Title

stata.com

stgen — Generate variables reflecting entire histories

Description

`stgen` provides a convenient way to generate new variables reflecting entire histories. These functions are intended for use with multiple-record survival data but may be used with single-record data. With single-record data, each function reduces to one `generate`, and `generate` would be a more natural way to approach the problem.

`stgen` can be used with single- or multiple-failure st data.

If you want to generate calculated values, such as the survivor function, see [ST] **sts**.

Quick start

Create binary indicator `newv1` equal to 1 in all records for a subject if `v1` = 1 at any time using multiple-record `stset` data

```
stgen newv1 = ever(v1==1)
```

Create `newv2` containing the time when `v2` is first greater than 5 for the subject

```
stgen newv2 = when(v2>5)
```

As above, but assume `v2` > 5 becomes true at the beginning instead of at the end of the corresponding record

```
stgen newv2 = when0(v2>5)
```

Create `newv3` containing the cumulative number of records with `v1` = 1 for the subject

```
stgen newv3 = count(v1==1)
```

As above, but assume `v1` = 1 becomes true at the beginning instead of at the end of the corresponding record

```
stgen newv3 = count0(v1==1)
```

Create `newv4` containing the cumulative number of gaps for the subject

```
stgen newv4 = ngaps()
```

Menu

Statistics > Survival analysis > Setup and utilities > Generate variable reflecting entire histories

Syntax

```
stgen [ type ] newvar = function
```

where *function* is

```
ever(exp)
never(exp)
always(exp)
min(exp)
max(exp)
when(exp)
when0(exp)
count(exp)
count0(exp)
minage(exp)
maxage(exp)
avgage(exp)
nfailures()
ngaps()
gaplen()
hasgap()
```

You must `stset` your data before using `stgen`; see [ST] **stset**.

Functions

In the description of the functions below, time units refer to the same units as *timevar* from `stset` *timevar*`,` .... For instance, if *timevar* is the number of days since 01 January 1960 (a Stata date), time units are days. If *timevar* is in years—years since 1960, years since diagnosis, or whatever—time units are years.

When we say variable X records a "time", we mean a variable that records when something occurred in the same units and with the same base as *timevar*. If *timevar* is a Stata date, "time" is correspondingly a Stata date.

t units, or analysis-time units, refer to a variable in the units *timevar*`/scale()` from `stset` *timevar*`, scale(...)` .... If you did not specify a `scale()`, t units are the same as time units. Alternatively, say that *timevar* is recorded as a Stata date and you specified `scale(365.25)`. Then t units are years. If you specified a nonconstant scale—`scale(myvar)`, where `myvar` varies from subject to subject—t units are different for every subject.

"An analysis time" refers to the time something occurred, recorded in the units (*timevar*`-origin())/scale()`. We speak about analysis time only in terms of the beginning and end of each time-span record.

Although in *Description* above we said that `stgen` creates variables reflecting entire histories, `stgen` restricts itself to the `stset` observations, so "entire history" means the entire history as it is currently `stset`. If you really want to use entire histories as recorded in the data, type `streset, past` or `streset, past future` before using `stgen`. Then type `streset` to reset to the original analysis sample.

The following functions are available:

`ever(`*exp*`)` creates *newvar* containing 1 (true) if the expression is ever true (nonzero) and 0 otherwise. For instance,

```
. stgen everlow = ever(bp<100)
```

would create `everlow` containing, for each subject, uniformly 1 or 0. Every record for a subject would contain `everlow` = 1 if, on any `stset` record for the subject, `bp` < 100; otherwise, `everlow` would be 0.

`never(`*exp*`)` is the reverse of `ever()`; it creates *newvar* containing 1 (true) if the expression is always false (0) and 0 otherwise. For instance,

```
. stgen neverlow = never(bp<100)
```

would create `neverlow` containing, for each subject, uniformly 1 or 0. Every record for a subject would contain `neverlow` = 1 if, on every `stset` record for the subject, `bp` < 100 is false.

`always(`*exp*`)` creates *newvar* containing 1 (true) if the expression is always true (nonzero) and 0 otherwise. For instance,

```
. stgen lowlow = always(bp<100)
```

would create `lowlow` containing, for each subject, uniformly 1 or 0. Every record for a subject would contain `lowlow` = 1 if, on every `stset` record for a subject, `bp` < 100.

`min(`*exp*`)` and `max(`*exp*`)` create *newvar* containing the minimum or maximum nonmissing value of *exp* within `id()`. `min()` and `max()` are often used with variables recording a time (see definition above), such as `min(visitdat)`.

`when(`*exp*`)` and `when0(`*exp*`)` create *newvar* containing the time when *exp* first became true within the previously `stset id()`. The result is in time, not t units; see the definition above.

`when()` and `when0()` differ about when the *exp* became true. Records record time spans $(time0, time1]$. `when()` assumes that the expression became true at the end of the time span, *time1*. `when0()` assumes that the expression became true at the beginning of the time span, *time0*.

Assume that you previously `stset myt, failure(`*eventvar*`=...)` .... `when()` would be appropriate for use with *eventvar*, and, presumably, `when0()` would be appropriate for use with the remaining variables.

`count(`*exp*`)` and `count0(`*exp*`)` create *newvar* containing the number of occurrences when *exp* is true within `id()`.

`count()` and `count0()` differ in when they assume that *exp* occurs. `count()` assumes that *exp* corresponds to the end of the time-span record. Thus even if *exp* is true in this record, the count would remain unchanged until the next record.

`count0()` assumes that *exp* corresponds to the beginning of the time-span record. Thus if *exp* is true in this record, the count is immediately updated.

For example, assume that you previously `stset myt, failure(`*eventvar*`=...)` .... `count()` would be appropriate for use with *eventvar*, and, presumably, `count0()` would be appropriate for use with the remaining variables.

`minage(`*exp*`)`, `maxage(`*exp*`)`, and `avgage(`*exp*`)` return the elapsed time, in time units, because *exp* is at the beginning, end, or middle of the record, respectively. *exp* is expected to evaluate to a time in time units. `minage()`, `maxage()`, and `avgage()` would be appropriate for use with the result of `when()`, `when0()`, `min()`, and `max()`, for instance.

Also see [ST] **stsplit**; `stsplit` will divide the time-span records into new time-span records that record specified intervals of ages.

`nfailures()` creates *newvar* containing the cumulative number of failures for each subject as of the entry time for the observation. `nfailures()` is intended for use with multiple-failure data; with single-failure data, `nfailures()` is always 0. In multiple-failure data,

```
. stgen nfail = nfailures()
```

might create, for a particular subject, the following:

id	time0	time1	fail	x	nfail
93	0	20	0	1	0
93	20	30	1	1	0
93	30	40	1	2	1
93	40	60	0	1	2
93	60	70	0	2	2
93	70	80	1	1	2

The total number of failures for this subject is 3, and yet the maximum of the new variable `nfail` is 2. At time 70, the beginning of the last record, there had been two failures previously, and there were two failures up to but not including time 80.

`ngaps()` creates *newvar* containing the cumulative number of gaps for each subject as of the entry time for the record. Delayed entry (an opening gap) is not considered a gap. For example,

```
. stgen ngap = ngaps()
```

might create, for a particular subject, the following:

id	time0	time1	fail	x	ngap
94	10	30	0	1	0
94	30	40	0	2	0
94	50	60	0	1	1
94	60	70	0	2	1
94	82	90	1	1	2

`gaplen()` creates *newvar* containing the time on gap, measured in analysis-time units, for each subject as of the entry time for the observation. Delayed entry (an opening gap) is not considered a gap. Continuing with the previous example,

```
. stgen gl = gaplen()
```

would produce

id	time0	time1	fail	x	ngap	gl
94	10	30	0	1	0	0
94	30	40	0	2	0	0
94	50	60	0	1	1	10
94	60	70	0	2	1	0
94	82	90	1	1	2	12

`hasgap()` creates *newvar* containing uniformly 1 if the subject ever has a gap and 0 otherwise. Delayed entry (an opening gap) is not considered a gap.

Remarks and examples

stata.com

`stgen` does nothing you cannot do in other ways, but it is convenient.

Consider how you would obtain results like those created by `stgen` should you need something that `stgen` will not create for you. Say that we have an st dataset for which we have previously

```
. stset t, failure(d) id(id)
```

Assume that these are some of the data:

```
id          t          d         bp
27         30          0         90
27         50          0        110
27         60          1         85
28         11          0        120
28         40          1        130
```

If we were to type

```
. stgen everlow = ever(bp<100)
```

the new variable, `everlow`, would contain for these two subjects

```
id          t          d         bp   everlow
27         30          0         90         1
27         50          0        110         1
27         60          1         85         1
28         11          0        120         0
28         40          1        130         0
```

Variable `everlow` is 1 for subject 27 because, in two of the three observations, `bp` $<$ 100, and `everlow` is 0 for subject 28 because `everlow` is never less than 100 in either observation.

Here is one way we could have created `everlow` for ourselves:

```
. generate islow = bp<100
. sort id
. by id: generate sumislow = sum(islow)
. by id: generate everlow = sumislow[_N]>0
. drop islow sumislow
```

The generic term for code like this is explicit subscripting; see **[U] 13.7 Explicit subscripting**.

Anyway, that is what `stgen` did for us, although, internally, `stgen` used denser code that was equivalent to

```
. by id, sort: generate everlow=sum(bp<100)
. by id: replace everlow = everlow[_N]>0
```

Obtaining things like the time on gap is no more difficult. When we `stset` the data, `stset` created variable `_t0` to record the entry time. `stgen`'s `gaplen()` function is equivalent to

```
. sort id _t
. by id: generate gaplen = _t0-_t[_n-1]
. by id: replace gaplen = 0 if _n == 1
```

Seeing this, you should realize that if all you wanted was the cumulative length of the gap before the current record, you could type

```
. sort id _t
. by id: generate curgap = sum(_t0-_t[_n-1])
```

If, instead, you wanted a variable that was 1 if there were a gap just before this record and 0 otherwise, you could type

```
. sort id _t
. by id: generate iscurgap = (_t0-_t[_n-1])>0
```

Example 1

Let's use the stgen commands to real effect. We have a multiple-record, multiple-failure dataset.

```
. use http://www.stata-press.com/data/r15/mrmf, clear
. st
-> stset t, id(id) failure(d) time0(t0) exit(time .) noshow
                id:  id
     failure event:  d != 0 & d < .
obs. time interval:  (t0, t]
 exit on or before:  time .
. stdescribe
```

Category	total	per subject: mean	min	median	max
no. of subjects	926				
no. of records	1734	1.87257	1	2	4
(first) entry time		0	0	0	0
(final) exit time		470.6857	1	477	960
subjects with gap	6				
time on gap if gap	411	68.5	16	57.5	133
time at risk	435444	470.2419	1	477	960
failures	808	.8725702	0	1	3

Also in this dataset are two covariates, x1 and x2. We wish to fit a Cox model on these data but wish to assume that the baseline hazard for first failures is different from that for second and later failures.

Our data contain six subjects with gaps. Because failures might have occurred during the gap, we begin by dropping those six subjects:

```
. stgen hg = hasgap()
. drop if hg
(14 observations deleted)
```

The six subjects had 14 records among them. We can now create variable nf containing the number of failures and, from that, create variable group, which will be 0 when subjects have experienced no previous failures and 1 thereafter:

```
. stgen nf = nfailures()
. generate byte group = nf>0
```

We can now fit our stratified model:

```
. stcox x1 x2, strata(group) vce(robust)
Iteration 0:   log pseudolikelihood = -4499.9966
Iteration 1:   log pseudolikelihood = -4444.7797
Iteration 2:   log pseudolikelihood = -4444.4596
Iteration 3:   log pseudolikelihood = -4444.4596
Refining estimates:
Iteration 0:   log pseudolikelihood = -4444.4596
Stratified Cox regr. -- Breslow method for ties
No. of subjects      =          920             Number of obs    =       1,720
No. of failures      =          800
Time at risk         =       432153
                                                 Wald chi2(2)     =      102.78
Log pseudolikelihood =   -4444.4596             Prob > chi2      =      0.0000
                                    (Std. Err. adjusted for 920 clusters in id)
------------------------------------------------------------------------------
          _t | Haz. Ratio   Robust Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   2.087903   .1961725     7.84   0.000     1.736738    2.510074
          x2 |   .2765613    .052277    -6.80   0.000     .1909383    .4005806
------------------------------------------------------------------------------
                                                          Stratified by group
```

◁

Also see

[ST] **stci** — Confidence intervals for means and percentiles of survival time

[ST] **sts** — Generate, graph, list, and test the survivor and cumulative hazard functions

[ST] **stset** — Declare data to be survival-time data

[ST] **stvary** — Report variables that vary over time