Maintaining and optimizing dependencies between statistical calculations

John Darrington

PSPP, Gnubik

August, 2013

Abstract

Statistical calculations involve iterating a (possibly very large) dataset one or more times. The designer of a statistical analysis tool wants to ensure that no more iterations than necessary are performed. Whereas, on a case by case basis, a statistical calculation can be optimised by inspection, this is not practical in a general purpose statistics tool, where a set of several statistical calculations are to be determined and the elements of the set are, at time of design, unknown. This presentation shows how the use of caching, a dependency graph the optimal number and order of iterations can be determined.
An implementation is presented, which demonstrates how the use of lisp can obviate the need for the programmer to maintain the dependency relationships. Instead, they are extracted from the implicit information contained within the program itself.

The Problem

- Statistical analysis requires iterating the data. Sometimes several iterations are required. However for large datasets, data iteration is expensive.
- Can we find a way to perform the minimum number of iterations and no more?
- Often calculations are unnecessarily repeated. These calculations involve iterating the data at least once. For large datasets, that is expensive. We want to minimize the number of passes.
- PSPP's backend provides an efficient means of iterating large datasets. However PSPP's front end is nasty.

PSPP Background Information

PSPP has few active developers, but (we think) quite a lot of users. Most users are probably windows users. (SF downloads are high. Debian Popcon score is low).

Debian popcon:

inst	vote	old	recent	no-files
330	48	236	46	0

Pspp4windows Downloads in May 2013: 10,323

Users:

- Social scientists (*eg.* Psychologists)
- Students
- Govt. Statisticians

Iterating a dataset

Datasets are streams:

case (i)	value (x_i)	value (y_i)	weight (w_i)
1	9	0.67	1
2	3	1.09	1
3	5	-109	1

Streams are :

- Sequential access. Can be read in-order only.
- Possibly of indeterminate length.
- Immutable. You cannot write to them.
- Single-Use. Once a case has been read, it cannot be read again.

The benefits of caching

Example: The PSPP descriptives command:

```
DESCRIPTIVES VARIABLES = x
/STATISTICS = MEAN.
```

- This user wants to know the arithmetic mean of x.

The benefits of caching

Example: The PSPP descriptives command:

```
DESCRIPTIVES VARIABLES = x
/STATISTICS = MEAN.

DESCRIPTIVES VARIABLES = x
/STATISTICS = SUM.
```

- This user wants to know the arithmetic mean of x.
- Now the user wants to know the sum x as well.

The benefits of starting the processing late

Calculating highest / lowest values in a dataset.

```
EXAMINE VARIABLES = x
/STATISTICS = EXTREME(3).
```

- One solution is to use binary trees to keep the 3 highest/lowest values.
- If we know that the data is sorted on x then the problem is a lot simpler. In general however, we don't know that.

The benefits of starting the processing late

Calculating highest / lowest values in a dataset.

```
EXAMINE VARIABLES = x
/STATISTICS = EXTREME(3)
/PERCENTILE = 10 20 30 40 50 60 70 80 90 .
```

- One solution is to use binary trees to keep the 3 highest/lowest values.
- If we know that the data is sorted on x then the problem is a lot simpler. In general however, we don't know that.
- But! Certain options demand that the data is sorted.

Conclusions so far

- All statistics require iterating the data at least once.
- Some statistics depend on others as intermediate results. Caching makes sense.
- Some dependencies are required before the calculation starts (a priori) whereas others are required only before the calculation can be completed.

Calculating Statistics: Some Examples

Mean:

$$\text{mean} = \frac{\sum x_i w_i}{\sum w_i} = \frac{\text{sum}}{\text{count}}$$

Naive implementation: 2 passes; Optimal implementation: 1 pass.

Variance:

$$\text{variance} = \frac{\sum(\text{mean} - x_i w_i)^2}{\text{count} - 1}$$

Optimal (and stable and simple) implementation: 2 passes.
'Count' needs to be available before the calculation can finish. but 'mean' must be available before the calculation can start.

A complex example

The test statistic, L, is defined as follows:

$$L = \frac{(N-k)}{(k-1)} \frac{\sum_{i=1}^{k} N_i (Z_{i\cdot} - Z_{\cdot\cdot})^2}{\sum_{i=1}^{k} \sum_{j=1}^{N_i} (Z_{ij} - Z_{i\cdot})^2},$$

where:

k	is the number of different groups to which the samples belong,
N	is the total number of samples,
N_i	is the number of samples in the ith group,
Y_{ij}	is the value of the jth sample from the ith group,
Z_{ij}	$= \lvert Y_{ij} - \bar{Y}_{i\cdot} \rvert$,
$\bar{Y}_{i\cdot}$	is the mean of i-th group.
$Z_{\cdot\cdot}$	$= \frac{1}{N} \sum_{i=1}^{k} \sum_{j=1}^{N_i} Z_{ij}$
$Z_{i\cdot}$	$= \frac{1}{N_i} \sum_{j=1}^{N_i} Z_{ij}$

Optimal: 4 ? passes

The Solution

Clever mathematics can reduce some multi-pass algorithms. But multi-pass algorithms are a fact of life.
How to devise a framework to ensure that we are not doing more passes than necessary?
Solution:

- Whenever possible, express calculation of a statistic in terms of simpler statistics.
- Cache all the intermediate statistics.
- Don't start the calculation before you know everything that is required.
- Statistical calculations have three parts
  1. Before (before iteration of the data starts) Eg: $s \leftarrow 0$;
  2. During (during the iteration - once per datum) Eg: $s \leftarrow s + x$;
  3. After (after the iteration is done) Eg: $s \leftarrow s/\mathrm{count}$;

Examples

Example: Count $\sum w_i$

1. Before: $s \leftarrow 0$;
2. During: $s \leftarrow s + w_i$;
3. After: null-op

Examples

Example: Count $\sum w_i$

1. Before: $s \leftarrow 0$;
2. During: $s \leftarrow s + w_i$;
3. After: null-op

Example: Sum $\sum x_i w_i$

1. Before: $s \leftarrow 0$;
2. During: $s \leftarrow s + x_i w_i$;
3. After: null-op

Examples

Example: Count $\sum w_i$

1. Before: $s \leftarrow 0$;
2. During: $s \leftarrow s + w_i$;
3. After: null-op

Example: Sum $\sum x_i w_i$

1. Before: $s \leftarrow 0$;
2. During: $s \leftarrow s + x_i w_i$;
3. After: null-op

Example: Mean $\frac{\sum x_i w_i}{\sum w_i}$

1. Before: null-op
2. During: null-op
3. After: $s \leftarrow \mathrm{Sum}/\mathrm{Count}$.

Statistics have Dependent Relationships

$$\text{variance} = \frac{\sum(\text{mean} - x_i w_i)}{\text{count}}$$
$$\text{mean} = \frac{\text{sum}}{\text{count}}$$

$$\begin{tikzcd}
& \boxed{\text{Mean}} \arrow[dl] \arrow[dr] & & \boxed{\text{Variance}} \arrow[ll, shift left] \arrow[ll, shift right] \arrow[dl] \\
\boxed{\text{Sum}} & & \boxed{\text{Count}} &
\end{tikzcd}$$

How to calculate a statistic in scheme

The three facets of a statistic can be represented by a scheme `alist`.

```
‘(arithmetic-mean . (
     (POST . ,(lambda (r acc) (/ (cached r ’sum ’\#(0))
 (cached r ’count ’\#()))))
     ))

‘(count . (
   (CALC . ,(lambda (r acc x w)  (+ acc w)))
   ))

‘(sum . (
 (CALC . ,(lambda (r acc x w) (+ acc (* (vector-ref x 0) w
 ))
```

Representing dependencies in Scheme

A list of lists defines the dependencies:

```
`(
 (count sum mean)
 (count variance)
)
```

Statistics which post-depend on others can be appended at the end of the same list.

```
`(
 (count sum mean)
 (count variance stddev)
)
```

Optimise! Each statistic need only be calculated once:

```
`(
 (count sum mean)
 (variance stddev)
)
```

If the list is ill-formed (*ie* a dependency is missing) an error will occur. We can use scheme itself to determine the dependencies and generate, and optimise the list automatically.

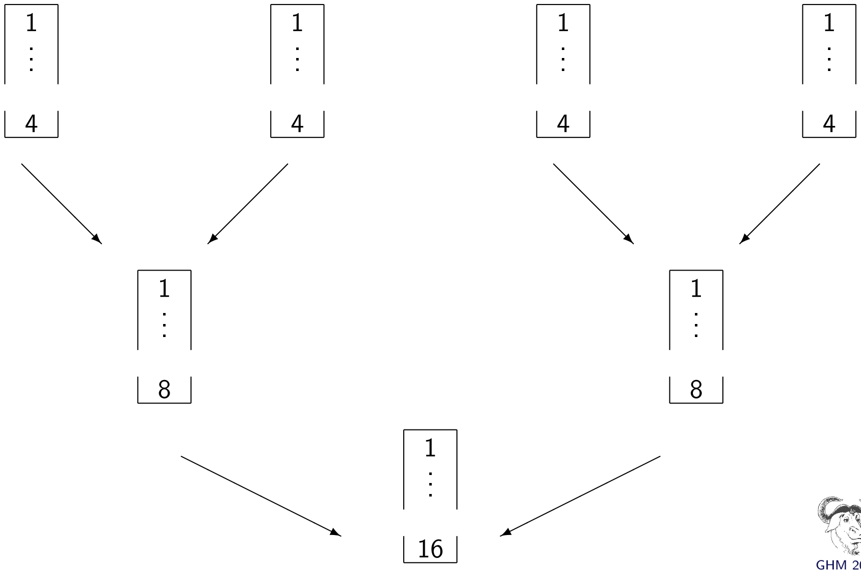
```
;; Return a list of the statistics which are immediate
;; post-dependencies of the  statistic STAT
(define (stat-deps stat)
  (let*(
  (proc (hashq-ref the-statistics (car stat)))
  (ppost (assq-ref proc 'POST))
  (deps (if ppost (get-deps (procedure-source ppost) (cadr stat)) '()))
  )
    (stats-with-deps deps)))


;; Return a list of the stastistics which are immediate
;; pre-dependencies of STAT
(define (immediate-pre-dep stat)
  (let* (
   (s (hashq-ref the-statistics (car stat)))
   (pre (assq-ref s 'CALC))
   (deps (if pre (get-deps (procedure-source pre) (cadr stat)) '()))
   )
  deps))
```

Future Work

Iterative statistics
Some statistics can only be calculated be passing through the data an indeterminate number of times (usually with an upper bound) until some convergence condition is reached.
For example Logistic Regression or Sorting:

Review of the merge sort



What does it mean for GNU ?

- Thoughtful combination of Guile and C can provide an efficient yet flexible statistical analysis system.
- PSPP's backend + Guile could retain the efficiency, yet make writing new procedures accessible to non-hackers.
- R is a free replacement for S. PSPP is a free replacement for SPSS. DAP is a free replacement for SAS. We could create a statistical analysis tool which combines the advantages of these - which is truly unique to GNU.