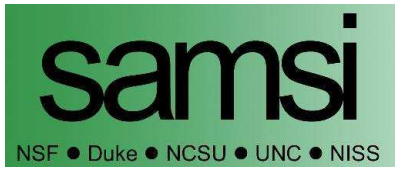
Scalability in Data Mining

David Banks

Institute of Statistics & Decision Sciences
Duke University



1. Introduction

Modern data mining requires one to make many hard searches. These searches are needed for variable selection, model selection, and robustness.

Different kinds of searches have different structures, and it is important to exploit the properties of the problem in selecting the search strategy.

This talk focuses on scalability in search, as this relates to research undertaken during the 2003-2004 program year at SAMSI. In particular, we discuss:

- variable selection,
- model selection,
- data subsetting.

2. Variable Selection

The canonical search problem is variable selection in multiple linear regression. Here one has p explanatory variables, and wants to find the parsimonious subset of the variables which provides the best (or good) explanation.

The usual approach is stepwise selection, but this is known to be imperfect (cf. Banks, Olszewski, and Maxion, 2003).

However, work on the large p small n problem at SAMSI makes it clear that aggressive variable selection is an essential part of any effort to model data when $p > 10$.

2.1 Gray Codes

In variable selection, eadh of the p variables may be included or excluded from the model. There are 2^p possible models, corresponding to all binary strings of length p. The combinatorial structure in this problem is to examine all of the possible strings.

The strings can be identified with the vertices of the p-dimensional unit hypercube. The $(0, 0, \ldots, 0)$ string is the origing vertex, and corresponds to a model that includes none of the explanatory variables. The $(1, 1, \ldots, 1)$ model is the opposite vertex, and corresponds to the model that includes all of the explanatory variables.

From this perspective, one can consider a search strategy that is based upon a smart traversal of the unit hypercube.

The Gray code was patented in 1953. It is an algorithm for traversing the unit hypercube that touches each vertex exactly once before returning to the starting point. At each step, it changes only one of the digits in the string from zero to one or from one to zero.

There are many such traversals, but the Gray code has the additional property that, to the greatest extent possible, it makes only local moves. In a sense, it takes only short steps.

Using Gray codes, one search strategy is to take r steps along the cube according to the algorithm, and then stop and fit a model. By repeating this until one returns to the starting point, one draws a systematic but well-dispersed sample of the model fits.

Here r should be about 1/1000th of 2^p.

If one analyzes the R^2 values (or some other measure of fit) from these $[2^p/r]$ models, one can find the variables that are most important.

As a technical note, there are reasons not to let r be a power of 2, but this makes little difference in practice.

Gray Code Vertex Rank, Binary Rank, and Vertex String.

0	0000	0000	9	1001	1101
1	0001	0001	10	1010	1111
2	0010	0011	11	1011	1110
3	0011	0010	12	1100	1010
4	0100	0110	13	1101	1011
5	0101	0111	14	1110	1001
6	0110	0101	15	1111	1000
7	0111	0100			
8	1000	1100			

Note that the rank in binary is not the same as the binary string for the corresponding vertex.

Ranking Theorem (Wilf, 1989).

Let $m = \sum a_i 2^i$ be the binary representation of m. Let $\ldots b_3 b_2 b_1$ be the vertex string of rank m in the Gray code. Then

$$b_i = a_i + a_{i+1} \pmod 2.$$

This means that one can quickly generate the vertex string at any given point on the Gray code list.

Yuen's Theorem (1974).

If the Hamming distance between vertex strings v_1 and v_2 is greater than or equal to k, then the difference in their ranks in the Gray code list is at least $\left[\frac{2k}{3}\right]$.

This means that large steps in rank correspond to large changes in the vertex string.

Gray code search is somewhat like the 'Leaps and Bounds' idea put forward by Breiman in 1996, but is specialized to variable selection.

There are a number of modifications to Gray code search that can improve performance when p is very large. For example, one can do adaptive searches in which the step-size is a function of the R^2 value—thus one takes large steps in bad neighborhoods.

Gray codes exist for other kinds of search problems with combinatorial structures, such as trees or partitions. But there are not always analogues of the ranking theorem or Yuen's theorem to enable easy application or guarantee good theoretical search properties.

2.2 Random Restart

A second way to search the hypercube is random restart. Here one picks a vertex uniformly at random, and then looks at all neighboring vertices to find the one that most improves R^2 (or whatever measure of fit is being used).

The search moves to the best neighboring vertex, and repeats. The search continues in this greedy hill-climbing way.

When a local maximum is found, the analyst restarts the process and finds another local maximum. By repeating this search many times and keeping track of the distinct local maxima and the corresponding fitness values, one can maek probability statements about the number of unfound local optima and their fitness values.

2.3 Designed Experiments

A third strategy is to treat the p explanatory variables as factors in a 2^{p-k} factorial designed experiment. This approach was initially suggested by Clyde (1999).

For a highly fractionated design, one calculates the R^2 values for each of the design points, and then estimates the main effects and (maybe) the low-order interactions among the p explanatory variables.

Variables with no significant effect are discarded. But this method does not scale well with p.

3. Unstructured Model Selection

Variable selection could use algorithms that took smart advantage of the special combinatorial structure or the relationship to experimental design. But in general, many kinds of search do not have exploitable structure.

Bertrand Clarke calls these situations list searches. There are no natural neighborhood relations among the elements on the list.

In the context of data mining, this kind of problem often arises in model selection.

3.1 Racing

Racing is a strategy that was developed by Maron and Moore (1997). It applies to many situations, but is especially useful in the context of list searches.

Suppose one wants to find the model on a list that gives the best fit to the data. It is helpfult to note that when comparing two models on the list, it is not necessary to fit all of the data. Often it is sufficient to fit a percent or two in order to determine which is better.

Racing fits a small percent of the data to each model, throws out the bad models, and then fits another small portion of the data. Typically this scales the seach up by a factor of about 100.

3.2 Structuring Lists

Sometimes it is possible to impose meaningful structure on lists. When this can be done, then one has the hope of finding better search strategies.

As an example, suppose one can find a neighborhood structure for the list. Perhaps two elements of the list are neighbors because they have almost the same variables, or very similar basis elements, or similar interaction terms.

With a neighborhood structure, one can use simulated annealing. But that is not necessarily a very good solution for problems in general, and scalability is a real issue.

A second strategy for dealing with strucutured lists uses genetic algorithms.

This approach requires that the models in the list be decomposable into simpler models in such a way that each model on the list can be assembled from the simpler models. Then one uses traditional genetic algorithm methods to let natural selection determine the best model on the list.

In my experience, this is highly problematic. But there are people who love genetic algorithms and seem able to make them work. I am particularly surprised and impressed that people are able to find such delicate constructions as D-optimal designs using these techniques.

Regarding scalability, I have no real information. But it is hard to imagine that genetic algorithms would be very competitive.

3.4 List Search

With list search, there is no exploitable structure that links the elements of the list, and the list is usually so long that exhaustive search is infeasible.

There is not much that one can do. If one tests entries on the list at random, then one can try some of the following:

- Estimate the proportion of list entries that give results above some threshold.
- Use some modeling to estimate the maximum value on the list from a random sample of list entries.
- Estimate the probability that further search will discover a new maximum within a fixed amount of time.

A strategy invented by computer scientists (Maron and Moore, 1997, *Artificial Intelligence Review*, **11** 193-225) is to **race** the testing.

One does pairwise comparisons of models. At first, one fits only a small random fraction of the data (say a random 1%) to each model on the list. Usually this is sufficent to discover which model is best and one discards the other.

If that small fraction does not distinguish the models, then one fits another small fraction. Only very rarely is it necessary to fit all mor most of the data to select the better model.

Racing is an easy way to extend one's search capability by about 100-fold.

4. Robust Structure Extraction

Suppose one has a large, complex dataset that contains multiple kinds of structures and/or noise, e.g.:

- 40% of the data follow $Y = \alpha_0 + \sum_{i=1}^{p} \alpha_i X_i + \epsilon$
- 30% of the data follow $Y = \beta_0 + \sum_{i=1}^{p} \beta_i X_i + \epsilon$
- 30% of the data are noise.

What can one do to analyze cases like this? One should assume that the data miner has little prior knowledge of the kinds of structure that might be present.

The standard approach in linear regression is to use S-estimators, which look for the thinnest strip (think of a transparent ruler) which covers some prespecified (but larger than 50%) fraction of the data.

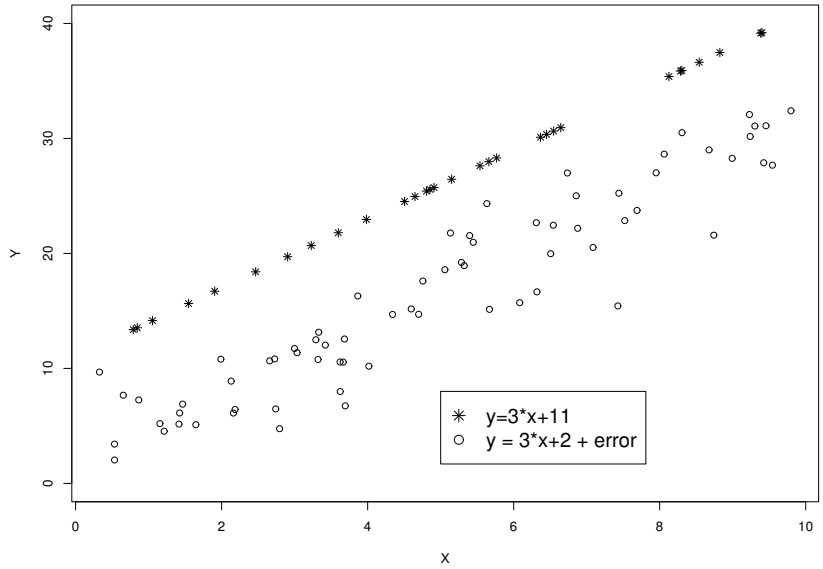
This strategy breaks down in high dimensions, or when the structure contains less than 50% of the data, or when fitting complex models.

One wants a solution strategy for harder problems:

- linear and non-linear regression in high dimensions,
- multidimensional scaling,
- cluster analysis.

4.1 Hidden Structure in Regression

The graph below shows data that come from a mixture of two regression models. Any naive analysis will miss the point and find some kind of average solution.



For observations $\{Y_i, \mathbf{X}_i\}$ for $i = 1, \ldots, n$ assume that Q percent of these follow the model

$$Y_i = \beta_0 + \beta_1 X_{i1} + \ldots + \beta_p X_{ip} + \epsilon_i \quad \text{where} \quad \epsilon_i \sim N(0, \sigma)$$

where Q, $\boldsymbol{\beta}$, and σ are unknown. One can say $Q\%$ of the data as "good" and the rest are "bad".

Simple Idea:

Start small, with a subsample of only **good** observations

$\Rightarrow$ add only **good** observations

$\Rightarrow$ end with a large subsample of **good** observations.

General procedure:

1. Choose d starting subsamples S_j, each of size m
2. Grow the subsamples by adding consistent data
3. Select the largest subsample.

The algorithm for choosing the starting subsamples and growing them efficiently is important in practice.

One starts with a guess about Q, the fraction of good data. In general, this is unknown, so one might pick a value that is reasonable given

- domain knowledge about the data collection
- scientific interest in a fraction of the data.

From the full dataset $\{Y_i, \mathbf{X}_i\}$ one selects, without replacement, d subsamples S_j of size m.

One needs to choose d and m to ensure that at least one of the starting subsamples S_j has a very high probability C of consisting entirely of good data (i.e., data that come from the model).

Preset a probability C that determines the chance that the algorithm will work.

The value m, which is the size of the starting-point random subsamples, should be the smallest value that allows one to calculate a goodness-of-fit. For multiple linear regression, m is $p + 2$ and a natural goodness-of-fit measure is R^2.

One solves the following equation for d:

$$C = \mathbb{P}[\text{ at least one of } S_1, \ldots, S_d \text{ is all good }] = 1 - (1 - Q^{p+2})^d.$$

Example: $Q = .8$, $c = .95$, $m = 3$ $(p = 1)$:

$$.95 = 1 - [1 - (.8)^{p+2}]^d \quad \rightarrow \quad d = 5$$

Given the d starting-point subsamples S_j, one grows each one of them by adding observations that do not lower the goodness-of-fit statistic (R^2).

Conceptually, for a particular S_j, one could cycle through all of the observations, and on each cycle augment S_j by adding the observation that provided the largest value of R^2. This cycling would continue until no observation can be added to S_j without decreasing the R^2.

One does this for all of the subsamples S_j. At the end of the process, each augmented S_j would have size m_j and goodness-of-fit R_j^2. The augmented subsample that achieves a large value of m_j and a large value of R_j^2 is the one that captures the most important structure in the data.

Then one can remove the data in S_j and iterate to find the next-most important structure in the dataset.

Fitting one observation per cycle is slow when n is large or one has a complex model (e.g., MARS fits). So we use a two-step procedure to add data.

Fast Search

- Sequentially sweep through all observations not in S_i.
- If the observation improves the fitness measure (or lowers it by only a small amount η), then
  $\rightarrow$ add observation to S_j
  $\rightarrow$ set $m_j = m_j + 1$.
  If $m_j < kn$ then implement slow search.

Slow Search

- Add the observation that improves the FM the most or decreases FM the least.
- Repeat until $m_j = kn$.

The analyst may pick an η that seems appropriately small, and a value for k that seems appropriately large. These choices affect the runtime of the algorithm.

The fast search is greedy, and the order of observations in the cycling matters. The slow search is less greedy; order does not matter, but it adds myopically. Fast search can add many observations per cycle, but slow search always adds exactly one.

If speed is needed, then there are other ways to accelerate the algorithm. One is to increase the number of starting-point subsamples and combine those that provide similar models and fits as they grow. The main concern is not to enumerate all $\binom{n}{Qn/100}$ possible subsamples.

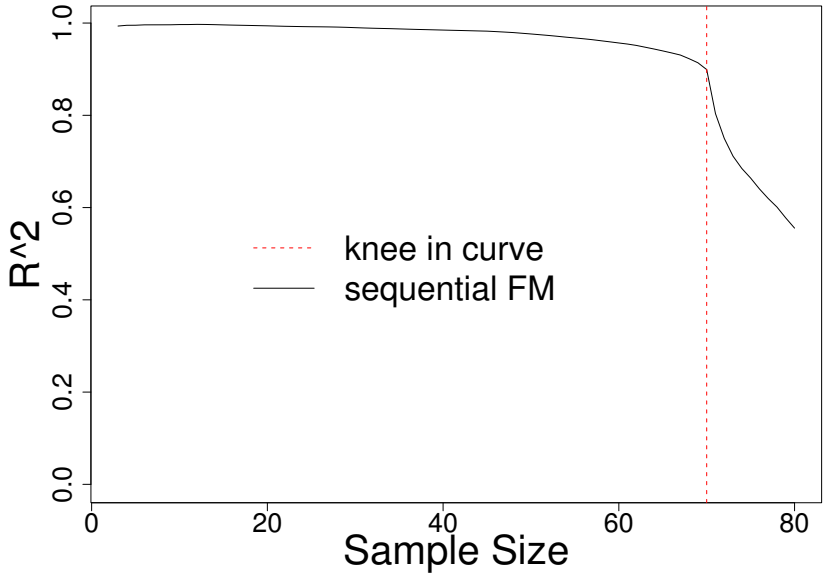
Note that:

1. One need not terminate the search at some preset value kn; one can grow until the goodness-of-fit measure deteriorates. When one starts to add bad data, there is a visible "slippery-slope" effect.

2. The goodness-of-fit measure should not depend upon the sample size. For R^2 this is easy, since it is just the proportion of variation in Y explained by X. For larger p, if one is doing stepwise regression to select variables, then one wants to use an AIC or Mallows' C_p statistic to adjust the tradeoff in fit between the number of variables and the sample size.

3. Other measures of fit are appropriate for nonparametric regression, such as cross-validated within-subsample squared error.

To see how the slippery-slope occurs, and the value of monitoring fit as a function of order of selection, consider the plot below. This plot is based on using R^2 for fitness with the double-line data shown previously. The total sample size is 80, and 70 observations were generated exactly on a line, as indicated by the knee in the curve.



4.2 Hidden Structure in MDS

Multidimensional scaling (MDS) starts with a proximity matrix that gives approximate distances between all pairs in a set of objects. These distances are often close to a true metric.

MDS finds a low-dimensional plot of the objects such that the inter-object distances are as close as possible to the values in the proximity matrix. Thus it puts similar objects near each other. This is done by a least squares fit to the values in the proximity matrix, minimizing the stress function:

$$\textbf{Stress}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \left[\sum_{i \neq i'} (d_{ii'} - \|\boldsymbol{z}_i - \boldsymbol{z}_{i'}\|) \right]^{1/2}$$

where $\boldsymbol{z}_i$ is the location assigned to pseudo-object i in the low-dimensional space and $d_{ii'}$ is the entry in proximity matrix.

The classic example is to take the entries in the proximity matrix to be the drive-time between pairs of cities. This is not a perfect metric, since roads curve, but it is approximately correct. MDS finds a plot in which the relative position of the cities looks like it would on a map (except the map can be in any orientation; north and south are not relevant).

MDS seeks is extremely susceptible to bad data. For example, if one had a flight tire while driving from Rockville to DC, this would create a seemingly large distance. The MDS algorithm would distort the entire map in an effort to put Rockville far from DC and still respect other inter-city drive times.

A very small proportion of outliers, or objects that do not fit well in a low-dimensional representation, can completely wreck the interpretability of an MDS plot. In many applications, such as text retrieval, this is a serious problem.

4.2.1 MDS Example

To test cherry-picking for MDS, consider the latitudes and longitudes of 99 eastern U.S. cities. The Euclidean distances between these cities gave the proximity matrix; the only stress in the MDS map is due to the curvature of the earth.

Perturb the proximity matrix by inflating a random proportion $1 - Q$ of the entries:

Bad Data	Distortion (%)	Stress
2	150	1.028
	500	2.394
10	150	1.791
	500	28.196
30	150	3.345
	500	9.351

To make things more interesting, we use not the traditional MDS using the stress measure defined previously, but rather Kruskal-Shephard non-metric scaling, in which one finds $\{\boldsymbol{z}_i\}$ to minimize

$$\textbf{Stress}_{KS}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \frac{\sum_{i \neq i'} [\theta((\|\boldsymbol{z}_i - \boldsymbol{z}_{i'}\|) - d_{ii'}]^2}{\sum_{i \neq i'} d_{ii'}^2}$$

where $\theta(\cdot)$ is an arbitrary increasing function fit during the minimization. The result is invariant to monotonic transformations of the data, which is why it is nonparametric.

This minimization uses an alternating algorithm that first fixes $\theta(\cdot)$ and finds the $\{\boldsymbol{z}_i\}$, and then fixes the $\{\boldsymbol{z}_i\}$ and uses isotonic regression to find $\theta(\cdot)$. This shows that the algorithm can be used in complex fits.

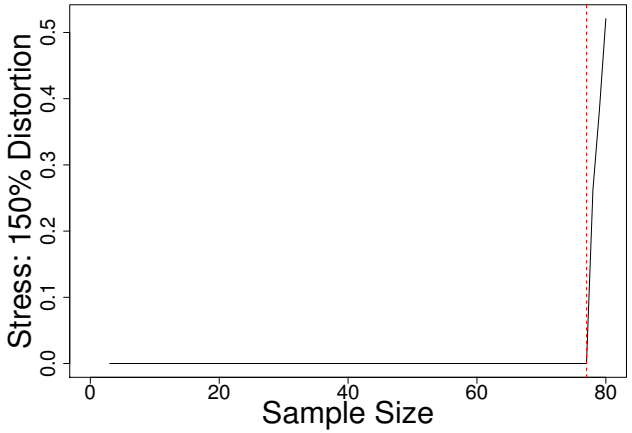
Our goal is to cherry-pick the largest subset of cities whose intercity distances can be represented with little stress.

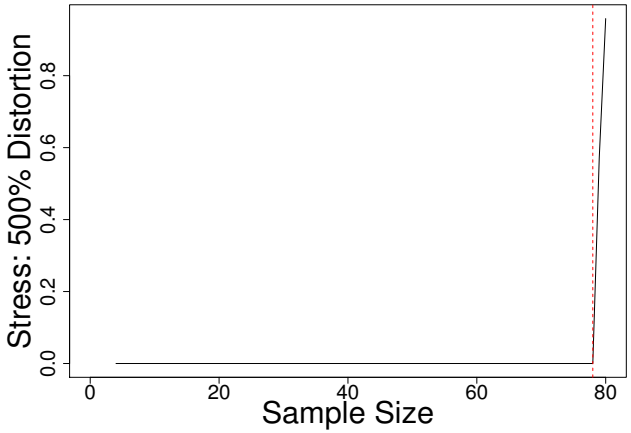
In MDS, the size m of the initial subsamples is 4 (since three points are always coplanar). We took $C = .99$ as the prespecified chance of getting at least one good subsample, and the table below shows the results.

True $1-Q$ (%)	Distance Distortion (%)	Original Stress	n^a	n^*	Final Stress
2	150	1.028	80	80	4.78e-12
	500	2.394	80	80	4.84e-12
10	150	1.791	80	80	4.86e-12
	500	28.196	80	80	4.81e-12
30	150	3.345	80	77	4.86e-12
	500	9.351	80	78	4.78e-12

- Note: The stress of the undistorted dataset was 8.42×10^{-12}.

As before, one should inspect order-of-entry plots that display the stress against the cities chosen for inclusion. The following two plots are typical, and show the knee in the curve that occurs when one begins to add bad cities.





- This is a simple strategy for identifing stucture in complex datasets.
- It is practical in computer-intensive fits, but one needs greedy search algorithms to select observations for inclusion.
- The main computational problem is to scale the algorithm to accommodate very large samples.
- One can make probabilistic statements about the chance of having a good starting-point subsample, and this almost leads to a probabilistic guarantee on the result, but not quite.
- Simulation indicates this works well across a range of problems and situations.
- Once structure is found, it can be removed and the process repeated to find second-order structure.