Imperial College London

GA²LEN Global Allergy and Asthma European Network
Network of Excellence

Creating factor variables in resultssets and other datasets

Roger B. Newson
r.newson@imperial.ac.uk
http://www.imperial.ac.uk/nhli/r.newson/

National Heart and Lung Institute, Imperial College London

19th UK Stata Users' Group Meeting, 12–13 September, 2013
Downloadable from the conference website at
http://ideas.repec.org/s/boc/usug13.html

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *thing*s should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *thing*s should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *thing*s should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *things* should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *thing*s should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *thing*s should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

The importance of string–factor (and factor–string) conversion

- Traditionally, Stata documentation has encouraged users to `encode` string variables to value–labelled integer variables to save space.
- Nowadays, a commoner reason to `encode` is that string variables cannot be axis variables in graphs.
- More generally, a well–formed Stata dataset should have one observation per *thing* and data on *attributes_of _things*.
- And the *thing*s should be identified by **key variables**, by which the observations are sorted and identified uniquely.
- And, if these key variables are string, then they cannot be sorted non–alphabetically.
- *On the other hand*, numeric variables (labelled or otherwise) often need the addition of prefixes, suffixes and/or conversion of exponents before being output to TEX, HTML, RTF or SMCL.

Some SSC programs for string–factor and factor–string conversion

Official Stata's `encode`, `decode`, `destring` and `tostring` commands seemed insufficient for what I wanted to do. So, over time, I accumulated some conversion packages of my own:

Command	*Description*
String–factor:	
`sencode`	"Super" version of `encode`
`fvregen`	Extract factors from a parameter–name string variable in a resultsset
`factext`	Extract factors from a parameter–label string variable in a resultsset
Factor–string:	
`sdecode`	"Super" version of `decode`
`msdecode`	Multi–factor version of `sdecode`
`factmerg`	Generate factor name, label and level string variables from multiple input factors
`insingap`	Insert labelled gap observations at start of by–groups
`bmjcip`	Decode estimates, confidence limits, and *P*– and *Q*–values

Resultssets may be generated by the SSC package `parmest`. The programs `msdecode`, `factmerg`, `insingap` and `bmjcip` all use `sdecode`.

The sencode package for string–factor conversion

- sencode is a "super" version of encode, downloadable (and frequently downloaded) from SSC.
- It has a replace option, so the output numeric variable can inherit the name and position of the input string variable.
- It has a gsort() option, specifying a list of existing variables (defaulting to _n), which determine the primary, non–alphabetic order in which the output numeric values will be allocated (breaking ties alphabetically).
- It has a manyto1 option, specifying that multiple gsort() groups of observations with the same input string value can have different output numeric values, instead of being combined in order of first appearance of the input string value.
- sencode has been evolving since 2001. *However*, we will be presenting some useful tips, accumulated since then, that are not immediately obvious.

The sencode package for string–factor conversion

- `sencode` is a "super" version of `encode`, downloadable (and frequently downloaded) from SSC.
- It has a `replace` option, so the output numeric variable can inherit the name and position of the input string variable.
- It has a `gsort()` option, specifying a list of existing variables (defaulting to `_n`), which determine the primary, non–alphabetic order in which the output numeric values will be allocated (breaking ties alphabetically).
- It has a `manyto1` option, specifying that multiple `gsort()` groups of observations with the same input string value can have different output numeric values, instead of being combined in order of first appearance of the input string value.
- `sencode` has been evolving since 2001. *However*, we will be presenting some useful tips, accumulated since then, that are not immediately obvious.

The sencode package for string–factor conversion

- `sencode` is a "super" version of `encode`, downloadable (and frequently downloaded) from SSC.
- It has a `replace` option, so the output numeric variable can inherit the name and position of the input string variable.
- It has a `gsort()` option, specifying a list of existing variables (defaulting to `_n`), which determine the primary, non–alphabetic order in which the output numeric values will be allocated (breaking ties alphabetically).
- It has a `manyto1` option, specifying that multiple `gsort()` groups of observations with the same input string value can have different output numeric values, instead of being combined in order of first appearance of the input string value.
- `sencode` has been evolving since 2001. *However*, we will be presenting some useful tips, accumulated since then, that are not immediately obvious.

The sencode package for string–factor conversion

- `sencode` is a "super" version of `encode`, downloadable (and frequently downloaded) from SSC.
- It has a `replace` option, so the output numeric variable can inherit the name and position of the input string variable.
- It has a `gsort()` option, specifying a list of existing variables (defaulting to `_n`), which determine the primary, non–alphabetic order in which the output numeric values will be allocated (breaking ties alphabetically).
- It has a `manytol` option, specifying that multiple `gsort()` groups of observations with the same input string value can have different output numeric values, instead of being combined in order of first appearance of the input string value.
- `sencode` has been evolving since 2001. *However*, we will be presenting some useful tips, accumulated since then, that are not immediately obvious.

The sencode package for string–factor conversion

- `sencode` is a "super" version of `encode`, downloadable (and frequently downloaded) from SSC.
- It has a `replace` option, so the output numeric variable can inherit the name and position of the input string variable.
- It has a `gsort()` option, specifying a list of existing variables (defaulting to `_n`), which determine the primary, non–alphabetic order in which the output numeric values will be allocated (breaking ties alphabetically).
- It has a `manyto1` option, specifying that multiple `gsort()` groups of observations with the same input string value can have different output numeric values, instead of being combined in order of first appearance of the input string value.
- `sencode` has been evolving since 2001. *However*, we will be presenting some useful tips, accumulated since then, that are not immediately obvious.

The sencode package for string–factor conversion

- `sencode` is a "super" version of `encode`, downloadable (and frequently downloaded) from SSC.
- It has a `replace` option, so the output numeric variable can inherit the name and position of the input string variable.
- It has a `gsort()` option, specifying a list of existing variables (defaulting to `_n`), which determine the primary, non–alphabetic order in which the output numeric values will be allocated (breaking ties alphabetically).
- It has a `manyto1` option, specifying that multiple `gsort()` groups of observations with the same input string value can have different output numeric values, instead of being combined in order of first appearance of the input string value.
- `sencode` has been evolving since 2001. *However*, we will be presenting some useful tips, accumulated since then, that are not immediately obvious.

Example: Keying and plotting the `auto` data

- The familiar `auto` dataset, loadable using `sysuse`, is not really a well–formed Stata dataset.
- The `describe` command shows that it is sorted by the binary variable `foreign`, which indicates US or non–US origin for a car model, but does not identify the car models uniquely.
- The models are in fact identified uniquely by the string variable `make`.
- We might like to encode the variable `make` to a labelled numeric variable, which we can then sort by, and plot against other variables in the dataset.
- And, to show off, we will order the new variable primarily by descending weight, breaking tied weights alphabetically.

Example: Keying and plotting the `auto` data

- The familiar `auto` dataset, loadable using `sysuse`, is not really a well–formed Stata dataset.
- The `describe` command shows that it is sorted by the binary variable `foreign`, which indicates US or non–US origin for a car model, but does not identify the car models uniquely.
- The models are in fact identified uniquely by the string variable `make`.
- We might like to encode the variable `make` to a labelled numeric variable, which we can then sort by, and plot against other variables in the dataset.
- And, to show off, we will order the new variable primarily by descending weight, breaking tied weights alphabetically.

Example: Keying and plotting the `auto` data

- The familiar `auto` dataset, loadable using `sysuse`, is not really a well–formed Stata dataset.
- The `describe` command shows that it is sorted by the binary variable `foreign`, which indicates US or non–US origin for a car model, but does not identify the car models uniquely.
- The models are in fact identified uniquely by the string variable `make`.
- We might like to encode the variable `make` to a labelled numeric variable, which we can then sort by, and plot against other variables in the dataset.
- And, to show off, we will order the new variable primarily by descending weight, breaking tied weights alphabetically.

Example: Keying and plotting the `auto` data

- The familiar `auto` dataset, loadable using `sysuse`, is not really a well–formed Stata dataset.
- The `describe` command shows that it is sorted by the binary variable `foreign`, which indicates US or non–US origin for a car model, but does not identify the car models uniquely.
- The models are in fact identified uniquely by the string variable `make`.
- We might like to encode the variable `make` to a labelled numeric variable, which we can then sort by, and plot against other variables in the dataset.
- And, to show off, we will order the new variable primarily by descending weight, breaking tied weights alphabetically.

Example: Keying and plotting the `auto` data

- The familiar `auto` dataset, loadable using `sysuse`, is not really a well–formed Stata dataset.
- The `describe` command shows that it is sorted by the binary variable `foreign`, which indicates US or non–US origin for a car model, but does not identify the car models uniquely.
- The models are in fact identified uniquely by the string variable `make`.
- We might like to encode the variable `make` to a labelled numeric variable, which we can then sort by, and plot against other variables in the dataset.
- And, to show off, we will order the new variable primarily by descending weight, breaking tied weights alphabetically.

Example: Keying and plotting the `auto` data

- The familiar `auto` dataset, loadable using `sysuse`, is not really a well–formed Stata dataset.
- The `describe` command shows that it is sorted by the binary variable `foreign`, which indicates US or non–US origin for a car model, but does not identify the car models uniquely.
- The models are in fact identified uniquely by the string variable `make`.
- We might like to encode the variable `make` to a labelled numeric variable, which we can then sort by, and plot against other variables in the dataset.
- And, to show off, we will order the new variable primarily by descending weight, breaking tied weights alphabetically.

The `auto` data

We load the `auto` dataset, and then `describe` it:

```
. sysuse auto, clear;
(1978 Automobile Data)

. describe;

Contains data from C:\Program Files (x86)\Stata12\ado\base/a/auto.dta
  obs:            74                          1978 Automobile Data
 vars:            12                          13 Apr 2011 17:45
 size:         3,182                          (_dta has notes)
------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------------------------
make            str18  %-18s                  Make and Model
price           int    %8.0gc                 Price
mpg             int    %8.0g                  Mileage (mpg)
rep78           int    %8.0g                  Repair Record 1978
headroom        float  %6.1f                  Headroom (in.)
trunk           int    %8.0g                  Trunk space (cu. ft.)
weight          int    %8.0gc                 Weight (lbs.)
length          int    %8.0g                  Length (in.)
turn            int    %8.0g                  Turn Circle (ft.)
displacement    int    %8.0g                  Displacement (cu. in.)
gear_ratio      float  %6.2f                  Gear Ratio
foreign         byte   %8.0g       origin     Car type
------------------------------------------------------------------------------
Sorted by:  foreign
```

It is sorted by `foreign` (2 values), but has 74 observations. *So*. . .

Generating a new ID variable using `sencode`

. . .we then use `sencode`, with the `gsort()` option, to generate a new factor variable `make2`, ordered by descending weight. Then, we sort the dataset by this new ID variable:

```
. sencode make, gsort(-weight) generate(make2);

. describe make2;

              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------------------------
make2           byte   %17.0g      make2      Make and Model

. keyby make2;
```

The `keyby` package can be downloaded from SSC. It is an extension of `sort`, and checks that the sort key variables uniquely identify the observations, and moves the key variables to the start of the variable order (unless the user specifies the `noorder` option).

The new improved `auto` dataset

We now `describe` the improved `auto` dataset:

```
. describe;

Contains data from C:\Program Files (x86)\Stata12\ado\base/a/auto.dta
  obs:            74                          1978 Automobile Data
 vars:            13                          13 Apr 2011 17:45
 size:         3,256                          (_dta has notes)
------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------------------------
make2           byte   %17.0g      make2      Make and Model
make            str18  %-18s                  Make and Model
price           int    %8.0gc                 Price
mpg             int    %8.0g                  Mileage (mpg)
rep78           int    %8.0g                  Repair Record 1978
headroom        float  %6.1f                  Headroom (in.)
trunk           int    %8.0g                  Trunk space (cu. ft.)
weight          int    %8.0gc                 Weight (lbs.)
length          int    %8.0g                  Length (in.)
turn            int    %8.0g                  Turn Circle (ft.)
displacement    int    %8.0g                  Displacement (cu. in.)
gear_ratio      float  %6.2f                  Gear Ratio
foreign         byte   %8.0g       origin     Car type
------------------------------------------------------------------------------
Sorted by:  make2
     Note:  dataset has changed since last saved
```

The dataset is now sorted (and keyed) by the new labelled numeric ID variable `make2`, which is now the first variable. *However*...

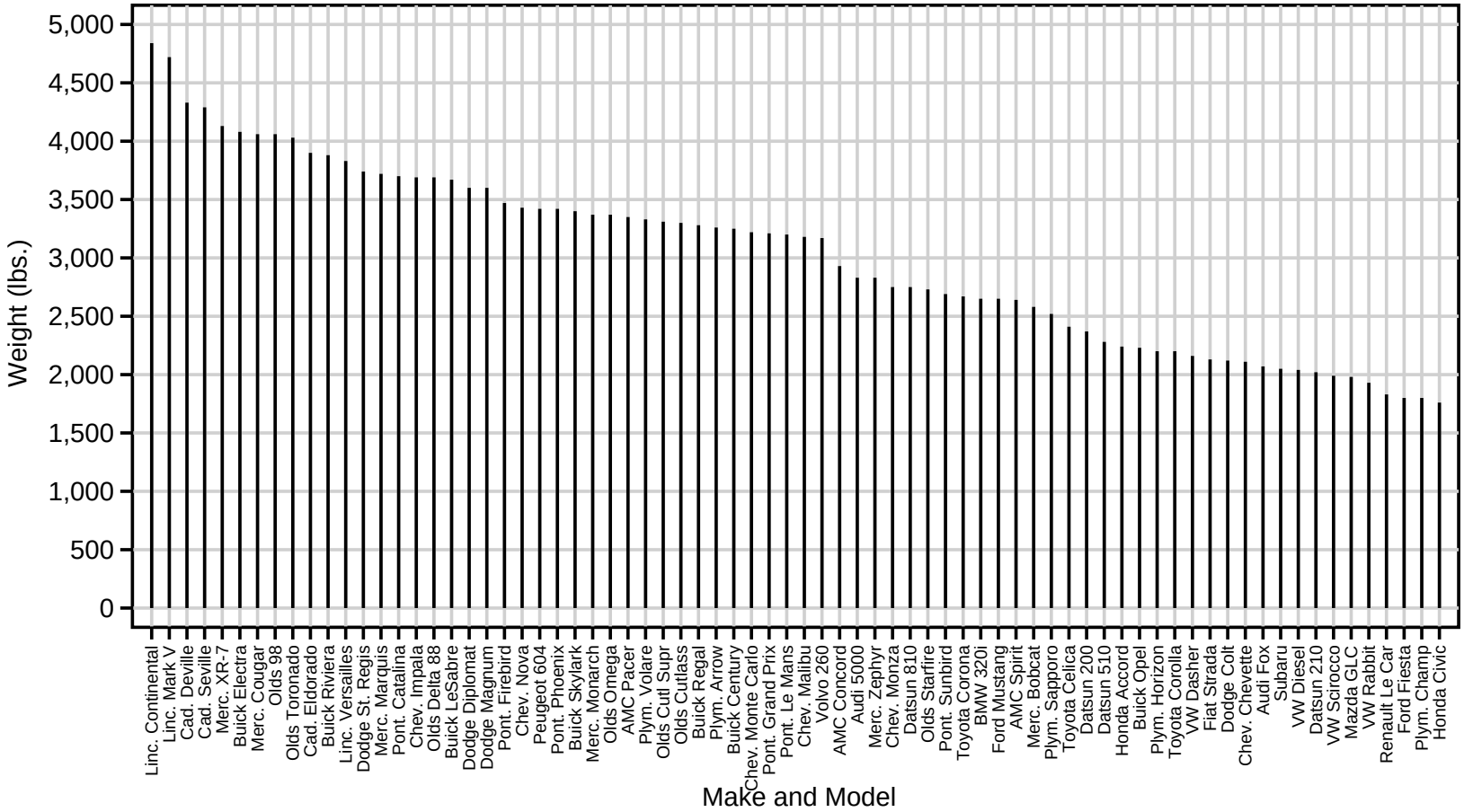
Using a `sencode` output factor on a graph axis

. . . to see what this new factor looks like, we use it to make a spike graph of car weights, using `twoway spike`:

```
. twoway spike weight make2,
>    ylabel(0(500)5000)
>    xlabel(1(1)74, labsize(2) angle(90))
>    xsize(6.5) ysize(3.75);
```

(Note that we could not have done this with the original string variable `make`.)

Weights of cars in the `auto` dataset



The models are ordered by descending weight, with the very few tied weights broken alphabetically.

Producing subset plots by decoding and re–encoding

- We often want to plot subsets of a dataset.
- *For instance*, we might want to produce a version of the previous plot, restricted to cars costing at least 9,000 1978 US dollars.
- When we do this, it often makes sense to decode the factor to string for the subset (using `sdecode`), and then to encode it back (using `sencode`).
- `sdecode`ing and `sencode`ing back is a commonly–used trick, as we shall see later.

Producing subset plots by decoding and re–encoding

- We often want to plot subsets of a dataset.
- *For instance*, we might want to produce a version of the previous plot, restricted to cars costing at least 9,000 1978 US dollars.
- When we do this, it often makes sense to decode the factor to string for the subset (using `sdecode`), and then to encode it back (using `sencode`).
- `sdecode`ing and `sencode`ing back is a commonly–used trick, as we shall see later.

Producing subset plots by decoding and re–encoding

- We often want to plot subsets of a dataset.
- *For instance*, we might want to produce a version of the previous plot, restricted to cars costing at least 9,000 1978 US dollars.
- When we do this, it often makes sense to decode the factor to string for the subset (using `sdecode`), and then to encode it back (using `sencode`).
- `sdecode`ing and `sencode`ing back is a commonly–used trick, as we shall see later.

Producing subset plots by decoding and re–encoding

- We often want to plot subsets of a dataset.
- *For instance*, we might want to produce a version of the previous plot, restricted to cars costing at least 9,000 1978 US dollars.
- When we do this, it often makes sense to decode the factor to string for the subset (using `sdecode`), and then to encode it back (using `sencode`).
- `sdecode`ing and `sencode`ing back is a commonly–used trick, as we shall see later.

Producing subset plots by decoding and re–encoding

- We often want to plot subsets of a dataset.
- *For instance*, we might want to produce a version of the previous plot, restricted to cars costing at least 9,000 1978 US dollars.
- When we do this, it often makes sense to decode the factor to string for the subset (using `sdecode`), and then to encode it back (using `sencode`).
- `sdecode`ing and `sencode`ing back is a commonly–used trick, as we shall see later.

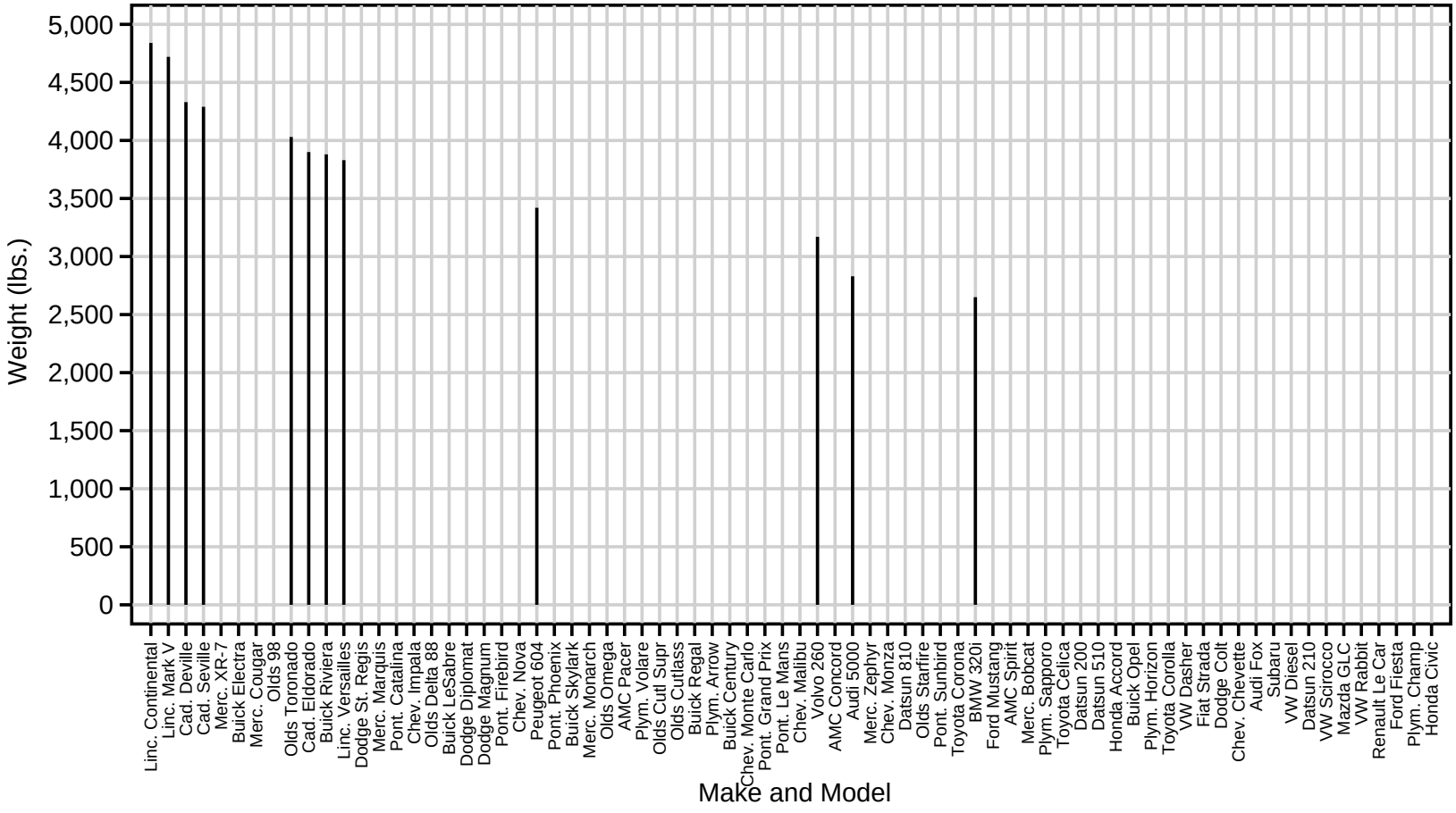
How do we make our subset plot? (Take 1)

We might be tempted to repeat the previous `twoway spike` command, adding only an `if` qualifier, as follows:

```
. twoway spike weight make2 if price>=9000,
>    ylabel(0(500)5000)
>    xlabel(1(1)74, labsize(2) angle(90))
>    xsize(6.5) ysize(3.75);
```

This might seem sensible at first. *However...*

Weights of cars costing at least 9,000 US dollars (take 1)



. . .this is not the graph we really wanted. The subset X–labels are unevenly spaced, and unwanted models are still listed, because the cars are still numbered as before. So what *should* we have done?

How do we make our subset plot? (Take 2)

A better way is to decode `make2`, with the `if` qualifier, to a string variable `make3`, and to encode `make3` back to numeric with the same ordering. This method has the added advantage that we can *italicize* the axis labels, using the `prefix()` and `suffix()` options of `sdecode` to add SMCL prefixes and suffixes:

```
. sdecode make2 if price>=9000, generate(make3) prefix("{it:") suffix("}");

. sencode make3, replace gsort(make2);

. describe make2 make3;

              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------------------------
make2           byte   %17.0g      make2      Make and Model
make3           byte   %22.0g      make3      Make and Model
```

We now have 2 factor variables with different value labels.

How do we make our subset plot? (Take 2, continued)

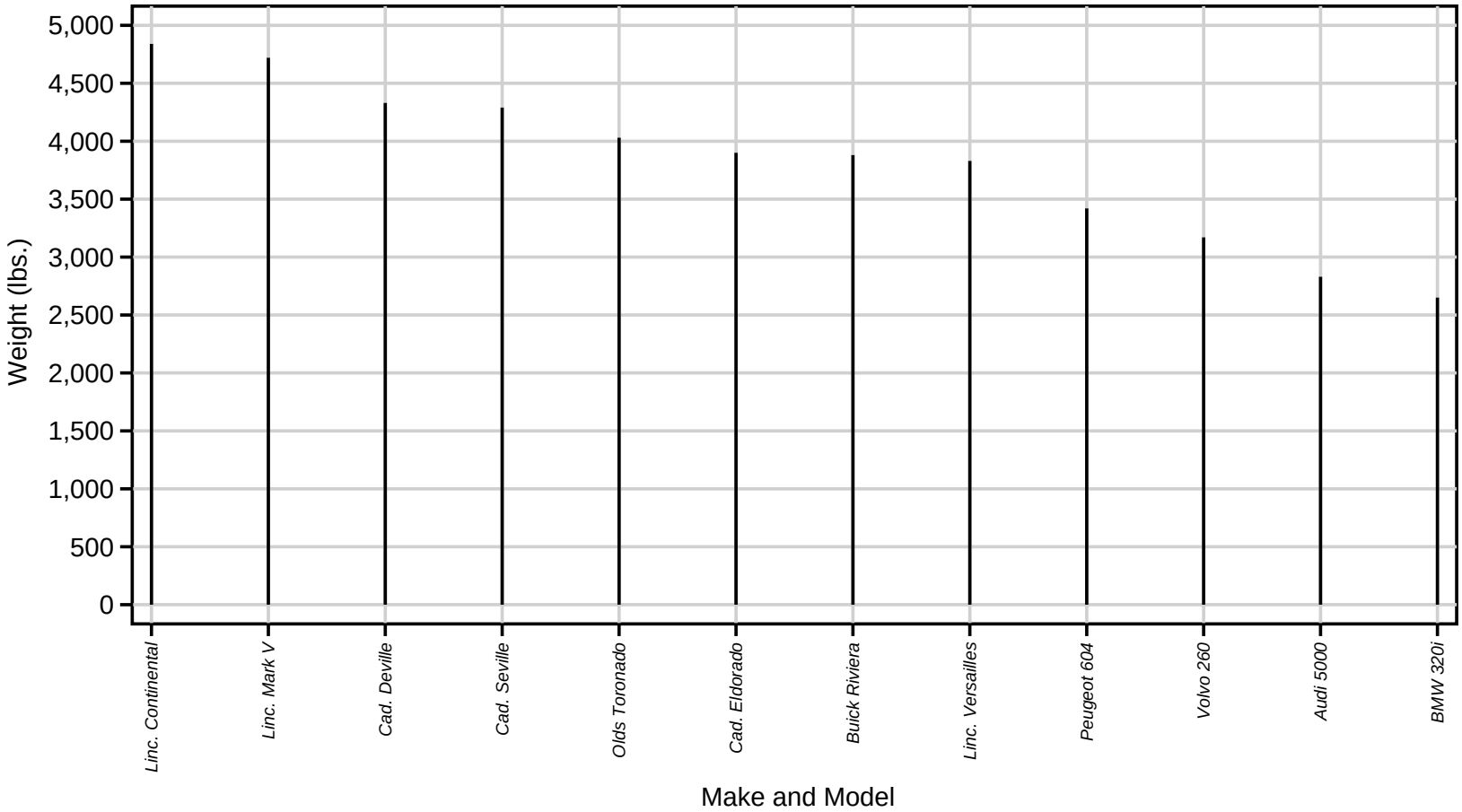
And, this time, when we make the plot, we use `levelsof` to extract the list of values of `make3` to a local macro `xlabs`, and use this macro to specify the *X*–axis labels for our `twoway spike` graph:

```
. levelsof make3, local(xlabs);
1 2 3 4 5 6 7 8 9 10 11 12

. twoway spike weight make3,
>   ylabel(0(500)5000)
>   xlabel(`xlabs', labsize(2) angle(90))
>   xsize(6.5) ysize(3.75);
```

So what does this graph look like?

Weights of cars costing at least 9,000 US dollars (take 2)



This looks more like the graph we wanted. The *X*–axis labels are now subsetted, evenly spaced *and* italicized.

Bringing order to concatenated resultssets

- A **resultsset** is a Stata dataset produced as output by a Stata command, such as the ones in the SSC package `parmest`[1].
- These resultssets are frequently **concatenated**, using `append`, especially when we make multiple resultssets containing parameter estimates from multiple model fits.
- And they often contain a **string ID variable**, specified by the `idstr()` option, identifying which resultsset an observation came from.
- In the concatenated resultsset, we usually use `sencode` to extract numeric factors from the string ID variable.
- These factors are then used for sorting and/or plotting the concatenated resultsset.

Bringing order to concatenated resultssets

- A **resultsset** is a Stata dataset produced as output by a Stata command, such as the ones in the SSC package `parmest`[1].
- These resultssets are frequently **concatenated**, using `append`, especially when we make multiple resultssets containing parameter estimates from multiple model fits.
- And they often contain a **string ID variable**, specified by the `idstr()` option, identifying which resultsset an observation came from.
- In the concatenated resultsset, we usually use `sencode` to extract numeric factors from the string ID variable.
- These factors are then used for sorting and/or plotting the concatenated resultsset.

Bringing order to concatenated resultssets

- A **resultsset** is a Stata dataset produced as output by a Stata command, such as the ones in the SSC package `parmest`[1].
- These resultssets are frequently **concatenated**, using `append`, especially when we make multiple resultssets containing parameter estimates from multiple model fits.
- And they often contain a **string ID variable**, specified by the `idstr()` option, identifying which resultsset an observation came from.
- In the concatenated resultsset, we usually use `sencode` to extract numeric factors from the string ID variable.
- These factors are then used for sorting and/or plotting the concatenated resultsset.

Bringing order to concatenated resultssets

- A **resultsset** is a Stata dataset produced as output by a Stata command, such as the ones in the SSC package `parmest`[1].
- These resultssets are frequently **concatenated**, using `append`, especially when we make multiple resultssets containing parameter estimates from multiple model fits.
- And they often contain a **string ID variable**, specified by the `idstr()` option, identifying which resultsset an observation came from.
- In the concatenated resultsset, we usually use `sencode` to extract numeric factors from the string ID variable.
- These factors are then used for sorting and/or plotting the concatenated resultsset.

Bringing order to concatenated resultssets

- A **resultsset** is a Stata dataset produced as output by a Stata command, such as the ones in the SSC package `parmest`[1].
- These resultssets are frequently **concatenated**, using `append`, especially when we make multiple resultssets containing parameter estimates from multiple model fits.
- And they often contain a **string ID variable**, specified by the `idstr()` option, identifying which resultsset an observation came from.
- In the concatenated resultsset, we usually use `sencode` to extract numeric factors from the string ID variable.
- These factors are then used for sorting and/or plotting the concatenated resultsset.

Bringing order to concatenated resultssets

- A **resultsset** is a Stata dataset produced as output by a Stata command, such as the ones in the SSC package `parmest`[1].
- These resultssets are frequently **concatenated**, using `append`, especially when we make multiple resultssets containing parameter estimates from multiple model fits.
- And they often contain a **string ID variable**, specified by the `idstr()` option, identifying which resultsset an observation came from.
- In the concatenated resultsset, we usually use `sencode` to extract numeric factors from the string ID variable.
- These factors are then used for sorting and/or plotting the concatenated resultsset.

Code to produce a concatenated resultsset in the `auto` dataset

This alien–looking code (which you do *not* need to memorize) uses the `parmby` module of the `parmest` package to fit unadjusted and weight–adjusted regression models of `mpg` with respect to `foreign`, with confidence limits from the unequal–variance and equal–variance formulas. The parameters from these 4 estimations are saved in 4 temporary `parmby` resultssets, identified using the `idstr()` option, which are then concatenated into the memory using `append`, after the old dataset has been cleared:

```
tempfile tf1 tf2 tf3 tf4;
parmby "regress mpg foreign, vce(robust)",
  idstr("Unequal&Unadjusted") saving(`"`tf1'"', replace);
parmby "regress mpg foreign weight, vce(robust)",
  idstr("Unequal&Adjusted") saving(`"`tf2'"', replace);
parmby "regress mpg foreign",
  idstr("Equal&Unadjusted") saving(`"`tf3'"', replace);
parmby "regress mpg foreign weight",
  idstr("Equal&Adjusted") saving(`"`tf4'"', replace);
clear all;
append using `"`tf1'"' `"`tf2'"' `"`tf3'"' `"`tf4'"';
```

Variables in the concatenated resultsset

When we `describe` the concatenated resultsset, we see that it contains parameter estimates, confidence limits and *P*–values:

```
. describe;

Contains data
  obs:            10
 vars:            10
 size:           750
-------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
-------------------------------------------------------------------------------
parmseq         byte   %12.0g                 Parameter sequence number
idstr           str18  %18s                   String ID
parm            str7   %9s                    Parameter name
estimate        double %10.0g                 Parameter estimate
stderr          double %10.0g                 SE of parameter estimate
dof             byte   %10.0g                 Degrees of freedom
t               double %10.0g                 t-test statistic
p               double %10.0g                 P-value
min95           double %10.0g                 Lower 95% confidence limit
max95           double %10.0g                 Upper 95% confidence limit
-------------------------------------------------------------------------------
Sorted by:
     Note:  dataset has changed since last saved
```

However, this resultsset does not seem to be sorted by anything!

Confidence intervals in the concatenated resultsset

When we `list` the concatenated resultsset, there seems to be more hope of some order being established:

```
. list idstr parmseq parm estimate min* max*, sepby(idstr);
```

	idstr	parmseq	parm	estimate	min95	max95
1.	Unequal&Unadjusted	1	foreign	4.9458042	1.8670625	8.0245459
2.	Unequal&Unadjusted	2	_cons	19.826923	18.510426	21.14342
3.	Unequal&Adjusted	1	foreign	-1.6500291	-3.9083006	.6082424
4.	Unequal&Adjusted	2	weight	-.00658789	-.00767698	-.00549879
5.	Unequal&Adjusted	3	_cons	41.679702	38.095484	45.26392
6.	Equal&Unadjusted	1	foreign	4.9458042	2.2303837	7.6612247
7.	Equal&Unadjusted	2	_cons	19.826923	18.346341	21.307505
8.	Equal&Adjusted	1	foreign	-1.6500291	-3.7955004	.49544223
9.	Equal&Adjusted	2	weight	-.00658789	-.00785825	-.00531752
10.	Equal&Adjusted	3	_cons	41.679702	37.361724	45.997681

The string ID variable `idstr` identifies the 4 estimations, and the numeric variable `parmseq` gives the parameter sequence order within each estimation.

Bringing order to the concatenated resultsset

This is done using the `split` command to split the string ID variable (at the ampersand) into 2 new string variables (`S_1` and `S_2`), and then using `sencode` to encode them to 2 numeric variables (`vartype` and `adjtype`), which are given variable labels and used to key the resultsset, after the old string variables have been `drop`ped:

```
. split idstr, parse(&) generate(S_);
variables created as string:
S_1  S_2

. sencode S_1, gene(vartype);

. sencode S_2, gene(adjtype);

. lab var vartype "Variance type";

. lab var adjtype "Adjustment type";

. drop idstr S_*;

. keyby vartype adjtype parmseq;
```

Note that `sencode` encodes string values in order of appearance, if no `gsort()` option is specified.

Variables in the improved concatenated resultsset

These are now as follows:

```
. describe;

Contains data
  obs:            10
 vars:            11
 size:           590
-------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
-------------------------------------------------------------------------------
vartype         byte   %8.0g       vartype    Variance type
adjtype         byte   %10.0g      adjtype    Adjustment type
parmseq         byte   %12.0g                 Parameter sequence number
parm            str7   %9s                    Parameter name
estimate        double %10.0g                 Parameter estimate
stderr          double %10.0g                 SE of parameter estimate
dof             byte   %10.0g                 Degrees of freedom
t               double %10.0g                 t-test statistic
p               double %10.0g                 P-value
min95           double %10.0g                 Lower 95% confidence limit
max95           double %10.0g                 Upper 95% confidence limit
-------------------------------------------------------------------------------
Sorted by:  vartype  adjtype  parmseq
     Note:  dataset has changed since last saved
```

This resultsset is sorted (and keyed) by 3 numeric variables.

Confidence intervals in the improved concatenated resultsset

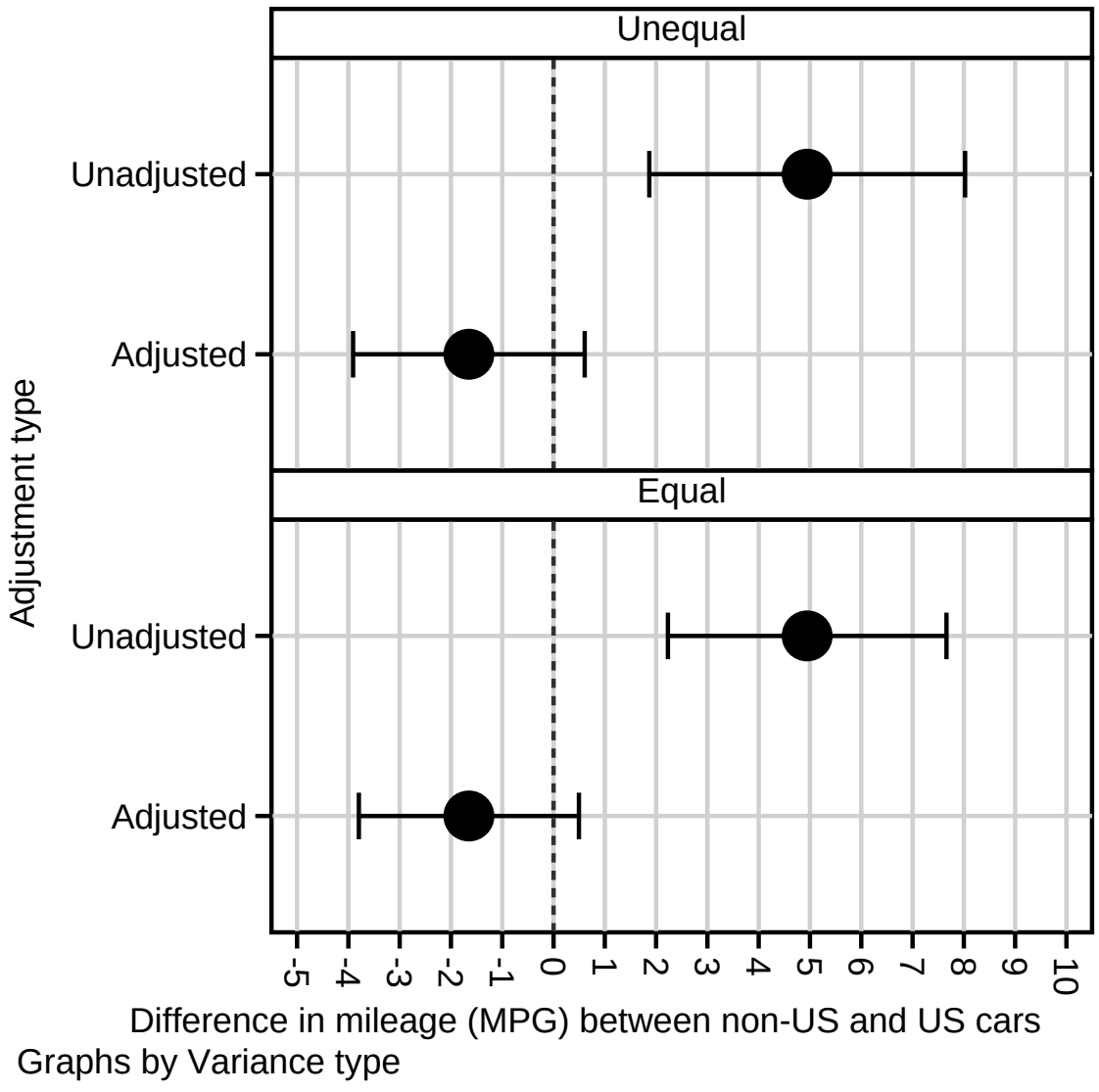
The dataset also looks better when listed (key variables first). . .

```
. list vartype adjtype parmseq parm estimate min* max*, sepby(vartype adjtype);
```

	vartype	adjtype	parmseq	parm	estimate	min95	max95
1.	Unequal	Unadjusted	1	foreign	4.9458042	1.8670625	8.0245459
2.	Unequal	Unadjusted	2	_cons	19.826923	18.510426	21.14342
3.	Unequal	Adjusted	1	foreign	-1.6500291	-3.9083006	.6082424
4.	Unequal	Adjusted	2	weight	-.00658789	-.00767698	-.00549879
5.	Unequal	Adjusted	3	_cons	41.679702	38.095484	45.26392
6.	Equal	Unadjusted	1	foreign	4.9458042	2.2303837	7.6612247
7.	Equal	Unadjusted	2	_cons	19.826923	18.346341	21.307505
8.	Equal	Adjusted	1	foreign	-1.6500291	-3.7955004	.49544223
9.	Equal	Adjusted	2	weight	-.00658789	-.00785825	-.00531752
10.	Equal	Adjusted	3	_cons	41.679702	37.361724	45.997681

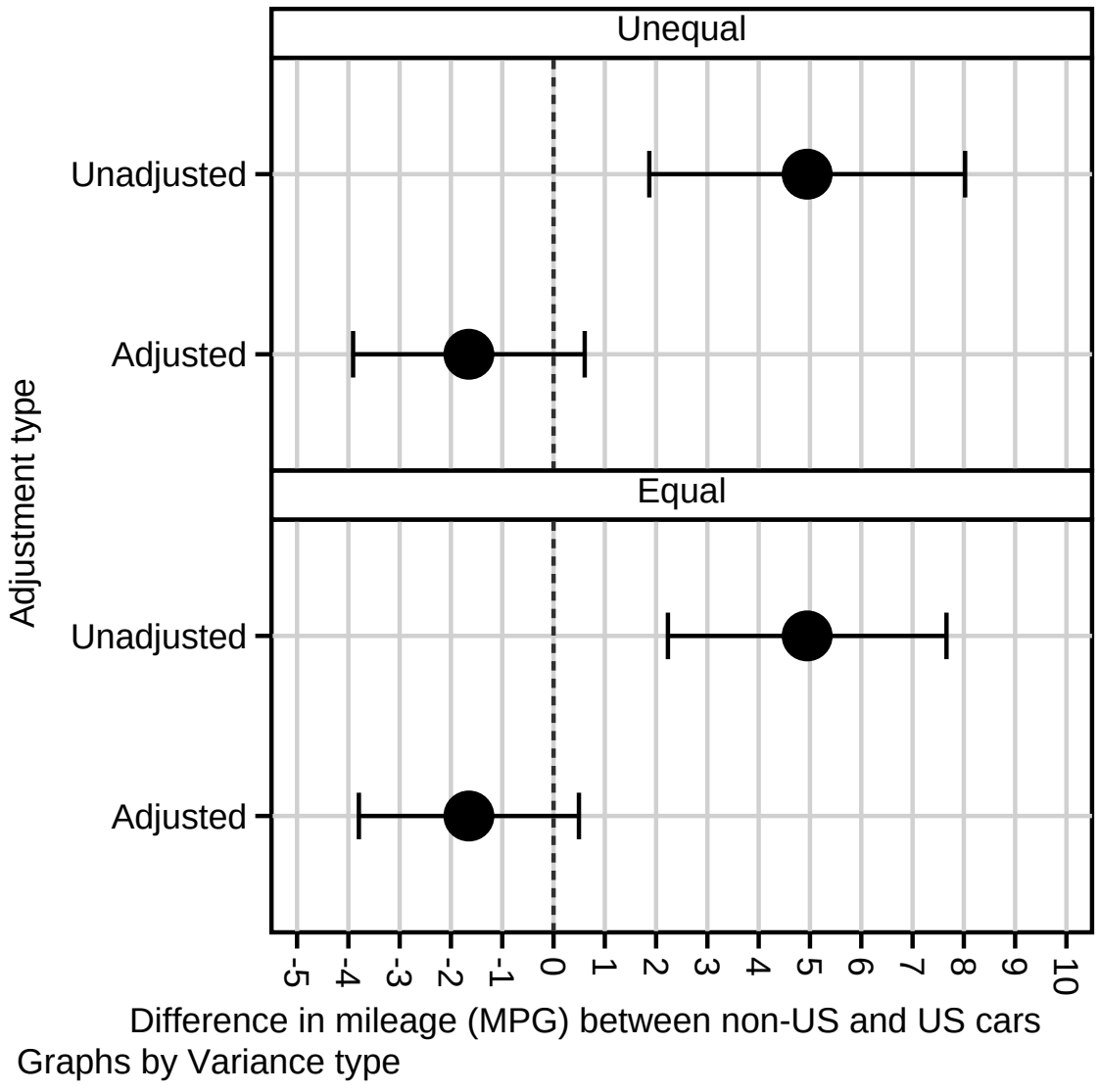
Mileage differences between non–US and US cars (plotted using the SSC package `eclplot`)

- . . .and better still when we plot only the `foreign` effects.
- We see that non–US cars do more miles per gallon than US cars.
- *However*, this difference vanishes after adjusting for weight.
- Of course, `split` and `sencode` can extract more than 2 factors from the same string ID.



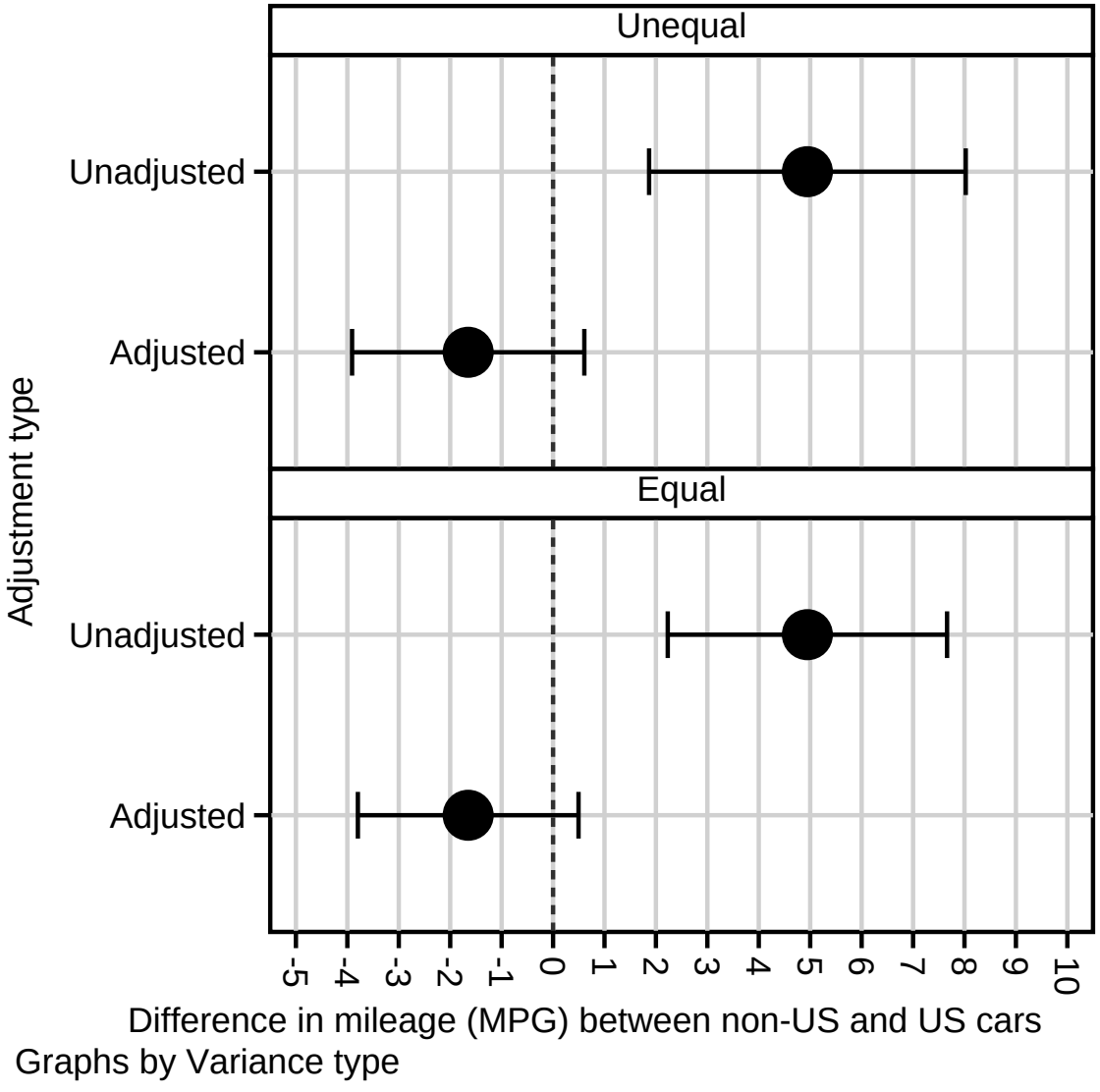
Mileage differences between non–US and US cars (plotted using the SSC package `eclplot`)

- . . .and better still when we plot only the `foreign` effects.
- We see that non–US cars do more miles per gallon than US cars.
- *However*, this difference vanishes after adjusting for weight.
- Of course, `split` and `sencode` can extract more than 2 factors from the same string ID.



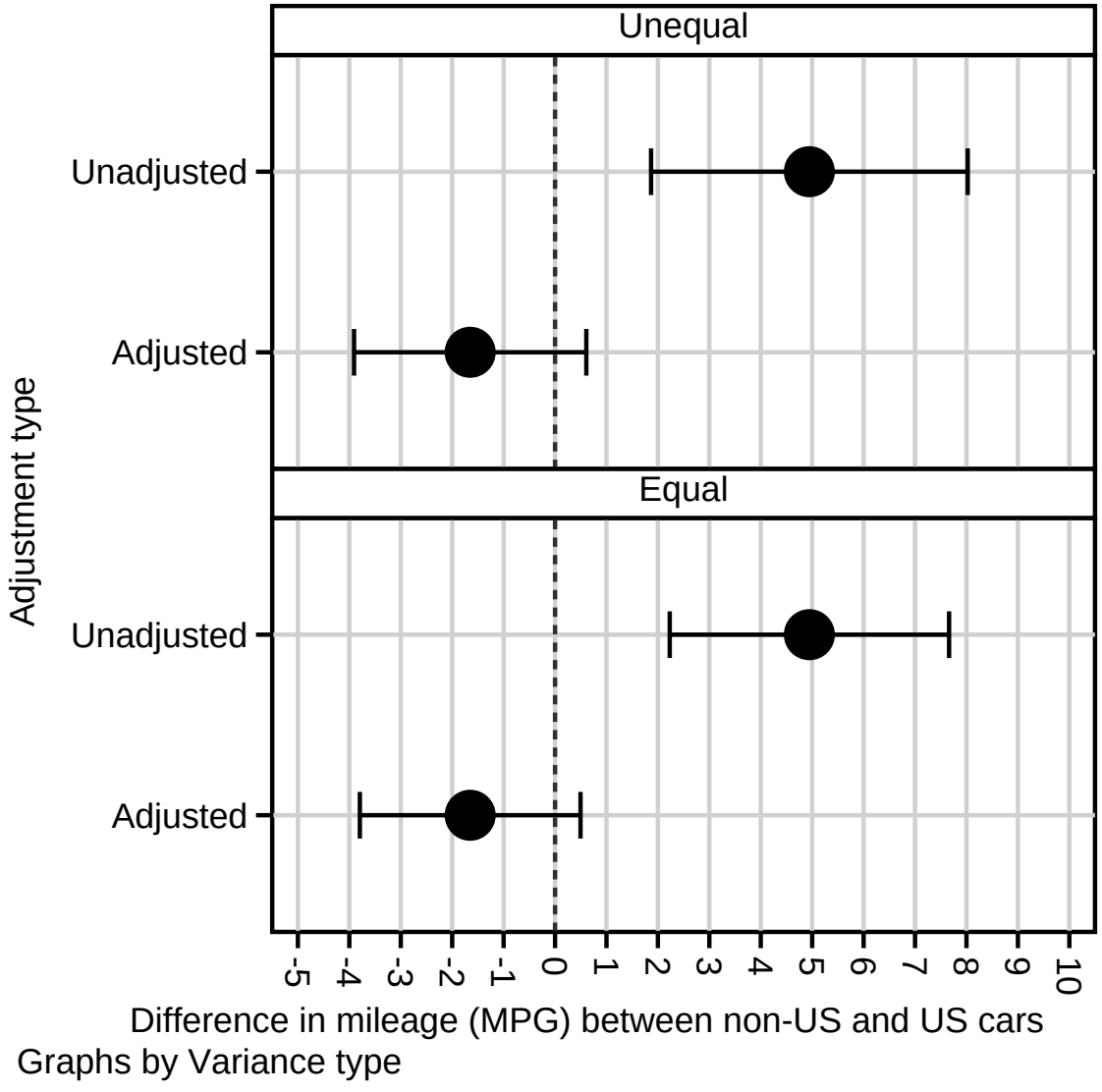
Mileage differences between non–US and US cars (plotted using the SSC package `eclplot`)

- . . .and better still when we plot only the `foreign` effects.
- We see that non–US cars do more miles per gallon than US cars.
- *However*, this difference vanishes after adjusting for weight.
- Of course, `split` and `sencode` can extract more than 2 factors from the same string ID.



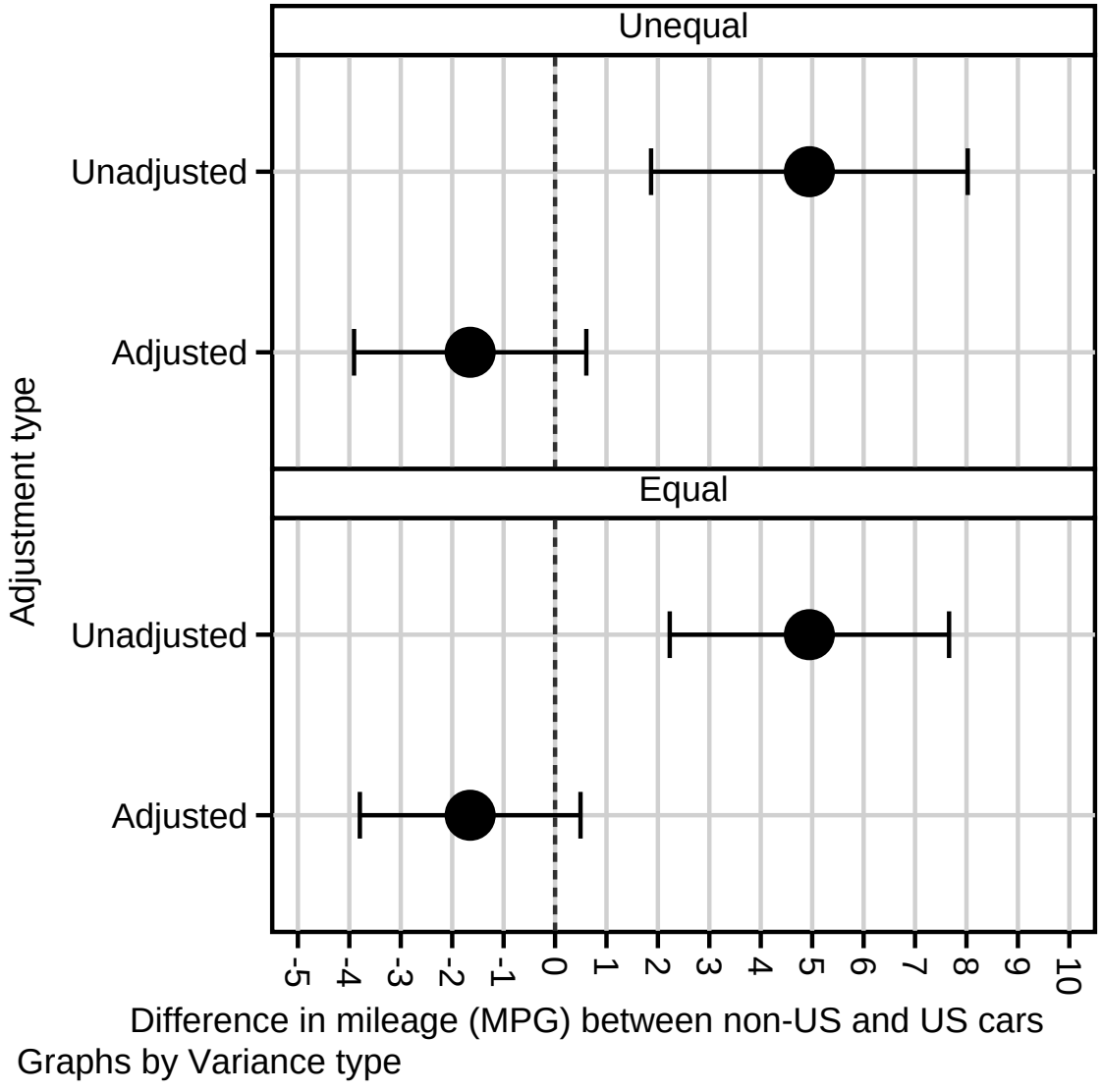
Mileage differences between non–US and US cars (plotted using the SSC package `eclplot`)

- . . .and better still when we plot only the `foreign` effects.
- We see that non–US cars do more miles per gallon than US cars.
- *However*, this difference vanishes after adjusting for weight.
- Of course, `split` and `sencode` can extract more than 2 factors from the same string ID.



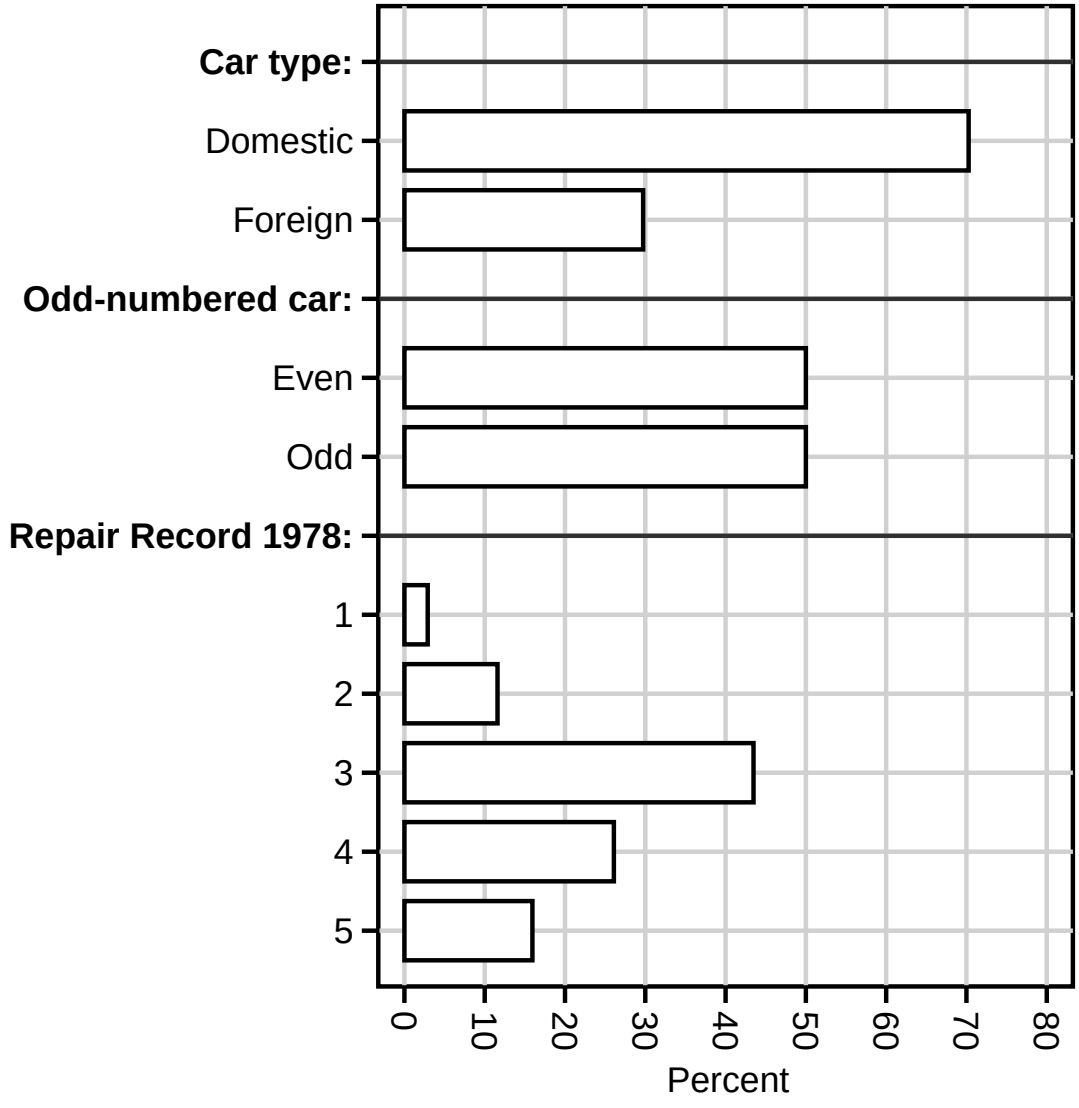
Mileage differences between non–US and US cars (plotted using the SSC package `eclplot`)

- . . .and better still when we plot only the `foreign` effects.
- We see that non–US cars do more miles per gallon than US cars.
- *However*, this difference vanishes after adjusting for weight.
- Of course, `split` and `sencode` can extract more than 2 factors from the same string ID.



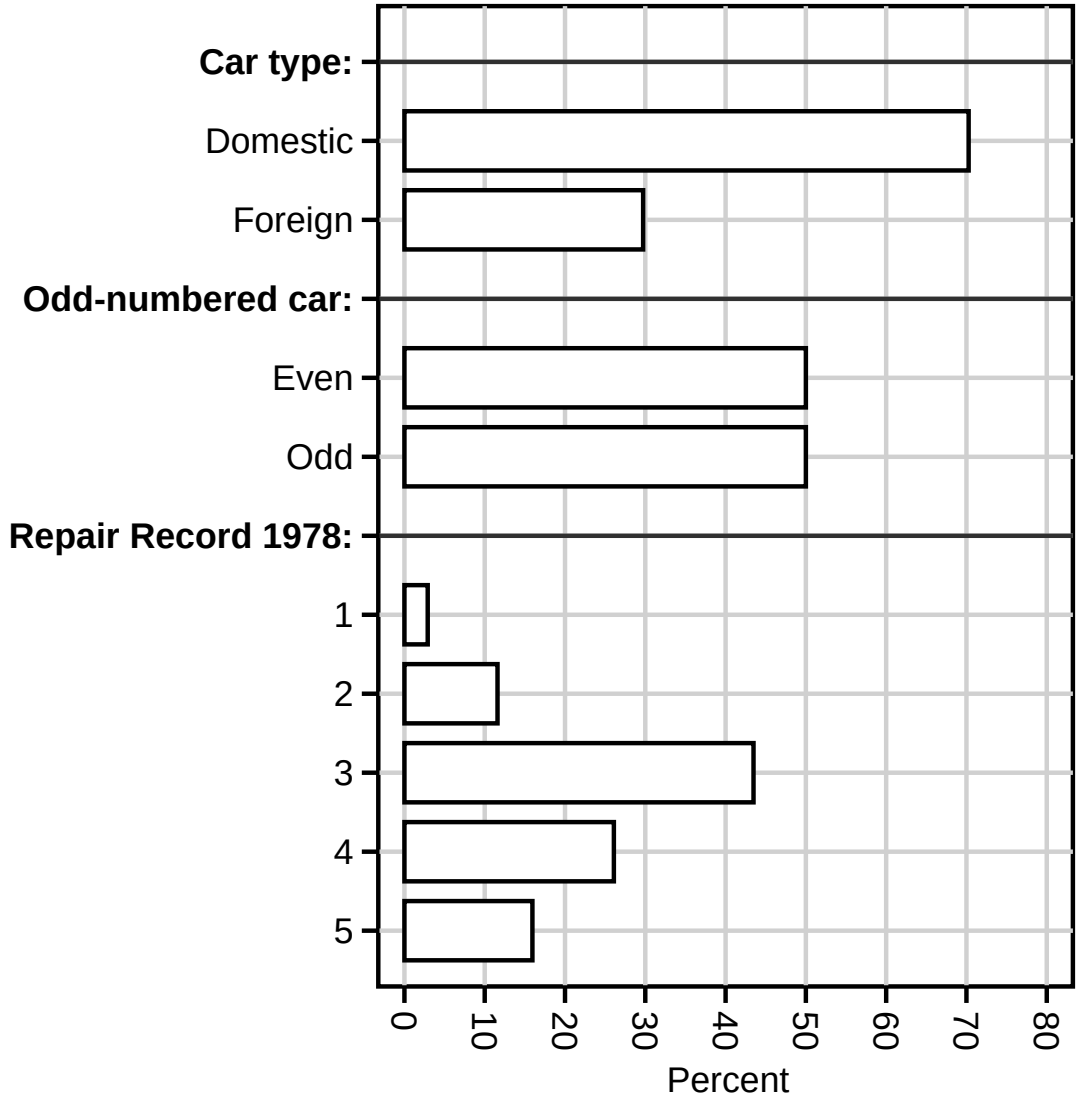
Advanced example: Frequency distributions of 3 factors in the `auto` dataset

- This histogram describes the distributions of 3 factors, with different levels and numbers of levels.
- These factors are `foreign`, `rep78`, and the added factor `odd=mod(_n,2)`.
- So how do we produce a multi–factor histogram like this?



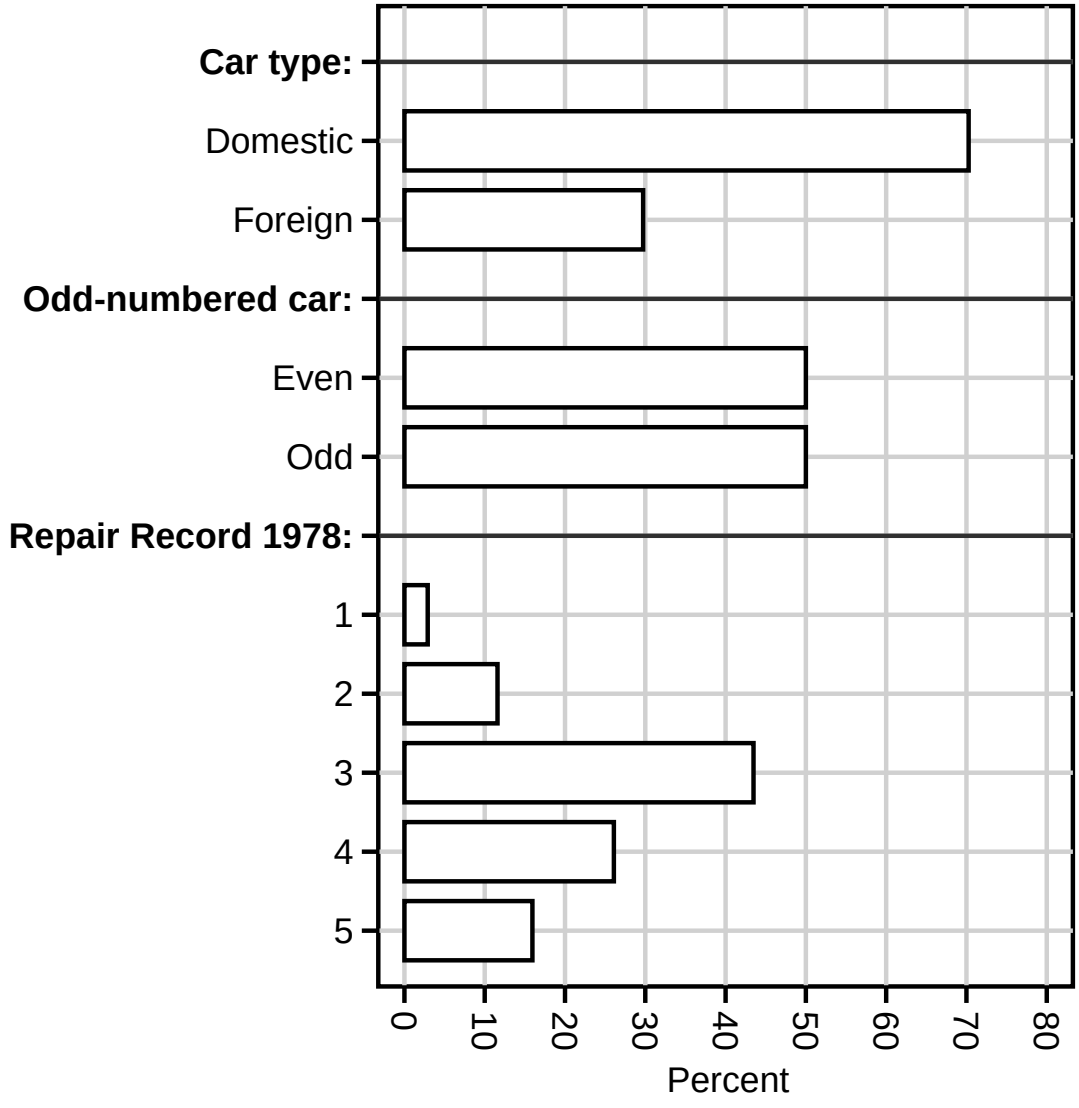
Advanced example: Frequency distributions of 3 factors in the `auto` dataset

- This histogram describes the distributions of 3 factors, with different levels and numbers of levels.
- These factors are `foreign`, `rep78`, and the added factor `odd=mod(_n,2)`.
- So how do we produce a multi–factor histogram like this?



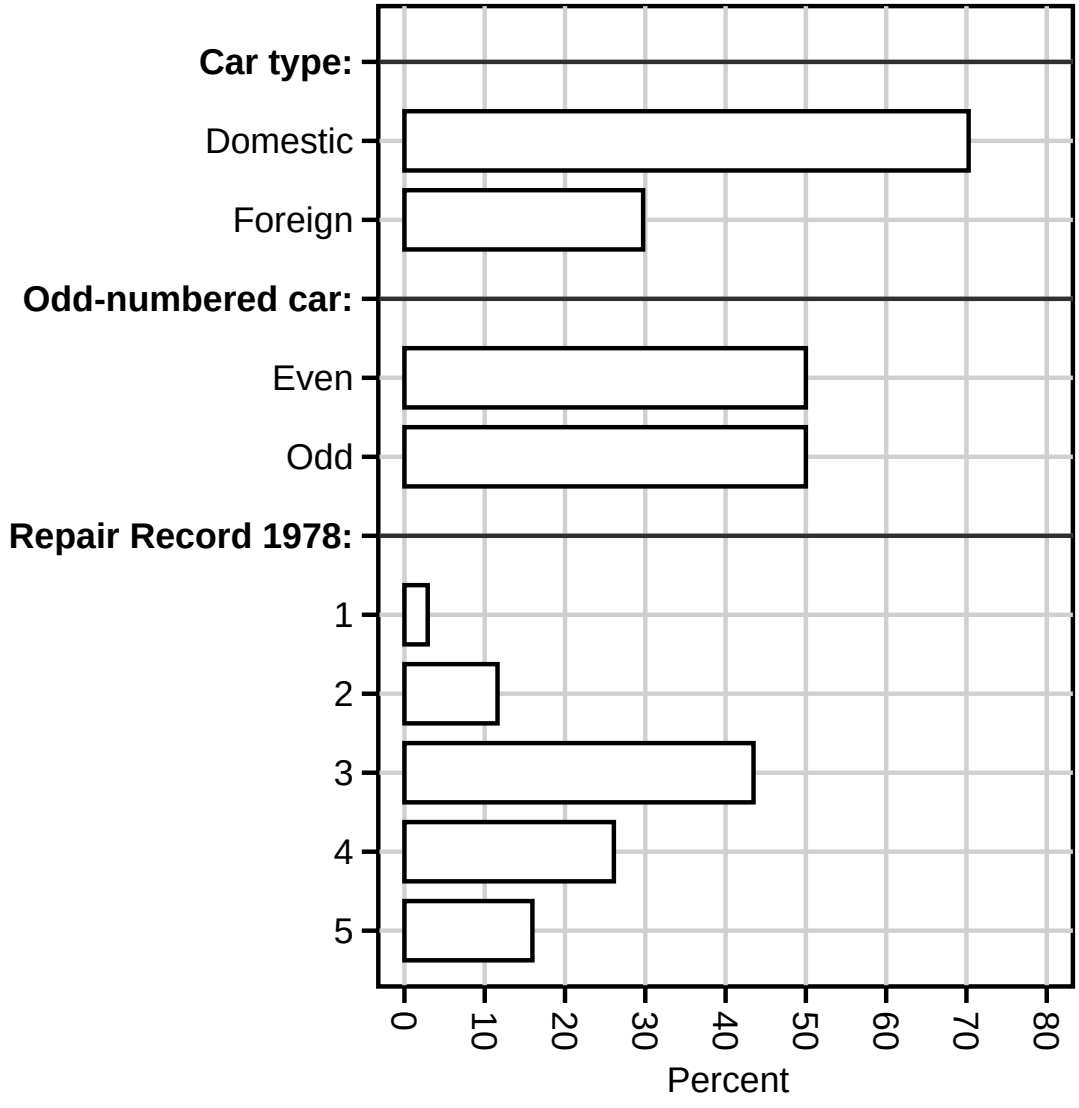
Advanced example: Frequency distributions of 3 factors in the `auto` dataset

- This histogram describes the distributions of 3 factors, with different levels and numbers of levels.
- These factors are `foreign`, `rep78`, and the added factor `odd=mod(_n,2)`.
- So how do we produce a multi–factor histogram like this?



Advanced example: Frequency distributions of 3 factors in the `auto` dataset

- This histogram describes the distributions of 3 factors, with different levels and numbers of levels.
- These factors are `foreign`, `rep78`, and the added factor `odd=mod(_n,2)`.
- So how do we produce a multi–factor histogram like this?



Creating the multi–factor histogram using `sdecode` and `sencode` (repeatedly)

- We start by using the SSC package `xcontract` (an extended version of `contract`) to make 3 frequency resultssets, one for each factor, which we `append` into the memory.
- We then replace these 3 old factors with 2 new key factors, indicating the old factors and the old–factor levels, respectively.
- We then replace the 2 new key factors with a single key factor, which we plot against the variable `_percent` to make the histogram.
- This kind of destructive mutilation of resultssets is usually done in a do–file, between a `preserve` statement and a `restore` statement. (*Not* interactively.)

Creating the multi–factor histogram using `sdecode` and `sencode` (repeatedly)

- We start by using the SSC package `xcontract` (an extended version of `contract`) to make 3 frequency resultssets, one for each factor, which we `append` into the memory.
- We then replace these 3 old factors with 2 new key factors, indicating the old factors and the old–factor levels, respectively.
- We then replace the 2 new key factors with a single key factor, which we plot against the variable `_percent` to make the histogram.
- This kind of destructive mutilation of resultssets is usually done in a do–file, between a `preserve` statement and a `restore` statement. (*Not* interactively.)

Creating the multi–factor histogram using `sdecode` and `sencode` (repeatedly)

- We start by using the SSC package `xcontract` (an extended version of `contract`) to make 3 frequency resultssets, one for each factor, which we `append` into the memory.
- We then replace these 3 old factors with 2 new key factors, indicating the old factors and the old–factor levels, respectively.
- We then replace the 2 new key factors with a single key factor, which we plot against the variable `_percent` to make the histogram.
- This kind of destructive mutilation of resultssets is usually done in a do–file, between a `preserve` statement and a `restore` statement. (*Not* interactively.)

Creating the multi–factor histogram using `sdecode` and `sencode` (repeatedly)

- We start by using the SSC package `xcontract` (an extended version of `contract`) to make 3 frequency resultssets, one for each factor, which we `append` into the memory.
- We then replace these 3 old factors with 2 new key factors, indicating the old factors and the old–factor levels, respectively.
- We then replace the 2 new key factors with a single key factor, which we plot against the variable `_percent` to make the histogram.
- This kind of destructive mutilation of resultssets is usually done in a do–file, between a `preserve` statement and a `restore` statement. (*Not* interactively.)

Creating the multi–factor histogram using `sdecode` and `sencode` (repeatedly)

- We start by using the SSC package `xcontract` (an extended version of `contract`) to make 3 frequency resultssets, one for each factor, which we `append` into the memory.
- We then replace these 3 old factors with 2 new key factors, indicating the old factors and the old–factor levels, respectively.
- We then replace the 2 new key factors with a single key factor, which we plot against the variable `_percent` to make the histogram.
- This kind of destructive mutilation of resultssets is usually done in a do–file, between a `preserve` statement and a `restore` statement. (*Not* interactively.)

The first concatenated resultsset (with 3 key factors)

This was made by `appending` 3 `xcontract` resultssets (one for each factor). It has 3 factors, 1 observation per level per factor, and a lot of missing factor values.

```
. list, sepa(0);

     +-----------------------------------------------+
     |  foreign    odd   rep78   _freq   _percent |
     |-----------------------------------------------|
  1. | Domestic      .       .      52      70.27 |
  2. |  Foreign      .       .      22      29.73 |
  3. |         .   Even       .      37      50.00 |
  4. |         .    Odd       .      37      50.00 |
  5. |         .      .       1       2       2.90 |
  6. |         .      .       2       8      11.59 |
  7. |         .      .       3      30      43.48 |
  8. |         .      .       4      18      26.09 |
  9. |         .      .       5      11      15.94 |
     +-----------------------------------------------+
```

The second concatenated resultsset (with 2 key factors)

This was made by replacing the 3 factors with 2 factors, `factor` and `faclev`, created by the decoding–encoding command sequence:

```
factmerge foreign odd rep78, flabel(factor) fvalue(faclev);
sencode factor, replace;
sencode faclev, replace manyto1 gsort(factor foreign odd rep78);
```

It still has 1 observation per level per factor. *However*, it is slimmer, with no missing factor values.

```
. list, sepby(factor);
```

	factor	faclev	_freq	_percent
1.	Car type	Domestic	52	70.27
2.	Car type	Foreign	22	29.73
3.	Odd-numbered car	Even	37	50.00
4.	Odd-numbered car	Odd	37	50.00
5.	Repair Record 1978	1	2	2.90
6.	Repair Record 1978	2	8	11.59
7.	Repair Record 1978	3	30	43.48
8.	Repair Record 1978	4	18	26.09
9.	Repair Record 1978	5	11	15.94

The third concatenated resultsset (with 1 key factor)

This was made by replacing the 2 factors with 1 factor `row`, created by the decoding–encoding command sequence:

```
sdecode faclev, gene(row);
insingap factor, rowlabel(row) grdecode(factor) inner(faclev)
  neworder(rowseq1) gapindicator(gapstat) prefix("{bf:") suffix(":}");
sencode row, replace manyto1;
```

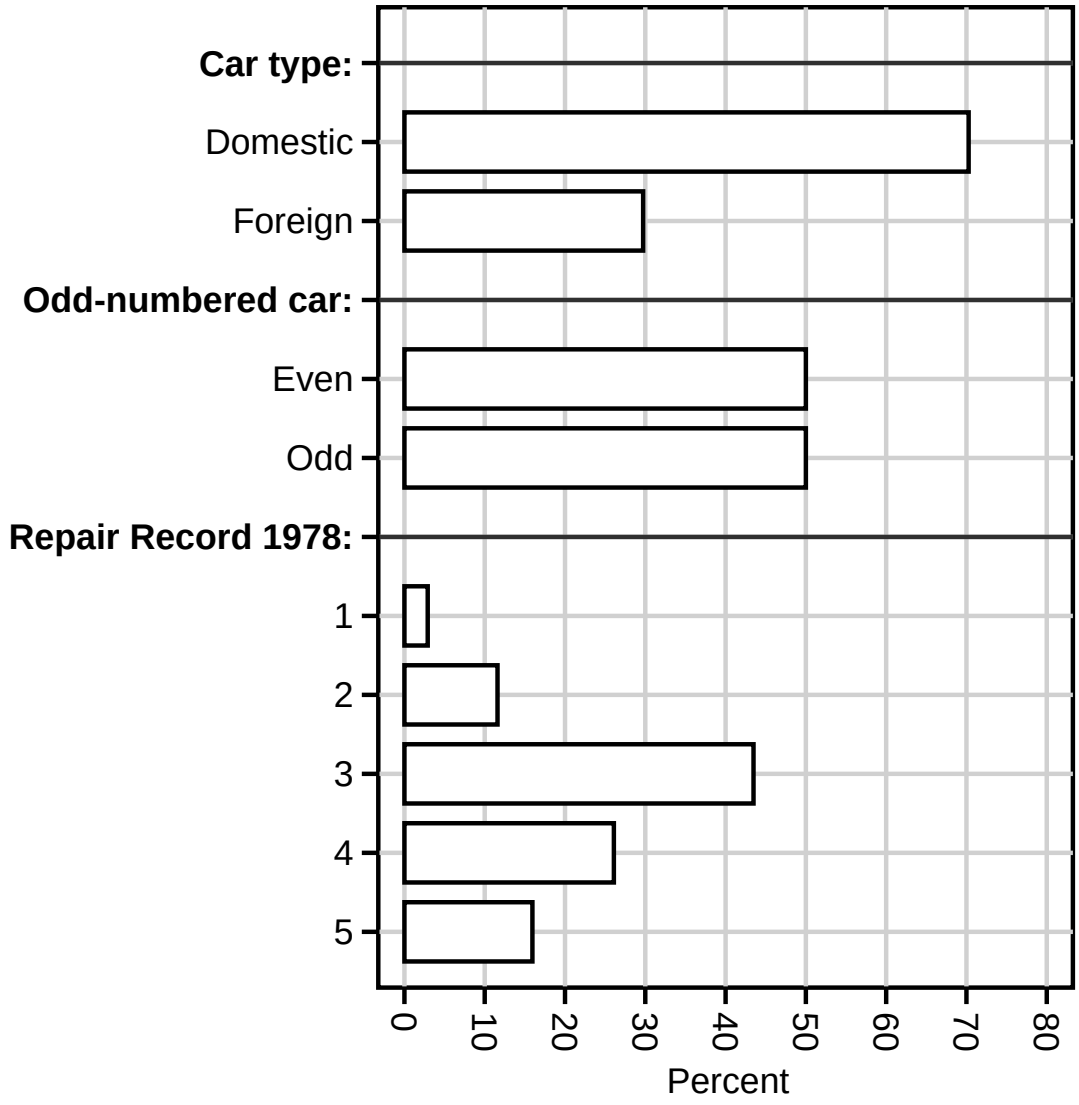
It is even slimmer, taller, and richer, with added SMCL **gap observations**, introducing each of the 3 original factors. *So*, it now has 1 observation per row of the planned multi–factor histogram. . .

```
. list row _freq _percent, sepby(factor);
```

	row	_freq	_percent
1.	{bf:Car type:}	.	.
2.	Domestic	52	70.27
3.	Foreign	22	29.73
4.	{bf:Odd-numbered car:}	.	.
5.	Even	37	50.00
6.	Odd	37	50.00
7.	{bf:Repair Record 1978:}	.	.
8.	1	2	2.90
9.	2	8	11.59
10.	3	30	43.48
11.	4	18	26.09
12.	5	11	15.94

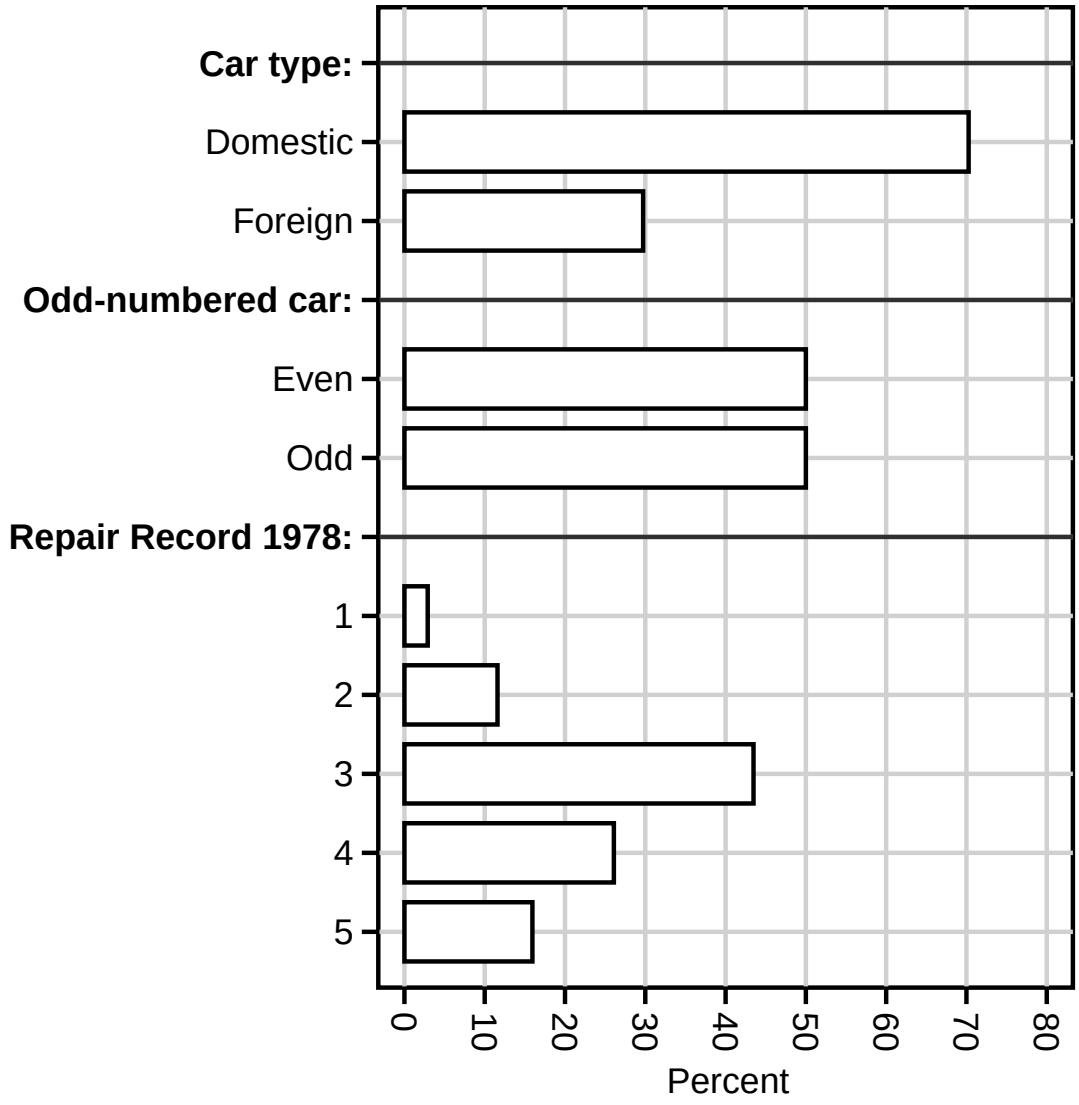
Frequency distributions of 3 factors in the `auto` dataset

- . . .which we then create using a `twoway bar` command, plotting `_percent` against `row`.
- Note that each factor has a bold–font heading, made using SMCL.
- Similar decoding and encoding sequences can be used to produce TEX, HTML or RTF tables[3].



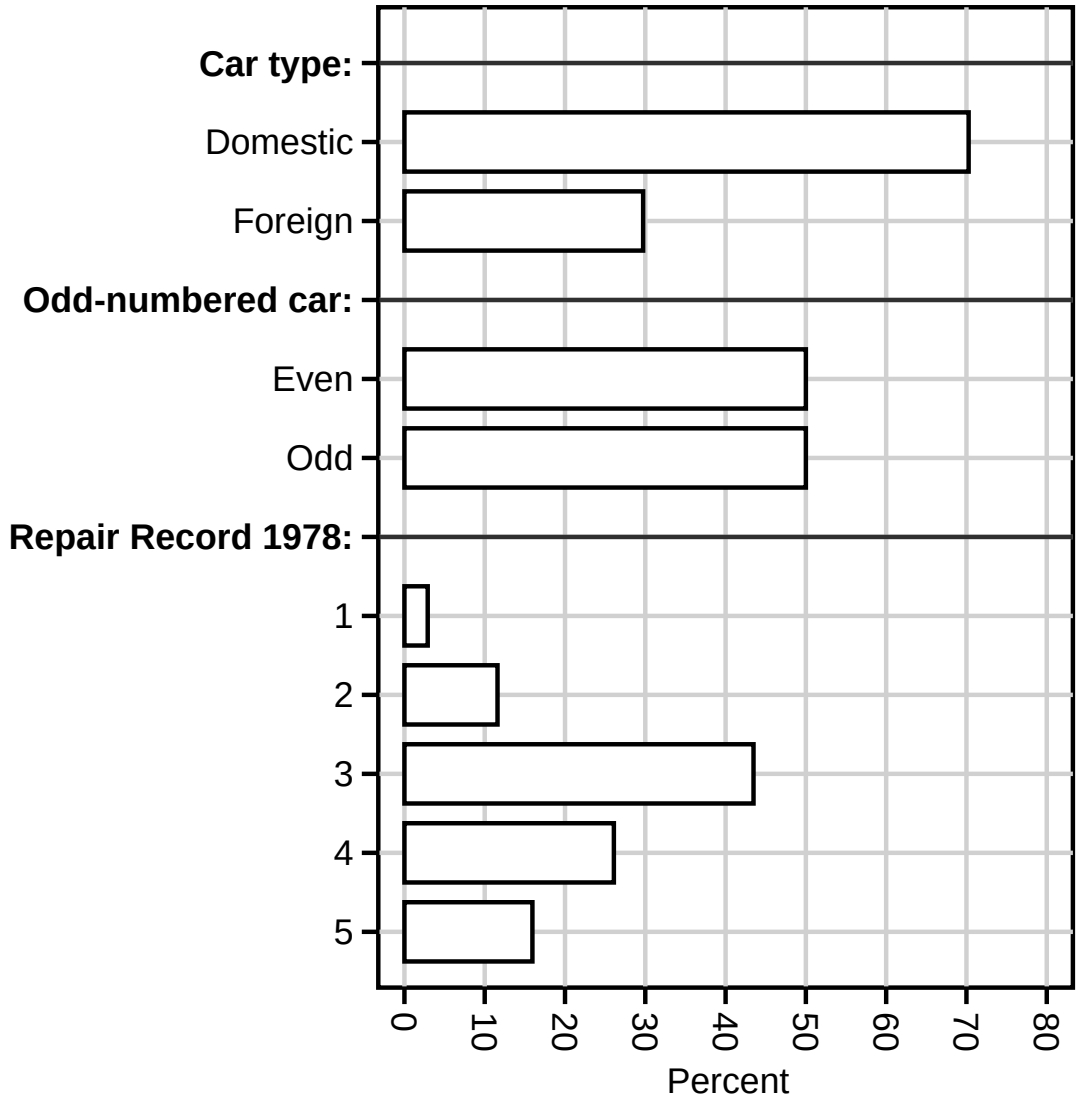
Frequency distributions of 3 factors in the `auto` dataset

- . . .which we then create using a `twoway bar` command, plotting `_percent` against `row`.
- Note that each factor has a bold–font heading, made using SMCL.
- Similar decoding and encoding sequences can be used to produce TEX, HTML or RTF tables[3].



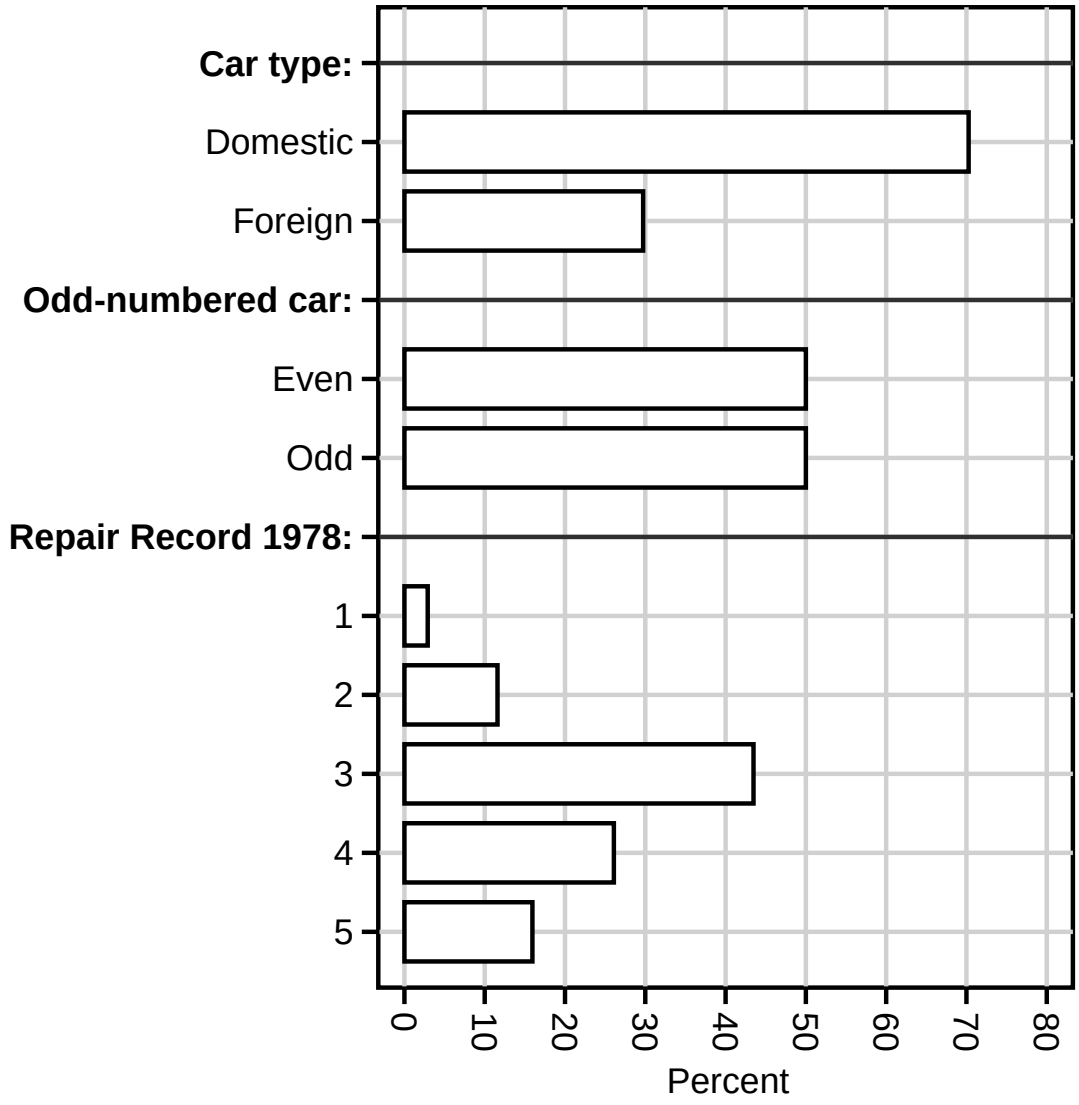
Frequency distributions of 3 factors in the `auto` dataset

- . . .which we then create using a `twoway bar` command, plotting `_percent` against `row`.
- Note that each factor has a bold–font heading, made using SMCL.
- Similar decoding and encoding sequences can be used to produce TEX, HTML or RTF tables[3].



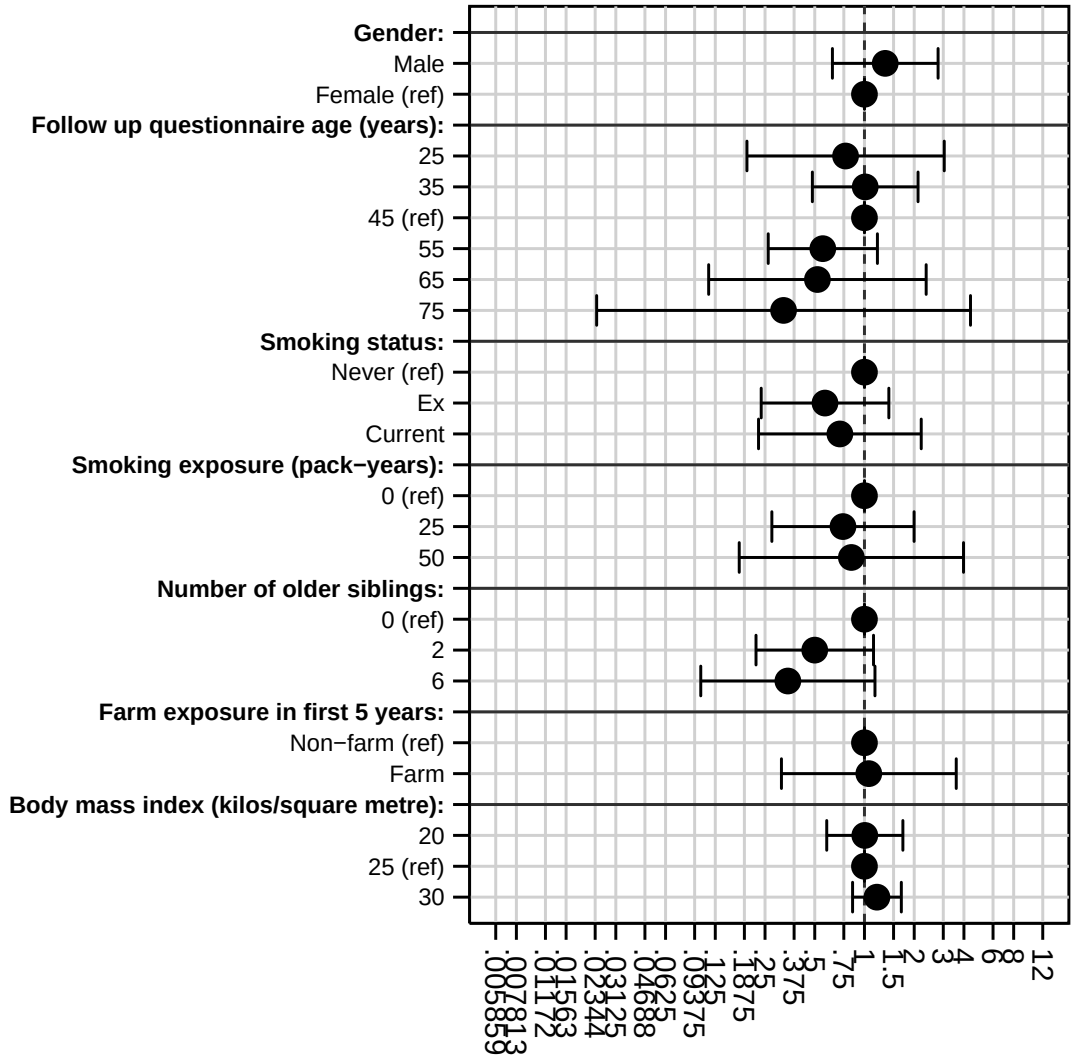
Frequency distributions of 3 factors in the `auto` dataset

- . . .which we then create using a `twoway bar` command, plotting `_percent` against `row`.
- Note that each factor has a bold–font heading, made using SMCL.
- Similar decoding and encoding sequences can be used to produce TEX, HTML or RTF tables[3].



Odds ratios for *Parietaria* pollen allergy in the GA^{2}LEN survey

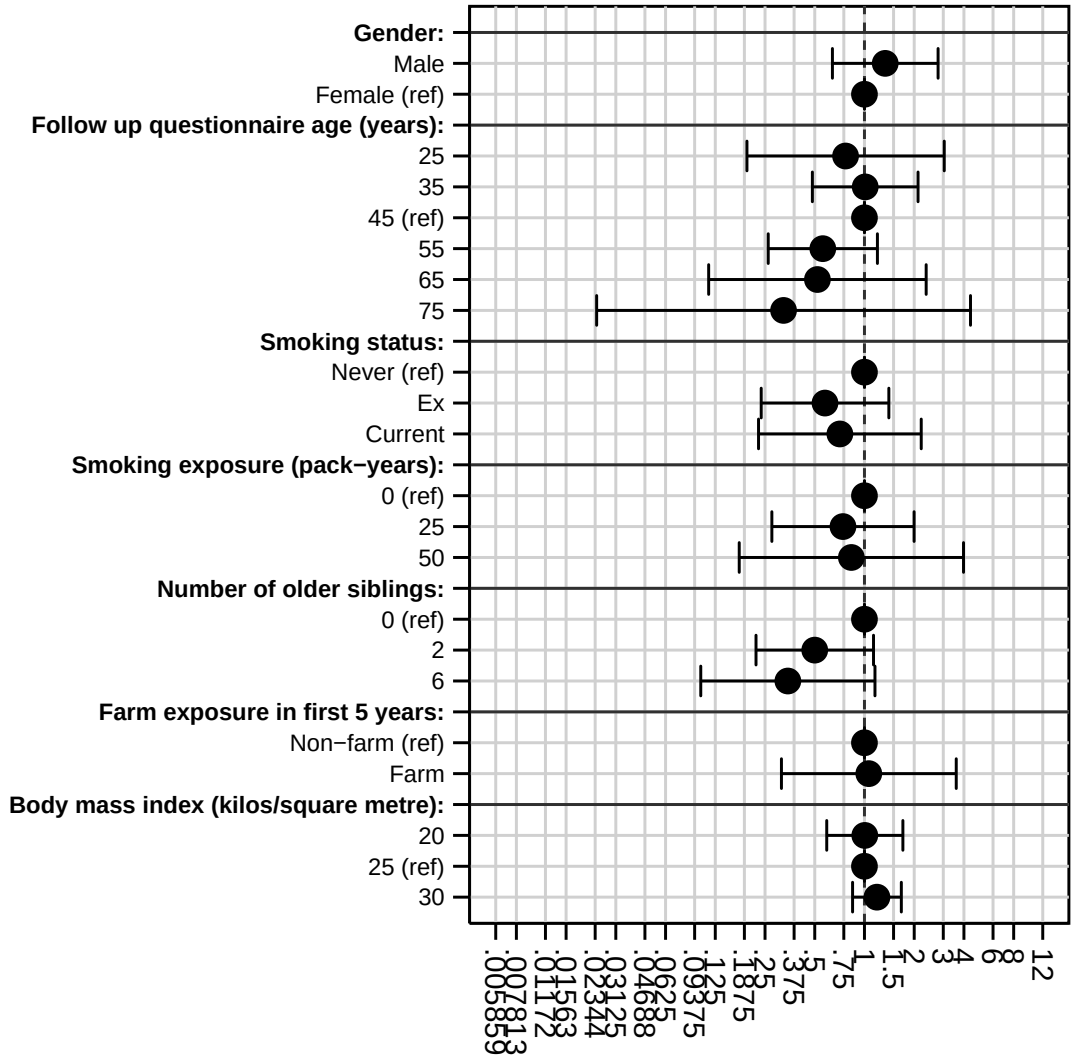
- And similar tricks also work with factors regenerated in `parmest` resultssets, using `fvregen`[2] and `factext`[1].
- In the GA^{2}LEN survey, we fitted multi–factor logistic models, predicting skin–prick allergies using discrete and continuous factors.
- Continuous factors were modelled using additive reference splines[4].



OR (95% CI) for sensitivity to: Parietaria (98 cases, 2920 subjects)

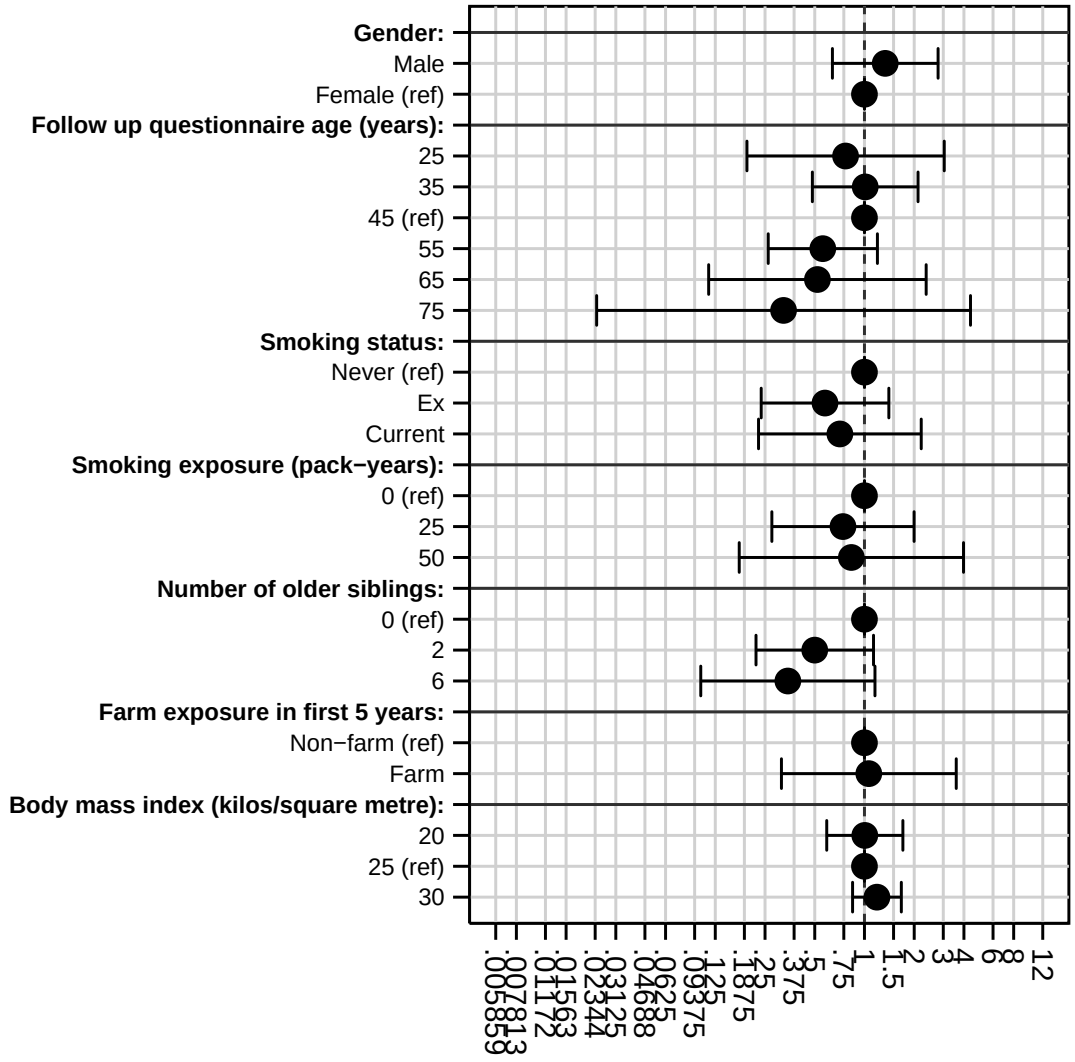
Odds ratios for *Parietaria* pollen allergy in the GA^{2}LEN survey

- And similar tricks also work with factors regenerated in `parmest` resultssets, using `fvregen`[2] and `factext`[1].
- In the GA^{2}LEN survey, we fitted multi–factor logistic models, predicting skin–prick allergies using discrete and continuous factors.
- Continuous factors were modelled using additive reference splines[4].



Odds ratios for *Parietaria* pollen allergy in the GA^{2}LEN survey

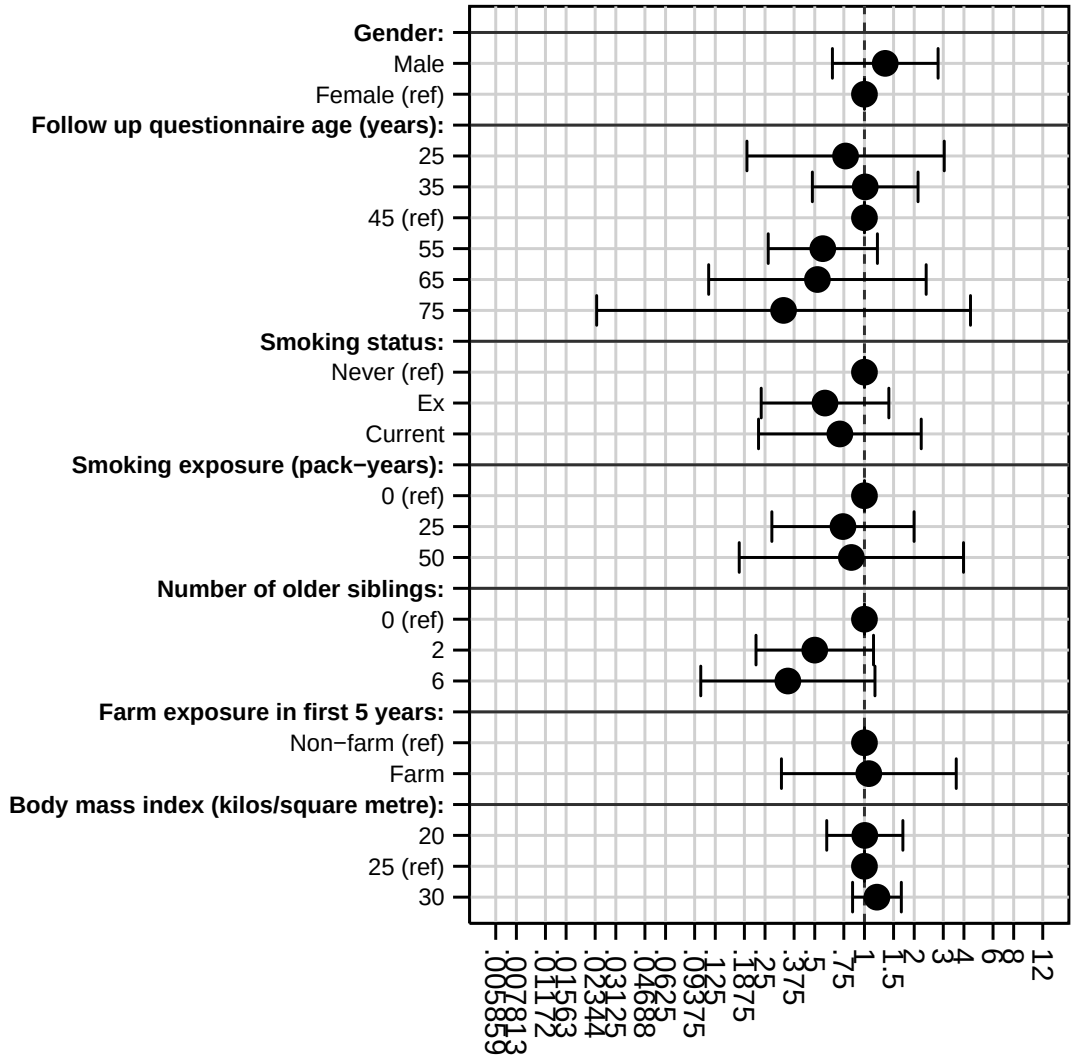
- And similar tricks also work with factors regenerated in `parmest` resultssets, using `fvregen`[2] and `factext`[1].
- In the GA^{2}LEN survey, we fitted multi–factor logistic models, predicting skin–prick allergies using discrete and continuous factors.
- Continuous factors were modelled using additive reference splines[4].



OR (95% CI) for sensitivity to: Parietaria (98 cases, 2920 subjects)

Odds ratios for *Parietaria* pollen allergy in the GA^{2}LEN survey

- And similar tricks also work with factors regenerated in `parmest` resultssets, using `fvregen`[2] and `factext`[1].
- In the GA^{2}LEN survey, we fitted multi–factor logistic models, predicting skin–prick allergies using discrete and continuous factors.
- Continuous factors were modelled using additive reference splines[4].



OR (95% CI) for sensitivity to: Parietaria (98 cases, 2920 subjects)

References

[1] Newson, R. 2004. From datasets to resultssets in Stata. Presented at the *10th UK Stata User Meeting*, 28–29 June, 2004. Downloadable from the conference website at *http://ideas.repec.org/s/boc/usug04.html*

[2] Newson, R. B. 2010. Post–`parmest` peripherals: `fvregen`, `invcise`, and `qqvalue`. . Presented at the *16th UK Stata User Meeting*, 9–10 September, 2010. Downloadable from the conference website at *http://ideas.repec.org/s/boc/usug10.html*

[3] Newson, R. B. 2012. From resultssets to resultstables in Stata. *The Stata Journal* **12(2)**: 479–504.

[4] Newson, R. B. 2012. Sensible parameters for univariate and multivariate splines. *The Stata Journal* **12(3)**: 479–504.

This presentation, and the do–file producing the examples in the `auto` data, can be downloaded from the conference website at *http://ideas.repec.org/s/boc/usug13.html*

The packages used in this presentation can be downloaded from SSC, using the `ssc` command.