set of points is considered certain to belong to the cluster. All the points in this set are replaced by some simple statistics, described below. Note: although called "discarded" points, these points in truth have a significant effect throughout the running of the algorithm, since they determine collectively where the centroid is and what the standard deviation of the cluster is in each dimension.

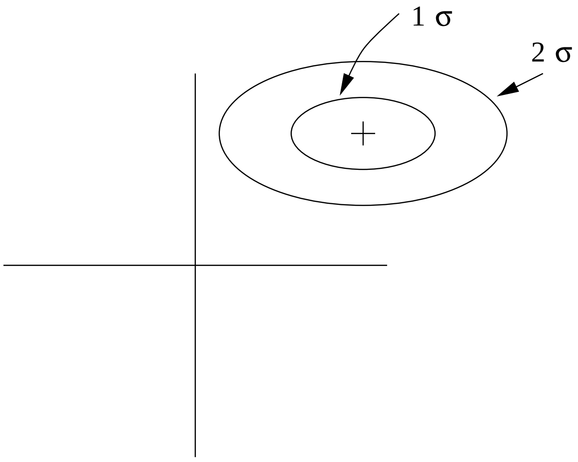


Figure 17: Points in a cluster are assumed to have been generated by taking a centroid for the cluster and adding to it in each dimension a normally distributed random variable with mean 0

2. Surrounding subgalaxies, collectively called the *Compression set* (CS). Each subcluster in the CS consists of a group of points that are sufficiently close to each other that they can be replaced by their statistics, just like the DS for a cluster is. However, they are sufficiently far away from any cluster's centroid, that we are not yet sure which cluster they belong to.

3. Individual stars that are not part of a galaxy or subgalaxy, the *Retained set* (RS). These points can neither be assigned to any cluster nor can they be grouped into a subcluster of the CS. They are stored in main memory, as individual points, along with the statistics of the DS and CS.

The statistics used to represent each cluster of the DS and each subcluster of the CS are:

1. The count of the number of points, N.

2. The vector of sums of the coordinates of the points in each dimension. The vector is called SUM, and the component in the ith dimension is SUM_i.

3. The vector of sums of squares of the coordinates of the points in each dimension, called $SUMSQ$. The component in dimension i is $SUMSQ_i$.

Note that these three pieces of information, totaling $2k+1$ numbers if there are k dimensions, are sufficient to compute important statistics of a cluster or subcluster, and they are more convenient to maintain as points are added to clusters than would, say, be the mean and variance in each dimension. For instance:

- The coordinate μ_i of the centroid of the cluster in dimension i is SUM_i/N.
- The variance in dimension i is

$$\frac{SUMSQ_i}{N} - \left(\frac{SUM_i}{N}\right)^2$$

  and the standard deviation σ_i is the square root of that.

7.8 Processing a Main-Memory-Full of Points in BFR

With the first load of main memory, BFR selects the k cluster centroids, using some chosen main-memory algorithm, e.g., pick a sample of points, optimize the clusters exactly, and chose their centroids as the initial centroids. The entire main-memory full of points is then processed like any subsequent memory-load of points, as follows:

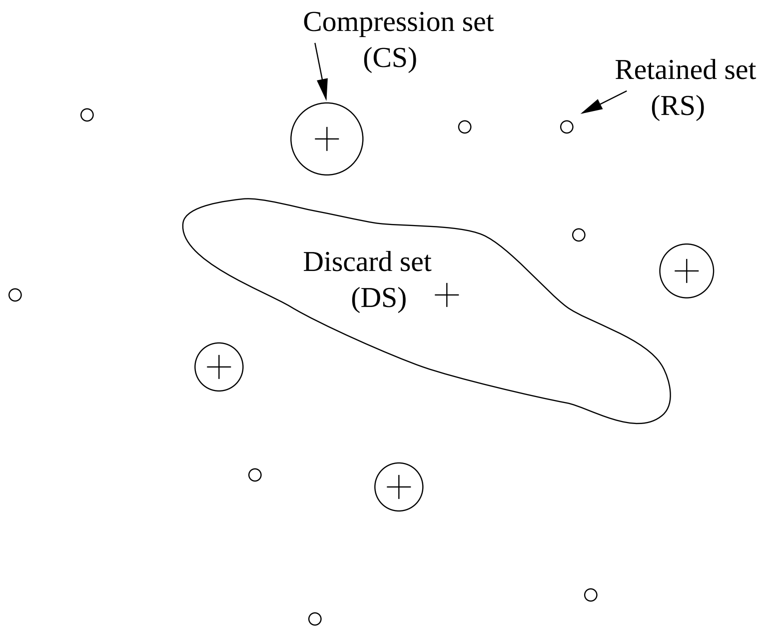


Figure 18: Examples of a cluster (DS), subclusters (CS), and individual points (RS)

1. Determine which points are sufficiently close to a current centroid that they may be taken into the DS and their statistics (N, SUM, $SUMSQ$) combined with the prior statistics of the cluster. BFR suggests two ways to determine whether a point is close enough to a centroid to be taken into the DS:

   (a) Take all points whose *Mahalanobis radius* is below a certain threshold, say four times the standard deviation of the cluster. The Mahalanobis radius is essentially the distance from the centroid, scaled in each dimension by σ_i, the standard deviation in that dimension. More precisely, if μ_i is the mean in dimension i, then the radius of point $y = [y_1, y_2, \ldots]$ is

   $$\sqrt{\sum_i \left(\frac{y_i - \mu_i}{\sigma_i}\right)^2}$$

   (b) Based on the number of points in the various clusters, ask whether it is likely (say at the 95% level) that the currently closest centroid will in the future move sufficiently far away from the point y, and another centroid will move sufficiently close to y that the latter is then closer to y than the former. If it is unlikely that the closest centroid for y will ever be different, then assign y to the DS, and place it in the cluster of the closest centroid.

2. Adjust the statistics N, SUM and $SUMSQ$ for each cluster to include the DS points just included.

3. In main memory, attempt to cluster the points that have not yet been placed in the DS, including points of the RS from previous rounds. If we find a cluster of points whose variance is below a chosen threshold, then we shall regard these points as a subcluster, replace them by their statistics, and consider them part of the CS. All other points are placed in the RS.

4. Consider merging a new subcluster with a previous subcluster of the CS. The test for whether it is desirable to do so is that the combined set of points will have a variance below a threshold. Note that the statistics kept for CS subclusters is sufficient to compute the variance of the combined set.

5. If we are at the last round, i.e., there is no more data, then we can assign subclusters in CS and points in RS to their nearest cluster, even though they will of necessity be fairly far away from any cluster centroid.

8 More About Clustering

We continue our discussion of large-scale clustering algorithms, covering:

1. Fastmap, and other ways to create a Euclidean space from an arbitrary distance measure.

2. The GRGPF algorithm for clustering without a Euclidean space.

3. The CURE algorithm for clustering odd-shaped clusters in a Euclidean space.

8.1 Simple Approaches to Building a Euclidean Space From a Distance Measure

Any n points can be placed in $(n-1)$-dimensional space, with distances preserved exactly.

Example 8.1: Figure 19 shows how we can place three points a, b, and c, with only a distance measure D given, in 2-dimensional space. Start by placing a and b distance $D(a,b)$ apart. Draw a circle of radius $D(a,c)$ around a and a circle of radius $D(b,c)$ around b. By the triangle inequality, which D must obey, these circles intersect. Pick one of the two points of intersection as c. □

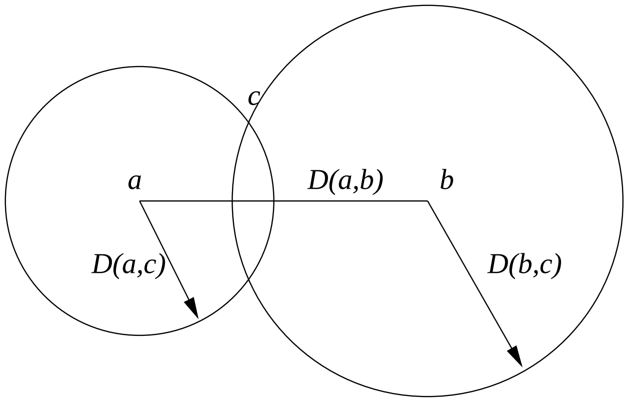


Figure 19: Placing three points in two dimensions

However, we usually have far too many points to try to place them in one fewer dimension. A more brute-force approach to placing n points in a k-dimensional space, where $k << n$, is called *multidimensional scaling.*

1. Start with the n points placed in k-dim space at random.

2. Take as the "error" the *energy* of a system of springs, each of length $D(x,y)$, that we imagine are strung between each pair of points x and y. The energy in a spring is the square of the difference between its actual length and $D(x,y)$.

3. Visit each point in turn, and try to place it in a position that minimizes the total energy of the springs. Since moving points around can affect the optimal position of other points, we must visit points repeatedly, until no more improvements can be made. At this point, we are in a local optimum, which may or may not be the global optimum.

Example 8.2: Suppose we have three points in a 3-4-5 "right triangle," as suggested in Fig. 20. If we try to place these points in one dimension, there must be some stretching and shrinking of springs. The optimum configuration, also shown in Fig. 20, is when the springs of length 3 and 4 are compressed to 7/3 and 10/3, while the spring of length 5 is stretched to 17/3. In this configuration, the total energy of the system is $(3-\frac{7}{3})^2 + (4-\frac{10}{3})^2 + (5-\frac{17}{3})^2 = 4/3$. □

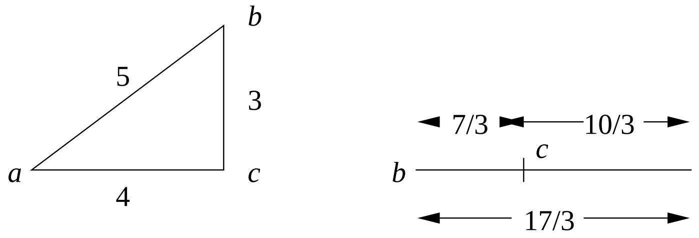


Figure 20: Optimal placement of three points in one dimension

8.2 Fastmap

Problem: multidimensional scaling requires at least $O(n^2)$ time to handle n points, since we must compute all distances at least once — perhaps more than once. *Fastmap* is a method for creating k pseudo-axes that can serve to place n points in a k-dim space in $O(nk)$ time.
In outline, Fastmap picks k pairs of points (a_i, b_i), each of which pairs serves as the "ends" of one of the k axes of the k-dim space. Using the law of cosines, we can calculate the "projection" x of any point c onto the line ab, using only the distances between points, not any assumed coordinates of these points in a plane. The diagram is shown in Fig. 21, and the formula is

$$x = \frac{D^2(a,c) + D^2(a,b) - D^2(b,c)}{2D(a,b)}$$

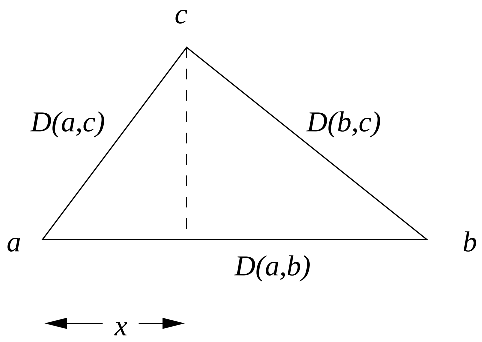


Figure 21: Computing the projection of point c onto line ab

Having picked a pair of points (a, b) as an axis, part of the distance between any two points c and d is accounted for by the projections of c and d onto line ab, and the remainder of the distance is in other dimensions. If the projections of c and d are x and y, respectively, then in the future (as we select other axes), the distance $D_{current}(c, d)$ should be related to the given distance function D by

$$D^2_{current}(c,d) = D^2(c,d) - (x-y)^2$$

The explanation is suggested by Fig. 22.
Here, then, is the outline of the Fastmap algorithm. It computes for each point c, k projections, which we shall refer to as $c^{(1)}, c^{(2)}, \ldots, c^{(k)}$ onto the k axes, which are determined by pairs of points (a_1, b_1), $(a_2, b_2), \ldots, (a_k, b_k)$. For $i = 1, 2, \ldots, k$ do the following:

1. Using the current distance $D_{current}$, Pick a_i and b_i, as follows:
   (a) Pick a random point c.
   (b) Pick a_i to be the point as far as possible from c, using distance $D_{current}$.
   (c) Pick b_i to be the point as far as possible from a_i.

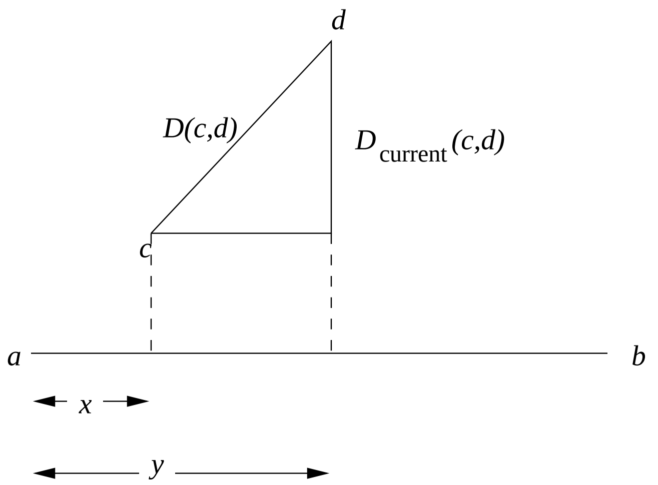


Figure 22: Subtracting the distance along axis ab to get the "current" distance between points c and d in other dimensions

2. For each point x, compute $x^{(i)}$, using the law-of-cosines formula described above.

3. Change the definition of $D_{current}$ to subtract the distance in the ith dimension as well as previous dimensions. That is

$$D_{current}(x, y) = \sqrt{D^2(x, y) - \sum_{j \leq i} (x^{(j)} - y^{(j)})^2}$$

   Note that no computation is involved in this step; we just use this formula when we need to find the current distance between specific pairs of points in steps (1) and (2).

8.3 Ways to Use Fastmap in Clustering Algorithms

1. Map the data to k dimensions in $O(nk)$ time. Cluster the points using any method that works for Euclidean spaces. Hope the resulting clusters are good (unlikely, according to GRGPF).

2. Improvement: Map to k dimensions using Fastmap, but use the Euclidean space only to estimate the *clustroids*: points that are closest on average to the other members of their cluster (like a centroid in a Euclidean space, but it has to be a particular point of the cluster, not a location that has no point). If there is a lot of data, we can use a sample for this stage. Then, assign points to clusters based on their true distance (using the distance measure, not the Euclidean space) to the clustroids.

8.4 Hierarchical Clustering

A general technique that, if we are not careful, will take $O(n^2)$ time to cluster n points, is *hierarchical* clustering.

1. Start with each point in a cluster by itself.

2. Repeatedly select two clusters to merge. In general, we want to pick the two clusters that are closest, but there are various ways we could measure "closeness." Some possibilities:

   (a) Distance between their centroids (or if the space is not Euclidean, between their clustroids).

   (b) Minimum distance between nodes in the clusters.

   (c) Maximum distance between nodes in the clusters.

   (d) Average distance between nodes of the clusters.

3. End the merger process when we have "few enough" clusters. Possibilities:

(a) Use a k-means approach — merge until only k clusters remain.

(b) Stop merging clusters when the only clusters that can result from merging fail to meet some criterion of compactness, e.g., the average distance of nodes to their clustroid or centroid is too high.

8.5 The GRGPF Algorithm

This algorithm assumes there is a distance measure D, but no Euclidean space. It also assumes that there is too much data to fit in main memory. The data structure it uses to store clusters is like an R-tree. Nodes of the tree are disk blocks, and we store different things at leaf and interior nodes:

- In leaf blocks, we store *cluster features* that summarize a cluster in a manner similar to BFR. However, since there is no Euclidean space, the "features are somewhat different, as follows:
  (a) The number of points in the cluster, N.
  (b) The *clustroid*: that point in the cluster that minimizes the *rowsum*, i.e., the sum of the squares of the distances to the other points of the cluster.
    - If C is a cluster, $\hat{C}$ will denote its clustroid.
    - Thus, the rowsum of the clustroid is $\sum_{X \ in \ C} D(\hat{C}, X)$.
    - Notice that the rowsum of the clustroid is analogous to the statistic $SUMSQ$ that was used in BFR. However, $SUMSQ$ is relative to the origin of the Euclidean space, while GRGPF assumes no such space. The rowsum can be used to compute a statistic, the *radius* of the cluster that is analogous to the standard deviation of a cluster in BFR. The formula is $radius = \sqrt{rowsum/N}$.
  (c) The p points in the cluster that are closest to the clustroid and their rowsums, for some chosen constant p.
  (d) The p points in the cluster that are farthest from the clustroid.
- In interior nodes, we keep samples of the clustroids of the clusters represented by the descendants of this tree node. An effort is made to keep the clusters in each subtree close. As in an R-tree, the interior nodes thus inform about the approximate region in which clusters at their descendants are found. When we need to insert a point into some cluster, we start at the root and proceed down the tree, choosing only those paths along which a reasonably close cluster might be found, judging from the samples at each interior node.

8.6 Maintenance of Cluster Features by GRGPF

There is an initialization phase on a main-memory full of data, where the first estimate of clusters is made, and the clusters themselves are arranged in a hierarchy corresponding to the "R-tree." The need for this hierarchy was mentioned in item 8.5 above.

We then examine all the other points and try to insert them into the existing clusters. The choice of clusters is reconsidered if there are too many to fit the representations in main memory, or if a cluster gets too big (too high a radius). There are many details about what happens when new points are added. Here are some of the key points:

(a) A new point X is placed in the cluster C such that $D(\hat{C}, X)$ is a minimum. Use the samples at the interior nodes of the tree to avoid searching the entire tree and all the clusters.

(b) If point X is added to cluster C, we add to each of the $2p + 1$ rowsums maintained for that cluster (rowsums of $\hat{C}$ and the p closest and p furthest points) the square of the distance from that point to X. We also estimate the rowsum of X as $Nr^2 + ND(\hat{C}, X)$, where r is the radius of the cluster. The validity of this formula, as an approximation is, based on the property of the "curse of dimensionality" we discussed in Section 7.2. That is, for each of the N points Y in the cluster, the distance between X and Y can be measured by going to the clustroid (the term

$D(\hat{C}, X)$), and then from the clustroid to Y (which is Y's contribution to r). If we assume that the lines (in a hypothetical Euclidean space) from $\hat{C}$ to X and Y are likely to be almost perpendicular, then the formula is justified by the Pythagorean theorem.

(c) We must consider the possibility that after adding X, cluster C now hasa different clustroid, which could be X or one of the p points closest to $\hat{C}$. If, after accounting for X, one of these points has a lower rowsum, it becomes the clustroid. obviously, this process cannot be maintained indefinitely, since eventually the clustroid will migrate outside the p closest points. Thus, there may need to be periodic recomputation of the cluster features for a cluster, which is possible — at the cost of disk I/O's — because all points in the cluster are stored on disk, even if they are not in the main-memory tree that stores the cluster features and samples.

8.7 Splitting and Merging Clusters in GRGPF

Sometimes, the radius of the clustroid exceeds a given threshold, and the algorithm decides to split the cluster into two. This process requires bringing the entire cluster into main memory and performing some split algorithm on just these points in memory.

An additional consequence is that the number of clusters in one leaf node increases by 1. If the node (disk block) overflows, then it must be split, as in a B-tree. That, in turn, may cause nodes up the tree to be split, and in the worst case, there is no room in main memory to hold the entire tree any more.

In that case, the solution is to raise the threshold that the radius of a cluster may have, and consider merging clusters throughout the tree. Suppose we consider merging clusters C_i and C_j. We wish to avoid having to bring all of the points of these clusters into main memory, so we guess at the clustroid and the rowsums as follows:

(a) Assume that the clustroid of the combined cluster will be one of the points furthest from the clustroid of either C_i or C_j. Remember that these points are all kept in the tree with their cluster features.

(b) To decide on the new clustroid, we need to estimate the rowsum for each point X in either cluster. Suppose for example that X is in C_i. Then we estimate:

$$Rowsum(X) = Rowsum_i(X) + N_j\big(D^2(X, \hat{C}_i) + D^2(\hat{C}_i, \hat{C}_j)\big) + Rowsum_j(\hat{C}_j)$$

Here, N_j is the number of points in C_j, and as always, hats indicate the clustroid of a cluster. The rationale for the above formula is that:

i. The first term represents the distances from X to all the nodes in its cluster.

ii. The middle term represents part of the paths from X to each point in C_j. We assume that the path starts out going from X to $\hat{C}_i$, the clustroid of C_i. From there, it goes to $\hat{C}_j$. Note that, by the "curse of dimensionality," we may assume that the lines from X to $\hat{C}_i$ and from $\hat{C}_i$ to $\hat{C}_j$ are "perpendicular" and combine them using the Pythagorean theorem.

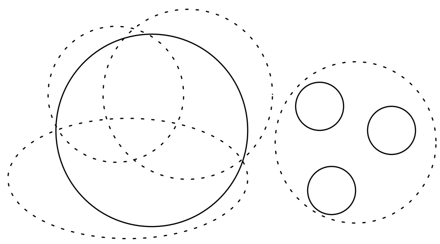
iii. The final term represents the component of the paths from X to all the members of C_j that continues after $\hat{C}_j$ is reached from X. Again, it is the "curse of dimensionality" assumption that lets us assume all the lines from $\hat{C}_j$ to members of C_j are orthogonal to the line from $\hat{C}_i$ to $\hat{C}_j$.

(c) Having decided on the new clustroid, compute the rowsums for all points that we retain in the cluster features of the new cluster using the same formula as in (2).

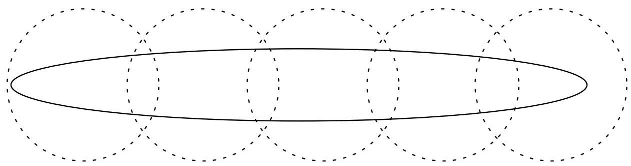
8.8 CURE

We now return to clustering in a Euclidean space. The special problem solved by CURE is that when clusters are not neatly expressed as Gaussian noise around a central point (as BFR assume) many things can go wrong in a k-means approach.

Example 8.3: Figure 23 suggests two possible problems. In (a), even though $k = 4$ is the right number of clusters (solid circles represent the extent of the clusters), an algorithm that tries to minimize distances to a centroid might well cluster points as suggested by the dotted lines, with the large cluster broken into three parts, and the three small clusters combined into one. In (b), a long cluster could be interpreted as many round clusters, since that would minimize the average distance of points to the cluster centroids (Notice, however, that by using the Mahalanobis radius, as in Section 7.8, a cluster that is long is one dimension is recognized as "circular," and we would not expect BFR to make the mistake of Fig. 23(b). □



(a) 4 clusters are correct, but they're the wrong 4!



(b) One long cluster looks like many small ones!

Figure 23: Problems with centroid-based clustering

Here is an outline of the CURE algorithm:

1. Start with a main memory full of random points. Cluster these points using the hierarchical approach of Section 8.4. Note that hierarchical clustering tends to avoid the problems of Fig. 23, as long as the true clusters are dense with points throughout.

2. For each cluster, choose c "sample" points for some constant c. These points are picked to be as dispersed as possible, then moved slightly closer to the mean, as follows:

   (a) Pick the first sample point to be the point of the cluster farthest from the centroid.

   (b) Repeatedly pick additional sample points by choosing that point of the cluster whose minimum distance to an already chosen sample point is as great as possible.

   (c) When c sample points are chosen, move all the samples toward the centroid by some fractional distance, e.g., 20% of the way toward the centroid. As a result, the sample points need not be real points of the cluster, but that fact is unimportant. The net effect is that the samples are "typical" points, well dispersed around the cluster, no matter what the cluster's shape is.

3. Assign all points, including those involved in steps (1) and (2) to the nearest cluster, where "nearest" means shortest distance to some sample point.

Example 8.4: Figure 24 suggests the process of picking $c = 6$ sample points from an elongated cluster, and then moving htem 20% of the way toward the centroid. □

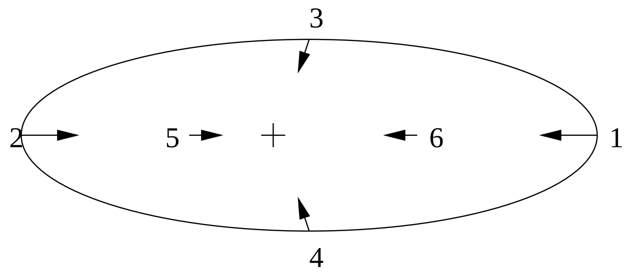


Figure 24: Selecting sample points in CURE

9 Sequence Matching

Sequences are lists of values $S = (x_1, x_2, \ldots, x_k)$, although we shall often think of the same sequence as a continuous function defined on the interval 0-to-1. That is, the sequence S can be thought of as sample values from a continuous function $S(t)$, with $x_i = S(i/k)$.

9.1 Sequence-Matching Problems

In the simplest case, we are given a collection of sequences $\{S_1, S_2, \ldots, S_n\}$, and a query sequence Q, each of the same length. Our problem is to find that sequence S_i whose *distance* from Q is the minimum, where "distance" is defined by the "energy" of the difference of the sequences; i.e., $D(S,T) = \int_0^1 \big(S(t) - T(t)\big)^2 dt$. For instance, the S_i's might be records of the prices of various stocks, and Q is the price of IBM stock, delayed by one day. If we found some S_i that was very similar to Q, we could use the price of the stock S_i to predict the price of IBM stock the next day, Notes:

- Do not try this at home. Anything easy to mine about stock prices is already being done, and the market has adjusted to whatever knowledge can be gleaned.
- Sequence matching is a great opportunity to violate the Bonferroni principal, since there has to be a "closest sequence." For instance, a famous mistake was looking in the UN book of world statistics to find the statistic that best predicted the Dow-Jones average. It was "cotton production in Bangladesh."

9.2 Fourier Transforms as Indexes for Sequences

We could treat sequences of length k as points in a k-dim vector space, but doing so is not likely to be useful. Usually, k will be so high, that spacial index techniques like kd-trees or R trees will be useless. The trick adopted by Faloutsos and his colleagues is to map sequences to the first few terms of their Fourier transforms. Formally, the jth term of the *Fourier Expansion* of the function $S(t)$ is $\int_0^1 S(t)e^{2\pi jit}dt$. Recall that the imaginary exponential e^ix is defined to be $\sin x + i \cos x$. Thus, the real and imaginary parts of X_j tell how well $S(t)$ matches sine and cosine functions that have j periods within the interval 0-to-1, i.e., $\sin 2\pi jt$ and $\cos 2\pi jt$.

Example 9.1: Figure 25 suggests a simple function $S(t)$ (solid) and compares it to the single-period sine function (dotted) and single-period cosine function (dashed). The integral of the product $S(t) \sin 2\pi t + iS(t) \cos 2\pi t$ is the complex number X_1. $\square$

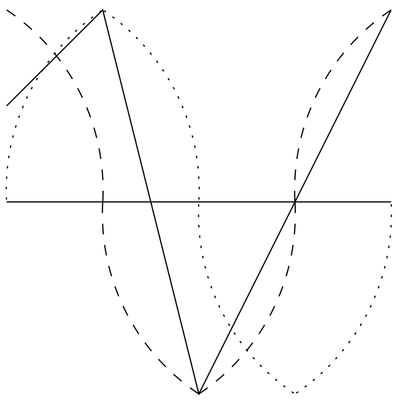


Figure 25: Function $S(t)$ matches the sine only slightly; the cosine better but not perfectly

- Key point: *Parsival's Theorem* states that the energy in a signal $S(t)$ is the same as the energy in its Fourier transform: $\sum_{i \geq 0} |X_i|^2$. Note $|X|$ is the magnitude of a complex number X.

- Key application: If $S(t)$ is actually the difference of two sequences, then the distance between those sequences, which is $\int_0^1 S^2(t)dt$ is the same as the sum of the squares of the magnitudes of the differences of the Fourier coefficients of the two sequences.
- Since the square of a magnitude is always positive, we can find all the sequences whose distance from a given query sequence Q is no more than ϵ^2 by finding all those sequences S the sum of whose magnitudes of the differences of the first m Fourier coefficients is no more than ϵ^2.
  - We shall retrieve some "false drops" — sequences that are close to Q in their first m coefficients, but differ greatly in their higher coefficients.
  - There is some justification for the rule that "all the energy is in the low-order Fourier coefficients," so retrieval based on the first few terms is quite accurate in practice.

9.3 Matching Queries to Sequences of the Same Length

As long as stored sequences and queries are of the same length, we can arrange the sequences in a simple, low-dimension data structure, as follows:

1. Compute the first m Fourier coefficients of each sequence S_i, for some small, fixed m. It is often sufficient to use $m = 2$ or $m = 3$. Since Fourier coefficients are complex, that is the equivalent of a 4- or 6-dim space.
2. Place each sequence S_i in this space according to the values of its first m Fourier coefficients. The points themselves are stored in some suitable multidimensional data structure.
3. To search for all sequences whose distance from query Q is no more than ϵ^2, compute the first m Fourier coefficients of Q, and look for points in the space no more distant than ϵ from the point corresponding to Q.
4. Since there are false drops, compare each retrieved sequence with Q to be sure that the distance is truly no greater than ϵ^2.

In some applications, it may make sense to "normalize" sequences and queries to make the 0th Fourier coefficient (which is the average value) 0 and perhaps to make the variance (i.e., the "energy") of each sequence the same.

Example 9.2: In Fig. 26(a) are two sequences that have a large difference, but are in some sense the same sequence, shifted and scaled vertically. We could shift them vertically to make their averages be 0, as in Fig. 26(b). However, that change still leaves one sequence always having twice the value of the other. If we also scaled them by multiplying the more slowly varying by 2, they would become the same sequence. □

9.4 Queries That are Shorter than the Sequences

Assume that queries are at least of length w. The *sequential scan* method for finding close (within distance ϵ^2) matches to a query Q is as follows:

1. Store the first m coefficients of the Fourier transform of each subsequence of length w for each sequence. As a result, if sequences are of length k, we store $k - w + 1$ points for each sequence.
2. Given a query Q, take the first w values of Q and compute the Fourier transform of that sequence of length w. Retrieve each sequence S and position p that matches within ϵ.
3. For each such S and p, compare the entire query Q with sequence S starting at position p. If the distance is no more than ϵ^2, report the match.

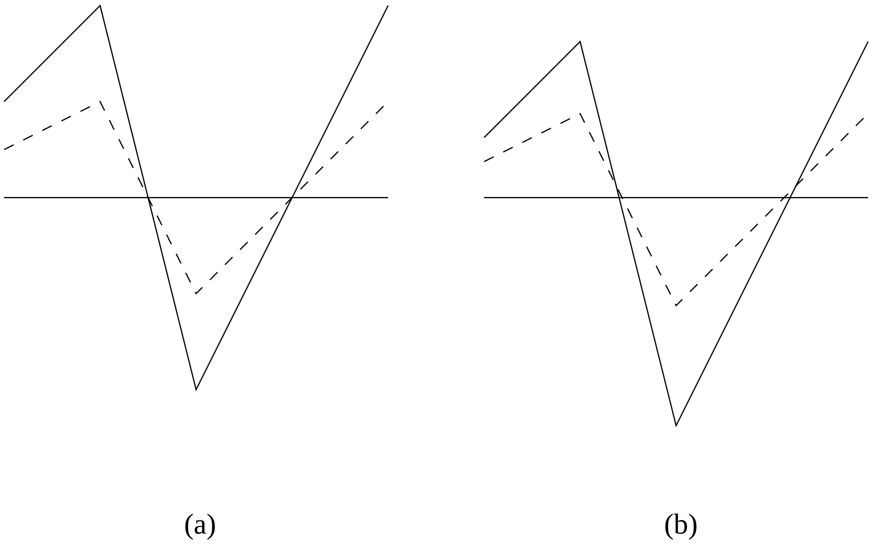


Figure 26: Shifting and scaling can make two sequence that look rather different become the same

9.5 Trails

The problem with sequential-scan is that the number of points that must be stored is almost the product of the number of sequences and the length of the sequences. The FRM paper takes advantage of the fact that as we slide a window of length w across a sequence S, the low-order Fourier coefficients will not vary too rapidly. We thus get a *trail* if we plot the points corresponding to consecutive windows of length w, as suggested in Fig. 27.

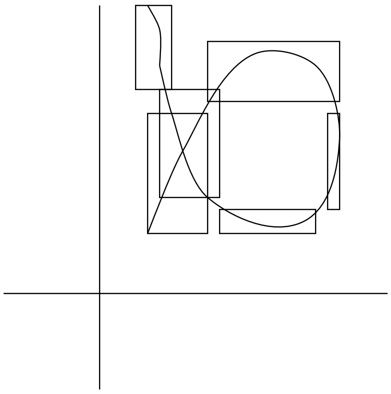


Figure 27: Trials and the rectangles that bound them

Instead of storing individual points, we store rectangles (in the appropriate number of dimensions) that bound the points in one segment of the trail, which is also suggested by Fig. 27. A rectangle can be stored using two opposite vertices, for example. Thus, if rectangles tend to represent many points, then the space ued to store the rectangles is much less than the space needed to store individual points.

- To retrieve the points whose distance from a query Q of length w is no more than ϵ, find the rectangles that are within ϵ of Q. Match Q only against the sequences that correspond to at least one retrieved rectangle, and only at begining positions represented by those rectangles.
- Partitioning trails into rectangles is an opimization problem. FRM uses as the cost of storing a rectangle whose side in dimension i is L_i (as a fraction of the length of the total distance along the ith axis): $\prod_i (L + i + 0.5)$. For example, a rectangle of fractional sides 1/4 and 1/3 would have cost

$(3/4)(5/6) = 5/8$. Start from the beginning of the trail, and form rectangles by adding points as we treavel along the trail. When deciding whether or not to add another point to the current rectangle, make sure that the cost of the new rectangle per point covered decreases. If it increases, then start a new rectangle instead.

9.6 Matching Queries of Arbitrary Length

If the query Q has length w, just find the rectangles within distance ϵ from Q, as discussed in Section 9.5. Note that Q may actually be within a rectangle, and even so, Q may be distant from actual points on the trail, as suggested in Fig. 28.

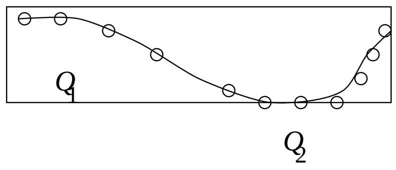


Figure 28: Two examples of queries near a rectangle; note that the query inside the rectangle is actually further from the points on the trail than the query outside the rectangle, but both need to be considered

However, if the query Q has length greater than w, say pw for some constant $p > 1$, there are several things we can do:

1. Search for just the first w values of Q, retrieve the matching sequences and their positions, and then compare the entire Q with the sequence starting at that position. This method is like sequential scan, but it takes advantage of the storage efficiency of trails.

2. Search for the rectangles that are sufficiently close (within ϵ) to the first w values of Q, the next w values of Q, and so on. Only a sequence that has at least one rectangle that is within ϵ of *each* of these subsequences of Q is a candidate match. Check the candidate matches.

3. Like (2), but check a sequence S only if it has a rectangle within distance $\epsilon/\sqrt{p}$ from *at least one* of the p subsequences of Q. Note that the distance of sequences must be at most ϵ^2, and if all p subsequences of Q are at least $\epsilon/\sqrt{p}$ from any point on the trail of S, then $p(\epsilon/\sqrt{p})^2 = \epsilon^2$ is a lower bound on the distance between Q and any subsequence of S.

10 Mining Episodes

In the episode model, the data is a history of *events*; each event has a *type* and a time of occurrence. An example of event type might be: "switch 34 became overloaded and had to drop a packet." It is probably too general to have an event time of the form "some switch became overloaded." Several events may occur at the same time, and there is no guarantee that some event happens at every time unit.

10.1 Applications of Epsiode Mining

- Mining "epsiodes," that is, sequences or sets of event types that occur within a short window, has been used to help predict outages in the Finnish power grid; the paper of MTV is an abstraction of this work.

- The same ideas could be used to monitor packet-switching or other communication networks, to develop rules for rerouting data in advance of congestion. It may also be possible to mine for rules that predict failures from combinations of manifestations in a variety of complex systems.

10.2 Epsiodes

- A *parallel episode* is a set of event types, e.g., $\{A, B, C\}$. In diagrams, these episodes are represented by a vertical box with A, B, and C within. The intent of a parallel episode is that each of the events in the episode occurs (within a window of time), but the order is not important.

- A *serial episode* is a list of event types, e.g., (A, B, C). In diagrams, these events are shown in a horizontal box, in order. The intent is that within a window of time, these events occur in order. Note that a single event may be thought of as both a parallel and a serial episode.

- A *composite episode* is built recursively from events by serial and parallel composition. That is, a composite episode is either:

  - An event,
  - The serial composition of two or more events, or
  - The parallel composition of two or more events.

  Thus. every serial and parallel episode is also a composite episode, but there are episodes that are composite, yet not serial or parallel.

Example 10.1 : Figure 29 shows a composite episode. It is the serial composition of three episodes. The first is the single event A. Then comes the parallel epsiode $\{B, C, D\}$. The third is a composite episode consisting of the parallel composition of the serial episodes (E, F) and (G, H). Two examples of orders of these 8 events that are consistent with this episode are $ABCDEGFH$ and $ACDBGHEF$. $\Box$

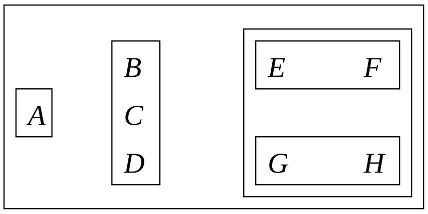


Figure 29: A composite episode

10.3 Monotonicity of Episodes and the A-Priori Algorithm

Given a window length w (an amount of time during which an episode may occur), an episode is *frequent* if it occurs in at least s windows, where s is the support threshold. Note that the same sequence of events may show up as an episode in several consecutive windows; we count one unit for each such window. However, no window can be credited with having the same episode occur more than once, even if we can construct the episode from several different sets of events in the same window.
Like frequent itemsets, frequent episodes are monotone: if an episode E is frequent, then so is any episode formed by deleting some events from E. Thus, we can construct all frequent episodes "levelwise," using a trick that is very much like a-priori.

1. Let C_i be the candidate episodes of size (number of events) i and let L_i be the frequent episodes of size i.
2. For a basis, C_1 is the set of all event types.
3. Construct C_{i+1} from L_i by putting in C_{i+1} exactly those episodes E of size $i+1$ such that deleting any one event from E yields an event from $L+i$.

Example 10.2: The epsiode of size 3 in Fig. 30(a) can be frequent (i.e., in C_3) only if the three episodes of Fig. 30(b) are frequent (i.e., in L_2). □

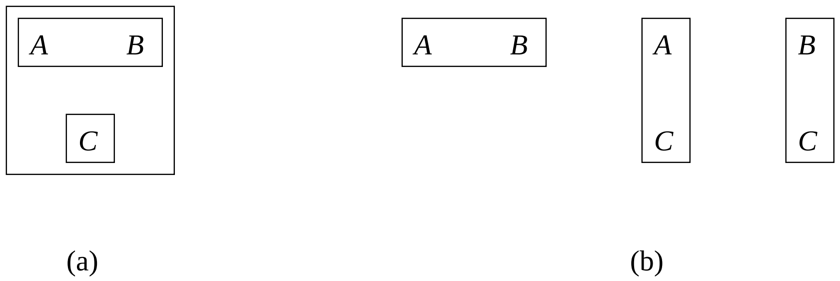


Figure 30: Episode (a) is a candidate in C_3 only if all three episodes (b) are in L_2

10.4 Checking Parallel Episodes

The big problem is converting C_i into L_i by going once through the data and counting the support for each episode in C_i. A dumb algorithm will look at each window of length w in turn, and for each candidate episode check whether it is there; if so, add 1 to the support for the episode.
The goal of MTV is to perform work that is only proportional to the sum over all events of the number of candidate episodes (including the subepisodes of composite episodes) that have that event. The following algorithm was developed in class; it seems less restrictive than the one given in the paper, since it does not assume the impossibility of an event type appearing several times in a window. We describe how the data is scanned once, to compute L_i from C_i, considering each window in turn, from the beginning of the sequence. The data structure needed:

1. For each event type A:
   (a) A count `A.count` of the number of times A has been seen in the present window.
   (b) A linked list `A.contains` of all the episodes E that contain A.
2. For each candidate episode E:
   (a) A time `E.startingTime` that is the beginning of a consecutive sequence of windows in which E has always been present, up to the present window.

(b) An integer `E.support`, the number of windows in which E has appeared, not including windows since `startingTime`.

(c) An integer `E.missing` giving the number of events A of E that are *not* in the present window. Note that E is present in the window if and only if `E.missing==0`.

The heart of the algorithm is what we do when we slide the window one time unit forward. We must consider what happens when an event A drops out of the beginning, and what happens when an event B is included at the front. If A drops out of the window:

```
    A.count--;
    if(A.count==0) /* we just lost A */
 for(all E on A.contains) {
            E.missing++;
            if(E.missing==1) /* we just lost E */
                E.support += (E.startingTime - currentTime);
        }
```

If B enters the window:

```
    B.count++;
    if(B.count==1) /* we just gained B */
        for(all E on B.contains) {
            E.missing--;
            if(E.missing==0) /* we just gained E */
                E.startingTime = currentTime;
        }
```

10.5 Checking Serial Episodes

To check serial episode $(A_1, A_2, \ldots, A_n)$ we simulate a nondeterministic finite automaton that recognizes the string $. * A_1 A_2 \cdots A_n$, as suggested by Fig. 31. As we scan the events in the data, we keep track of the set of states that the NFA is in.

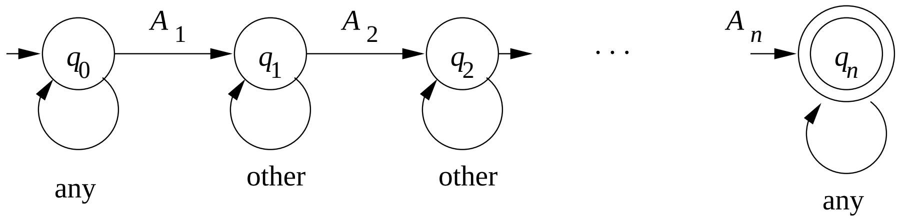


Figure 31: An NFA for recognizing a serial episode

Here is a rough description of how the NFA's for the various events E are used, assuming a data structure similar to that described in Section 10.4 for parallel episodes.

- Note that the NFA stays in the same set of states it was in if the input symbol is an event that is not part of its episode. Thus, simulating this automaton requires 0 time unless its episode is on a list like `A.contains` for the current event A (see the data structure in Section 10.4).

- As we simulate, we keep for each state the NFA is currently in the *most recent* point at which we could have started the NFA and still gotten to that state. This value is:

  1. The current time if we enter state q_1.
  2. The time of q_{i-1} if we enter state q_i by reading A_i.

3. The same time as previously, if we were already in q_i and the next input is other than A_i.

- If the NFA for episode E enters the accepting state q_n, but was not previously in that state, then set `E.startingTime` to the time currently associated with q_n.
- If any time associated with a state is less that the current time minus w (the window length), then delete that state from the set of states the NFA is in. If the accepting state is thus lost, add `E.startingTime` minus the current time to `E.support`.

Example 10.3: Suppose the event E is (A, B, C). The NFA for E is as in Fig. 32.

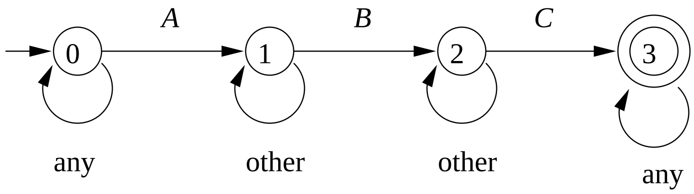


Figure 32: NFA for the episode (A, B, C)

Let the data include the following occurrences of A, B, and C, along with other event types, not shown: (A, B, A, B, C); all these events are within the current window. Then the states of the NFA after each of these events is as shown in Fig. 33. Solid lines show how each state is derived from the previous one by transitions of the NFA, and dashed lines indicate the associated time for each state. □

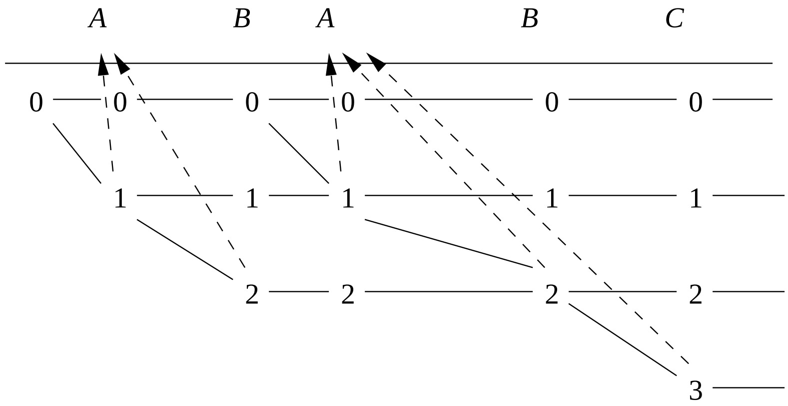


Figure 33: Maintaining the set of states and their associated times

10.6 Counting Composite Events

To extend the above ideas to composite episodes, we must keep a "machine" of some sort for each subepisode. It is the job of each machine to report to the machine(s) for the superepisode(s) of which it is a part whether or not it is present, and if so what is the most recent time at which it can be construed to have begun.

- If the subepisode is the parallel composition of events and/or other subepisodes, we use a data structure like that of Section 10.4, but we replace the count for an event by the most recent beginning time for a composite episode.
- For serial episodes, we maintain an automaton, but the inputs are occurrences of its subepisodes.
- The starting time and support only need to be maintained for episodes in C_i, not for subepisodes.