Package ‘fixest’

November 24, 2023

Type Package
Title Fast Fixed-Effects Estimations
Version 0.11.2
Imports stats, graphics, grDevices, tools, utils, methods, numDeriv,
  nlm, sandwich, Rcpp(>= 1.0.5), dreamerr(>= 1.2.3)
Suggests knitr, rmarkdown, data.table, plm, MASS, pander, ggplot2,
  lfe, tinytex, pdfTools
LinkingTo Rcpp
Depends R(>= 3.5.0)
Description Fast and user-friendly estimation of econometric models with multiple fixed-effects. Includes ordinary least squares (OLS), generalized linear models (GLM) and the negative binomial. The core of the package is based on optimized parallel C++ code, scaling especially well for large data sets. The method to obtain the fixed-effects coefficients is based on Berge (2018) <https://github.com/lrberge/fixest/blob/master/_DOCS/FENmlm_paper.pdf>. Further provides tools to export and view the results of several estimations with intuitive design to cluster the standard-errors.
License GPL-3
BugReports https://github.com/lrberge/fixest/issues
URL https://lrberge.github.io/fixest/,
 https://github.com/lrberge/fixest
VignetteBuilder knitr
LazyData true
RoxygenNote 7.2.3
Encoding UTF-8
NeedsCompilation yes
Author Laurent Berge [aut, cre],
  Sebastian Krantz [ctb],
  Grant McDermott [ctb] (<https://orcid.org/0000-0001-7883-8573>)
Maintainer Laurent Berge <laurent.berge@u-bordeaux.fr>
## R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate.fixest</td>
<td>4</td>
</tr>
<tr>
<td>AIC.fixest</td>
<td>6</td>
</tr>
<tr>
<td>as.dict</td>
<td>7</td>
</tr>
<tr>
<td>as.list.fixest_multi</td>
<td>8</td>
</tr>
<tr>
<td>base_did</td>
<td>9</td>
</tr>
<tr>
<td>base_stagg</td>
<td>10</td>
</tr>
<tr>
<td>BIC.fixest</td>
<td>10</td>
</tr>
<tr>
<td>bin</td>
<td>11</td>
</tr>
<tr>
<td>bread.fixest</td>
<td>15</td>
</tr>
<tr>
<td>check_conv_feols</td>
<td>16</td>
</tr>
<tr>
<td>coef.fixest</td>
<td>17</td>
</tr>
<tr>
<td>coef.fixest_multi</td>
<td>19</td>
</tr>
<tr>
<td>coefplot</td>
<td>21</td>
</tr>
<tr>
<td>coeftable</td>
<td>32</td>
</tr>
<tr>
<td>coeftable.default</td>
<td>33</td>
</tr>
<tr>
<td>coeftable.fixest</td>
<td>35</td>
</tr>
<tr>
<td>coeftable.fixest_multi</td>
<td>39</td>
</tr>
<tr>
<td>collinearity</td>
<td>42</td>
</tr>
<tr>
<td>confint.fixest</td>
<td>43</td>
</tr>
<tr>
<td>confint.fixest_multi</td>
<td>45</td>
</tr>
<tr>
<td>degrees_freedom</td>
<td>47</td>
</tr>
<tr>
<td>demean</td>
<td>49</td>
</tr>
<tr>
<td>deviance.fixest</td>
<td>52</td>
</tr>
<tr>
<td>did_means</td>
<td>53</td>
</tr>
<tr>
<td>dsb</td>
<td>56</td>
</tr>
<tr>
<td>estfun.fixest</td>
<td>64</td>
</tr>
<tr>
<td>est_env</td>
<td>65</td>
</tr>
<tr>
<td>etable</td>
<td>67</td>
</tr>
<tr>
<td>extralines_register</td>
<td>89</td>
</tr>
<tr>
<td>f</td>
<td>91</td>
</tr>
<tr>
<td>fdim</td>
<td>92</td>
</tr>
<tr>
<td>feglm</td>
<td>93</td>
</tr>
<tr>
<td>femlm</td>
<td>105</td>
</tr>
<tr>
<td>feNmlm</td>
<td>116</td>
</tr>
<tr>
<td>feols</td>
<td>128</td>
</tr>
<tr>
<td>fitstat</td>
<td>142</td>
</tr>
<tr>
<td>fitstat_register</td>
<td>145</td>
</tr>
<tr>
<td>fitted.fixest</td>
<td>147</td>
</tr>
<tr>
<td>fixef.fixest</td>
<td>148</td>
</tr>
<tr>
<td>fixest_startup_msg</td>
<td>150</td>
</tr>
<tr>
<td>formula.fixest</td>
<td>151</td>
</tr>
<tr>
<td>hatvalues.fixest</td>
<td>152</td>
</tr>
<tr>
<td>R topics documented:</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--</td>
</tr>
<tr>
<td>i</td>
<td>153</td>
</tr>
<tr>
<td>lag.formula</td>
<td>156</td>
</tr>
<tr>
<td>logLik.fixest</td>
<td>159</td>
</tr>
<tr>
<td>model.matrix.fixest</td>
<td>160</td>
</tr>
<tr>
<td>models</td>
<td>162</td>
</tr>
<tr>
<td>nobs.fixest</td>
<td>163</td>
</tr>
<tr>
<td>n_unik</td>
<td>164</td>
</tr>
<tr>
<td>obs</td>
<td>167</td>
</tr>
<tr>
<td>osize</td>
<td>168</td>
</tr>
<tr>
<td>panel</td>
<td>168</td>
</tr>
<tr>
<td>plot.fixest.fixef</td>
<td>170</td>
</tr>
<tr>
<td>predict.fixest</td>
<td>172</td>
</tr>
<tr>
<td>print.fixest</td>
<td>175</td>
</tr>
<tr>
<td>print.fixest_fitstat</td>
<td>177</td>
</tr>
<tr>
<td>print.fixest_multi</td>
<td>178</td>
</tr>
<tr>
<td>r2</td>
<td>179</td>
</tr>
<tr>
<td>ref</td>
<td>180</td>
</tr>
<tr>
<td>rep.fixest</td>
<td>183</td>
</tr>
<tr>
<td>resid.fixest</td>
<td>185</td>
</tr>
<tr>
<td>resid.fixest_multi</td>
<td>186</td>
</tr>
<tr>
<td>sample_df</td>
<td>187</td>
</tr>
<tr>
<td>setFixest_coefplot</td>
<td>188</td>
</tr>
<tr>
<td>setFixest_dict</td>
<td>193</td>
</tr>
<tr>
<td>setFixest_estimation</td>
<td>194</td>
</tr>
<tr>
<td>setFixest_fml</td>
<td>197</td>
</tr>
<tr>
<td>setFixest_notes</td>
<td>202</td>
</tr>
<tr>
<td>setFixest_nthreads</td>
<td>202</td>
</tr>
<tr>
<td>setFixest_vcov</td>
<td>203</td>
</tr>
<tr>
<td>sigma.fixest</td>
<td>205</td>
</tr>
<tr>
<td>ssc</td>
<td>206</td>
</tr>
<tr>
<td>stepwise</td>
<td>209</td>
</tr>
<tr>
<td>style.df</td>
<td>210</td>
</tr>
<tr>
<td>style.tex</td>
<td>213</td>
</tr>
<tr>
<td>summary.fixest</td>
<td>217</td>
</tr>
<tr>
<td>summary.fixest.fixef</td>
<td>222</td>
</tr>
<tr>
<td>summary.fixest_multi</td>
<td>223</td>
</tr>
<tr>
<td>sunab</td>
<td>225</td>
</tr>
<tr>
<td>terms.fixest</td>
<td>229</td>
</tr>
<tr>
<td>to_integer</td>
<td>230</td>
</tr>
<tr>
<td>trade</td>
<td>231</td>
</tr>
<tr>
<td>unpanel</td>
<td>232</td>
</tr>
<tr>
<td>update.fixest</td>
<td>233</td>
</tr>
<tr>
<td>vcov.fixest</td>
<td>234</td>
</tr>
<tr>
<td>vcov_cluster</td>
<td>239</td>
</tr>
<tr>
<td>vcov_conley</td>
<td>241</td>
</tr>
<tr>
<td>vcov_hac</td>
<td>242</td>
</tr>
<tr>
<td>wald</td>
<td>245</td>
</tr>
<tr>
<td>weights.fixest</td>
<td>247</td>
</tr>
</tbody>
</table>
aggregate.fixest

Aggregates the values of DiD coefficients a la Sun and Abraham

Description

Simple tool that aggregates the value of CATT coefficients in staggered difference-in-difference setups (see details).

Usage

```r
## S3 method for class 'fixest'
aggregate(x, agg, full = FALSE, use_weights = TRUE, ...)
```

Arguments

- `x`: A fixest object.
- `agg`: A character scalar describing the variable names to be aggregated, it is pattern-based. For sunab estimations, the following keywords work: "att", "period", "cohort" and FALSE (to have full disaggregation). All variables that match the pattern will be aggregated. It must be of the form "(root)", the parentheses must be there and the resulting variable name will be "root". You can add another root with parentheses: "(root1)regex(root2)", in which case the resulting name is "root1::root2". To name the resulting variable differently you can pass a named vector: c("name" = "pattern") or c("name" = "pattern(root2)"). It’s a bit intricate sorry, please see the examples.
- `full`: Logical scalar, defaults to FALSE. If TRUE, then all coefficients are returned, not only the aggregated coefficients.
- `use_weights`: Logical, default is TRUE. If the estimation was weighted, whether the aggregation should take into account the weights. Basically if the weights reflected frequency it should be TRUE.
- `...`: Arguments to be passed to `summary.fixest`.

Details

This is a function helping to replicate the estimator from Sun and Abraham (2020). You first need to perform an estimation with cohort and relative periods dummies (typically using the function i), this leads to estimators of the cohort average treatment effect on the treated (CATT). Then you can use this function to retrieve the average treatment effect on each relative period, or for any other way you wish to aggregate the CATT.

Note that contrary to the SA article, here the cohort share in the sample is considered to be a perfect measure for the cohort share in the population.
Value

It returns a matrix representing a table of coefficients.

Author(s)

Laurent Berge

References


Examples

```r
# # DiD example #
#
data(base_stagg)

# 2 kind of estimations:
# - regular TWFE model
# - estimation with cohort x time_to_treatment interactions, later aggregated

# Note: the never treated have a time_to_treatment equal to -1000

# Now we perform the estimation
res_twfe = feols(y ~ x1 + i(time_to_treatment, treated, ref = c(-1, -1000)) | id + year, base_stagg)

# we use the "i." prefix to force year_treated to be considered as a factor
res_cohort = feols(y ~ x1 + i(time_to_treatment, i.year_treated, ref = c(-1, -1000)) | id + year, base_stagg)

# Displaying the results
iplot(res_twfe, ylim = c(-6, 8))
att_true = tapply(base_stagg$treatment_effect_true, base_stagg$time_to_treatment, mean)[-1]
points(-9:8 + 0.15, att_true, pch = 15, col = 2)

# The aggregate effect for each period
agg_coef = aggregate(res_cohort, "(ti.*nt)::(-?[[:digit:]]+)")
x = c(-9:-2, 0:8) + .35
points(x, agg_coef[, 1], pch = 17, col = 4)
segments(x0 = x, y0 = agg_coef[, 1] - 1.96 * agg_coef[, 2], x1 = x, y1 = agg_coef[, 1] + 1.96 * agg_coef[, 2], col = 4)

legend("topleft", col = c(1, 2, 4), pch = c(20, 15, 17), legend = c("TWFE", "True", "Sun & Abraham"))
```
# The ATT
aggregate(res_cohort, c("ATT" = "treatment::[^-]"))
with(base_stagg, mean(treatment_effect_true[time_to_treatment >= 0]))

# The total effect for each cohort
aggregate(res_cohort, c("cohort" = "::[^-].*year_treated::([[:digit:]]+)"))

---

AIC.fixest

Aikake’s an information criterion

Description

This function computes the AIC (Aikake’s, an information criterion) from a fixest estimation.

Usage

```r
## S3 method for class 'fixest'
AIC(object, ..., k = 2)
```

Arguments

- **object**
  A fixest object. Obtained using the functions `femlm`, `feols` or `feglm`.
- **...**
  Optionally, more fitted objects.
- **k**
  A numeric, the penalty per parameter to be used; the default k = 2 is the classical AIC (i.e. AIC=-2*LL+k*nparams).

Details

The AIC is computed as:

\[
AIC = -2 \times LogLikelihood + k \times nbParams
\]

with k the penalty parameter.

You can have more information on this criterion on AIC.

Value

It return a numeric vector, with length the same as the number of objects taken as arguments.

Author(s)

Laurent Berge
See Also

See also the main estimation functions `femlm`, `feols` or `feglm`. Other statistics methods: `BIC.fixest`, `logLik.fixest`, `nobs.fixest`.

Examples

```r
# two fitted models with different expl. variables:
res1 = femlm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width | Species, iris)
res2 = femlm(Sepal.Length ~ Petal.Width | Species, iris)

AIC(res1, res2)
BIC(res1, res2)
```

---

**as.dict**

Transforms a character string into a dictionary

**Description**

Transforms a single character string containing a dictionary in a textual format into a proper dictionary, that is a named character vector.

**Usage**

```r
as.dict(x)
```

**Arguments**

- `x` A character scalar of the form "variable 1: definition \n variable 2: definition" etc. Each line of this character must contain at most one definition with, on the left the variable name, and on the right its definition. The separation between the variable and its definition must be a colon followed with a single space (i.e. ": "). You can stack definitions within a single line by making use of a semi colon: "var1: def; var2: def". White spaces on the left and right are ignored. You can add commented lines with a "#". Non-empty, non-commented lines that don’t have the proper format will raise an error.

**Details**

This function is mostly used in combination with `setFixest_dict` to set the dictionary to be used in the function `etable`.

**Value**

It returns a named character vector.
as.list.fixest_multi  
Transforms a fixest_multi object into a list

Description
Extracts the results from a fixest_multi object and place them into a list.

Usage
## S3 method for class 'fixest_multi'
as.list(x, ...)

Arguments
- x: A fixest_multi object, obtained from a fixest estimation leading to multiple results.
- ...: Not currently used.

Value
Returns a list containing all the results of the multiple estimations.

See Also
The main fixest estimation functions: feols, fepois, fenegbin, feglm, feNmlm. Tools for multiple fixest estimations: summary.fixest_multi, print.fixest_multi, as.list.fixest_multi, sub-sub-.fixest_multi, sub-.fixest_multi.
**Examples**

```r
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# Multiple estimation
res = feols(y ~ csw(x1, x2, x3), base, split = ~species)

# All the results at once
as.list(res)
```

---

**base_did**

*Sample data for difference in difference*

**Description**

This data has been generated to illustrate the use of difference in difference functions in package `fixest`. This is a balanced panel of 104 individuals and 10 periods. About half the individuals are treated, the treatment having a positive effect on the dependent variable `y` after the 5th period. The effect of the treatment on `y` is gradual.

**Usage**

data(base_did)

**Format**

`base_did` is a data frame with 1,040 observations and 6 variables named `y`, `x1`, `id`, `period`, `post` and `treat`.

- `y` The dependent variable affected by the treatment.
- `x1` An explanatory variable.
- `id` Identifier of the individual.
- `period` From 1 to 10
- `post` Indicator taking value 1 if the period is strictly greater than 5, 0 otherwise.
- `treat` Indicator taking value 1 if the individual is treated, 0 otherwise.

**Source**

This data has been generated from R.
base_stagg  

Sample data for staggered difference in difference

Description

This data has been generated to illustrate the Sun and Abraham (Journal of Econometrics, forthcoming) method for staggered difference-in-difference. This is a balanced panel of 95 individuals and 10 periods. Half the individuals are treated. For those treated, the treatment date can vary from the second to the last period. The effect of the treatment depends on the time since the treatment: it is first negative and then increasing.

Usage

data(base_stagg)

Format

base_stagg is a data frame with 950 observations and 7 variables:

- id: panel identifier.
- year: from 1 to 10.
- year_treated: the period at which the individual is treated.
- time_to_treatment: different between the year and the treatment year.
- treated: indicator taking value 1 if the individual is treated, 0 otherwise.
- x1: explanatory variable, correlated with the period.
- y: the dependent variable affected by the treatment.

Source

This data has been generated from R.

BIC.fixest  

Bayesian information criterion

Description

This function computes the BIC (Bayesian information criterion) from a fixest estimation.

Usage

## S3 method for class 'fixest'
BIC(object, ...)

**Arguments**

- **object**: A `fixest` object. Obtained using the functions `femlm`, `feols` or `feglm`.
- ... Optionally, more fitted objects.

**Details**

The BIC is computed as follows:

\[
BIC = -2 \times \text{LogLikelihood} + \log(nobs) \times nbParams
\]

with \( k \) the penalty parameter.

You can have more information on this criterion on `AIC`.

**Value**

It return a numeric vector, with length the same as the number of objects taken as arguments.

**Author(s)**

Laurent Berge

**See Also**

See also the main estimation functions `femlm`, `feols` or `feglm`. Other statistics functions: `AIC.fixest`, `logLik.fixest`.

**Examples**

```r
# two fitted models with different expl. variables:
res1 = femlm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width | Species, iris)
res2 = femlm(Sepal.Length ~ Petal.Width | Species, iris)
AIC(res1, res2)
BIC(res1, res2)
```

---

**bin**

Bins the values of a variable (typically a factor)

**Description**

Tool to easily group the values of a given variable.

**Usage**

`bin(x, bin)`
Arguments

x A vector whose values have to be grouped. Can be of any type but must be atomic.

bin A list of values to be grouped, a vector, a formula, or the special values "bin::digit" or "cut::values". To create a new value from old values, use `bin = list("new_value"=old_values)` with old_values a vector of existing values. You can use .() for list(). It accepts regular expressions, but they must start with an "@", like in bin="@Aug|Dec". It accepts one-sided formulas which must contain the variable x, e.g. bin=list("<2" = ~x < 2). The names of the list are the new names. If the new name is missing, the first value matched becomes the new name. In the name, adding "@d", with d a digit, will relocate the value in position d: useful to change the position of factors. Use "@" as first item to make subsequent items be located first in the factor. Feeding in a vector is like using a list without name and only a single element. If the vector is numeric, you can use the special value "bin::digit" to group every digit element. For example if x represents years, using bin="bin::2" creates bins of two years. With any data, using "!bin::digit" groups every digit consecutive values starting from the first value. Using "!!bin::digit" is the same but starting from the last value. With numeric vectors you can: a) use "cut::n" to cut the vector into n equal parts, b) use "cut::a]b[" to create the following bins: [min, a], ]a, b[, [b, max]. The latter syntax is a sequence of number/quartile (q0 to q4)/percentile (p0 to p100) followed by an open or closed square bracket. You can add custom bin names by adding them in the character vector after 'cut::values'. See details and examples. Dot square bracket expansion (see dsb) is enabled.

Value

It returns a vector of the same length as x.

"Cutting" a numeric vector

Numeric vectors can be cut easily into: a) equal parts, b) user-specified bins.

Use "cut::n" to cut the vector into n (roughly) equal parts. Percentiles are used to partition the data, hence some data distributions can lead to create less than n parts (for example if P0 is the same as P50).

The user can specify custom bins with the following syntax: "cut::a]b[". Here the numbers a, b, c, etc, are a sequence of increasing numbers, each followed by an open or closed square bracket. The numbers can be specified as either plain numbers (e.g. "cut::5]12["), quartiles (e.g. "cut::q1]q3["), or percentiles (e.g. "cut::p10]p15]p90"). Values of different types can be mixed: "cut::5]q2[p80[" is valid provided the median (q2) is indeed greater than 5, otherwise an error is thrown.

The square bracket right of each number tells whether the numbers should be included or excluded from the current bin. For example, say x ranges from 0 to 100, then "cut::5]" will create two bins: one from 0 to 5 and a second from 6 to 100. With "cut::5]\" the bins would have been 0-4 and 5-100.

A factor is always returned. The labels always report the min and max values in each bin.
To have user-specified bin labels, just add them in the character vector following 'cut::values'. You don’t need to provide all of them, and NA values fall back to the default label. For example, `bin = c("cut::4", "Q1", NA, "Q3")` will modify only the first and third label that will be displayed as "Q1" and "Q3".

**bin vs ref**

The functions `bin` and `ref` are able to do the same thing, then why use one instead of the other? Here are the differences:

- `ref` always returns a factor. This is in contrast with `bin` which returns, when possible, a vector of the same type as the vector in input.
- `ref` always places the values modified in the first place of the factor levels. On the other hand, `bin` tries to not modify the ordering of the levels. It is possible to make `bin` mimic the behavior of `ref` by adding an "@" as the first element of the list in the argument `bin`.
- when a vector (and not a list) is given in input, `ref` will place each element of the vector in the first place of the factor levels. The behavior of `bin` is totally different, `bin` will transform all the values in the vector into a single value in x (i.e. it's binning).

**Author(s)**

Laurent Berge

**See Also**

To re-factor variables: `ref`.

**Examples**

data(airquality)
month_num = airquality$Month
table(month_num)

# Grouping the first two values
table(bin(month_num, 5:6))

# ... plus changing the name to '10'
table(bin(month_num, list("10" = 5:6)))

# ... and grouping 7 to 9
table(bin(month_num, list("g1" = 5:6, "g2" = 7:9)))

# Grouping every two months
table(bin(month_num, "bin::2"))

# ... every 2 consecutive elements
table(bin(month_num, "!bin::2"))

# ... idem starting from the last one
table(bin(month_num, "!!bin::2"))
# Using .() for list:
table(bin(month_num, .("g1" = 5:6)))

# with non numeric data
month_lab = c("may", "june", "july", "august", "september")
month_fact = factor(month_num, labels = month_lab)

# Grouping the first two elements
table(bin(month_fact, c("may", "jun")))

# ... using regex
table(bin(month_fact, @may|jun))

# ... changing the name
table(bin(month_fact, list("spring" = @may|jun)))

# Grouping every 2 consecutive months
table(bin(month_fact, !bin::2))

# ... idem but starting from the last
table(bin(month_fact, !!bin::2))

# Relocating the months using @d in the name
table(bin(month_fact, .("@5" = "may", @1 summer" = "aug|jul")))

# Putting @ as first item means subsequent items will be placed first
table(bin(month_fact, .("@", "aug", "july")))

# "Cutting" numeric data

data(iris)
plen = iris$Petal.Length

# 3 parts of (roughly) equal size
table(bin(plen, "cut::3"))

# Three custom bins
table(bin(plen, "cut::2[5]"))

# .. same, excluding 5 in the 2nd bin
table(bin(plen, "cut::2[5["))

# Using quartiles
table(bin(plen, "cut::q1|q2|q3"))

# Using percentiles
table(bin(plen, "cut::p20[p50][p70][p90]"))

# Mixing all
table(bin(plen, "cut::2[q2][p90]"))

# NOTA:
# -> the labels always contain the min/max values in each bin

# Custom labels can be provided, just give them in the char. vector
# NA values lead to the default label

# With a formula

data(iris)
plen = iris$Petal.Length

# We need to use "x"

# table(bin(plen, list("< 2" = ~x < 2, ">= 2" = ~x >= 2)))

bread.fixest  Extracts the bread matrix from fixest objects

Description
Extracts the bread matrix from fixest objects to be used to compute sandwich variance-covariance matrices.

Usage
## S3 method for class 'fixest'
bread(x, ...)

Arguments

x  A fixest object, obtained for instance from feols.

...  Not currently used.

Value
Returns a matrix of the same dimension as the number of variables used in the estimation.
Examples

```r
est = feols(Petal.Length ~ Petal.Width + Sepal.Width, iris)
bread(est)
```

---

**check_conv_feols**  
*Check the fixed-effects convergence of a feols estimation*

**Description**

Checks the convergence of a `feols` estimation by computing the first-order conditions of all fixed-effects (all should be close to 0)

**Usage**

```r
check_conv_feols(x)
```

```
## S3 method for class 'fixest_check_conv'
summary(object, type = "short", ...)
```

**Arguments**

- `x`  
  A `feols` estimation that should contain fixed-effects.
- `object`  
  An object returned by `check_conv_feols`.
- `type`  
  Either "short" (default) or "detail". If "short", only the maximum absolute FOC are displayed, otherwise the 2 smallest and the 2 largest FOC are reported for each fixed-effect and each variable.
- `...`  
  Not currently used.

Note that this function first re-demeans the variables, thus possibly incurring some extra computation time.

**Value**

It returns a list of `N` elements, `N` being the number of variables in the estimation (dependent variable + explanatory variables +, if IV, endogenous variables and instruments). For each variable, all the first-order conditions for each fixed-effect are returned.

**Examples**

```r
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
base$FE = rep(1:30, 5)

# one estimation with fixed-effects + varying slopes
est = feols(y ~ x1 | species[x2] + FE[x3], base)
```
# Checking the convergence
conv = check_conv_feols(est)

# We can check that all values are close to 0
summary(conv)
summary(conv, "detail")

calculation

### coef.fixest

*Extracts the coefficients from a fixest estimation*

**Description**

This function extracts the coefficients obtained from a model estimated with `femlm`, `feols` or `feglm`.

**Usage**

```r
## S3 method for class 'fixest'
coef(object, keep, drop, order, collin = FALSE, agg = TRUE, ...)

## S3 method for class 'fixest'
coefficients(object, keep, drop, order, collin = FALSE, agg = TRUE, ...)
```

**Arguments**

- **object**: A fixest object. Obtained using the functions `femlm`, `feols` or `feglm`.
- **keep**: Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument `dict`). Example: you have the variable `x1` to `x55` and want to display only `x1` to `x9`, then you could use `keep = "x[[:digit:]]\$"`. If the first character is an exclamation mark, the effect is reversed (e.g. `keep = "!Intercept"` means: every variable that does not contain “Intercept” is kept). See details.
- **drop**: Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument `dict`). Example: you have the variable `x1` to `x55` and want to display only `x1` to `x9`, then you could use `drop = "x[[:digit:]][2]"`. If the first character is an exclamation mark, the effect is reversed (e.g. `drop = "!Intercept"` means: every variable that does not contain “Intercept” is dropped). See details.
```r
order
Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see `base::regex` help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument `dict`). Example: you have the following variables: `month1` to `month6`, then `x1` to `x5`, then `year1` to `year6`. If you want to display first the `x`'s, then the years, then the months you could use: `order = c("x", "year")`. If the first character is an exclamation mark, the effect is reversed (e.g. `order = "!Intercept"` means: every variable that does not contain “Intercept” goes first). See details.

collin
Logical, default is `FALSE`. Whether the coefficients removed because of collinearity should be also returned as `NA`. It cannot be used when coefficients aggregation is also used.

agg
Logical scalar, default is `TRUE`. If the coefficients of the estimation have been aggregated, whether to report the aggregated coefficients. If `FALSE`, the raw coefficients will be returned.

... Not currently used.

Details

The coefficients are the ones that have been found to maximize the log-likelihood of the specified model. More information can be found on the models from the estimations help pages: `femlm`, `feols` or `feglm`.

Note that if the model has been estimated with fixed-effects, to obtain the fixed-effect coefficients, you need to use the function `fixef.fixest`.

Value

This function returns a named numeric vector.

Author(s)

Laurent Berge

See Also

See also the main estimation functions `femlm`, `feols` or `feglm`. `summary.fixest`, `confint.fixest`, `vcov.fixest`, `etable`, `fixef.fixest`.

Examples

```r
# simple estimation on iris data, using "Species" fixed-effects
res = femlm(Sepal.Length ~ Sepal.Width + Petal.Length +
            Petal.Width | Species, iris)

# the coefficients of the variables:
coef(res)
```
# the fixed-effects coefficients:
fixef(res)

## S3 method for class 'fixest_multi'
coef(
  object, 
  keep, 
  drop, 
  order, 
  collin = FALSE, 
  long = FALSE, 
  na.rm = TRUE, 
  ... 
)

## S3 method for class 'fixest_multi'
coefficients(
  object, 
  keep, 
  drop, 
  order, 
  collin = FALSE, 
  long = FALSE, 
  na.rm = TRUE, 
  ... 
)

### Arguments

- **object**
  A `fixest_multi` object. Obtained from a multiple estimation.

- **keep**
  Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument `dict`). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use `keep = "x[[:digit:]]$"`. If the first character is an exclamation mark, the effect is
reversed (e.g. keep = "!Intercept" means: every variable that does not contain "Intercept" is kept). See details.

drop Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see base::regex help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument dict). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use drop = "x[[[:digit:]]]{2}". If the first character is an exclamation mark, the effect is reversed (e.g. drop = "!Intercept" means: every variable that does not contain "Intercept" is dropped). See details.

order Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see base::regex help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument dict). Example: you have the following variables: month1 to month6, then x1 to x5, then year1 to year6. If you want to display first the x’s, then the years, then the months you could use: order = c("x", "year"). If the first character is an exclamation mark, the effect is reversed (e.g. order = "!Intercept" means: every variable that does not contain "Intercept" goes first). See details.

collin Logical, default is FALSE. Whether the coefficients removed because of collinearity should be also returned as NA. It cannot be used when coefficients aggregation is also used.

long Logical, default is FALSE. Whether the results should be displayed in a long format.

na.rm Logical, default is TRUE. Only applies when long = TRUE: whether to remove the coefficients with NA values.

... Not currently used.

Examples

```r
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# A multiple estimation
est = feols(y ~ x1 + csw0(x2, x3), base)

# Getting all the coefficients at once, # each row is a model
coeff(est)

# Example of keep/drop/order
coeff(est, keep = "Int|x1", order = "x1")

# To change the order of the model, use fixest_multi # extraction tools:
```
coefplot

Plots confidence intervals and point estimates

Description

This function plots the results of estimations (coefficients and confidence intervals). The function `iplot` restricts the output to variables created with `i`, either interactions with factors or raw factors.

Usage

c coefplot(
  object,
  ..., 
  style = NULL, 
  sd, 
  ci_low, 
  ci_high, 
  df.t = NULL, 
  x, 
  x.shift = 0, 
  horiz = FALSE, 
  dict = getFixest_dict(), 
  keep, 
  drop, 
  order, 
  ci.width = "1%", 
  ci_level = 0.95, 
  add = FALSE, 
  pt.pch = c(20, 17, 15, 21, 24, 22), 
)
pt.bg = NULL,
cex = 1,
pt.cex = cex,
col = 1:8,
pt.col = col,
ci.col = col,
lwd = 1,
pt.lwd = lwd,
ci.lwd = lwd,
ci.lty = 1,
grid = TRUE,
grid.par = list(lty = 3, col = "gray"),
zero = TRUE,
zero.par = list(col = "black", lwd = 1),
pt.join = FALSE,
pt.join.par = list(col = pt.col, lwd = lwd),
ci.join = FALSE,
ci.join.par = list(lwd = lwd, col = col, lty = 2),
ci.fill = FALSE,
ci.fill.par = list(col = "lightgray", alpha = 0.5),
ref = "auto",
ref.line = "auto",
ref.line.par = list(col = "black", lty = 2),
lab.cex,
lab.min.cex = 0.85,
lab.max.mar = 0.25,
lab.fit = "auto",
xlim.add,
ylim.add,
only.params = FALSE,
sep,
as.multiple = FALSE,
bg,
group = "auto",
group.par = list(lwd = 2, line = 3, tcl = 0.75),
main = "Effect on __depvar__",
value.lab = "Estimate and __ci__ Conf. Int.",
ylab = NULL,
xlab = NULL,
sub = NULL
)

iplot(
    object,
    ...
    i.select = 1,
    style = NULL,
    sd,
ci_low,
  ci_high,
  df.t = NULL,
  x,
  x.shift = 0,
  horiz = FALSE,
  dict = getFixest_dict(),
  keep,
  drop,
  order,
  ci.width = "1%",
  ci_level = 0.95,
  add = FALSE,
  pt.pch = c(20, 17, 15, 21, 24, 22),
  pt.bg = NULL,
  cex = 1,
  pt.cex = cex,
  col = 1:8,
  pt.col = col,
  ci.col = col,
  lwd = 1,
  pt.lwd = lwd,
  ci.lwd = lwd,
  ci.lty = 1,
  grid = TRUE,
  grid.par = list(lty = 3, col = "gray"),
  zero = TRUE,
  zero.par = list(col = "black", lwd = 1),
  pt.join = FALSE,
  pt.join.par = list(col = pt.col, lwd = lwd),
  ci.join = FALSE,
  ci.join.par = list(lwd = lwd, col = col, lty = 2),
  ci.fill = FALSE,
  ci.fill.par = list(col = "lightgray", alpha = 0.5),
  ref = "auto",
  ref.line = "auto",
  ref.line.par = list(col = "black", lty = 2),
  lab.cex,
  lab.min.cex = 0.85,
  lab.max.mar = 0.25,
  lab.fit = "auto",
  xlim.add,
  ylim.add,
  only.params = FALSE,
  sep,
  as.multiple = FALSE,
  bg,
  group = "auto",}
group.par = list(lwd = 2, line = 3, tcl = 0.75),
main = "Effect on __depvar__",
value.lab = "Estimate and __ci__ Conf. Int.",
ylab = NULL,
xlab = NULL,
sub = NULL
)

Arguments

object Can be either: i) an estimation object (obtained for example from `feols`), ii) a list of estimation objects (several results will be plotted at once), iii) a matrix of coefficients table, iv) a numeric vector of the point estimates – the latter requiring the extra arguments `sd` or `ci_low` and `ci_high`.

other Other arguments to be passed to `summary`, if `object` is an estimation, and/or to the function `plot` or `lines` (if `add = TRUE`).

style A character scalar giving the style of the plot to be used. You can set styles with the function `setFixest_coefplot`, setting all the default values of the function. If missing, then it switches to either "default" or "iplot", depending on the calling function.

sd The standard errors of the estimates. It may be missing.

ci_low If `sd` is not provided, the lower bound of the confidence interval. For each estimate.

ci_high If `sd` is not provided, the upper bound of the confidence interval. For each estimate.

df.t Integer scalar or NULL (default). The degrees of freedom (DoF) to use when computing the confidence intervals with the Student t. By default it tries to capture the DoF from the estimation. To use a Normal law to compute the confidence interval, use `df.t = Inf`.

x The value of the x-axis. If missing, the names of the argument `estimate` are used.

x.shift Shifts the confidence intervals bars to the left or right, depending on the value of `x.shift`. Default is 0.

horiz A logical scalar, default is FALSE. Whether to display the confidence intervals horizontally instead of vertically.

dict A named character vector or a logical scalar. It changes the original variable names to the ones contained in the dictionary. E.g. to change the variables named `a` and `b3` to (resp.) "$\log(a)$" and "$\text{bonus}^3$", use `dict=c(a="$\log(a)$",b3="$\text{bonus}^3$")`. By default, it is equal to `getFixest_dict()`, a default dictionary which can be set with `setFixest_dict()`. You can use `dict = FALSE` to disable it. By default `dict` modifies the entries in the global dictionary, to disable this behavior, use "reset" as the first element (ex: `dict=c("reset", mpg="Miles per gallon")`).

keep Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This
argument is applied post aliasing (see argument dict). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use keep = "x[:digit:]]$". If the first character is an exclamation mark, the effect is reversed (e.g. keep = "!Intercept" means: every variable that does not contain "Intercept" is kept). See details.

**drop**

Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see base::regex help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument dict). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use drop = "x[:digit:]]{2}". If the first character is an exclamation mark, the effect is reversed (e.g. drop = "!Intercept" means: every variable that does not contain "Intercept" is dropped). See details.

**order**

Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see base::regex help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument dict). Example: you have the following variables: month1 to month6, then x1 to x5, then year1 to year6. If you want to display first the x’s, then the years, then the months you could use: order = c("x", "year"). If the first character is an exclamation mark, the effect is reversed (e.g. order = "!Intercept" means: every variable that does not contain "Intercept" goes first). See details.

**ci.width**

The width of the extremities of the confidence intervals. Default is 0.1.

**ci_level**

Scalar between 0 and 1: the level of the CI. By default it is equal to 0.95.

**add**

Default is FALSE, if the intervals are to be added to an existing graph. Note that if it is the case, then the argument x MUST be numeric.

**pt.pch**

The patch of the coefficient estimates. Default is 1 (circle).

**pt.bg**

The background color of the point estimate (when the pt.pch is in 21 to 25). Defaults to NULL.

**cex**

Numeric, default is 1. Expansion factor for the points

**pt.cex**

The size of the coefficient estimates. Default is the other argument cex.

**col**

The color of the points and the confidence intervals. Default is 1 ("black"). Note that you can set the colors separately for each of them with pt.col and ci.col.

**pt.col**

The color of the coefficient estimates. Default is equal to the other argument col.

**ci.col**

The color of the confidence intervals. Default is equal to the other argument col.

**lwd**

General line with. Default is 1.

**pt.lwd**

The line width of the coefficient estimates. Default is equal to the other argument lwd.

**ci.lwd**

The line width of the confidence intervals. Default is equal to the other argument lwd.

**ci.lty**

The line type of the confidence intervals. Default is 1.
grid Logical, default is TRUE. Whether a grid should be displayed. You can set the display of the grid with the argument grid.par.

grid.par List. Parameters of the grid. The default values are: `lty = 3` and `col = "gray"`. You can add any graphical parameter that will be passed to `graphics::abline`. You also have two additional arguments: use `horiz = FALSE` to disable the horizontal lines, and use `vert = FALSE` to disable the vertical lines. Eg: `grid.par = list(vert = FALSE, col = "red", lwd = 2).

zero Logical, default is TRUE. Whether the 0-line should be emphasized. You can set the parameters of that line with the argument zero.par.

den.par List. Parameters of the zero-line. The default values are `col = "black"` and `lwd = 1`. You can add any graphical parameter that will be passed to `graphics::abline`. Example: `zero.par = list(col = "darkblue", lwd = 3).

pt.join Logical, default is FALSE. If TRUE, then the coefficient estimates are joined with a line.

den.join.par List. Parameters of the line joining the coefficients. The default values are: `col = pt.col` and `lwd = lwd`. You can add any graphical parameter that will be passed to `lines`. Eg: `pt.join.par = list(lty = 2).

ci.join Logical default to FALSE. Whether to join the extremities of the confidence intervals. If TRUE, then you can set the graphical parameters with the argument ci.join.par.

den.join.par A list of parameters to be passed to `graphics::lines`. Only used if `ci.join=TRUE`. By default it is equal to `list(lwd = lwd, col = col, lty = 2).

ci.fill Logical default to FALSE. Whether to fill the confidence intervals with a color. If TRUE, then you can set the graphical parameters with the argument ci.fill.par.

den.fill.par A list of parameters to be passed to `graphics::polygon`. Only used if `ci.fill=TRUE`. By default it is equal to `list(col = "lightgray", alpha = 0.5)`. Note that alpha is a special parameter that adds transparency to the color (ranges from 0 to 1).

den Logical or numeric, default is "auto", whose behavior depends on the situation. It is TRUE only if: i) interactions are plotted, ii) the x values are numeric and iii) a reference is found. If TRUE, then a vertical line is drawn at the level of the reference value. Otherwise, if numeric a vertical line will be drawn at that specific value.

den.line Logical or numeric, default is "auto", whose behavior depends on the situation. It is TRUE only if: i) interactions are plotted, ii) the x values are numeric and iii) a reference is found. If TRUE, then a vertical line is drawn at the level of the reference value. Otherwise, if numeric a vertical line will be drawn at that specific value.

den.line.par List. Parameters of the vertical line on the reference. The default values are: `col = "black"` and `lty = 2`. You can add any graphical parameter that will be passed to `graphics::abline`. Eg: `den.line.par = list(lty = 1, lwd = 3).`
The size of the labels of the coefficients. Default is missing. It is automatically set by an internal algorithm which can go as low as `lab.min.cex` (another argument).

The minimum size of the coefficients labels, as set by the internal algorithm. Default is 0.85.

The maximum size the left margin can take when trying to fit the coefficient labels into it (only when `horiz = TRUE`). This is used in the internal algorithm fitting the coefficient labels. Default is 0.25.

The method to fit the coefficient labels into the plotting region (only when `horiz = FALSE`). Can be "auto" (the default), "simple", "multi" or "tilted". If "simple", then the classic axis is drawn. If "multi", then the coefficient labels are fit horizontally across several lines, such that they don’t collide. If "tilted", then the labels are tilted. If "auto", an automatic choice between the three is made.

A numeric vector of length 1 or 2. It represents an extension factor of `xlim`, in percentage. Eg: `xlim.add = c(0, 0.5)` extends `xlim` of 50% on the right. If of length 1, positive values represent the right, and negative values the left (Eg: `xlim.add = -0.5` is equivalent to `xlim.add = c(0.5, 0)`).

A numeric vector of length 1 or 2. It represents an extension factor of `ylim`, in percentage. Eg: `ylim.add = c(0, 0.5)` extends `ylim` of 50% on the top. If of length 1, positive values represent the top, and negative values the bottom (Eg: `ylim.add = -0.5` is equivalent to `ylim.add = c(0.5, 0)`).

Logical, default is `FALSE`. If `TRUE` no graphic is displayed, only the values of `x` and `y` used in the plot are returned.

The distance between two estimates – only when argument `object` is a list of estimation results.

Logical: default is `FALSE`. Only when `object` is a single estimation result: whether each coefficient should have a different color, line type, etc. By default they all get the same style.

Background color for the plot. By default it is white.

A list, default is missing. Each element of the list reports the coefficients to be grouped while the name of the element is the group name. Each element of the list can be either: i) a character vector of length 1, ii) of length 2, or ii) a numeric vector. If equal to: i) then it is interpreted as a pattern: all element fitting the regular expression will be grouped (note that you can use the special character "^^" to clean the beginning of the names, see example), if ii) it corresponds to the first and last elements to be grouped, if iii) it corresponds to the coefficients numbers to be grouped. If equal to a character vector, you can use a percentage to tell the algorithm to look at the coefficients before aliasing (e.g. "%varname"). Example of valid uses: `group=list(group_name="pattern")`, `group=list(group_name=c("var_start", "var_end"), group=list(group_name=1:2))`. See details.

A list of parameters controlling the display of the group. The parameters controlling the line are: `lwd`, `tcl` (length of the tick), `line.adj` (adjustment of the position, default is 0), `tick` (whether to add the ticks), `lwd.ticks`, `col.ticks`. Then the parameters controlling the text: `text.adj` (adjustment of the position, default is 0), `text.cex`, `text.font`, `text.col`. 
The title of the plot. Default is "Effect on __depvar__". You can use the special variable __depvar__ to set the title (useful when you set the plot default with setFixest_coefplot).

The label to appear on the side of the coefficient values. If horiz = FALSE, the label appears in the y-axis. If horiz = TRUE, then it appears on the x-axis. The default is equal to "Estimate and __ci__ Conf. Int.", with __ci__ a special variable giving the value of the confidence interval.

The label of the y-axis, default is NULL. Note that if horiz = FALSE, it overrides the value of the argument value.lab.

The label of the x-axis, default is NULL. Note that if horiz = TRUE, it overrides the value of the argument value.lab.

A subtitle, default is NULL.

Integer scalar, default is 1. In iplot, used to select which variable created with i() to select. Only used when there are several variables created with i. This is an index, just try increasing numbers to hopefully obtain what you want. Note that it works much better when the variables are "pure" i() variables and not interacted with other variables. For example: i(species, x1) is good while i(species):x1 isn't. The latter will also work but the index may feel weird in case there are many i() variables.

Functions

• iplot(): Plots the coefficients generated with i()

Setting custom default values

The function coefplot dispose of many arguments to parametrize the plots. Most of these arguments can be set once an for all using the function setFixest_coefplot. See Example 3 below for a demonstration.

iplot

The function iplot restricts coefplot to interactions or factors created with the function i. Only one of the i-variables will be plotted at a time. If you have several i-variables, you can navigate through them with the i.select argument.

The argument i.select is an index that will go through all the i-variables. It will work well if the variables are pure, meaning not interacted with other variables. If the i-variables are interacted, the index may have an odd behavior but will (in most cases) work all the same, just try some numbers up until you (hopefully) obtain the graph you want.

Note, importantly, that interactions of two factor variables are (in general) disregarded since they would require a 3-D plot to be properly represented.

Arguments keep, drop and order

The arguments keep, drop and order use regular expressions. If you are not aware of regular expressions, I urge you to learn it, since it is an extremely powerful way to manipulate character strings (and it exists across most programming languages).
For example drop = "Wind" would drop any variable whose name contains "Wind". Note that variables such as "Temp:Wind" or "StrongWind" do contain "Wind", so would be dropped. To drop only the variable named "Wind", you need to use drop = "^Wind$" (with "^" meaning beginning, resp. "$" meaning end, of the string => this is the language of regular expressions).

Although you can combine several regular expressions in a single character string using pipes, drop also accepts a vector of regular expressions.

You can use the special character "!" (exclamation mark) to reverse the effect of the regular expression (this feature is specific to this function). For example drop = "!Wind" would drop any variable that does not contain "Wind".

You can use the special character "%" (percentage) to make reference to the original variable name instead of the aliased name. For example, you have a variable named "Month6", and use a dictionary dict = c(Month6="June"). Thus the variable will be displayed as "June". If you want to delete that variable, you can use either drop="June", or drop="%Month6" (which makes reference to its original name).

The argument order takes in a vector of regular expressions, the order will follow the elements of this vector. The vector gives a list of priorities, on the left the elements with highest priority. For example, order = c("Wind", "!Inter", "!Temp") would give highest priorities to the variables containing "Wind" (which would then appear first), second highest priority is the variables not containing "Inter", last, with lowest priority, the variables not containing "Temp". If you had the following variables: (Intercept), Temp:Wind, Wind, Temp you would end up with the following order: Wind, Temp:Wind, Temp, (Intercept).

Author(s)
Laurent Berge

See Also
See setFixest_coefplot to set the default values of coefplot, and the estimation functions: e.g. feols, fepois, feglm, fenegbin.

Examples

```r
# Example 1: Stacking two sets of results on the same graph
#
# Estimation on Iris data with one fixed-effect (Species)
est = feols(Petal.Length ~ Petal.Width + Sepal.Length + Sepal.Width | Species, iris)
est_clu = summary(est)
est_std = summary(est, se = "iid")
```

# You can plot the two results at once
coefplot(list(est_clu, est_std))

# Alternatively, you can use the argument x.shift
# to do it sequentially:

# First graph with clustered standard-errors
coefplot(est, x.shift = -.2)

# 'x.shift' was used to shift the coefficients on the left.

# Second set of results: this time with
# standard-errors that are not clustered.
coefplot(est, se = "iid", x.shift = .2,
         add = TRUE, col = 2, ci.lty = 2, pch=15)

# Note that we used 'se', an argument that will
# be passed to summary.fixest

legend("topright", col = 1:2, pch = 20, lwd = 1, lty = 1:2,
       legend = c("Clustered", "IID"), title = "Standard-Errors")

#

# Example 2: Interactions
#

# Now we estimate and plot the "yearly" treatment effects

data(base_did)
base_inter = base_did

# We interact the variable 'period' with the variable 'treat'
est_did = feols(y ~ x1 + i(period, treat, 5) | id+period, base_inter)

# In the estimation, the variable treat is interacted
# with each value of period but 5, set as a reference

# coefplot will show all the coefficients:
coefplot(est_did)

# Note that the grouping of the coefficients is due to 'group = "auto"

# If you want to keep only the coefficients
# created with i() (ie the interactions), use iplot
iplot(est_did)

# When estimations contain interactions, as before,
# the default behavior of coefplot changes,
# it now only plots interactions:
coefplot(est_did)
# We can see that the graph is different from before:
# - only interactions are shown,
# - the reference is present,
# => this is fully flexible

iplot(est_did, ref.line = FALSE, pt.join = TRUE)

# What if the interacted variable is not numeric?
# Let's create a "month" variable
all_months = c("aug", "sept", "oct", "nov", "dec", "jan",
        "feb", "mar", "apr", "may", "jun", "jul")
base_inter$period_month = all_months[base_inter$period]

# The new estimation
est = feols(y ~ x1 + i(period_month, treat, "oct") | id+period, base_inter)
# Since 'period_month' of type character, coefplot sorts it
iplot(est)

# To respect a plotting order, use a factor
base_inter$month_factor = factor(base_inter$period_month, levels = all_months)
est = feols(y ~ x1 + i(month_factor, treat, "oct") | id+period, base_inter)
iplot(est)

# Example 3: Setting defaults

# coefplot has many arguments, which makes it highly flexible.
# If you don't like the default style of coefplot. No worries,
# you can set *your* default by using the function
# setFixest_coefplot()

dict = c("Petal.Length"="Length (Petal)", "Petal.Width"="Width (Petal)",
        "Sepal.Length"="Length (Sepal)", "Sepal.Width"="Width (Sepal)"")

setFixest_coefplot(ci.col = 2, pt.col = "darkblue", ci.lwd = 3,
        pt.cex = 2, pt.pch = 15, ci.width = 0, dict = dict)
est = feols(Petal.Length ~ Petal.Width + Sepal.Length +
        Sepal.Width + i(Species), iris)
# And that's it
coefplot(est)

# You can set separate default values for iplot
setFixest_coefplot("iplot", pt.join = TRUE, pt.join.par = list(lwd = 2, lty = 2))
iplot(est)

# To reset to the default settings:
setFixest_coefplot("all", reset = TRUE)
coefplot(est)
#
# Example 4: group + cleaning
#
# You can use the argument group to group variables
# You can further use the special character "^^" to clean
# the beginning of the coef. name: particularly useful for factors

est = feols(Petal.Length ~ Petal.Width + Sepal.Length +
            Sepal.Width + Species, iris)

# No grouping:
coefplot(est)

# now we group by Sepal and Species
coefplot(est, group = list(Sepal = "Sepal", Species = "Species"))

# now we group + clean the beginning of the names using the special character ^^
coefplot(est, group = list(Sepal = "^^Sepal.", Species = "^^Species"))

---

coeftable

*Extracts the coefficients table from an estimation*

**Description**

Methods to extracts the coefficients table and its sub-components from an estimation.

**Usage**

coeftable(object, ...)

se(object, ...)

pvalue(object, ...)

tstat(object, ...)

**Arguments**

object       An estimation (fitted model object), e.g. a fixest object.
...

Other arguments to the methods.

**Value**

Returns a matrix (coeftable) or vectors.
coeftable.default

See Also

Please look at the coeftable.fixest page for more detailed information.

Examples

```r
est = lm(mpg ~ cyl, mtcars)
coeftable(est)
```

## coeftable.default

Extracts the coefficients table from an estimation

Description

Default method to extracts the coefficients table and its sub-components from an estimation.

Usage

```r
## Default S3 method:
coeftable(object, keep, drop, order, ...)

## Default S3 method:
se(object, keep, drop, order, ...)

## Default S3 method:
tstat(object, keep, drop, order, ...)

## Default S3 method:
pvalue(object, keep, drop, order, ...)

## S3 method for class 'matrix'
se(object, keep, drop, order, ...)
```

Arguments

- `object`: The result of an estimation (a fitted model object). Note that this function is made to work with fixest objects so it may not work for the specific model you provide.

- `keep`: Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument `dict`). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use keep = "x[[:digit:]]". If the first character is an exclamation mark, the effect is reversed (e.g. keep = "!Intercept" means: every variable that does not contain “Intercept” is kept). See details.
drop

Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument `dict`). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use `drop = "x[[[:digit:]]]{2}"`. If the first character is an exclamation mark, the effect is reversed (e.g. `drop = "!Intercept"` means: every variable that does not contain “Intercept” is dropped). See details.

order

Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see `base::regex` help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument `dict`). Example: you have the following variables: `month1` to `month6`, then `x1` to `x5`, then `year1` to `year6`. If you want to display first the `x`'s, then the years, then the months you could use: `order = c("x", "year")`. If the first character is an exclamation mark, the effect is reversed (e.g. `order = "!Intercept"` means: every variable that does not contain “Intercept” goes first). See details.

... Other arguments that will be passed to summary.

First the method `summary` is applied if needed, then the coefficients table is extracted from its output.

The default method is very naive and hopes that the resulting coefficients table contained in the summary of the fitted model is well formed: this assumption is very often wrong. Anyway, there is no development intended since the `coeftable/se/pvalue/tstat` series of methods is only intended to work well with `fixest` objects. To extract the coefficients table from fitted models in a general way, it’s better to use `tidy from broom`.

Value

Returns a matrix (`coeftable`) or vectors.

Functions

- `se(default)`: Extracts the standard-errors from an estimation
- `tstat(default)`: Extracts the standard-errors from an estimation
- `pvalue(default)`: Extracts the p-values from an estimation
- `se(matrix)`: Extracts the standard-errors from a VCOV matrix

Examples

```r
# NOTA: This function is really made to handle fixest objects
# The default methods works for simple structures, but you'd be
# likely better off with broom::tidy for other models

est = lm(mpg ~ cyl, mtcars)
coeftable(est)
```
se(est)

---

**coeftable.fixest**

*Obtain various statistics from an estimation*

---

**Description**

Set of functions to directly extract some commonly used statistics, like the p-value or the table of coefficients, from estimations. This was first implemented for `fixest` estimations, but has some support for other models.

**Usage**

```r
## S3 method for class 'fixest'
coeftable(
  object,
  vcov = NULL,
  ssc = NULL,
  cluster = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  list = FALSE,
  ...
)

## S3 method for class 'fixest'
se(
  object,
  vcov = NULL,
  ssc = NULL,
  cluster = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  ...
)

## S3 method for class 'fixest'
tstat(
  object,
  vcov = NULL,
  ssc = NULL,
  cluster = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  ...
)
```
## Arguments

**object**  
A fixest object. For example an estimation obtained from `feols`.

**vcov**  
A function to be used to compute the standard-errors of each fixest object. You can pass extra arguments to this function using the argument `vcov_args`. See the example.

**ssc**  
An object of class `ssc.type` obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: `adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE)`. See the help of the function `ssc` for details.

**cluster**  
Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over `var1` and `var2` contained in the data.frame `base` used for the estimation. All the following `cluster` arguments are valid and do the same thing: `cluster = base[, c("var1, "var2")], cluster = c("var1, "var2"), cluster = ~var1+var2`. If the two variables were used as clusters in the estimation, you could further use `cluster = 1:2` or leave it blank with `se = "twoway"` (assuming `var1` [resp. `var2`] was the 1st [resp. 2nd] cluster).

**keep**  
Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument `dict`). Example: you have the variable `x1` to `x55` and want to display only `x1` to `x9`, then you could use `keep = "x[[:digit:]]$"`. If the first character is an exclamation mark, the effect is
reversed (e.g. keep = "!Intercept" means: every variable that does not contain "Intercept" is kept). See details.

drop  Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument `dict`). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use drop = "x[[:digit:]]{2}". If the first character is an exclamation mark, the effect is reversed (e.g. drop = "!Intercept" means: every variable that does not contain "Intercept" is dropped). See details.

order  Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see `base::regex` help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument `dict`). Example: you have the following variables: month1 to month6, then x1 to x5, then year1 to year6. If you want to display first the x’s, then the years, then the months you could use: order = c("x", "year"). If the first character is an exclamation mark, the effect is reversed (e.g. order = "!Intercept" means: every variable that does not contain "Intercept" goes first). See details.

list  Logical, default is FALSE. If TRUE, then a nested list is returned, the first layer is accessed with the coefficients names; the second layer with the following values: coef, se, tstat, pvalue. Note that the variable "(Intercept)" is renamed into "constant".

...  Other arguments to be passed to `summary.fixest`.

Details

This set of tiny functions is primarily constructed for fixest estimations.

Value

Returns a table of coefficients, with in rows the variables and four columns: the estimate, the standard-error, the t-statistic and the p-value.

If list = TRUE then a nested list is returned, the first layer is accessed with the coefficients names; the second layer with the following values: coef, se, tstat, pvalue. For example, with res = coeftable(est, list = TRUE) you can access the SE of the coefficient x1 with res$x1$se; and its coefficient with res$x1$coef, etc.

Functions

• se(fixest): Extracts the standard-error of an estimation
• tstat(fixest): Extracts the t-statistics of an estimation
• pvalue(fixest): Extracts the p-value of an estimation
Examples

```r
# Some data and estimation
data(trade)
est = fe pois(Euros ~ log(dist_km) | Origin^Product + Year, trade)
#
# Coeftable/se/tstat/pvalue
#
# Default is clustering along Origin^Product
coeftable(est)
se(est)
tstat(est)
pvalue(est)

# Now with two-way clustered standard-errors
# and using coeftable()

coeftable(est, cluster = ~Origin + Product)
se(est, cluster = ~Origin + Product)
pvalue(est, cluster = ~Origin + Product)
tstat(est, cluster = ~Origin + Product)

# Or you can cluster only once:
est_sum = summary(est, cluster = ~Origin + Product)
coeftable(est_sum)
se(est_sum)
tstat(est_sum)
pvalue(est_sum)

# You can use the arguments keep, drop, order
# to rearrange the results
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")
est_iv = feols(y ~ x1 | x2 ~ x3, base)
tstat(est_iv, keep = "x1")
coeftable(est_iv, keep = "x1|Int")

coeftable(est_iv, order = "!Int")

# Using lists
#
# Returning the coefficients table as a list can be useful for quick
# reference in markdown documents.
# Note that the "(Intercept)" is renamed into "constant"
```
res = coef_table(est_iv, list = TRUE)

# coefficient of the constant:
res$constant$coef

# pvalue of x1
res$x1$pvalue

description

Series of methods to extract the coefficients table or its sub-components from a fixest_multi objects (i.e. the outcome of multiple estimations).

usage

## S3 method for class 'fixest_multi'
coef_table(
  object,
  vcov = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  long = FALSE,
  wide = FALSE,
  ...
)

## S3 method for class 'fixest_multi'
se(
  object,
  vcov = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  long = FALSE,
  wide = FALSE,
  ...
)

## S3 method for class 'fixest_multi'
tstat(
  object,
  vcov = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  long = FALSE,
  wide = FALSE,
  ...
)
```r
vcov = NULL,
keep = NULL,
drop = NULL,
order = NULL,
long = FALSE,
```

```r
## S3 method for class 'fixest_multi'

pvalue(
  object,
  vcov = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  long = FALSE,
  ...
)
```

**Arguments**

- **object**
  A `fixest_multi` object, coming from a `fixest` multiple estimation.

- **vcov**
  A function to be used to compute the standard-errors of each fixest object. You can pass extra arguments to this function using the argument `.vcov_args`. See the example.

- **keep**
  Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument `dict`). Example: you have the variable `x1` to `x55` and want to display only `x1` to `x9`, then you could use `keep = "x\[[:digit:]]\$"`. If the first character is an exclamation mark, the effect is reversed (e.g. `keep = "!Intercept"` means: every variable that does not contain “Intercept” is kept). See details.

- **drop**
  Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument `dict`). Example: you have the variable `x1` to `x55` and want to display only `x1` to `x9`, then you could use `drop = "x\[[:digit:]]\{2\}"`. If the first character is an exclamation mark, the effect is reversed (e.g. `drop = "!Intercept"` means: every variable that does not contain “Intercept” is dropped). See details.

- **order**
  Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see `base::regex` help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument `dict`). Example: you have the following variables: `month1` to `month6`, then `x1` to `x5`, then `year1` to `year6`. If you want to display first the `x`’s, then the years, then the months
you could use: `order = c("x", "year")`. If the first character is an exclamation mark, the effect is reversed (e.g. `order = "!Intercept"` means: every variable that does not contain “Intercept” goes first). See details.

**long**

Logical scalar, default is FALSE. If TRUE, then all the information is stacked, with two columns containing the information: "param" and "value". The column `param` contains the values `coef/se/tstat/pvalue`.

**wide**

A logical scalar, default is FALSE. If TRUE, then a list is returned: the elements of the list are `coef/se/tstat/pvalue`. Each element of the list is a wide table with a column per coefficient.

... Other arguments to be passed to `summary.fixest`.

### Value

It returns a `data.frame` containing the coefficients tables (or just the `se/pvalue/tstat`) along with the information on which model was estimated.

If `wide` = TRUE, then a list is returned. The elements of the list are `coef/se/tstat/pvalue`. Each element of the list is a wide table with a column per coefficient.

If `long` = TRUE, then all the information is stacked. This removes the 4 columns containing the coefficient estimates to the p-values, and replace them with two new columns: "param" and "value". The column `param` contains the values `coef/se/tstat/pvalue`, and the column `values` the associated numerical information.

### Functions

- `se(fixest_multi)`: Extracts the standard-errors from `fixest_multi` estimations
- `tstat(fixest_multi)`: Extracts the t-stats from `fixest_multi` estimations
- `pvalue(fixest_multi)`: Extracts the p-values from `fixest_multi` estimations

### Examples

```r
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
est_multi = feols(y~ csw(x.[,1:3]), base, split = ~species)

# we get all the coefficient tables at once
coeftable(est_multi)

# Now just the standard-errors
se(est_multi)

# wide = TRUE => leads to a list of wide tables
coeftable(est_multi, wide = TRUE)

# long = TRUE, all the information is stacked
coeftable(est_multi, long = TRUE)
```
Description

In some occasions, the optimization algorithm of `femlm` may fail to converge, or the variance-covariance matrix may not be available. The most common reason of why this happens is collinearity among variables. This function helps to find out which set of variables is problematic.

Usage

collinearity(x, verbose)

Arguments

x
A fixest object obtained from, e.g. functions `femlm`, `feols` or `feglm`.

verbose
An integer. If higher than or equal to 1, then a note is prompted at each step of the algorithm. By default `verbose = 0` for small problems and to 1 for large problems.

Details

This function tests: 1) collinearity with the fixed-effect variables, 2) perfect multi-collinearity between the variables, 4) perfect multi-collinearity between several variables and the fixed-effects, and 4) identification issues when there are non-linear in parameters parts.

Value

It returns a text message with the identified diagnostics.

Author(s)

Laurent Berge

Examples

# Creating an example data base:
set.seed(1)
fe_1 = sample(3, 100, TRUE)
fe_2 = sample(20, 100, TRUE)
x = rnorm(100, fe_1)**2
y = rnorm(100, fe_2)**2
z = rnorm(100, 3)**2
dep = rpois(100, x*y*z)
base = data.frame(fe_1, fe_2, x, y, z, dep)

# creating collinearity problems:
base$v1 = base$v2 = base$v3 = base$v4 = 0
base$v1[base$fe_1 == 1] = 1
base$v2[base$fe_1 == 2] = 1
base$v3[base$fe_1 == 3] = 1
base$v4[base$fe_2 == 1] = 1

# Estimations:

# Collinearity with the fixed-effects:
res_1 = femlm(dep ~ log(x) + v1 + v2 + v4 | fe_1 + fe_2, base)
collinearity(res_1)

# => collinearity with the first fixed-effect identified, we drop v1 and v2
res_1bis = femlm(dep ~ log(x) + v4 | fe_1 + fe_2, base)
collinearity(res_1bis)

# Multi-Collinearity:
res_2 = femlm(dep ~ log(x) + v1 + v2 + v3 + v4, base)
collinearity(res_2)

---

confint.fixest  Confidence interval for parameters estimated with fixest

Description

This function computes the confidence interval of parameter estimates obtained from a model estimated with femlm, feols or feglm.

Usage

## S3 method for class 'fixest'
confint(
  object,
  parm,
  level = 0.95,
  vcov,
  se,
  cluster,
  ssc = NULL,
  coef.col = FALSE,
  ...
)

Arguments

object  A fixest object. Obtained using the functions femlm, feols or feglm.
`confint.fixest`

parm  The parameters for which to compute the confidence interval (either an integer vector or a character vector with the parameter name). If missing, all parameters are used.

level  The confidence level. Default is 0.95.

vcov  Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the `vcov` documentation in the vignette.

se  Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "three-way" or "four-way"? By default if there are clusters in the estimation: `se = "cluster"`, otherwise `se = "iid"`. Note that this argument is deprecated, you should use `vcov` instead.

cluster  Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over `var1` and `var2` contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: `cluster = base[, c("var1", "var2")], cluster = c("var1", "var2")], cluster = ~var1+var2`. If the two variables were used as fixed-effects in the estimation, you can leave it blank with `vcov = "twoway"` (assuming `var1` [resp. `var2`] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using `^` with the following syntax: `cluster = ~var1^var2` or `cluster = "var1^var2"`.

ssc  An object of class `ssc.type` obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: `adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE`). See the help of the function `ssc` for details.

test.col  Logical, default is FALSE. If TRUE the column coefficient is inserted in the first position containing the coefficient names.

...  Not currently used.

Value

Returns a data.frame with two columns giving respectively the lower and upper bound of the confidence interval. There is as many rows as parameters.

Author(s)

Laurent Berge
**Examples**

```r
# Load trade data
data(trade)

# We estimate the effect of distance on trade (with 3 fixed-effects)
est_pois = femlm(Euros ~ log(dist_km) + log(Year) | Origin + Destination + Product, trade)

# confidence interval with "normal" VCOV
confint(est_pois)

# confidence interval with "clustered" VCOV (w.r.t. the Origin factor)
confint(est_pois, se = "cluster")
```

---

**confint.fixest_multi**  
*Confidence intervals for fixest_multi objects*

**Description**

Computes the confidence intervals of parameter estimates for `fixest`'s multiple estimation objects (aka `fixest_multi`).

**Usage**

```r
## S3 method for class 'fixest_multi'
confint(
  object,
  parm,
  level = 0.95,
  vcov = NULL,
  se = NULL,
  cluster = NULL,
  ssc = NULL,
  ...)
```

**Arguments**

- `object`  
  A `fixest_multi` object obtained from a multiple estimation in `fixest`.

- `parm`  
  The parameters for which to compute the confidence interval (either an integer vector OR a character vector with the parameter name). If missing, all parameters are used.

- `level`  
  The confidence level. Default is 0.95.
vcov  
Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: vcov_type ~ variables. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from vcov_cluster, vcov_NW, NW, vcov_DK, DK, vcov_conley and conley. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the vcov documentation in the vignette.

se  
Character scalar. Which kind of standard error should be computed: “standard”, “hetero”, “cluster”, “twoway”, “threeway” or “fourway”? By default if there are clusters in the estimation: se = “cluster”, otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

cluster  
Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

ssc  
An object of class ssc.type obtained with the function ssc. Represents how the degree of freedom correction should be done. You must use the function ssc for this argument. The arguments and defaults of the function ssc are: adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE). See the help of the function ssc for details.

...  
Not currently used.

Value  
It returns a data frame whose first columns indicate which model has been estimated. The last three columns indicate the coefficient name, and the lower and upper confidence intervals.

Examples

base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
est = feols(y ~ csw(x.[,1:3]) | sw0(species), base, vcov = "iid")

confint(est)
# focusing only on the coefficient 'x3'
confint(est, "x3")

# the 'id' provides the index of the estimation
est[c(3, 6)]
degrees_freedom

Gets the degrees of freedom of a fixest estimation

Description

Simple utility to extract the degrees of freedom from a fixest estimation.

Usage

degrees_freedom(
  x,
  type,
  vars = NULL,
  vcov = NULL,
  se = NULL,
  cluster = NULL,
  ssc = NULL,
  stage = 2
)

degrees_freedom_iid(x, type)

Arguments

x  A fixest estimation.

type  Character scalar, equal to "k", "resid", "t". If "k", then the number of regressors is returned. If "resid", then it is the "residuals degree of freedom", i.e. the number of observations minus the number of regressors. If "t", it is the degrees of freedom used in the t-test. Note that these values are affected by how the VCOV of x is computed, in particular when the VCOV is clustered.

vars  A vector of variable names, of the regressors. This is optional. If provided, then type is set to 1 by default and the number of regressors contained in vars is returned. This is only useful in the presence of collinearity and we want a subset of the regressors only. (Mostly for internal use.)

vcov  Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: vcov_type ~ variables. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from vcov_cluster, vcov_NW, NW, vcov_DK, DK, vcov_conley and conley. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the vcov documentation in the vignette.

se  Character scalar. Which kind of standard error should be computed: “standard”, “hetero”, “cluster”, “twoway”, “threeway” or “fourway”? By default if there are clusters in the estimation: se = “cluster”, otherwise se = “iid”. Note that this argument is deprecated, you should use vcov instead.
**cluster**

Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over `var1` and `var2` contained in the data.frame `base` used for the estimation. All the following cluster arguments are valid and do the same thing: `cluster = base[, c("var1", "var2")], cluster = c("var1", "var2").cluster = ~var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with `vcov = "twoway"` (assuming `var1` [resp. `var2`] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using `^` with the following syntax: `cluster = ~var1^var2` or `cluster = "var1^var2"`.

**ssc**

An object of class `ssc.type` obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: `adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE`). See the help of the function `ssc` for details.

**stage**

Either 1 or 2. Only concerns IV regressions, which stage to look at. The type of VCOV can have an influence on the degrees of freedom. In particular, when the VCOV is clustered, the DoF returned will be in accordance with the way the small sample correction was performed when computing the VCOV. That type of value is in general not what we have in mind when we think of “degrees of freedom”. To obtain the ones that are more intuitive, please use `degrees_freedom_iid` instead.

**Functions**

- `degrees_freedom_iid()`: Gets the degrees of freedom of a `fixest` estimation

**Examples**

```r
# First: an estimation
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")
est = feols(y ~ x1 + x2 | species, base)

# "Normal" standard-errors (SE)
est_standard = summary(est, se = "st")

# Clustered SEs
est_clustered = summary(est, se = "clu")

# The different degrees of freedom

# => different type 1 DoF (because of the clustering)
degrees_freedom(est_standard, type = "k")
degrees_freedom(est_clustered, type = "k") # fixed-effects are excluded

# => different type 2 DoF (because of the clustering)
```
demean

Centers a set of variables around a set of factors

Description

User-level access to internal demeaning algorithm of fixest.

Usage

demean(
  X,
  f,
  slope.vars,
  slope.flag,
  data,
  weights,
  nthreads = getFixest_nthreads(),
  notes = getFixest_notes(),
  iter = 2000,
  tol = 1e-06,
  fixef.reorder = TRUE,
  na.rm = TRUE,
  as.matrix = is.atomic(X),
  im_confident = FALSE,
  ...
)

Arguments

X
A matrix, vector, data.frame or a list OR a formula OR a feols estimation. If equal to a formula, then the argument data is required, and it must be of the type: x1 + x2 ~ f1 + fe2 with on the LHS the variables to be centered, and on the RHS the factors used for centering. Note that you can use variables with varying slopes with the syntax fe[v1, v2] (see details in feols). If a feols estimation, all variables (LHS+RHS) are demeaned and then returned (only if it was estimated with fixed-effects). Otherwise, it must represent the data to be centered. Of course the number of observations of that data must be the same as the factors used for centering (argument f).

f
A matrix, vector, data.frame or list. The factors used to center the variables in argument X. Matrices will be coerced using as.data.frame.
demean

slope.vars  A vector, matrix or list representing the variables with varying slopes. Matrices will be coerced using as.data.frame. Note that if this argument is used it MUST be in conjunction with the argument slope.flag that maps the factors to which the varying slopes are attached. See examples.

slope.flag  An integer vector of the same length as the number of variables in f (the factors used for centering). It indicates for each factor the number of variables with varying slopes to which it is associated. Positive values mean that the raw factor should also be included in the centering, negative values that it should be excluded. Sorry it’s complicated... but see the examples it may get clearer.

data  A data.frame containing all variables in the argument X. Only used if X is a formula, in which case data is mandatory.

weights  Vector, can be missing or NULL. If present, it must contain the same number of observations as in X.

nthreads  Number of threads to be used. By default it is equal to getFixest_nthreads().

notes  Logical, whether to display a message when NA values are removed. By default it is equal to getFixest_notes().

iter  Number of iterations, default is 2000.

tol  Stopping criterion of the algorithm. Default is $1e^{-6}$. The algorithm stops when the maximum absolute increase in the coefficients values is lower than tol.

fixef.reorder  Logical, default is TRUE. Whether to reorder the fixed-effects by frequencies before feeding them into the algorithm. If FALSE, the original fixed-effects order provided by the user is maintained. In general, reordering leads to faster and more precise performance.

na.rm  Logical, default is TRUE. If TRUE and the input data contains any NA value, then any observation with NA will be discarded leading to an output with less observations than the input. If FALSE, if NAs are present the output will also be filled with NAs for each NA observation in input.

as.matrix  Logical, if TRUE a matrix is returned, if FALSE it will be a data.frame. The default depends on the input, if atomic then a matrix will be returned.

im_confident  Logical, default is FALSE. FOR EXPERT USERS ONLY! This argument allows to skip some of the preprocessing of the arguments given in input. If TRUE, then X MUST be a numeric vector/matrix/list (not a formula!), f MUST be a list, slope.vars MUST be a list, slope.vars MUST be consistent with slope.flag, and weights, if given, MUST be numeric (not integer!). Further there MUST be not any NA value, and the number of observations of each element MUST be consistent. Non compliance to these rules may simply lead your R session to break.

...  Not currently used.

Value

It returns a data.frame of the same number of columns as the number of variables to be centered.

If na.rm = TRUE, then the number of rows is equal to the number of rows in input minus the number of NA values (contained in X, f, slope.vars or weights). The default is to have an output of the same number of observations as the input (filled with NAs where appropriate).

A matrix can be returned if as.matrix = TRUE.
Varying slopes

You can add variables with varying slopes in the fixed-effect part of the formula. The syntax is as follows: `fixef_var[var1, var2]`. Here the variables `var1` and `var2` will be with varying slopes (one slope per value in `fixef_var`) and the fixed-effect `fixef_var` will also be added.

To add only the variables with varying slopes and not the fixed-effect, use double square brackets: `fixef_var[[var1, var2]]`.

In other words:

- `fixef_var[var1, var2]` is equivalent to `fixef_var + fixef_var[[var1]] + fixef_var[[var2]]`
- `fixef_var[[var1, var2]]` is equivalent to `fixef_var[[var1]] + fixef_var[[var2]]`

In general, for convergence reasons, it is recommended to always add the fixed-effect and avoid using only the variable with varying slope (i.e. use single square brackets).

Examples

```r
# Illustration of the FWL theorem
data(trade)
based = trade
base$ln_dist = log(base$dist_km)
based$ln_euros = log(base$Euros)

# We center the two variables ln_dist and ln_euros
# on the factors Origin and Destination
X_demean = demean(X = based[, c("ln_dist", "ln_euros")],
    f = based[, c("Origin", "Destination")])
based[, c("ln_dist_dm", "ln_euros_dm") = X_demean

est = feols(ln_euros_dm ~ ln_dist_dm, based)
est_fe = feols(ln_euros ~ ln_dist | Origin + Destination, based)

# The results are the same as if we used the two factors
# as fixed-effects
etable(est, est_fe, se = "st")

# Variables with varying slopes

# You can center on factors but also on variables with varying slopes

# Let's have an illustration
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# We center y and x1 on species and x2 * species

# using a formula
```
deviance.fixest

Extracts the deviance of a fixest estimation

deviance.fixest

Description

Returns the deviance from a fixest estimation.
did_means

Usage

```r
## S3 method for class 'fixest'
deviance(object, ...)
```

Arguments

- **object**: A `fixest` object.
- **...**: Not currently used.

Value

Returns a numeric scalar equal to the deviance.

See Also

- `feols`, `fepois`, `feglm`, `fenegbin`, `feNmlm`

Examples

```r
est = feols(Petal.Length ~ Petal.Width, iris)
deviance(est)

est_pois = fepois(Petal.Length ~ Petal.Width, iris)
deviance(est_pois)
```

did_means

## Treated and control sample descriptives

Description

This function shows the means and standard-deviations of several variables conditional on whether they are from the treated or the control group. The groups can further be split according to a pre/post variable. Results can be seamlessly be exported to Latex.

Usage

```r
did_means(
  fml,
  base,
  treat_var,
  post_var,
  tex = FALSE,
  treat_dict,
  dict = getFixest_dict(),
  file,
```
Arguments

- **fml**: Either a formula of the type `var1 + ... + varN ~ treat` or `var1 + ... + varN ~ treat | post`. Either a data.frame/matrix containing all the variables for which the means are to be computed (they must be numeric of course). Both the treatment and the post variables must contain only exactly two values. You can use a point to select all the variables of the data set: `. ~ treat`.

- **base**: A data base containing all the variables in the formula `fml`.

- **treat_var**: Only if argument `fml` is not a formula. The vector identifying the treated and the control observations (the vector can be of any type but must contain only two possible values). Must be of the same length as the data.

- **post_var**: Only if argument `fml` is not a formula. The vector identifying the periods (pre/post) of the observations (the vector can be of any type but must contain only two possible values). The first value (in the sorted sense) of the vector is taken as the pre period. Must be of the same length as the data.

- **tex**: Should the result be displayed in Latex? Default is `FALSE`. Automatically set to `TRUE` if the table is to be saved in a file using the argument `file`.

- **treat_dict**: A character vector of length two. What are the names of the treated and the control? This should be a dictionary: e.g. `c("1"="Treated", "0" = "Control")`.

- **dict**: A named character vector. A dictionary between the variables names and an alias. For instance dict=c("x"="Inflation Rate") would replace the variable name x by “Inflation Rate”.

- **file**: A file path. If given, the table is written in Latex into this file.

- **replace**: Default is `TRUE`, which means that when the table is exported, the existing file is not erased.

- **title**: Character string giving the Latex title of the table. (Only if exported.)

- **label**: Character string giving the Latex label of the table. (Only if exported.)

- **raw**: Logical, default is `FALSE`. If `TRUE`, it returns the information without formatting.

- **indiv**: Either the variable name of individual identifiers, a one sided formula, or a vector. If the data is that of a panel, this can be used to track the number of individuals per group.

- **treat_first**: Which value of the 'treatment' vector should appear on the left? By default the max value appears first (e.g. if the treatment variable is a 0/1 vector, 1 appears first).
did_means

prepostnames

Only if there is a 'post' variable. The names of the pre and post periods to be displayed in Latex. Default is c("Before", "After").

diff.inv

Logical, default to FALSE. Whether to inverse the difference.

Details

By default, when the user tries to apply this function to non-numeric variables, an error is raised. The exception is when the all variables are selected with the dot (like in . ~ treat. In this case, non-numeric variables are automatically omitted (with a message).

NAs are removed automatically: if the data contains NAs an information message will be prompted. First all observations containing NAs relating to the treatment or post variables are removed. Then if there are still NAs for the variables, they are excluded separately for each variable, and a new message detailing the NA breakup is prompted.

Value

It returns a data.frame or a Latex table with the conditional means and statistical differences between the groups.

Examples

# Playing around with the DiD data
data(base_did)

# means of treat/control
did_means(y+x1+period~treat, base_did)

# same but inverting the difference
did_means(y+x1+period~treat, base_did, diff.inv = TRUE)

# now treat/control, before/after
did_means(y+x1+period~treat|post, base_did)

# same but with a new line giving the number of unique "indiv" for each case
did_means(y+x1+period~treat|post, base_did, indiv = "id")

# same but with the treat case "0" coming first
did_means(y+x1+period~treat|post, base_did, indiv = ~id, treat_first = 0)

# Selecting all the variables with "."
did_means(.~treat|post, base_did, indiv = "id")
dsb

Simple and powerful string manipulation with the dot square bracket operator

Description

Compactly performs many low level string operations. Advanced support for pluralization.

Usage

```r
dsb(
  ..., 
  frame = parent.frame(),
  sep = "",
  vectorize = FALSE,
  nest = TRUE,
  collapse = NULL
)
```

Arguments

- `...` Character scalars that will be collapsed with the argument `sep`. You can use ".[x]" within each character string to insert the value of `x` in the string. You can add string operations in each ".[]" instance with the syntax "'arg' op ? x" (resp. "'arg'op ! x") to apply the operation `op` with the argument `arg` to `x` (resp. the verbatim of `x`). Otherwise, what to say? Ah, nesting is enabled, and since there’s over 30 operators, it’s a bit complicated to sort you out in this small space. But type `dsb("--help")` to prompt an (almost) extensive help.

- `frame` An environment used to evaluate the variables in ".[]".

- `sep` Character scalar, default is ".". It is used to collapse all the elements in 

- `vectorize` Logical, default is `FALSE`. If `TRUE`, further elements in 

- `nest` Logical, default is `TRUE`. Whether the original character strings should be nested into a ".[]". If `TRUE`, then things like `dsb("S!one, two")` are equivalent to `dsb(".[S!one, two]")` and hence create the vector `c("one", "two")`.

- `collapse` Character scalar or `NULL` (default). If provided, the resulting character vector will be collapsed into a character scalar using this value as a separator.

There are over 30 basic string operations, it supports pluralization, it’s fast (e.g. faster than `glue` in the benchmarks), string operations can be nested (it may be the most powerful feature), operators have sensible defaults.

See detailed help on the console with `dsb("--help")`. The real help is in fact in the "Examples" section.

Value

It returns a character vector whose length depends on the elements and operations in ".[]".
Examples

# # BASIC USAGE ####
#

x = c("Romeo", "Juliet")

# .[x] inserts x
dsb("Hello .[x]!")

# elements in ... are collapsed with "" (default)
dsb("Hello .[x[1]], ",
    "how is .[x[2]] doing?")

# Splitting a comma separated string
# The mechanism is explained later
dsb("/J. Mills, David, Agnes, Dr Strong")

# Nota: this is equivalent to (explained later)
dsb("', *'S !J. Mills, David, Agnes, Dr Strong")

# # Applying low level operations to strings
#

# Two main syntax:
#
# A) expression evaluation
# .[operation ? x]
#    \-> the expression to be evaluated
#    ? means that the expression will be evaluated

# B) verbatim
# .[operation ! x]
#    \-> the expression taken as verbatim (here ' x')
#    ! means that the expression is taken as verbatim

# operation: usually 'arg'op with op an operation code.

# Example: splitting
x = "hello dear"
dsb(".['s ? x]")
# x is split by ',

dsb(".['s !hello dear]")
# 'hello dear' is split by ',
# had we used ?, there would have been an error

# By default, the string is nested in .[], so in that case no need to use .[]:
dsb("' 's ? x")
dsbb("' 's !hello dear")

# There are 35 string operators
# Operators usually have a default value
# Operations can be chained by separating them with a comma
# Example: default of 's' is ' ' + chaining with collapse
dsbb("s, ' my 'c!hello dear")

#
# Nesting
#
# .[operations ! s1.[expr]s2]
#   |   |
#   | \-> expr will be evaluated then added to the string
#   \-> nesting requires verbatim evaluation: '!

dsbb("The variables are: .[C!x.[1:4]].")

# This one is a bit ugly but it shows triple nesting
dsbb("The variables are: .[w, C!.][2* ! x.[1:4]].[S, 4** ! , _sq]].")

#
# Splitting
#
# s: split with fixed pattern, default is '

dsbb("s !a b c")
dsbb("' b 's !a b c")

# S: split with regex pattern, default is ', *

dsbb("S !a, b, c")
dsbb("[[[:punct:]]]S !a! b; c")

#
# Collapsing
#
# c and C do the same, their default is different
# syntax: 's1||s2' with
# - s1 the string used for collapsing
# - s2 (optional) the string used for the last collapse
# c: default is '

dsbb("c?1:3")

# C: default is ', || and '

dsbb("C?1:3")

dsbb("', || or 'c?1:4")
# Extraction

# x: extracts the first pattern
# X: extracts all patterns
# syntax: 'pattern'x
# Default is '[[:alnum:]]+'

x = "This years is... 2020"

dsb("x ? x")
dsb("X ? x")
dsb("'\d+'x ? x")

# STRING FORMATTING ####

# u, U: uppercase first/all letters

# first letter
dsb("u!julia mills")

# title case: split -> upper first letter -> collapse
dsb("s, u, c!julia mills")

# upper all letters
dsb("U!julia mills")

# L: lowercase
dsb("L!JULIA MILLS")

# q, Q: single or double quote
dsb("S, q, C!Julia, David, Wilkins")
dsb("S, Q, C!Julia, David, Wilkins")

# f, F: formats the string to fit the same length

score = c(-10, 2050)

nm = c("Wilkins", "David")
dsb("Monopoly scores:
.

\n
\n
\n
c ! - .[f ? nm]: .[F ? score] US$")

# OK that example may have been a bit too complex,
# let's make it simple:
dsb("Scores: [.f ? score"])
dsbf("Names: [.f ? nm"])

#
# w, W: reformat the white spaces
# w: suppresses trimming white spaces + normalizes successive white spaces
# W: same but also includes punctuation

dsbf("w ! The white spaces are now clean. ")
dsbf("W ! I, really -- truly; love punctuation!!")

#
# %: applies sprintf formatting

dsbf("pi = .[.2f % ? pi"])

#
# a: appends text on each item
# syntax: 's1's2'a', adds s1 at the beginning and s2 at the end of the string
# It accepts the special values :1:, :i:, :I:, :a:, :A:
# These values create enumerations (only one such value is accepted)

# appending square brackets
dsbf("[][|]a, 'c!x.[1:4"])

# Enumerations
acad = dsbf("/you like admin, you enjoy working on weekends, you really love emails")
dsbf("Main reasons to pursue an academic career:
 .[i:]: a, C ? acad.")

#
# A: same as 'a' but adds at the begging/end of the full string (not on the elements)
# special values: :n:, :N:, give the number of elements

characters = dsbf("/David, Wilkins, Dora, Agnes")
dsbf("There are .[N: characters: 'A, C ? characters.")

#
# stop: removes basic English stopwords
# the list is from the Snowball project: http://snowball.tartarus.org/algorithms/english/stop.txt

dsbf("stop, w!It is a tale told by an idiot, full of sound and fury, signifying nothing.")

#
# k: keeps the first n characters
# syntax: nk: keeps the first n characters
# 'n|s'k: same + adds 's' at the end of shortened strings
# 'n|s'k: same but 's' counts in the n characters kept

words = dsbf("/short, constitutional")
dsbf("$k ? words")
dsb("'5|..'k ? words")

dsb("'5|..'k ? words")

# K: keeps the first n elements
# syntax: nK: keeps the first n elements
# 'n|s'K: same + adds the element 's' at the end
# 'n||s'K: same but 's' counts in the n elements kept
# Special values :rest: and :REST:, give the number of items dropped

bx = dsb("/Pessac Leognan, Saint Emilion, Marguaux, Saint Julien, Pauillac")

dsb("Bordeaux wines I like: .[3K, ', 'C ? bx].")

ds("Bordeaux wines I like: .[3|etc..'K, ', 'C ? bx].")

ds("Bordeaux wines I like: .[3|etc..'K, ', 'C ? bx].")

ds("Bordeaux wines I like: .[3|and at least :REST: others'K, ', 'C ? bx].")

# Ko, KO: special operator which keeps the first n elements and adds "others"
# syntax: nKo
# KO gives the rest in letters

ds("Bordeaux wines I like: .[4KO, C ? bx].")

# r, R: string replacement
# syntax: 's'R: deletes the content in 's' (replaces with the empty string)
# 's1 => s2'R replaces s1 into s2
# r: fixed / R: perl = TRUE

ds("'e'r !The letter e is deleted")

# adding a perl look-behind
dsb("'(?<! )e'R !The letter e is deleted")

ds("'e => a'r !The letter e becomes a")

ds("'('[[:alpha:]][[:alpha:]]+ => \\
\1.'R !Trimming the words")

# *, *c, **, **c: replication, replication + collapse
# syntax: n* or n*c
# ** is the same as * but uses "each" in the replication

ds("N.[10*c!o]!")

ds("3*c ? 1:3")

ds("3**c ? 1:3")
# d: replaces the items by the empty string
# -> useful in conditions

dsb("d!I am going to be annihilated")

# # ELEMENT MANIPULATION ####
#
#
# D: deletes all elements
# -> useful in conditions

x = dsb("/I'll, be, deleted")
dsb("D ? x")

# # i, I: inserts an item
# syntax: 's1|s2': inserts s1 first and s2 last
# I: is the same as i but is 'invisibly' included

characters = dsb("/David, Wilkins, Dora, Agnes, Trotwood")
dsb("'/Heep|Spenlow'i, C ? characters")
dsb("'/Heep|Spenlow'I, C ? characters")

# # PLURALIZATION ####
#
#
# There is support for pluralization
#
# # *s, *s_: adds 's' or 's' depending on the number of elements

nb = 1:5
dsb("Number.[*s, D ? nb]: .[C ? nb]"")
dsb("Number.[*s, D ? 2 ]: .[C ? 2 ]"")

# or
dsb("Number.[*s, 'A, C ? nb"]")

# # v, V: adds a verb at the beginning/end of the string
# syntax: 'verb've

# Unpopular opinion?
brand = c("Apple", "Samsung")
dsb(".[V, C ? brand] overrated.")
dsb(".[V, C ? brand[1]] overrated.")
win = dsb("/Peggotty, Agnes, Emily")
dsb("The winner.[s, v, C ? win].")
dsb("The winner.[s, v, C ? win[1]].")

# Other verbs

dsb("."[.' have'V, C ? win] won a prize.")
dsb("."[.' have'V, C ? win[1]] won a prize.")

dsb("."[.' was'V, C ? win] unable to come.")
dsb("."[.' was'V, C ? win[1]] unable to come.")

# *A: appends text depending on the length of the vector
# syntax: 's1|s2 / s3|s4'
#   if length == 1: applies 's1|s2'A
#   if length > 1: applies 's3|s4'A

win = dsb("/Barkis, Micawber, Murdstone")
dsb("The winner.[.' is /s are '*A, C ? win].")
dsb("The winner.[.' is /s are '*A, C ? win[1]].")

# # CONDITIONS ####
# # Conditions can be applied with 'if' statements.
# The syntax is 'type comp value if(true : false), with
# - type: either 'len', 'char', 'fixed' or 'regex'
# - len: number of elements in the vector
# - char: number of characters
# - fixed: fixed pattern
# - regex: regular expression pattern
# - comp: a comparator:
#   - valid for len/char: >, <, >=, <=, !=, ==
#   - valid for fixed/regex: !=, ==
# - value: a value for which the comparison is applied.
# - true: operations to be applied if true (can be void)
# - false: operations to be applied if false (can be void)

dsb("'char <= 2'if('(|) a : '[[]]a), ' + 'c ? c(1, 12, 123)"")

sentence = "This is a sentence with some longish words."
dsb("s, 'char<4'if(D), c ? sentence")
dsb("s, 'fixed == e'if(D), c ! Only words with an e are selected.")

# # ARGUMENTS FROM THE FRAME ####
# # Arguments can be evaluated from the calling frame.
# Simply use backticks instead of quotes.
dollar = 6
reason = "glory"
dsb("Why do you develop packages? For .\`dollar\`*c!$?",  
     "For money? No... For .[U,"s, c?reason]!", sep = "\n")

---

estfun.fixest  
*Extracts the scores from a fixest estimation*

**Description**

Extracts the scores from a fixest estimation.

**Usage**

```r
## S3 method for class 'fixest'
estfun(x, ...)
```

**Arguments**

- `x`  
  A `fixest` object, obtained for instance from `feols`.
- `...`  
  Not currently used.

**Value**

Returns a matrix of the same number of rows as the number of observations used for the estimation, and the same number of columns as there were variables.

**Examples**

```r
data(iris)
est = feols(Petal.Length ~ Petal.Width + Sepal.Width, iris)
head(estfun(est))
```
est_env

Estimates a fixest estimation from a fixest environment

Description

This is a function advanced users which allows to estimate any fixest estimation from a fixest environment obtained with only.env = TRUE in a fixest estimation.

Usage

est_env(env, y, X, weights, endo, inst)

Arguments

e

An environment obtained from a fixest estimation with only.env = TRUE. This is intended for advanced users so there is no error handling: any other kind of input will fail with a poor error message.

y

A vector representing the dependent variable. Should be of the same length as the number of observations in the initial estimation.

X

A matrix representing the independent variables. Should be of the same dimension as in the initial estimation.

weights

A vector of weights (i.e. with only positive values). Should be of the same length as the number of observations in the initial estimation. If identical to the scalar 1, this will mean that no weights will be used in the estimation.

endo

A matrix representing the endogenous regressors in IV estimations. It should be of the same dimension as the original endogenous regressors.

inst

A matrix representing the instruments in IV estimations. It should be of the same dimension as the original instruments.

Details

This function has been created for advanced users, mostly to avoid overheads when making simulations with fixest.

How can it help you make simulations? First make a core estimation with only.env = TRUE, and usually with only.coef = TRUE (to avoid having extra things that take time to compute). Then loop while modifying the appropriate things directly in the environment. Beware that if you make a mistake here (typically giving stuff of the wrong length), then you can make the R session crash because there is no more error-handling! Finally estimate with est_env(env = core_env) and store the results.

Instead of est_env, you could use directly fixest estimations too, like feols, since they accept the env argument. The function est_env is only here to add a bit of generality to avoid the trouble to the user to write conditions (look at the source, it’s just a one liner).

Objects of main interest in the environment are:

lhs The left hand side, or dependent variable.
The matrix of the right-hand-side, or explanatory variables.

The matrix of the endogenous variables in IV regressions.

The matrix of the instruments in IV regressions.

The vector of weights.

I strongly discourage changing the dimension of any of these elements, or else crash can occur. However, you can change their values at will (given the dimension stay the same). The only exception is the weights, which tolerates changing its dimension: it can be identical to the scalar 1 (meaning no weights), or to something of the length the number of observations.

I also discourage changing anything in the fixed-effects (even their value) since this will almost surely lead to a crash.

Note that this function is mostly useful when the overheads/estimation ratio is high. This means that OLS will benefit the most from this function. For GLM/Max.Lik. estimations, the ratio is small since the overheads is only a tiny portion of the total estimation time. Hence this function will be less useful for these models.

It returns the results of a `fixest` estimation: the one that was summoned when obtaining the environment.

Laurent Berge

# Let's make a short simulation
# Inspired from Grant McDermott bboot function
# See https://twitter.com/grant_mcdermott/status/1487528757418102787

# Simple function that computes a Bayesian bootstrap
bboot = function(x, n_sim = 100){
  # We bootstrap on the weights
  # Works with fixed-effects/IVs
  # and with any fixest function that accepts weights

  core_env = update(x, only.coef = TRUE, only.env = TRUE)
  n_obs = x$nobs

  res_all = vector("list", n_sim)
  for(i in 1:n_sim){
    # # begin: NOT RUN
    # # We could directly assign in the environment:
    # assign("weights.value", rexp(n_obs, rate = 1), core_env)
    # res_all[[i]] = est_env(env = core_env)
    # end: NOT RUN

    # Instead we can use the argument weights, which does the same
etable

\[
\text{etable} \quad \text{Estimations table (export the results of multiples estimations to a DF or to Latex)}
\]

Description

Aggregates the results of multiple estimations and displays them in the form of either a Latex table or a data.frame. Note that you will need the booktabs package for the Latex table to render properly.

Usage

etable(
  \ldots,
  vcov = NULL,
  stage = 2,
  agg = NULL,
  se = NULL,
  ssc = NULL,
  cluster = NULL,
  .vcov = NULL,
  .vcov_args = NULL,
  digits = 4,
  digits.stats = 5,
  tex,
  fitstat = NULL,
  title = NULL,
)
etable

coefstat = "se",
ci = 0.95,
se.row = NULL,
se.below = NULL,
keep = NULL,
drop = NULL,
order = NULL,
dict = TRUE,
file = NULL,
replace = FALSE,
convergence = NULL,
signif.code = NULL,
label = NULL,
float = NULL,
headers = list("auto"),
fixef_sizes = FALSE,
fixef_sizes.simplify = TRUE,
keepFactors = TRUE,
family = NULL,
powerBelow = -5,
interaction.combine = NULL,
interaction.order = NULL,
i.equal = NULL,
depvar = TRUE,
style.tex = NULL,
style.df = NULL,
notes = NULL,
group = NULL,
eextralines = NULL,
fixef.group = NULL,
placement = "htbp",
drop.section = NULL,
poly_dict = c("", " square", " cube"),
postprocess.tex = NULL,
postprocess.df = NULL,
tpt = FALSE,
arraystretch = NULL,
adjustbox = NULL,
fontsize = NULL,
fit_format = "__var__",
coef.just = NULL,
tabular = "normal",
highlight = NULL,
coef.style = NULL,
meta = NULL,
meta.time = NULL,
meta.author = NULL,
meta.sys = NULL,
etable

meta.call = NULL,
meta.comment = NULL,
view = FALSE,
export = NULL,
markdown = NULL,
page.width = "fit",
div.class = "etable"
)

setFixest_etable(
digits = 4,
digits.stats = 5,
fitstat,
coefstat = c("se", "tstat", "confint"),
ct = 0.95,
se.below = TRUE,
keep,
drop,
order,
dict,
float,
fixef_sizes = FALSE,
fixef_sizes.simplify = TRUE,
family,
powerBelow = -5,
interaction.order = NULL,
depvar,
style.tex = NULL,
style.df = NULL,
notes = NULL,
group = NULL,
extralines = NULL,
fixef.group = NULL,
placement = "htbp",
drop.section = NULL,
view = FALSE,
markdown = NULL,
view.cache = FALSE,
page.width = "fit",
postprocess.tex = NULL,
postprocess.df = NULL,
fit_format = "__var__",
meta.time = NULL,
meta.author = NULL,
meta.sys = NULL,
meta.call = NULL,
meta.comment = NULL,
reset = FALSE,
save = FALSE
)

getFixest_etable()

## S3 method for class 'etable_tex'
print(x, ...)

## S3 method for class 'etable_df'
print(x, ...)

log_etable(type = "pdflatex")
esttable(
  ..., 
  vcov = NULL, 
  stage = 2, 
  agg = NULL, 
  se = NULL, 
  ssc = NULL, 
  cluster = NULL, 
  .vcov = NULL, 
  .vcov_args = NULL, 
  digits = 4, 
  digits.stats = 5, 
  fitstat = NULL, 
  coefstat = "se", 
  ci = 0.95, 
  se.row = NULL, 
  se.below = NULL, 
  keep = NULL, 
  drop = NULL, 
  order = NULL, 
  dict = TRUE, 
  file = NULL, 
  replace = FALSE, 
  convergence = NULL, 
  signif.code = NULL, 
  headers = list("auto"), 
  fixef_sizes = FALSE, 
  fixef_sizes.simplify = TRUE, 
  keepFactors = TRUE, 
  family = NULL, 
  powerBelow = -5, 
  interaction.combine = NULL, 
  interaction.order = NULL, 
  i.equal = NULL, 
  depvar = TRUE,
estyle.df = NULL,
group = NULL,
extralines = NULL,
fixef.group = NULL,
drop.section = NULL,
poly_dict = c("", "square", "cube"),
postprocess.df = NULL,
fit_format = "\_\_var\_\_",
coef.just = NULL,
highlight = NULL,
coef.style = NULL,
export = NULL,
page.width = "fit",
div.class = "etable"

esttex(
  ..., 
  vcov = NULL,
  stage = 2,
  agg = NULL,
  se = NULL,
  ssc = NULL,
  cluster = NULL,
  .vcov = NULL,
  .vcov_args = NULL,
  digits = 4,
  digits.stats = 5,
  fitstat = NULL,
  title = NULL,
  coefstat = "se",
  ci = 0.95,
  se.row = NULL,
  se.below = NULL,
  keep = NULL,
  drop = NULL,
  order = NULL,
  dict = TRUE,
  file = NULL,
  replace = FALSE,
  convergence = NULL,
  signif.code = NULL,
  label = NULL,
  float = NULL,
  headers = list("auto"),
  fixef_sizes = FALSE,
  fixef_sizes.simplify = TRUE,
  keepFactors = TRUE,
family = NULL,
powerBelow = -5,
interaction.combine = NULL,
interaction.order = NULL,
i.equal = NULL,
depvar = TRUE,
styler = NULL,
notes = NULL,
group = NULL,
extralines = NULL,
fixef.group = NULL,
placement = "htbp",
drop.section = NULL,
poly_dict = c("", "square", "cube"),
postprocess.tex = NULL,
tpt = FALSE,
arraystretch = NULL,
adjustbox = NULL,
fontsize = NULL,
fit_format = "__var__",
tabular = "normal",
highlight = NULL,
coef.style = NULL,
meta = NULL,
meta.time = NULL,
meta.author = NULL,
meta.sys = NULL,
meta.call = NULL,
meta.comment = NULL,
view = FALSE,
export = NULL,
markdown = NULL,
page.width = "fit",
div.class = "etable"
)

Arguments

... Used to capture different fixest estimation objects (obtained with feelm, feols or feglm). Note that any other type of element is discarded. Note that you can give a list of fixest objects.

vcov Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: vcov_type ~ variables. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from vcov_cluster, vcov_NW, NW, vcov_DK, DK, vcov_conley and conley. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the vcov documentation in the
etable vignette.

stage Can be equal to 2 (default), 1, 1:2 or 2:1. Only used if the object is an IV estimation: defines the stage to which summary should be applied. If stage = 1 and there are multiple endogenous regressors or if stage is of length 2, then an object of class fixest_multi is returned.

agg A character scalar describing the variable names to be aggregated, it is pattern-based. For sunab estimations, the following keywords work: "att", "period", "cohort" and FALSE (to have full disaggregation). All variables that match the pattern will be aggregated. It must be of the form "(root)", the parentheses must be there and the resulting variable name will be "root". You can add another root with parentheses: "(root1)regex(root2)", in which case the resulting name is "root1::root2". To name the resulting variable differently you can pass a named vector: c("name" = "pattern") or c("name" = "pattern(root2)"). It's a bit intricate sorry, please see the examples.

se Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "threeway" or "fourway"? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

ssc An object of class ssc.type obtained with the function ssc. Represents how the degree of freedom correction should be done. You must use the function ssc for this argument. The arguments and defaults of the function ssc are: adj = TRUE, fixef.K = "nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact = FALSE). See the help of the function ssc for details.

cluster Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

.vcov A function to be used to compute the standard-errors of each fixest object. You can pass extra arguments to this function using the argument .vcov_args. See the example.

.vcov_args A list containing arguments to be passed to the function .vcov.

digits Integer or character scalar. Default is 4 and represents the number of significant digits to be displayed for the coefficients and standard-errors. To apply rounding instead of significance use, e.g., digits = "r3" which will round at the first 3 decimals. If character, it must be of the form "rd" or "sd" with d a digit (r is for round and s is for significance). For the number of digits for the fit statistics, use digits.stats. Note that when significance is used it does not exactly display the number of significant digits: see details for its exact meaning.
digits.stats
Integer or character scalar. Default is 5 and represents the number of significant digits to be displayed for the fit statistics. To apply rounding instead of significance use, e.g., digits = "r3" which will round at the first 3 decimals. If character, it must be of the form "rd" or "sd" with d a digit (r is for round and s is for significance). Note that when significance is used it does not exactly display the number of significant digits: see details for its exact meaning.

tex
Logical: whether the results should be a data.frame or a Latex table. By default, this argument is TRUE if the argument file (used for exportation) is not missing; it is equal to FALSE otherwise.

fitstat
A character vector or a one sided formula (both with only lowercase letters).
A vector listing which fit statistics to display. The valid types are 'n', 'll', 'aic', 'bic' and r2 types like 'r2', 'pr2', 'war2', etc (see all valid types in \texttt{r2}). Also accepts valid types from the function \texttt{fitstat}. The default value depends on the models to display. Example of use: \texttt{fitstat=c('n', 'cor2', 'ar2', 'war2')}, or \texttt{fitstat=~n+cor2+ar2+war2} using a formula. You can use the dot to refer to default values: \texttt{~ . + ll} would add the log-likelihood to the default fit statistics.

title
(Tex only.) Character scalar. The title of the Latex table.

coeffstat
One of "se" (default), "tstat" or "confint". The statistic to report for each coefficient: the standard-error, the t-statistics or the confidence interval. You can adjust the confidence interval with the argument \texttt{ci}.

\texttt{ci}
Level of the confidence interval, defaults to 0.95. Only used if \texttt{coeffstat} = \texttt{confint}.

se.row
Logical scalar, default is \texttt{NULL}. Whether should be displayed the row with the type of standard-error for each model. When \texttt{tex = FALSE}, the default is \texttt{TRUE}. When \texttt{tex = FALSE}, the row is showed only when there is a table-footer and the types of standard-errors differ across models.

se.below
Logical or \texttt{NULL} (default). Should the standard-errors be displayed below the coefficients? If \texttt{NULL}, then this is \texttt{TRUE} for Latex and \texttt{FALSE} otherwise.

\texttt{keep}
Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see \texttt{base::regex} help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument \texttt{dict}). Example: you have the variable \texttt{x1} to \texttt{x55} and want to display only \texttt{x1} to \texttt{x9}, then you could use \texttt{keep = x\[[[:digit:]]\]$"}. If the first character is an exclamation mark, the effect is reversed (e.g. \texttt{keep = "!Intercept"} means: every variable that does not contain "Intercept" is kept). See details.

\texttt{drop}
Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see \texttt{base::regex} help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument \texttt{dict}). Example: you have the variable \texttt{x1} to \texttt{x55} and want to display only \texttt{x1} to \texttt{x9}, then you could use \texttt{drop = x\[[[:digit:]]\][2]$". If the first character is an exclamation mark, the effect is reversed (e.g. \texttt{drop = "!Intercept"} means: every variable that does not contain "Intercept" is dropped). See details.
**etable**

**order** Character vector. This element is used if the user wants the variables to be ordered in a certain way. This should be a vector of regular expressions (see `base::regex` help for more info). The variables satisfying the first regular expression will be placed first, then the order follows the sequence of regular expressions. This argument is applied post aliasing (see argument `dict`). Example: you have the following variables: `month1` to `month6`, then `x1` to `x5`, then `year1` to `year6`. If you want to display first the `x`'s, then the years, then the months you could use: `order = c("x", "year")`. If the first character is an exclamation mark, the effect is reversed (e.g. `order = "!Intercept"` means: every variable that does not contain “Intercept” goes first). See details.

**dict** A named character vector or a logical scalar. It changes the original variable names to the ones contained in the dictionary. E.g. to change the variables named `a` and `b3` to (resp.) “$\log(a)$” and to “$\text{bonus}^3$”, use `dict=c(a="$\log(a)$",b3="$\text{bonus}^3$")`. By default, it is equal to `getFixest_dict()`, a default dictionary which can be set with `setFixest_dict`. You can use `dict = FALSE` to disable it. By default `dict` modifies the entries in the global dictionary, to disable this behavior, use "reset" as the first element (ex: `dict=c("reset", mpg="\text{Miles per gallon}\)").

**file** A character scalar. If provided, the LaTeX (or data frame) table will be saved in a file whose path is `file`. If you provide this argument, then a LaTeX table will be exported, to export a regular data frame, use argument `tex = FALSE`.

**replace** Logical, default is FALSE. Only used if option `file` is used. Should the exported table be written in a new file that replaces any existing file?

**convergence** Logical, default is missing. Should the convergence state of the algorithm be displayed? By default, convergence information is displayed if at least one model did not converge.

**signif.code** Named numeric vector, used to provide the significance codes with respect to the p-value of the coefficients. Default is c("***"=0.01, "**"=0.05, "*"=0.10) for a LaTeX table and c("***"=0.001, "**"=0.01, "*"=0.05, "."=0.10) for a data.frame (to conform with R's default). To suppress the significance codes, use `signif.code=NA` or `signif.code=NULL`. Can also be equal to "letters", then the default becomes c("a"=0.01, "b"=0.05, "c"=0.10).

**label** (Tex only.) Character scalar. The label of the LaTeX table.

**float** (Tex only.) Logical. By default, if the argument `title` or `label` is provided, it is set to TRUE. Otherwise, it is set to FALSE.

**headers** Character vector or list. Adds one or more header lines in the table. A header line can be represented by a character vector or a named list of numbers where the names are the cell values and the numbers are the span. Example: `headers=list("M"=2, "F"=3)` will create a row with 2 times "M" and three time "F" (this is identical to `headers=rep(c("M", "F"), c(2, 3))`). You can stack header lines within a list, in that case the list names will be displayed in the leftmost cell. Example: `headers=list(Gender=list("M"=2, "F"=3), Country="US")` will create two header lines. When `tex = TRUE`, you can add a rule to separate groups by using "\_\_\_" somewhere in the row name (ex: `headers=list("\_\_\_Gender"=list("M"=2, "F"=3))`). You can monitor the placement by inserting a special character in the row name: "^" means at the top, "-" means in the middle (default) and "_" means at the bottom. Example: `headers=list("\_\_\_Country"="US")` will add the country row as the very last header row (after the model row). Finally, you can use
the special value "auto" to include automatic headers when the data contains
split sample estimations. By default it is equal to list("auto"). You can use
(). instead of list().

fixef_sizes  (Tex only.) Logical, default is FALSE. If TRUE and fixed-effects were used in the
models, then the number of "units" per fixed-effect dimension is also displayed.

fixef_sizes.simplify  Logical, default is TRUE. Only used if fixef_sizes = TRUE. If TRUE, the fixed-
effects sizes will be displayed in parentheses instead of in a separate line if there
is no ambiguity (i.e. if the size is constant across models).

keepFactors  Logical, default is TRUE. If FALSE, then factor variables are displayed as fixed-
effects and no coefficient is shown.

family  Logical, default is missing. Whether to display the families of the models. By
default this line is displayed when at least two models are from different fami-
lies.

powerBelow  (Tex only.) Integer, default is -5. A coefficient whose value is below $10^{\text{powerBelow}+1}$
is written with a power in Latex. For example 0.0000456 would be written
4.56$\times 10^{-5}$ by default. Setting powerBelow = -6 would lead to
0.00004 in Latex.

interaction.combine  Character scalar, defaults to "$\times$" for Tex and to " = " otherwise.
When the estimation contains interactions, then the variables names (after alias-
ing) are combined with this argument. For example: if dict = c(x1="Wind",
x2="Rain") and you have the following interaction x1:x2, then it will be re-
named (by default) Wind $\times$ Rain – using interaction.combine = 
"*" would lead to Wind*Rain.

interaction.order  Character vector of regular expressions. Only affects variables that are inter-
acted like x1 and x2 in feols(y ~ x1*x2, data). You can change the order in
which the interacted variables are displayed: e.g. interaction.order = "x2"
would lead to "x1 x x2" instead of "x1 x x2". Please look at the argument 'order'
and the dedicated section in the help page for more information.

i.equal  Character scalar, defaults to "$=$" when tex = TRUE and to " = " otherwise. Only
affects factor variables created with the function i, tells how the variable should
be linked to its value. For example if you have the Species factor from the
iris data set, by default the display of the variable is Species = Setosa, etc. If
i.equal = ": " the display becomes Species: Setosa.

depvar  Logical, default is TRUE. Whether a first line containing the dependent variables
should be shown.

style.tex  An object created by the function style.tex. It represents the style of the Latex
table, see the documentation of style.tex.

style.df  An object created by the function style.df. It represents the style of the data
frame returned (if tex = FALSE), see the documentation of style.df.

notes  (Tex only.) Character vector. If provided, a "notes" section will be added at
the end right after the end of the table, containing the text of this argument. If
it is a vector, it will be collapsed with new lines. If tpt = TRUE, the behavior is
etable

different: each element of the vector is an item. If the first element of the vector starts with "@", then it will be included verbatim, and in case of tpt = TRUE, right before the first item. If that element is provided, it will replace the value defined in style.tex(notes.intro) or style.tex(notes.tpt.intro).

group

A list. The list elements should be vectors of regular expressions. For each elements of this list: A new line in the table is created, all variables that are matched by the regular expressions are discarded (same effect as the argument drop) and TRUE or FALSE will appear in the model cell, depending on whether some of the previous variables were found in the model. Example: group=list("Controls: personal traits"=c("gender", "height", "weight")) will create a new line with "Controls: personal traits" in the leftmost column, all three variables gender, height and weight are discarded, TRUE appearing in each model containing at least one of the three variables (the style of TRUE/FALSE is governed by the argument yesNo). You can control the placement of the new row by using 1 or 2 special characters at the start of the row name. The meaning of these special characters are: 1) "^": coef., "-": fixed-effect, "_": stats, section; 2) "^": 1st, "_": last, row. For example: group=list("^Controls"=stuff) will place the line at the top of the 'stats' section, and using group=list("^Controls"=stuff) will make the row appear at the bottom of the coefficients section. For details, see the dedicated section.

extralines

A vector, a list or a one sided formula. The list elements should be either a vector representing the value of each cell, a list of the form list("item1" = #item1, "item2" = #item2, etc) or a function. This argument can be many things, please have a look at the dedicated help section; a simplified description follows. For each elements of this list: A new line in the table is created, the list name being the row name and the vector being the content of the cells. For example: extralines=list("Sub-sample"=c("<20 yo", "all", ">50 yo")) will create an new line with "Sub-sample" in the leftmost cell, the vector filling the content of the cells for the three models. You can control the placement of the new row by using 1 or 2 special characters at the start of the row name. The meaning of these special characters are: 1) "^": coef., "-": fixed-effect, "_": stats, section; 2) "^": 1st, "_": last, row. For example: extralines=list("^Controls"=stuff) will place the line at the bottom of the stats section, and using extralines=list("^Controls"=stuff) will make the row appear at the top of the 'coefficients' section. For details, see the dedicated section. You can use \(\) instead of list().

fixef.group

Logical scalar or list (default is NULL). If equal to TRUE, then all fixed-effects always appearing jointly in models will be grouped in one row. If a list, its elements must be character vectors of regular expressions and the list names will be the row names. For ex. fixef.group=list(\"Dates fixed-effects\"="Month|Day\") will remove the "Month" and "Day" fixed effects from the display and replace them with a single row named "Dates fixed-effects". You can monitor the placement of the new row with two special characters telling where to place the row within a section: first in which section it should appear: "^" (coef.), "-" (fixed-effects), or "_" (stat.) section; then whether the row should be "^" (first), or "_" (last). These two special characters must appear first in the row names. Please see the dedicated section.

placement

(Tex only.) Character string giving the position of the float in Latex. Default is "htbp". It must consist of only the characters 'h', 't', 'b', 'p', 'H' and '!"
Reminder: h: here; t: top; b: bottom; p: float page; H: definitely here; !: prevents Latex to look for other positions. Note that it can be equal to the empty string (and you’ll get the default placement).

**drop.section**

Character vector which can be of length 0 (i.e. equal to NULL). Can contain the values "coef", "fixef", "slopes" or "stats". It would drop, respectively, the coefficients section, fixed-effects section, the variables with varying slopes section or the fit statistics section.

**poly_dict**

Character vector, default is c("", " square", " cube"). When raw polynomials (x^2, etc) are used, the variables are automatically renamed and poly_dict rules the display of the power. For powers greater than the number of elements of the vector, the value displayed is \(^{\text{pow}}\) in Latex and ^ pow in the R console.

**postprocess.tex**

A function that will postprocess the character vector defining the latex table. Only when tex = TRUE. By default it is equal to NULL, meaning that there is no postprocessing. When tex = FALSE, see the argument postprocess.df. See details.

**postprocess.df**

A function that will postprocess.tex the resulting data.frame. Only when tex = FALSE. By default it is equal to NULL, meaning that there is no postprocessing. When tex = TRUE, see the argument postprocess.tex.

**tpt**

(Tex only.) Logical scalar, default is FALSE. Whether to use the threeparttable environment. If so, the notes will be integrated into the tablenotes environment.

**arraystretch**

(Tex only.) A numeric scalar, default is NULL. If provided, the command \renewcommand*{\arraystretch{x}} is inserted, replacing x by the value of arraystretch. The changes are specific to the current table and do not affect the rest of the document.

**adjustbox**

(Tex only.) A logical, numeric or character scalar, default is NULL. If not NULL, the table is inserted within the adjustbox environment. By default the options are width = 1\textwidth, center (if TRUE). A numeric value changes the value before \textwidth. You can also add a character of the form "x tw" or "x th" with x a number and where tw (th) stands for text-width (text-height). Finally any other character value is passed verbatim as an adjustbox option.

**fontsize**

(Tex only.) A character scalar, default is NULL. Can be equal to tiny, scriptsize, footnotesize, small, normalsize, large, or Large. The change affect the table only (and not the rest of the document).

**fit_format**

Character scalar, default is "__var__". Only used in the presence of IVs. By default the endogenous regressors are named fit_varname in the second stage. The format of the endogenous regressor to appear in the table is governed by fit_format. For instance, by default, the prefix "fit_" is removed, leading to only varname to appear. If fit_format = "$\hat{__var__}$", then "$\hat{\text{varname}}$" will appear in the table.

**coef.just**

(DF only.) Either ".", ",", "1", "c" or "r", default is NULL. How the coefficients should be justified. If NULL then they are right aligned if se.below = FALSE and aligned to the dot if se.below = TRUE. The keywords stand respectively for dot-, parenthesis-, left-, center- and right-aligned.
**tabular**  (Tex only.) Character scalar equal to "normal" (default), "*" or "X". Represents the type of tabular environment to use: either tabular, tabular* or tabularx.

**highlight**  List containing coefficients to highlight. Highlighting is of the form .("options1" = "coefs1", "options2" = "coefs2", etc). The coefficients to be highlighted can be written in three forms: 1) row, eg "x1" will highlight the full row of the variable x1; 2) cells, use '0' after the coefficient name to give the column, it accepts ranges, eg "x1@2, 4-6, 8" will highlight only the columns 2, 4, 5, 6, and 8 of the variable x1; 3) range, by giving the top-left and bottom-right values separated with a semi-colon, eg "x1@2 ; x3@5" will highlight from the column 2 of x1 to the 5th column of x3. Coefficient names are partially matched, use a '%' first to refer to the original name (before dictionary) and use '0' first to use a regular expression. You can add a vector of row/cell/range. The options are a comma-separated list of items. By default the highlighting is done with a frame (a thick box) around the coefficient, use 'rowcol' to highlight with a row color instead. Here are the other options: 'se' to highlight the standard-errors too; 'square' to have a square box (instead of rounded); 'thick1' to 'thick6' to monitor the width of the box; 'sep0' to 'sep9' to monitor the inner spacing. Finally the remaining option is the color: simply add an R color (it must be a valid R color!). You can use "color!alpha" with "alpha" a number between 0 to 100 to change the alpha channel of the color.

**coef.style**  Named list containing styles to be applied to the coefficients. It must be of the form .("style1" = "coefs1", "style2" = "coefs2", etc). The style must contain the string ":coef:" (or ":coef_se:" to style both the coefficient and its standard-error). The string :coef: will be replaced verbatim by the coefficient value. For example use "\\textbf{:coef:}" to put the coefficient in bold. Note that markdown markup is enabled so "**:coef:**" would also put it in bold. The coefficients to be styled can be written in three forms: 1) row, eg "x1" will style the full row of the variable x1; 2) cells, use '0' after the coefficient name to give the column, it accepts ranges, eg "x1@2, 4-6, 8" will style only the columns 2, 4, 5, 6, and 8 of the variable x1; 3) range, by giving the top-left and bottom-right values separated with a semi-colon, eg "x1@2 ; x3@5" will style from the column 2 of x1 to the 5th column of x3. Coefficient names are partially matched, use a '%' first to refer to the original name (before dictionary) and use '0' first to use a regular expression. You can add a vector of row/cell/range.

**meta**  (Tex only.) A one-sided formula that shall contain the following elements: date or time, sys, author, comment and call. Default is NULL. This argument is a short-cut to controlling the meta information that can be displayed in comments before the table. Typically if the element is in the formula, it means that the argument will be equal to TRUE. Example: meta = ~time+call is equivalent to meta.time = TRUE and meta.call = TRUE. The "author" and "comment" elements are a bit special. Using meta = ~author("Mark") is equivalent to meta.author = "Mark" while meta=~author is equiv. to meta.author = TRUE. The "comment" must be used with a character string inside: meta = ~comment("this is a comment"). The order in the formula controls the order of appearance of the meta elements. It also has precedence over the meta.XX arguments.

**meta.time**  (Tex only.) Either a logical scalar (default is FALSE) or "time" or "date". Whether to include the time (if TRUE or "time") or the date (if "date") of creation of the
etable

meta.author (Tex only.) A logical scalar (default is FALSE) or a character vector. If TRUE then the identity of the author (deduced from the system user in Sys.info()) is inserted in a comment right before the table. If a character vector, then it should contain author names that will be inserted as comments before the table, prefixed with "Created by: ". For free-form comments see the argument meta.comment.

meta.sys (Tex only.) A logical scalar, default is FALSE. Whether to include system information (from Sys.info()) in a comment right before the table.

meta.call (Tex only.) Logical scalar, default is FALSE. If TRUE then the call to the function is inserted right before the table in a comment.

meta.comment (Tex only.) A character vector containing free-form comments to be inserted right before the table.

view Logical, default is FALSE. If TRUE, then the table generated in Latex by etable and then is displayed in the viewer pane. Note that for this option to work you need i) pdflatex, ii) imagemagick and iii) ghostscript. All three software must be installed and on the path.

export Character scalar giving the path to a PNG file to be created, default is NULL. If provided, the Latex table will be converted to PNG and copied to the export location. Note that for this option to work you need a working distribution of pdflatex, imagemagick and ghostscript.

markdown Character scalar giving the location of a directory, or a logical scalar. Default is NULL. This argument only works in Rmarkdown documents, when knitting the document. If provided: two behaviors depending on context. A) if the output document is Latex, the table is exported in Latex. B) if the output document is not Latex, the table will be exported to PNG at the desired location and inserted in the document via a markdown link. If equal to TRUE, the default location of the PNGs is a temporary folder for R > 4.0.0, or to "images/etable/" for earlier versions.

page.width Character scalar equal to 'fit' (default), 'a4' or 'us'; or a single Latex measure (like '17cm') or a double one (like "21, 2cm"). Only used when the Latex table is to be viewed (view = TRUE), exported (export != NULL) or displayed in Rmarkdown (markdown != NULL). It represents the text width of the page in which the Latex table will be inserted. By default, 'fit', the page fits exactly the table (i.e. text width = table width). If 'a4' or 'us', two times 2cm is removed from the page width to account for margins. Providing a page width and a margin width, like in "17in, 1in", enables a correct display of the argument adjustbox. Note that the margin width represent the width of a single side margin (and hence will be doubled).

div.class Character scalar, default is "etable". Only used in Rmarkdown documents when markdown = TRUE. The table in an image format is embedded in a <div> container, and that container is of class div.class.

view.cache Logical, default is FALSE. Only used when view = TRUE. Whether the PNGs of the tables should be cached.

reset (setFixest_etable only.) Logical, default is FALSE. If TRUE, this will reset all the default values that were already set by the user in previous calls.
Either a logical or equal to "reset". Default is FALSE. If TRUE then the value is set permanently at the project level, this means that if you restart R, you will still obtain the previously saved defaults. This is done by writing in the ".Renviron" file, located in the project's working directory, hence we must have write permission there for this to work, and only works with Rstudio. If equal to "reset", the default at the project level is erased. Since there is writing in a file involved, permission is asked to the user.

An object returned by etable.

digits

Character scalar equal to 'pdflatex' (default), 'magick', 'dir' or 'tex'. Which log file to report; if 'tex', the full source code of the tex file is returned, if 'dir': the directory of the log files is returned.

Details

The function esttex is equivalent to the function etable with argument tex = TRUE.

The function esttable is equivalent to the function etable with argument tex = FALSE.

To display the table, you will need the Latex package booktabs which contains the \toprule, \midrule and \bottomrule commands.

You can permanently change the way your table looks in Latex by using setFixest_etable. The following vignette gives an example as well as illustrates how to use the style and postprocessing functions: Exporting estimation tables.

When the argument postprocessing.tex is not missing, two additional tags will be included in the character vector returned by etable: "%start:tab\n" and "%end:tab\n". These can be used to identify the start and end of the tabular and are useful to insert code within the table environment.

Value

If tex = TRUE, the lines composing the Latex table are returned invisibly while the table is directly prompted on the console.

If tex = FALSE, the data.frame is directly returned. If the argument file is not missing, the data.frame is printed and returned invisibly.

Functions

- esttable(): Exports the results of multiple fixest estimations in a Latex table.
- esttex(): Exports the results of multiple fixest estimations in a Latex table.

How does digits handle the number of decimals displayed?

The default display of decimals is the outcome of an algorithm. Let's take the example of digits = 3 which "kind of" requires 3 significant digits to be displayed.

For numbers greater than 1 (in absolute terms), their integral part is always displayed and the number of decimals shown is equal to digits minus the number of digits in the integral part. This means that 12.345 will be displayed as 12.3. If the number of decimals should be 0, then a single decimal is displayed to suggest that the number is not whole. This means that 1234.56 will be displayed as 1234.5. Note that if the number is whole, no decimals are shown.
For numbers lower than 1 (in absolute terms), the number of decimals displayed is equal to digits except if there are only 0s in which case the first significant digit is shown. This means that 0.01234 will be displayed as 0.012 (first rule), and that 0.000123 will be displayed as 0.0001 (second rule).

**Arguments keep, drop and order**

The arguments `keep`, `drop` and `order` use regular expressions. If you are not aware of regular expressions, I urge you to learn it, since it is an extremely powerful way to manipulate character strings (and it exists across most programming languages).

For example `drop = "Wind"` would drop any variable whose name contains "Wind". Note that variables such as "Temp:Wind" or "StrongWind" do contain "Wind", so would be dropped. To drop only the variable named "Wind", you need to use `drop = "^Wind\$"` (with "^" meaning beginning, resp. "$" meaning end, of the string => this is the language of regular expressions).

Although you can combine several regular expressions in a single character string using pipes, `drop` also accepts a vector of regular expressions.

You can use the special character "!" (exclamation mark) to reverse the effect of the regular expression (this feature is specific to this function). For example `drop = "!Wind"` would drop any variable that does not contain "Wind".

You can use the special character "%" (percentage) to make reference to the original variable name instead of the aliased name. For example, you have a variable named "Month6", and use a dictionary `dict = c(Month6="June")`. Thus the variable will be displayed as "June". If you want to delete that variable, you can use either `drop="June"`, or `drop="^Month6"` (which makes reference to its original name).

The argument `order` takes in a vector of regular expressions, the order will follow the elements of this vector. The vector gives a list of priorities, on the left the elements with highest priority. For example, `order = c("Wind", "!Inter", "!Temp")` would give highest priorities to the variables containing "Wind" (which would then appear first), second highest priority is the variables not containing "Inter", last, with lowest priority, the variables not containing "Temp". If you had the following variables: (Intercept), Temp:Wind, Wind, Temp you would end up with the following order: Wind, Temp:Wind, Wind, Temp (Intercept).

**The argument extralines**

The argument `extralines` adds well... extra lines to the table. It accepts either a list, or a one-sided formula.

For each line, you can define the values taken by each cell using 4 different ways: a) a vector, b) a list, c) a function, and d) a formula.

If a vector, it should represent the values taken by each cell. Note that if the length of the vector is smaller than the number of models, its values are recycled across models, but the length of the vector is required to be a divisor of the number of models.

If a list, it should be of the form `list("item1" = #item1, "item2" = #item2, etc)`. For example `list("A"=2, "B"=3)` leads to `c("A", "A", "B", "B", "B")`. Note that if the number of items is 1, you don’t need to add = 1. For example `list("A"=2, "B")` is valid and leads to `c("A", "A", "B")`. As for the vector the values are recycled if necessary.

If a function, it will be applied to each model and should return a scalar (NA values returned are accepted).
If a formula, it must be one-sided and the elements in the formula must represent either `extralines` macros, either fit statistics (i.e., valid types of the function `fitstat`). One new line will be added for each element of the formula. To register `extralines` macros, you must first register them in `extralines_register`.

Finally, you can combine as many lines as wished by nesting them in a list. The names of the nesting list are the row titles (values in the leftmost cell). For example `extralines = list(~r2, Controls = TRUE, Group = list("A"=2, "B"))` will add three lines, the titles of which are "R2", "Controls" and "Group".

### Controlling the placement of extra lines

The arguments `group`, `extralines` and `fixef.group` allow to add customized lines in the table. They can be defined via a list where the list name will be the row name. By default, the placement of the extra line is right after the coefficients (except for `fixef.group`, covered in the last paragraph). For instance, `group = list("Controls" = "x[[:digit:]]")` will create a line right after the coefficients telling which models contain the control variables.

But the placement can be customized. The previous example (of the controls) will be used for illustration (the mechanism for `extralines` and `fixef.group` is identical).

The row names accept 2 special characters at the very start. The first character tells in which section the line should appear: it can be equal to "^", "-", or "_", meaning respectively the coefficients, the fixed-effects and the statistics section (which typically appear at the top, mid and bottom of the table). The second one governs the placement of the new line within the section: it can be equal to "^", meaning first line, or "_", meaning last line.

Let's have some examples. Using the previous example, writing "^-Controls" would place the new line at the top of the statistics section. Writing "-_Controls" places it as the last row of the fixed-effects section; "^^Controls" at the top row of the coefficients section; etc...

The second character is optional, the default placement being in the bottom. This means that "^-Controls" would place it at the bottom of the statistics section.

The placement in `fixef.group` is defined similarly, only the default placement is different. Its default placement is at the top of the fixed-effects section.

### Escaping special Latex characters

By default on all instances (with the notable exception of the elements of `style.tex`) special Latex characters are escaped. This means that `title="Exports in million \$."` will be exported as "Exports in million \\$.": the dollar sign will be escaped. This is true for the following characters: &,$,%,_," and #.

Note, importantly, that equations are NOT escaped. This means that `title="Functional form \$a_i \times x^b\$, variation in \%."` will be displayed as: "Functional form $a_i \times x^b\$, variation in \%.": only the last percentage will be escaped.

If for some reason you don’t want the escaping to take place, the arguments `headers` and `extralines` are the only ones allowing that. To disable escaping, add the special token ":tex:" in the row names. Example: in `headers=list(":tex:Row title="/ weird &\ 
	\ tex stuff\"\")`, the elements will be displayed verbatim. Of course, since it can easily ruin your table, it is only recommended to super users.
Markdown markup

Within anything that is Latex-escaped (see previous section), you can use a markdown-style markup to put the text in italic and/or bold. Use *text*, **text** or ***text*** to put some text in, respectively, italic (with \textit), bold (with \textbf) and italic-bold.

The markup can be escaped by using an backslash first. For example "***This: \***, are three stars***" will leave the three stars in the middle untouched.

Author(s)

Laurent Berge

See Also

See also the main estimation functions femlm, feols or feglm. Use summary.fixest to see the results with the appropriate standard-errors, fixef.fixest to extract the fixed-effects coefficients.

Examples

```r
est1 = feols(Ozone ~ i(Month) / Wind + Temp, data = airquality)
est2 = feols(Ozone ~ i(Month, Wind) + Temp | Month, data = airquality)

# Displaying the two results in a single table
etable(est1, est2)

# keep/drop: keeping only interactions
etable(est1, est2, keep = " x ")
# or using drop (see regexp help):
etable(est1, est2, drop = "^{\text{Month}|\text{Temp}}\)

# keep/drop: dropping interactions
etable(est1, est2, drop = " x ")
# or using keep ("!" reverses the effect):
etable(est1, est2, keep = "! x ")

# order: Wind variable first, intercept last (note the "!" to reverse the effect)
etable(est1, est2, order = c("Wind", "!Inter"))
# Month, then interactions, then the rest
etable(est1, est2, order = c("Month", " x "))

#
# dict
#
# You can rename variables with dict = c(var1 = alias1, var2 = alias2, etc)
# You can also rename values taken by factors.
# Here's a full example:
dict = c(Temp = "Temperature", "Month::5" = "May", 6 = "Jun")
etable(est1, est2, dict = dict)
# Note the difference of treatment between Jun and May
```
# Assume the following dictionary:
dict = c("Month::5"="May", "Month::6"="Jun", "Month::7"="Jul",
          "Month::8"="Aug", "Month::9"="Sep")

# We would like to keep only the Months, but now the names are all changed...
# How to do?
# We can use the special character '% Month' to make reference to the original names.
etable(est1, est2, dict = dict, keep = "%Month")

# signif.code
etable(est1, est2, signif.code = c(" A"=0.01, " B"=0.05, " C"=0.1, " D"=0.15, " F"=1))

# Using the argument style to customize Latex exports

# If you don't like the default layout of the table, no worries!
# You can modify many parameters with the argument style

# To drop the headers before each section, use:
# Note that a space adds an extra line
style_noHeaders = style.tex(var.title = "", fixef.title = "", stats.title = " ")
etable(est1, est2, dict = dict, tex = TRUE, style.tex = style_noHeaders)

# To change the lines of the table + dropping the table footer
style_lines = style.tex(line.top = \"\toprule\", line.bottom = \"\bottomrule",
                         tablefoot = FALSE)
etable(est1, est2, dict = dict, tex = TRUE, style.tex = style_lines)

# Or you have the predefined type "aer"
etable(est1, est2, dict = dict, tex = TRUE, style.tex = style.tex("aer"))

# Group and extralines

# Sometimes it's useful to group control variables into a single line
# You can achieve that with the group argument
setFixest_fml(..ctrl = ~ poly(Wind, 2) + poly(Temp, 2))
est_c0 = feols(Ozone ~ Solar.R, data = airquality)
est_c1 = feols(Ozone ~ Solar.R + ..ctrl, data = airquality)
est_c2 = feols(Ozone ~ Solar.R + Solar.R^2 + ..ctrl, data = airquality)
etable(est_c0, est_c1, est_c2, group = list(Controls = "poly"))

# 'group' here does the same as drop = "poly", but adds an extra line
# with TRUE/FALSE where the variables were found
etable est_all, est_sub1, est_sub2, est_sub3,
   extralines = list("Sub-sample" = c("All", "May-June", "Jul.-Aug.", "Sept."))

# You can monitor the placement of the new lines with two special characters
# at the beginning of the row name.
# 1) "^", "-" or "_" which mean the coefficients, the fixed-effects or the
# statistics section.
# 2) "^" or "_" which mean first or last line of the section
#
# Ex: starting with "_" will place the line at the top of the stat. section
# starting with "_" will place the line at the bottom of the FEs section
#
# You can use a single character which will represent the section,
# the line would then appear at the bottom of the section.

# Examples
etable(est_c0, est_c1, est_c2, group = list("_Controls" = "poly"))
etable(est_all, est_sub1, est_sub2, est_sub3,
   extralines = list("Sub-sample" = c("All", "May-June", "Jul.-Aug.", "Sept.")))

# headers
#
# You can add header lines with 'headers'
# These lines will appear at the top of the table

# first, 3 estimations
est_header = feols(c(Ozone, Solar.R, Wind) ~ poly(Temp, 2), airquality)

# header => vector: adds a line w/t title
etable(est_header, headers = c("A", "A", "B"))

# header => list: identical way to do the previous header
# The form is: list(item1 = #item1, item2 = #item2, etc)
etable(est_header, headers = list("A" = 2, "B" = 1))

# Adding a title +
# when an element is to be repeated only once, you can avoid the "= 1":
etable(est_header, headers = list(Group = list("A" = 2, "B")))
# To change the placement, add as first character:
# - "^" => top
# - "-" => mid (default)
# - "_" => bottom
# Note that "mid" and "top" are only distinguished when tex = TRUE

# Placing the new header line at the bottom
etable(est_header, headers = list("_Group" = c("A", "A", "B"),
                               "^Currency" = list("US $" = 2, "CA $" = 1)))

# In Latex, you can add "grouped underlines" (cmidrule from the booktabs package)
# by adding "_::_" in the title:
etable(est_header, tex = TRUE,
       headers = list("^:_:Group" = c("A", "A", "B")))

# extralines and headers: .() for list()
# In the two arguments extralines and headers, .() can be used for list()
# For example:
etable(est_header, headers = .("^Currency" = .("US $" = 2, "CA $" = 1)))

# fixef.group
# You can group the fixed-effects line with fixef.group
est_0fe = feols(Ozone ~ Solar.R + Temp + Wind, airquality)
est_1fe = feols(Ozone ~ Solar.R + Temp + Wind | Month, airquality)
est_2fe = feols(Ozone ~ Solar.R + Temp + Wind | Month + Day, airquality)

# A) automatic way => simply use fixef.group = TRUE
etable(est_0fe, est_2fe, fixef.group = TRUE)
# Note that when grouping would lead to inconsistencies across models,
# it is avoided
etable(est_0fe, est_1fe, est_2fe, fixef.group = TRUE)

# B) customized way => use a list
etable(est_0fe, est_2fe, fixef.group = list("Dates" = "Month\Day"))
# Note that when a user grouping would lead to inconsistencies,
# the term partial replaces yes/no and the fixed-effects are not removed.
etable(est_0fe, est_1fe, est_2fe, fixef.group = list("Dates" = "Month|Day"))

# Using customized placement => as with 'group' and 'extralines',
# the user can control the placement of the new line.
# See the previous 'group' examples and the dedicated section in the help.

# On top of the coefficients:
etable(est_0fe, est_2fe, fixef.group = list("^Dates" = "Month|Day"))

# Last line of the statistics
etable(est_0fe, est_2fe, fixef.group = list("_Dates" = "Month|Day"))

#
# Using custom functions to compute the standard errors
#

# You can use external functions to compute the VCOVs
# by feeding functions in the 'vcov' argument.
# Let's use some covariances from the sandwich package
etable(est_c0, est_c1, est_c2, vcov = sandwich::vcovHC)

# To add extra arguments to vcovHC, you need to write your wrapper:
etable(est_c0, est_c1, est_c2, vcov = function(x) sandwich::vcovHC(x, type = "HC0"))

#
# Customize which fit statistic to display
#
# You can change the fit statistics with the argument fitstat
# and you can rename them with the dictionary
etable(est1, est2, fitstat = ~ r2 + n + G)

# If you use a formula, '.' means the default:
etable(est1, est2, fitstat = ~ ll + .)

#
# Computing a different SE for each model
#
est = feols(Ozone ~ Solar.R + Wind + Temp, data = airquality)

# Method 1: use summary
s1 = summary(est, "iid")
s2 = summary(est, cluster = ~ Month)
s3 = summary(est, cluster = ~ Day)
s4 = summary(est, cluster = ~ Day + Month)
etable(list(s1, s2, s3, s4))

# Method 2: using a list in the argument 'vcov'

est_bis = feols(Ozone ~ Solar.R + Wind + Temp | Month, data = airquality)
etable(est, est_bis, vcov = list("hetero", ~ Month))

# When you have only one model, this model is replicated along the elements of the vcov list.
etable(est, vcov = list("hetero", ~ Month))

# Method 3: Using "each" or "times" in vcov

# If the first element of the list in 'vcov' is "each" or "times", then all models will be replicated and all the VCOVs will be applied to each model. The order in which they are replicated are governed by the each/times keywords.

each
etable(est, est_bis, vcov = list("each", "iid", ~ Month, ~ Day))

times
etable(est, est_bis, vcov = list("times", "iid", ~ Month, ~ Day))

# Notes and markup

# Notes can be also be set in a dictionary
# You can use markdown markup to put text into italic/bold
dict = c("note 1" = "*Notes:* This data is not really random.",
        "source 1" = "**Source:** the internet?"
)
est = feols(Ozone ~ csw(Solar.R, Wind, Temp), data = airquality)
etable(est, dict = dict, tex = TRUE, notes = c("note 1", "source 1"))
Description

This function is used to create extralines (which is an argument of `etable`) macros that can be easily summoned in `etable`.

Usage

`extralines_register(type, fun, alias)`

Arguments

- **type**: A character scalar giving the type-name.
- **fun**: A function to be applied to a fixest estimation. It must return a scalar.
- **alias**: A character scalar. This is the alias to be used in lieu of the type name to form the row name.

Details

You can register as many macros as you wish, the only constraint is that the type name should not conflict with a `fitstat` type name.

Examples

```r
# We register a function computing the standard-deviation of the dependent variable
my_fun = function(x) sd(model.matrix(x, type = "lhs"))
extralines_register("sdy", my_fun, "SD(y)")

# An estimation
data(iris)
est = feols(Petal.Length ~ Sepal.Length | Species, iris)

# Now we can easily create a row with the SD of y.
# We just "summon" it in a one-sided formula
etable(est, extralines = ~ sdy)

# We can change the alias on the fly:
etable(est, extralines = list("_Standard deviation of the dep. var." = ~ sdy))
```
f

Lags a variable in a fixest estimation

Description

Produce lags or leads in the formulas of fixest estimations or when creating variables in a data.table::data.table. The data must be set as a panel beforehand (either with the function panel or with the argument panel.id in the estimation).

Usage

f(x, lead = 1, fill = NA)
d(x, lag = 1, fill = NA)
l(x, lag = 1, fill = NA)

Arguments

x The variable.

lead A vector of integers giving the number of leads. Negative values lead to lags. This argument can be a vector when using it in fixest estimations. When creating variables in a data.table::data.table, it must be of length one.

fill A scalar, default is NA. How to fill the missing values due to the lag/lead? Note that in a fixest estimation, 'fill' must be numeric (not required when creating new variables).

lag A vector of integers giving the number of lags. Negative values lead to leads. This argument can be a vector when using it in fixest estimations. When creating variables in a data.table::data.table, it must be of length one.

Value

These functions can only be used i) in a formula of a fixest estimation, or ii) when creating variables within a fixest_panel object (obtained with function panel) which is also a data.table::data.table.

Functions

• f(): Forwards a variable (inverse of lagging) in a fixest estimation
• d(): Creates differences (i.e. x - lag(x)) in a fixest estimation

See Also

The function panel changes data.frames into a panel from which the functions l and f can be called. Otherwise you can set the panel 'live' during the estimation using the argument panel.id (see for example in the function feols).
Examples

```r
data(base_did)

# Setting a data set as a panel...
pdat = panel(base_did, ~ id + period)

# ...then using the functions l and f
est1 = feols(y ~ l(x1, 0:1), pdat)
est2 = feols(f(y) ~ l(x1, -1:1), pdat)
est3 = feols(l(y) ~ l(x1, 0:3), pdat)
etable(est1, est2, est3, order = c("f", "^x"), drop = "Int")

# or using the argument panel.id
feols(f(y) ~ l(x1, -1:1), base_did, panel.id = ~id + period)
feols(d(y) ~ d(x1), base_did, panel.id = ~id + period)

# l() and f() can also be used within a data.table:
if(require("data.table")){
  pdat_dt = panel(as.data.table(base_did), ~id+period)
  # Now since pdat_dt is also a data.table
  # you can create lags/leads directly
  pdat_dt[, x1_l1 := l(x1)]
  pdat_dt[, x1_d1 := d(x1)]
  pdat_dt[, c("x1_l1_fill0", "y_f2") := .(l(x1, fill = 0), f(y, 2))]
}
```

---

### fdim

**Formatted dimension**

---

**Description**

Prints the dimension of a data set, in an user-readable way

**Usage**

`fdim(x)`

**Arguments**

- `x` An R object, usually a data.frame (but can also be a vector).

**Value**

It does not return anything, the output is directly printed on the console.
**feglm**

**Author(s)**

Laurent Berge

**Examples**

```r
fdim(iris)

fdim(iris$Species)
```

---

**feglm**

*Fixed-effects GLM estimations*

**Description**

Estimates GLM models with any number of fixed-effects.

**Usage**

```r
feglm(
  fml,
  data,
  family = "gaussian",
  vcov,
  offset,
  weights,
  subset,
  split,
  fsplit,
  split.keep,
  split.drop,
  cluster,
  se,
  ssc,
  panel.id,
  start = NULL,
  etastart = NULL,
  mustart = NULL,
  fixef,
  fixef.rm = "perfect",
  fixef.tol = 1e-06,
  fixef.iter = 10000,
  collin.tol = 1e-10,
  glm.iter = 25,
)```
glm.tol = 1e-08,
nthreads = getFixest_nthreads(),
lean = FALSE,
warn = TRUE,
notes = getFixest_notes(),
verbose = 0,
only.coef = FALSE,
combine.quick,
mem.clean = FALSE,
only.env = FALSE,
env,
...

feglm.fit(
y, 
X, 
fixef_df, 
family = "gaussian", 
vcov, 
offset, 
split, 
split, 
fsplit, 
split.keep, 
split.drop, 
cluster, 
se, 
ssc, 
weights, 
subset, 
start = NULL, 
etastart = NULL, 
mustart = NULL, 
fixef.rm = "perfect", 
fixef.tol = 1e-06, 
fixef.iter = 10000, 
collin.tol = 1e-10, 
glm.iter = 25, 
glm.tol = 1e-08, 
nthreads = getFixest_nthreads(), 
lean = FALSE, 
warn = TRUE, 
notes = getFixest_notes(), 
mem.clean = FALSE, 
verbose = 0, 
only.env = FALSE, 
only.coef = FALSE, 
env,
Arguments

**fml**

A formula representing the relation to be estimated. For example: `fml = z~x+y`. To include fixed-effects, insert them in this formula using a pipe: e.g. `fml = z~x+y|fixef_1+fixef_2`. Multiple estimations can be performed at once: for multiple dep. vars, wrap them in `c()`: e.g. `c(y1, y2)`. For multiple indep. vars, use the stepwise functions: `ex x1 + csw(x2, x3)`. The formula `fml = c(y1, y2) ~ x1 + cw0(x2, x3)` leads to 6 estimation, see details. Square brackets starting...
with a dot can be used to call global variables: \( y_{[i]} \sim x_{[1:2]} \) will lead to \( y_3 \sim x_1 + x_2 \) if \( i \) is equal to 3 in the current environment (see details in \texttt{xpd}).

data

A data.frame containing the necessary variables to run the model. The variables of the non-linear right hand side of the formula are identified with this data.frame names. Can also be a matrix.

family

Family to be used for the estimation. Defaults to \texttt{gaussian()}. See \texttt{family} for details of family functions.

vcov

Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: \texttt{vcov_type ~ variables}. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from \texttt{vcov_cluster}, \texttt{vcov_NW}, \texttt{NW}, \texttt{vcov_DK}, \texttt{DK}, \texttt{vcov_conley} and \texttt{conley}. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the \texttt{vcov} documentation in the \texttt{vignette}.

offset

A formula or a numeric vector. An offset can be added to the estimation. If equal to a formula, it should be of the form (for example) \( \sim 0.5*x**2 \). This offset is linearly added to the elements of the main formula 'fml'.

weights

A formula or a numeric vector. Each observation can be weighted, the weights must be greater than 0. If equal to a formula, it should be one-sided: for example \( \sim \var_weight \).

subset

A vector (logical or numeric) or a one-sided formula. If provided, then the estimation will be performed only on the observations defined by this argument.

split

A one sided formula representing a variable (eg \texttt{split = ~var}) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. If you also want to include the estimation for the full sample, use the argument \texttt{fsplit} instead. You can use the special operators \%keep\% and \%drop\% to select only a subset of values for which to split the sample. E.g. \texttt{split = ~var \%keep\% c("v1", "v2")} will split the sample only according to the values v1 and v2 of the variable \texttt{var}; it is equivalent to supplying the argument \texttt{split.keep = c("v1", "v2")}. By default there is partial matching on each value, you can trigger a regular expression evaluation by adding a \'@\' first, as in: \texttt{~var \%drop\% \"@^v[12]\"} which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

fsplit

A one sided formula representing a variable (eg \texttt{split = ~var}) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. This argument is the same as \texttt{split} but also includes the full sample as the first estimation. You can use the special operators \%keep\% and \%drop\% to select only a subset of values for which to split the sample. E.g. \texttt{split = ~var \%keep\% c("v1", "v2")} will split the sample only according to the values v1 and v2 of the variable \texttt{var}; it is equivalent to supplying the argument \texttt{split.keep = c("v1", "v2")}. By default there is partial matching on each value, you can trigger a regular expression evaluation by adding an \'@\' first, as in: \texttt{~var \%drop\% \"@^v[12]\"} which will drop values starting with "v1" or "v2" (of course you need to know regexes!).
split.keep  A character vector. Only used when split, or fsplit, is supplied. If provided, then the sample will be split only on the values of split.keep. The values in split.keep will be partially matched to the values of split. To enable regular expressions, you need to add an "@" first. For example split.keep = c("v1", "@other|var") will keep only the value in split partially matched by "v1" or the values containing "other" or "var".

split.drop  A character vector. Only used when split, or fsplit, is supplied. If provided, then the sample will be split only on the values that are not in split.drop. The values in split.drop will be partially matched to the values of split. To enable regular expressions, you need to add an "@" first. For example split.drop = c("v1", "@other|var") will drop only the value in split partially matched by "v1" or the values containing "other" or "var".

cluster  Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2.

If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

se  Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "threeway" or "fourway"? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

ssc  An object of class ssc.type obtained with the function ssc. Represents how the degree of freedom correction should be done. You must use the function ssc for this argument. The arguments and defaults of the function ssc are: adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE). See the help of the function ssc for details.

panel.id  The panel identifiers. Can either be: i) a one sided formula (e.g. panel.id = ~id+time), ii) a character vector of length 2 (e.g. panel.id=c('id', 'time'), or iii) a character scalar of two variables separated by a comma (e.g. panel.id='id,time'). Note that you can combine variables with ^ only inside formulas (see the dedicated section in feols).

start  Starting values for the coefficients. Can be: i) a numeric of length 1 (e.g. start = 0), ii) a numeric vector of the exact same length as the number of variables, or iii) a named vector of any length (the names will be used to initialize the appropriate coefficients). Default is missing.

etastart  Numeric vector of the same length as the data. Starting values for the linear predictor. Default is missing.

mustart  Numeric vector of the same length as the data. Starting values for the vector of means. Default is missing.
**fixef**

Character vector. The names of variables to be used as fixed-effects. These variables should contain the identifier of each observation (e.g., think of it as a panel identifier). Note that the recommended way to include fixed-effects is to insert them directly in the formula.

**fixef.rm**

Can be equal to "perfect" (default), "singleton", "both" or "none". Controls which observations are to be removed. If "perfect", then observations having a fixed-effect with perfect fit (e.g. only 0 outcomes in Poisson estimations) will be removed. If "singleton", all observations for which a fixed-effect appears only once will be removed. The meaning of "both" and "none" is direct.

**fixef.tol**

Precision used to obtain the fixed-effects. Defaults to $1e^{-6}$. It corresponds to the maximum absolute difference allowed between two coefficients of successive iterations.

**fixef.iter**

Maximum number of iterations in fixed-effects algorithm (only in use for 2+ fixed-effects). Default is 10000.

**collin.tol**

Numeric scalar, default is $1e^{-10}$. Threshold deciding when variables should be considered collinear and subsequently removed from the estimation. Higher values means more variables will be removed (if there is presence of collinearity). One signal of presence of collinearity is t-stats that are extremely low (for instance when t-stats $< 1e^{-3}$).

**glm.iter**

Number of iterations of the glm algorithm. Default is 25.

**glm.tol**

Tolerance level for the glm algorithm. Default is $1e^{-8}$.

**nthreads**

The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function `setFixest_nthreads`.

**lean**

Logical, default is `FALSE`. If `TRUE` then all large objects are removed from the returned result: this will save memory but will block the possibility to use many methods. It is recommended to use the arguments `se` or `cluster` to obtain the appropriate standard-errors at estimation time, since obtaining different SEs won’t be possible afterwards.

**warn**

Logical, default is `TRUE`. Whether warnings should be displayed (concerns warnings relating to convergence state).

**notes**

Logical. By default, three notes are displayed: when NAs are removed, when some fixed-effects are removed because of only 0 (or 0/1) outcomes, or when a variable is dropped because of collinearity. To avoid displaying these messages, you can set `notes = FALSE`. You can remove these messages permanently by using `setFixest_notes(FALSE)`.

**verbose**

Integer. Higher values give more information. In particular, it can detail the number of iterations in the demeaning algorithm (the first number is the left-hand-side, the other numbers are the right-hand-side variables). It can also detail the step-halving algorithm.

**only.coef**

Logical, default is `FALSE`. If `TRUE`, then only the estimated coefficients are returned. Note that the length of the vector returned is always the length of the number of coefficients to be estimated: this means that the variables found to be collinear are returned with an NA value.
combine.quick Logical. When you combine different variables to transform them into a single fixed-effects you can do e.g. \( y \sim x \mid \text{paste}(\text{var1}, \text{var2}) \). The algorithm provides a shorthand to do the same operation: \( y \sim x \mid \text{var1}^\text{var2} \). Because pasting variables is a costly operation, the internal algorithm may use a numerical trick to hasten the process. The cost of doing so is that you lose the labels. If you are interested in getting the value of the fixed-effects coefficients after the estimation, you should use combine.quick = FALSE. By default it is equal to FALSE if the number of observations is lower than 50,000, and to TRUE otherwise.

mem.clean Logical, default is FALSE. Only to be used if the data set is large compared to the available RAM. If TRUE then intermediary objects are removed as much as possible and gc is run before each substantial C++ section in the internal code to avoid memory issues.

only.env (Advanced users.) Logical, default is FALSE. If TRUE, then only the environment used to make the estimation is returned.

eq (Advanced users.) A fixest environment created by a fixest estimation with only.env = TRUE. Default is missing. If provided, the data from this environment will be used to perform the estimation.

... Not currently used.

\( y \) Numeric vector/matrix/data.frame of the dependent variable(s). Multiple dependent variables will return a fixest_multi object.

\( X \) Numeric matrix of the regressors.

\( \text{fixef}_\text{df} \) Matrix/data.frame of the fixed-effects.

Details

The core of the GLM are the weighted OLS estimations. These estimations are performed with \texttt{feols}. The method used to demean each variable along the fixed-effects is based on Berge (2018), since this is the same problem to solve as for the Gaussian case in a ML setup.

Value

A fixest object. Note that fixest objects contain many elements and most of them are for internal use, they are presented here only for information. To access them, it is safer to use the user-level methods (e.g. \texttt{vcov.fixest}, \texttt{resid.fixest}, etc) or functions (like for instance \texttt{fitstat} to access any fit statistic).

\( \text{nobs} \) The number of observations.

\( \text{fml} \) The linear formula of the call.

\( \text{call} \) The call of the function.

\( \text{method} \) The method used to estimate the model.

\( \text{family} \) The family used to estimate the model.

\( \text{fml_all} \) A list containing different parts of the formula. Always contain the linear formula. Then, if relevant: \texttt{fixef}: the fixed-effects.

\( \text{npars} \) The number of parameters of the model.

\( \text{fixef}_{\text{vars}} \) The names of each fixed-effect dimension.
The list (of length the number of fixed-effects) of the fixed-effects identifiers for each observation.

The size of each fixed-effect (i.e. the number of unique identifier for each fixed-effect dimension).

(When relevant.) The dependent variable (used to compute the within-R2 when fixed-effects are present).

Logical, convergence status of the IRWLS algorithm.

The weights of the last iteration of the IRWLS algorithm.

(When relevant.) List containing vectors of integers. It represents the sequential selection of observation vis a vis the original data set.

(When relevant.) In the case there were fixed-effects and some observations were removed because of only 0/1 outcome within a fixed-effect, it gives the list (for each fixed-effect dimension) of the fixed-effect identifiers that were removed.

The named vector of estimated coefficients.

The table of the coefficients with their standard errors, z-values and p-values.

The loglikelihood.

Deviance of the fitted model.

Number of iterations of the algorithm.

Log-likelihood of the null model (i.e. with the intercept only).

Sum of the squared residuals of the null model (containing only with the intercept).

The adjusted pseudo R2.

The fitted values are the expected value of the dependent variable for the fitted model: that is \( E(Y|X) \).

The linear predictors.

The residuals (y minus the fitted values).

Squared correlation between the dependent variable and the expected predictor (i.e. fitted.values) obtained by the estimation.

The Hessian of the parameters.

The variance-covariance matrix of the parameters.

The standard-error of the parameters.

The matrix of the scores (first derivative for each observation).

The difference between the dependent variable and the expected predictor.

The sum of the fixed-effects coefficients for each observation.

(When relevant.) The offset formula.

(When relevant.) The weights formula.

(When relevant.) Vector containing the variables removed because of collinearity.

(When relevant.) Vector of coefficients, where the values of the variables removed because of collinearity are NA.
Combining the fixed-effects

You can combine two variables to make it a new fixed-effect using \(^{\wedge}\). The syntax is as follows: \(fe_1^{\wedge}fe_2\). Here you created a new variable which is the combination of the two variables \(fe_1\) and \(fe_2\). This is identical to doing \(\text{paste0}(fe_1, ",", fe_2)\) but more convenient.

Note that pasting is a costly operation, especially for large data sets. Thus, the internal algorithm uses a numerical trick which is fast, but the drawback is that the identity of each observation is lost (i.e. they are now equal to a meaningless number instead of being equal to \(\text{paste0}(fe_1, ",", fe_2)\)). These “identities” are useful only if you’re interested in the value of the fixed-effects (that you can extract with \(\text{fixef}\text{.fixest}\)). If you’re only interested in coefficients of the variables, it doesn’t matter. Anyway, you can use \(\text{combine\_quick} = \text{FALSE}\) to tell the internal algorithm to use \(\text{paste}\) instead of the numerical trick. By default, the numerical trick is performed only for large data sets.

Varying slopes

You can add variables with varying slopes in the fixed-effect part of the formula. The syntax is as follows: \(\text{fixef\_var}[\text{var1, var2}]\). Here the variables \(\text{var1}\) and \(\text{var2}\) will be with varying slopes (one slope per value in \(\text{fixef\_var}\)) and the fixed-effect \(\text{fixef\_var}\) will also be added.

To add only the variables with varying slopes and not the fixed-effect, use double square brackets: \(\text{fixef\_var}[[\text{var1, var2}]]\).

In other words:

- \(\text{fixef\_var}[\text{var1, var2}]\) is equivalent to \(\text{fixef\_var} + \text{fixef\_var}[[\text{var1}]] + \text{fixef\_var}[[\text{var2}]]\)
- \(\text{fixef\_var}[[\text{var1, var2}]]\) is equivalent to \(\text{fixef\_var}[[\text{var1}]] + \text{fixef\_var}[[\text{var2}]]\)

In general, for convergence reasons, it is recommended to always add the fixed-effect and avoid using only the variable with varying slope (i.e. use single square brackets).

Lagging variables

To use leads/lags of variables in the estimation, you can: i) either provide the argument \(\text{panel\_id}\), ii) either set your data set as a panel with the function \(\text{panel, f and d}\).

You can provide several leads/lags/differences at once: e.g. if your formula is equal to \(f(y) \sim 1(x, \sim1:1)\), it means that the dependent variable is equal to the lead of \(y\), and you will have as explanatory variables the lead of \(x1\), \(x1\) and the lag of \(x1\). See the examples in function \(l\) for more details.

Interactions

You can interact a numeric variable with a "factor-like" variable by using \(i(\text{factor\_var, continuous\_var}, \text{ref})\), where \(\text{continuous\_var}\) will be interacted with each value of \(\text{factor\_var}\) and the argument \(\text{ref}\) is a value of \(\text{factor\_var}\) taken as a reference (optional).

Using this specific way to create interactions leads to a different display of the interacted values in \(\text{etable}\). See examples.

It is important to note that if you do not care about the standard-errors of the interactions, then you can add interactions in the fixed-effects part of the formula, it will be incomparably faster (using the syntax \(\text{factor\_var}[\text{continuous\_var}]\), as explained in the section “Varying slopes”).

The function \(i\) has in fact more arguments, please see details in its associated help page.
On standard-errors

Standard-errors can be computed in different ways, you can use the arguments se and ssc in \texttt{summary.fixest} to define how to compute them. By default, in the presence of fixed-effects, standard-errors are automatically clustered.

The following vignette: \texttt{On standard-errors} describes in details how the standard-errors are computed in \texttt{fixest} and how you can replicate standard-errors from other software.

You can use the functions \texttt{setFixest_vcov} and \texttt{setFixest_ssc} to permanently set the way the standard-errors are computed.

Multiple estimations

Multiple estimations can be performed at once, they just have to be specified in the formula. Multiple estimations yield a \texttt{fixest_multi} object which is ‘kind of’ a list of all the results but includes specific methods to access the results in a handy way. Please have a look at the dedicated vignette: \texttt{Multiple estimations}.

To include multiple dependent variables, wrap them in \texttt{c()} (\texttt{list()} also works). For instance \texttt{fml = c(y1, y2) ~ x1} would estimate the model \texttt{fml = y1 ~ x1} and then the model \texttt{fml = y2 ~ x1}.

To include multiple independent variables, you need to use the stepwise functions. There are 4 stepwise functions: \texttt{sw}, \texttt{sw0}, \texttt{csw}, \texttt{csw0}, and \texttt{mvsw}. Of course \texttt{sw} stands for stepwise, and \texttt{csw} for cumulative stepwise. Finally \texttt{mvsw} is a bit special, it stands for multiverse stepwise. Let’s explain that. Assume you have the following formula: \texttt{fml = y ~ x1 + sw(x2, x3)}. The stepwise function \texttt{sw} will estimate the following two models: \texttt{y ~ x1 + x2} and \texttt{y ~ x1 + x3}. That is, each element in \texttt{sw()} is sequentially, and separately, added to the formula. Would have you used \texttt{sw0} in lieu of \texttt{sw}, then the model \texttt{y ~ x1} would also have been estimated. The \texttt{0} in the name means that the model without any stepwise element also needs to be estimated. The prefix \texttt{c} means cumulative: each stepwise element is added to the next. That is, \texttt{fml = y ~ x1 + csw(x2, x3)} would lead to the following models \texttt{y ~ x1 + x2} and \texttt{y ~ x1 + x2 + x3}. The \texttt{0} has the same meaning and would also lead to the model without the stepwise elements to be estimated: in other words, \texttt{fml = y ~ x1 + csw0(x2, x3)} leads to the following three models: \texttt{y ~ x1, y ~ x1 + x2} and \texttt{y ~ x1 + x2 + x3}. Finally \texttt{mvsw} will add, in a stepwise fashion all possible combinations of the variables in its arguments. For example \texttt{mvsw(x1, x2, x3)} is equivalent to \texttt{sw0(x1, x2, x3, x1 + x2, x1 + x3, x2 + x3, x1 + x2 + x3)}. The number of models to estimate grows at a factorial rate: so be cautious!

Multiple independent variables can be combined with multiple dependent variables, as in \texttt{fml = c(y1, y2) ~ cw(x1, x2, x3)} which would lead to 6 estimations. Multiple estimations can also be combined to split samples (with the arguments \texttt{split}, \texttt{fsplit}).

You can also add fixed-effects in a stepwise fashion. Note that you cannot perform stepwise estimations on the IV part of the formula (\texttt{feols} only).

If NAs are present in the sample, to avoid too many messages, only NA removal concerning the variables common to all estimations is reported.

A note on performance. The feature of multiple estimations has been highly optimized for \texttt{feols}, in particular in the presence of fixed-effects. It is faster to estimate multiple models using the formula rather than with a loop. For non-\texttt{feols} models using the formula is roughly similar to using a loop performance-wise.
Argument sliding

When the data set has been set up globally using `setFixest_estimation(data = data_set)`, the argument `vcov` can be used implicitly. This means that calls such as `feols(y ~ x, "HC1")`, or `feols(y ~ x, ~id)`, are valid: i) the data is automatically deduced from the global settings, and ii) the `vcov` is deduced to be the second argument.

Piping

Although the argument `data` is placed in second position, the data can be piped to the estimation functions. For example, with R >= 4.1, `mtcars |> feols(mpg ~ cyl)` works as `feols(mpg ~ cyl, mtcars).

Tricks to estimate multiple LHS

To use multiple dependent variables in `fixest` estimations, you need to include them in a vector: like in `c(y1, y2, y3)`.

First, if names are stored in a vector, they can readily be inserted in a formula to perform multiple estimations using the dot square bracket operator. For instance if `my_lhs = c("y1", "y2")`, calling `fixest with`, say `feols(. [my_lhs] ~ x1, etc)` is equivalent to using `feols(c(y1, y2) ~ x1, etc)`. Beware that this is a special feature unique to the left-hand-side of `fixest` estimations (the default behavior of the DSB operator is to aggregate with sums, see `xpd`).

Second, you can use a regular expression to grep the left-hand-sides on the fly. When the `..("regex")` feature is used naked on the LHS, the variables greppe are inserted into c(). For example `..("Pe") ~ Sepal.Length, iris` is equivalent to `c(Petal.Length, Petal.Width) ~ Sepal.Length, iris`. Beware that this is a special feature unique to the left-hand-side of `fixest` estimations (the default behavior of `..("regex")` is to aggregate with sums, see `xpd`).

Dot square bracket operator in formulas

In a formula, the dot square bracket (DSB) operator can: i) create manifold variables at once, or ii) capture values from the current environment and put them verbatim in the formula.

Say you want to include the variables `x1` to `x3` in your formula. You can use `xpd(y ~ x. [1:3])` and you’ll get `y ~ x1 + x2 + x3`.

To summon values from the environment, simply put the variable in square brackets. For example: `for(i in 1:3) xpd(y.[i] ~ x)` will create the formulas `y1 ~ x` to `y3 ~ x` depending on the value of `i`.

You can include a full variable from the environment in the same way: `for(y in c("a", "b")) xpd(.[y] ~ x)` will create the two formulas `a ~ x` and `b ~ x`.

The DSB can even be used within variable names, but then the variable must be nested in character form. For example `y ~ .["x. [1:2].sq"]` will create `y ~ x1_sq + x2_sq`. Using the character form is important to avoid a formula parsing error. Double quotes must be used. Note that the character string that is nested will be parsed with the function `dsb`, and thus it will return a vector.

By default, the DSB operator expands vectors into sums. You can add a comma, like in `. [, x]`, to expand with commas—the content can then be used within functions. For instance: `c(x.[, 1:2])` will create `c(x1, x2)` (and not `c(x1 + x2)`).

In all `fixest` estimations, this special parsing is enabled, so you don’t need to use `xpd`.
One-sided formulas can be expanded with the DSB operator: let \( x = \neg sepal + petal \), then \( xpd(y \sim [x]) \) leads to \( color \sim sepal + petal \).

You can even use multiple square brackets within a single variable, but then the use of nesting is required. For example, the following \( xpd(y \sim [".\{letters[1:2]\}_1[1:2]"]) \) will create \( y \sim a_1 + b_2 \). Remember that the nested character string is parsed with \texttt{dsb}, which explains this behavior.

When the element to be expanded i) is equal to the empty string or, ii) is of length 0, it is replaced with a neutral element, namely 1. For example, \( x = "" \); \( xpd(y \sim .[x]) \) leads to \( y \sim 1 \).

Author(s)

Laurent Berge

References


For models with multiple fixed-effects:

Gaure, Simen, 2013, "OLS with multiple high dimensional category variables", Computational Statistics & Data Analysis 66 pp. 8–18

See Also

See also \texttt{summary.fixest} to see the results with the appropriate standard-errors, \texttt{fixef.fixest} to extract the fixed-effects coefficients, and the function \texttt{etable} to visualize the results of multiple estimations. And other estimation methods: \texttt{feols, feqmlm, fenegbin, feNmlm}.

Examples

# Poisson estimation
res = feglm(Sepal.Length ~ Sepal.Width + Petal.Length | Species, iris, "poisson")

# You could also use fepois
res_pois = fepois(Sepal.Length ~ Sepal.Width + Petal.Length | Species, iris)

# With the fit method:
res_fit = feglm.fit(iris$Sepal.Length, iris[, 2:3], iris$Species, "poisson")

# All results are identical:
etable(res, res_pois, res_fit)

# Note that you have many more examples in feols

# # Multiple estimations:
#
# # 6 estimations
est_mult = fepois(c(Ozone, Solar.R) ~ Wind + Temp + csw0(Wind:Temp, Day), airquality)
# We can display the results for the first lhs:
etable(est_mult[lhs = 1])

# And now the second (access can be made by name)
etable(est_mult[lhs = "Solar.R"])

# Now we focus on the two last right hand sides
# (note that .N can be used to specify the last item)
etable(est_mult[rhs = 2:.N])

# Combining with split
est_split = fepois(c(Ozone, Solar.R) ~ sw(poly(Wind, 2), poly(Temp, 2)),
                   airquality, split = ~ Month)

# You can display everything at once with the print method
est_split

# Different way of displaying the results with "compact"
summary(est_split, "compact")

# You can still select which sample/LHS/RHS to display
est_split[sample = 1:2, lhs = 1, rhs = 1]

---

**femlm**  
*Fixed-effects maximum likelihood models*

**Description**

This function estimates maximum likelihood models with any number of fixed-effects.

**Usage**

```r
femlm(
fml,  
data,  
family = c("poisson", "negbin", "logit", "gaussian"),  
vcov,  
start = 0,  
fixef,  
fixef.rm = "perfect",  
offset,  
subset,  
split,  
fsplit,  
split.keep,  
split.drop,
```
fenegbin(
 .fml, 
  data, 
  vcov, 
  theta.init, 
  start = 0, 
  fixef, 
  fixef.rm = "perfect", 
  offset, 
  subset, 
  split, 
  fsplit, 
  split.keep, 
  split.drop, 
  cluster, 
  se, 
  ssc, 
  panel.id, 
  fixef.tol = 1e-05, 
  fixef.iter = 10000, 
  nthreads = getFixest_nthreads(), 
  lean = FALSE, 
  verbose = 0, 
  warn = TRUE, 
  notes = getFixest_notes(), 
  combine.quick, 
  mem.clean = FALSE, 
  only.env = FALSE, 
  only.coef = FALSE, 
  env, ...
)
only.coef = FALSE,
env,
...
)

**Arguments**

- **fml**: A formula representing the relation to be estimated. For example: `fml = z~x+y`. To include fixed-effects, insert them in this formula using a pipe: e.g. `fml = z~x+y|fixef_1+fixef_2`. Multiple estimations can be performed at once: for multiple dep. vars, wrap them in `c()`: ex `c(y1, y2)`. For multiple indep. vars, use the stepwise functions: ex `x1 + csw(x2, x3)`. The formula `fml = c(y1, y2) ~ x1 + csw(x2, x3)` leads to 6 estimation, see details. Square brackets starting with a dot can be used to call global variables: `y.[i] ~ x.[1:2]` will lead to `y3 ~ x1 + x2` if `i` is equal to 3 in the current environment (see details in `xpd`).

- **data**: A data.frame containing the necessary variables to run the model. The variables of the non-linear right hand side of the formula are identified with this data.frame names. Can also be a matrix.

- **family**: Character scalar. It should provide the family. The possible values are "poisson" (Poisson model with log-link, the default), "negbin" (Negative Binomial model with log-link), "logit" (LOGIT model with log-link), "gaussian" (Gaussian model).

- **vcov**: Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the `vcov` documentation in the vignette.

- **start**: Starting values for the coefficients. Can be: i) a numeric of length 1 (e.g. `start = 0`, the default), ii) a numeric vector of the exact same length as the number of variables, or iii) a named vector of any length (the names will be used to initialize the appropriate coefficients).

- **fixef**: Character vector. The names of variables to be used as fixed-effects. These variables should contain the identifier of each observation (e.g., think of it as a panel identifier). Note that the recommended way to include fixed-effects is to insert them directly in the formula.

- **fixef.rm**: Can be equal to "perfect" (default), "singleton", "both" or "none". Controls which observations are to be removed. If "perfect", then observations having a fixed-effect with perfect fit (e.g. only 0 outcomes in Poisson estimations) will be removed. If "singleton", all observations for which a fixed-effect appears only once will be removed. The meaning of "both" and "none" is direct.

- **offset**: A formula or a numeric vector. An offset can be added to the estimation. If equal to a formula, it should be of the form (for example) `~0.5*x**2`. This offset is linearly added to the elements of the main formula 'fml'.
subset

A vector (logical or numeric) or a one-sided formula. If provided, then the estimation will be performed only on the observations defined by this argument.

split

A one-sided formula representing a variable (e.g., `split = ~var`) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. If you also want to include the estimation for the full sample, use the argument `fsplit` instead. You can use the special operators `%keep%` and `%drop%` to select only a subset of values for which to split the sample. E.g., `split = ~var %keep% c("v1", "v2")` will split the sample only according to the values `v1` and `v2` of the variable `var`; it is equivalent to supplying the argument `split.keep = c("v1", "v2")`. By default there is partial matching on each value, you can trigger a regular expression evaluation by adding a `@` first, as in: `~var %drop% "@\^v\[12\]"` which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

fsplit

A one-sided formula representing a variable (e.g., `split = ~var`) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. This argument is the same as `split` but also includes the full sample as the first estimation. You can use the special operators `%keep%` and `%drop%` to select only a subset of values for which to split the sample. E.g., `split = ~var %keep% c("v1", "v2")` will split the sample only according to the values `v1` and `v2` of the variable `var`; it is equivalent to supplying the argument `split.keep = c("v1", "v2")`. By default there is partial matching on each value, you can trigger a regular expression evaluation by adding an `@` first, as in: `~var %drop% "@\^v\[12\]"` which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

split.keep

A character vector. Only used when `split`, or `fsplit`, is supplied. If provided, then the sample will be split only on the values of `split.keep`. The values in `split.keep` will be partially matched to the values of `split`. To enable regular expressions, you need to add an `@` first. For example `split.keep = c("v1", "@other|var")` will keep only the value in `split` partially matched by "v1" or the values containing "other" or "var".

split.drop

A character vector. Only used when `split`, or `fsplit`, is supplied. If provided, then the sample will be split only on the values that are not in `split.drop`. The values in `split.drop` will be partially matched to the values of `split`. To enable regular expressions, you need to add an `@` first. For example `split.drop = c("v1", "@other|var")` will drop only the value in `split` partially matched by "v1" or the values containing "other" or "var".

cluster

Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over `var1` and `var2` contained in the data.frame `base` used for the estimation. All the following `cluster` arguments are valid and do the same thing: `cluster = base[, c("var1", "var2")], cluster = c("var1", "var2") , cluster = ~var1+var2`. If the two variables were used as fixed-effects in the estimation, you can leave it blank with `vcov = "twoway"` (assuming `var1` [resp. `var2`] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using `^` with the following syntax: `cluster = ~var1^var2` or `cluster = "var1^var2"`.

se

Character scalar. Which kind of standard error should be computed: "standard", "robust", or "cluster" (if clustering is requested).
“hetero”, “cluster”, “twoway”, “threeway” or “fourway”? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

**ssc**

An object of class ssc.type obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: `adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE).` See the help of the function `ssc` for details.

**panel.id**

The panel identifiers. Can either be: i) a one sided formula (e.g. `panel.id = ~id+time`), ii) a character vector of length 2 (e.g. `panel.id=c('id', 'time')`, or iii) a character scalar of two variables separated by a comma (e.g. `panel.id='id,time').` Note that you can combine variables with ^ only inside formulas (see the dedicated section in *feols*).

**fixef.tol**

Precision used to obtain the fixed-effects. Defaults to 1e-5. It corresponds to the maximum absolute difference allowed between two coefficients of successive iterations. Argument `fixef.tol` cannot be lower than `10000*.Machine$double.eps`. Note that this parameter is dynamically controlled by the algorithm.

**fixef.iter**

Maximum number of iterations in fixed-effects algorithm (only in use for 2+ fixed-effects). Default is 10000.

**nthreads**

The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function `setFixest_nthreads`.

**lean**

Logical, default is FALSE. If TRUE then all large objects are removed from the returned result: this will save memory but will block the possibility to use many methods. It is recommended to use the arguments se or cluster to obtain the appropriate standard-errors at estimation time, since obtaining different SEs won’t be possible afterwards.

**verbose**

Integer, default is 0. It represents the level of information that should be reported during the optimisation process. If `verbose=0` nothing is reported. If `verbose=1`: the value of the coefficients and the likelihood are reported. If `verbose=2`: 1 + information on the computing time of the null model, the fixed-effects coefficients and the hessian are reported.

**warn**

Logical, default is TRUE. Whether warnings should be displayed (concerns warnings relating to convergence state).

**notes**

Logical. By default, two notes are displayed: when NAs are removed (to show additional information) and when some observations are removed because of only 0 (or 0/1) outcomes in a fixed-effect setup (in Poisson/Neg. Bin./Logit models). To avoid displaying these messages, you can set `notes = FALSE`. You can remove these messages permanently by using `setFixest_notes(FALSE).

**theta.init**

Positive numeric scalar. The starting value of the dispersion parameter if `family="negbin"`. By default, the algorithm uses as a starting value the theta obtained from the model with only the intercept.
logical. When you combine different variables to transform them into a single
fixed-effects you can do e.g. \( y \sim x | \text{paste(var1, var2)} \). The algorithm
provides a shorthand to do the same operation: \( y \sim x | \text{var1^var2} \). Because pasting
variables is a costly operation, the internal algorithm may use a numerical trick
to hasten the process. The cost of doing so is that you lose the labels. If you are
interested in getting the value of the fixed-effects coefficients after the estima-
tion, you should use `combine.quick = FALSE`. By default it is equal to `FALSE` if
the number of observations is lower than 50,000, and to `TRUE` otherwise.

**mem.clean**

Logical, default is `FALSE`. Only to be used if the data set is large compared to
the available RAM. If `TRUE` then intermediary objects are removed as much as
possible and `gc` is run before each substantial C++ section in the internal code
to avoid memory issues.

**only.env**

(Advanced users.) Logical, default is `FALSE`. If `TRUE`, then only the environment
used to make the estimation is returned.

**only.coef**

Logical, default is `FALSE`. If `TRUE`, then only the estimated coefficients are re-
turned. Note that the length of the vector returned is always the length of the
number of coefficients to be estimated: this means that the variables found to be
collinear are returned with an NA value.

**env**

(Advanced users.) A `fixest` environment created by a `fixest` estimation with
`only.env = TRUE`. Default is missing. If provided, the data from this environ-
ment will be used to perform the estimation.

... Not currently used.

**Details**

Note that the functions `feglm` and `femlm` provide the same results when using the same families
but differ in that the latter is a direct maximum likelihood optimization (so the two can really have
different convergence rates).

**Value**

A `fixest` object. Note that `fixest` objects contain many elements and most of them are for internal
use, they are presented here only for information. To access them, it is safer to use the user-level
methods (e.g. `vcov.fixest`, `resid.fixest`, etc) or functions (like for instance `fitstat` to access
any fit statistic).

**nobs**

The number of observations.

**fml**

The linear formula of the call.

**call**

The call of the function.

**method**

The method used to estimate the model.

**family**

The family used to estimate the model.

**fml_all**

A list containing different parts of the formula. Always contain the linear for-
formula. Then, if relevant: `fixef`: the fixed-effects; `NL`: the non linear part of the
formula.

**nparams**

The number of parameters of the model.

**fixef_vars**

The names of each fixed-effect dimension.
**femlm**

**fixef_id**  The list (of length the number of fixed-effects) of the fixed-effects identifiers for each observation.

**fixef_sizes**  The size of each fixed-effect (i.e. the number of unique identifier for each fixed-effect dimension).

**convStatus**  Logical, convergence status.

**message**  The convergence message from the optimization procedures.

**obs_selection**  (When relevant.) List containing vectors of integers. It represents the sequential selection of observation vis a vis the original data set.

**fixef_removed**  (When relevant.) In the case there were fixed-effects and some observations were removed because of only 0/1 outcome within a fixed-effect, it gives the list (for each fixed-effect dimension) of the fixed-effect identifiers that were removed.

**coefficients**  The named vector of estimated coefficients.

**coeftable**  The table of the coefficients with their standard errors, z-values and p-values.

**loglik**  The log-likelihood.

**iterations**  Number of iterations of the algorithm.

**ll_null**  Log-likelihood of the null model (i.e. with the intercept only).

**ll_fe_only**  Log-likelihood of the model with only the fixed-effects.

**ssr_null**  Sum of the squared residuals of the null model (containing only with the intercept).

**pseudo_r2**  The adjusted pseudo R2.

**fitted.values**  The fitted values are the expected value of the dependent variable for the fitted model: that is \( E(Y|X) \).

**residuals**  The residuals (y minus the fitted values).

**sq.cor**  Squared correlation between the dependent variable and the expected predictor (i.e. fitted.values) obtained by the estimation.

**hessian**  The Hessian of the parameters.

**cov.iid**  The variance-covariance matrix of the parameters.

**se**  The standard-error of the parameters.

**scores**  The matrix of the scores (first derivative for each observation).

**residuals**  The difference between the dependent variable and the expected predictor.

**sumFE**  The sum of the fixed-effects coefficients for each observation.

**offset**  (When relevant.) The offset formula.

**weights**  (When relevant.) The weights formula.

**Combining the fixed-effects**

You can combine two variables to make it a new fixed-effect using \(^\). The syntax is as follows: \( fe_1^fe_2 \). Here you created a new variable which is the combination of the two variables \( fe_1 \) and \( fe_2 \). This is identical to doing `paste0(fe_1, "_", fe_2)` but more convenient.
Note that pasting is a costly operation, especially for large data sets. Thus, the internal algorithm uses a numerical trick which is fast, but the drawback is that the identity of each observation is lost (i.e. they are now equal to a meaningless number instead of being equal to `paste0(fe_1, "_", fe_2)`). These “identities” are useful only if you’re interested in the value of the fixed-effects (that you can extract with `fixef.fixest`). If you’re only interested in coefficients of the variables, it doesn’t matter. Anyway, you can use `combine.quick = FALSE` to tell the internal algorithm to use `paste` instead of the numerical trick. By default, the numerical trick is performed only for large data sets.

### Lagging variables

To use leads/lags of variables in the estimation, you can: i) either provide the argument `panel.id`, ii) either set your data set as a panel with the function `panel, f` and `d`.

You can provide several leads/lags/differences at once: e.g. if your formula is equal to `f(y) ~ l(x, -1:1)`, it means that the dependent variable is equal to the lead of `y`, and you will have as explanatory variables the lead of `x1`, `x1` and the lag of `x1`. See the examples in function `l` for more details.

### Interactions

You can interact a numeric variable with a "factor-like" variable by using `i(factor_var, continuous_var, ref)`, where `continuous_var` will be interacted with each value of `factor_var` and the argument `ref` is a value of `factor_var` taken as a reference (optional).

Using this specific way to create interactions leads to a different display of the interacted values in `etable`. See examples.

It is important to note that if you do not care about the standard-errors of the interactions, then you can add interactions in the fixed-effects part of the formula, it will be incomparably faster (using the syntax `factor_var[continuous_var]`, as explained in the section “Varying slopes”).

The function `i` has in fact more arguments, please see details in its associated help page.

### On standard-errors

Standard-errors can be computed in different ways, you can use the arguments `se` and `ssc` in `summary.fixest` to define how to compute them. By default, in the presence of fixed-effects, standard-errors are automatically clustered.

The following vignette: On standard-errors describes in details how the standard-errors are computed in `fixest` and how you can replicate standard-errors from other software.

You can use the functions `setFixest_vcov` and `setFixest_ssc` to permanently set the way the standard-errors are computed.

### Multiple estimations

Multiple estimations can be performed at once, they just have to be specified in the formula. Multiple estimations yield a `fixest_multi` object which is ‘kind of’ a list of all the results but includes specific methods to access the results in a handy way. Please have a look at the dedicated vignette: Multiple estimations.
To include multiple dependent variables, wrap them in `c()` (`list()` also works). For instance \( \text{fml} = c(y_1, y_2) \sim x_1 \) would estimate the model \( \text{fml} = y_1 \sim x_1 \) and then the model \( \text{fml} = y_2 \sim x_1 \).

To include multiple independent variables, you need to use the stepwise functions. There are 4 stepwise functions: `sw`, `sw0`, `csw`, `csw0`, and `mvsw`. Of course `sw` stands for stepwise, and `csw` for cumulative stepwise. Finally `mvsw` is a bit special, it stands for multiverse stepwise. Let’s explain that. Assume you have the following formula: \( \text{fml} = y \sim x_1 + \text{sw}(x_2, x_3) \). The stepwise function `sw` will estimate the following two models: \( y \sim x_1 + x_2 \) and \( y \sim x_1 + x_3 \). That is, each element in `sw()` is sequentially, and separately, added to the formula. Would have you used `sw0` in lieu of `sw`, then the model \( y \sim x_1 \) would also have been estimated. The 0 in the name means that the model without any stepwise element also needs to be estimated. The prefix `c` means cumulative: each stepwise element is added to the next. That is, \( \text{fml} = y \sim x_1 + \text{csw}(x_2, x_3) \) would lead to the following models \( y \sim x_1 + x_2 \) and \( y \sim x_1 + x_2 + x_3 \). The 0 has the same meaning and would also lead to the model without the stepwise elements to be estimated: in other words, \( \text{fml} = y \sim x_1 + \text{csw0}(x_2, x_3) \) leads to the following three models: \( y \sim x_1, y \sim x_1 + x_2 \) and \( y \sim x_1 + x_2 + x_3 \). Finally `mvsw` will add, in a stepwise fashion all possible combinations of the variables in its arguments. For example \( \text{mvsw}(x_1, x_2, x_3) \) is equivalent to `sw0(x1, x2, x3, x1 + x2, x1 + x3, x2 + x3, x1 + x2 + x3)`. The number of models to estimate grows at a factorial rate: so be cautious!

Multiple independent variables can be combined with multiple dependent variables, as in \( \text{fml} = c(y_1, y_2) \sim \text{cw}(x_1, x_2, x_3) \) which would lead to 6 estimations. Multiple estimations can also be combined to split samples (with the arguments `split`, `fsplit`).

You can also add fixed-effects in a stepwise fashion. Note that you cannot perform stepwise estimations on the IV part of the formula (`feols` only).

If NAs are present in the sample, to avoid too many messages, only NA removal concerning the variables common to all estimations is reported.

A note on performance. The feature of multiple estimations has been highly optimized for `feols`, in particular in the presence of fixed-effects. It is faster to estimate multiple models using the formula rather than with a loop. For non-`feols` models using the formula is roughly similar to using a loop performance-wise.

**Argument sliding**

When the data set has been set up globally using `setFixest_estimation(data = data_set)`, the argument `vcov` can be used implicitly. This means that calls such as `feols(y ~ x, "HC1")`, or `feols(y ~ x, ~id)`, are valid: i) the data is automatically deduced from the global settings, and ii) the `vcov` is deduced to be the second argument.

**Piping**

Although the argument ‘data’ is placed in second position, the data can be piped to the estimation functions. For example, with R >= 4.1, `mtcars |> feols(mpg ~ cyl)` works as `feols(mpg ~ cyl, mtcars)`.

**Tricks to estimate multiple LHS**

To use multiple dependent variables in fixest estimations, you need to include them in a vector: like in `c(y_1, y_2, y_3)`.
First, if names are stored in a vector, they can readily be inserted in a formula to perform multiple estimations using the dot square bracket operator. For instance if \texttt{my_lhs} = c("y1", "y2"), calling \texttt{fixest} with, say \texttt{feols(my_lhs ~ x1, etc)} is equivalent to using \texttt{feols(c(y1, y2) ~ x1, etc)}. Beware that this is a special feature unique to the left-hand-side of \texttt{fixest} estimations (the default behavior of the DSB operator is to aggregate with sums, see \texttt{xpd}).

Second, you can use a regular expression to grep the left-hand-sides on the fly. When the \texttt{..("regex")} feature is used naked on the LHS, the variables grepped are inserted into \texttt{c()}. For example \texttt{..("Pe") ~ Sepal.Length, iris} is equivalent to \texttt{c(Petal.Length, Petal.Width) ~ Sepal.Length, iris}. Beware that this is a special feature unique to the left-hand-side of \texttt{fixest} estimations (the default behavior of \texttt{..("regex")} is to aggregate with sums, see \texttt{xpd}).

### Dot square bracket operator in formulas

In a formula, the dot square bracket (DSB) operator can: i) create manifold variables at once, or ii) capture values from the current environment and put them verbatim in the formula.

Say you want to include the variables \texttt{x1} to \texttt{x3} in your formula. You can use \texttt{xpd(y ~ x.[1:3])} and you'll get \texttt{y ~ x1 + x2 + x3}.

To summon values from the environment, simply put the variable in square brackets. For example: \texttt{for(i in 1:3) xpd(y.[i] ~ x)} will create the formulas \texttt{y1 ~ x} to \texttt{y3 ~ x} depending on the value of \texttt{i}.

You can include a full variable from the environment in the same way: \texttt{for(y in c("a","b")) xpd(.[y] ~ x)} will create the two formulas \texttt{a ~ x} and \texttt{b ~ x}.

The DSB can even be used within variable names, but then the variable must be nested in character form. For example \texttt{y ~ ["x.[1:2]_sq"]} will create \texttt{y ~ x1_sq + x2_sq}. Using the character form is important to avoid a formula parsing error. Double quotes must be used. Note that the character string that is nested will be parsed with the function \texttt{dsb}, and thus it will return a vector.

By default, the DSB operator expands vectors into sums. You can add a comma, like in \texttt{. [, x]}, to expand with commas—the content can then be used within functions. For instance: \texttt{c(x.[, 1:2])} will create \texttt{c(x1, x2)} (and not \texttt{c(x1 + x2)}).

In all \texttt{fixest} estimations, this special parsing is enabled, so you don’t need to use \texttt{xpd}.

One-sided formulas can be expanded with the DSB operator: let \texttt{x = ~sepal + petal}, then \texttt{xpd(y ~ .[x])} leads to \texttt{color ~ sepal + petal}.

You can even use multiple square brackets within a single variable, but then the use of nesting is required. For example, the following \texttt{xpd(y ~ .["[letters[1:2]_].[1:2]""])} will create \texttt{y ~ a_1 + b_2}. Remember that the nested character string is parsed with \texttt{dsb}, which explains this behavior.

When the element to be expanded i) is equal to the empty string or, ii) is of length 0, it is replaced with a neutral element, namely \texttt{1}. For example, \texttt{x = ""} ; \texttt{xpd(y ~ .[x])} leads to \texttt{y ~ 1}.

### Author(s)

Laurent Berge
femlm

References


For models with multiple fixed-effects:

Gaure, Simen, 2013, "OLS with multiple high dimensional category variables", Computational Statistics & Data Analysis 66 pp. 8–18

On the unconditional Negative Binomial model:


See Also

See also summary.fixest to see the results with the appropriate standard-errors, fixef.fixest to extract the fixed-effects coefficients, and the function etable to visualize the results of multiple estimations. And other estimation methods: feols, feglm, fepois, feNmlm.

Examples

# Load trade data
data(trade)

# We estimate the effect of distance on trade => we account for 3 fixed-effects
# 1) Poisson estimation
est_pois = femlm(Euros ~ log(dist_km) | Origin + Destination + Product, trade)

# 2) Log-Log Gaussian estimation (with same FEs)
est_gaus = update(est_pois, log(Euros+1) ~ ., family = "gaussian")

# Comparison of the results using the function etable
etable(est_pois, est_gaus)
# Now using two way clustered standard-errors
etable(est_pois, est_gaus, se = "twoway")

# Comparing different types of standard errors
sum_hetero = summary(est_pois, se = "hetero")
sum_oneway = summary(est_pois, se = "cluster")
sum_twoway = summary(est_pois, se = "twoway")
sum_threeway = summary(est_pois, se = "threeway")
etable(sum_hetero, sum_oneway, sum_twoway, sum_threeway)

# # Multiple estimations:
#
# # 6 estimations
est_mult = femlm(c(Ozone, Solar.R) ~ Wind + Temp + csw0(Wind:Temp, Day), airquality)
# We can display the results for the first lhs:
etable(est_mult[lhs = 1])

# And now the second (access can be made by name)
etable(est_mult[lhs = "Solar.R"])

# Now we focus on the two last right hand sides
# (note that .N can be used to specify the last item)
etable(est_mult[rhs = 2:.N])

# Combining with split
est_split = fepois(c(Ozone, Solar.R) - sw(poly(Wind, 2), poly(Temp, 2)),
                   airquality, split = ~ Month)

# You can display everything at once with the print method
est_split

# Different way of displaying the results with "compact"
summary(est_split, "compact")

# You can still select which sample/LHS/RHS to display
est_split[sample = 1:2, lhs = 1, rhs = 1]

---

### feNmlm

**Fixed effects nonlinear maximum likelihood models**

### Description

This function estimates maximum likelihood models (e.g., Poisson or Logit) with non-linear in parameters right-hand-sides and is efficient to handle any number of fixed effects. If you do not use non-linear in parameters right-hand-side, use `femlm` or `feglm` instead (their design is simpler).

### Usage

```r
defnmmlm(
  fml, 
  data, 
  family = c("poisson", "negbin", "logit", "gaussian"), 
  NL.fml, 
  vcov, 
  fixef, 
  fixef.rm = "perfect", 
  NL.start, 
  lower, 
  upper, 
  # other arguments...
)
```
feNmlm

Arguments

fml A formula. This formula gives the linear formula to be estimated (it is similar to a lm formula), for example: fml = z~x+y. To include fixed-effects variables, insert them in this formula using a pipe (e.g. fml = z~x+y|fixef_1+fixef_2). To include a non-linear in parameters element, you must use the argument NL.fml. Multiple estimations can be performed at once: for multiple dep. vars, wrap them in c(): ex c(y1, y2). For multiple indep. vars, use the stepwise functions: ex x1 + csw(x2, x3). This leads to 6 estimation fml = c(y1, y2) ~ x1 + cw0(x2, x3). See details. Square brackets starting with a dot can be used to call global variables: y.[i] ~ x.[i:2] will lead to y3 ~ x1 + x2 if i is equal to 3 in the current environment (see details in xpd).

data A data.frame containing the necessary variables to run the model. The variables of the non-linear right hand side of the formula are identified with this
data.frame names. Can also be a matrix.

**family**
Character scalar. It should provide the family. The possible values are "poisson" (Poisson model with log-link, the default), "negbin" (Negative Binomial model with log-link), "logit" (LOGIT model with log-link), "gaussian" (Gaussian model).

**NL.fml**
A formula. If provided, this formula represents the non-linear part of the right hand side (RHS). Note that contrary to the `fml` argument, the coefficients must explicitly appear in this formula. For instance, it can be \( -a \cdot \log(b \cdot x + c \cdot x^3) \), where \( a \), \( b \), and \( c \) are the coefficients to be estimated. Note that only the RHS of the formula is to be provided, and NOT the left hand side.

**vcov**
Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the `vcov` documentation in the vignette.

**fixef**
Character vector. The names of variables to be used as fixed-effects. These variables should contain the identifier of each observation (e.g., think of it as a panel identifier). Note that the recommended way to include fixed-effects is to insert them directly in the formula.

**fixef.rm**
Can be equal to "perfect" (default), "singleton", "both" or "none". Controls which observations are to be removed. If "perfect", then observations having a fixed-effect with perfect fit (e.g., only 0 outcomes in Poisson estimations) will be removed. If "singleton", all observations for which a fixed-effect appears only once will be removed. The meaning of "both" and "none" is direct.

**NL.start**
(For NL models only) A list of starting values for the non-linear parameters. ALL the parameters are to be named and given a staring value. Example: `NL.start=list(a=1,b=5,c=0)`. Though, there is an exception: if all parameters are to be given the same starting value, you can use a numeric scalar.

**lower**
(For NL models only) A list. The lower bound for each of the non-linear parameters that requires one. Example: `lower=list(b=0,c=0)`. Beware, if the estimated parameter is at its lower bound, then asymptotic theory cannot be applied and the standard-error of the parameter cannot be estimated because the gradient will not be null. In other words, when at its upper/lower bound, the parameter is considered as 'fixed'.

**upper**
(For NL models only) A list. The upper bound for each of the non-linear parameters that requires one. Example: `upper=list(a=10,c=50)`. Beware, if the estimated parameter is at its upper bound, then asymptotic theory cannot be applied and the standard-error of the parameter cannot be estimated because the gradient will not be null. In other words, when at its upper/lower bound, the parameter is considered as 'fixed'.

**NL.start.init**
(For NL models only) Numeric scalar. If the argument `NL.start` is not provided, or only partially filled (i.e. there remain non-linear parameters with no starting value), then the starting value of all remaining non-linear parameters is set to `NL.start.init`. 
offset
A formula or a numeric vector. An offset can be added to the estimation. If equal to a formula, it should be of the form (for example) \( \sim 0.5*x**2 \). This offset is linearly added to the elements of the main formula ‘fml’.

subset
A vector (logical or numeric) or a one-sided formula. If provided, then the estimation will be performed only on the observations defined by this argument.

split
A one-sided formula representing a variable (eg split = \( \sim \)var) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. If you also want to include the estimation for the full sample, use the argument fsplit instead. You can use the special operators %keep% and %drop% to select only a subset of values for which to split the sample. E.g. split = \( \sim \)var %keep% c("v1", "v2") will split the sample only according to the values v1 and v2 of the variable var; is equivalent to supplying the argument split.keep = c("v1", "v2"). By default there is partial matching on each value, you can trigger a regular expression evaluation by adding a ‘@’ first, as in: \( \sim \)var %drop% "@^v[12]" which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

fsplit
A one-sided formula representing a variable (eg split = \( \sim \)var) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. This argument is the same as split but also includes the full sample as the first estimation. You can use the special operators %keep% and %drop% to select only a subset of values for which to split the sample. E.g. split = \( \sim \)var %keep% c("v1", "v2") will split the sample only according to the values v1 and v2 of the variable var; is equivalent to supplying the argument split.keep = c("v1", "v2"). By default there is partial matching on each value, you can trigger a regular expression evaluation by adding an ‘@’ first, as in: \( \sim \)var %drop% "@^v[12]" which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

split.keep
A character vector. Only used when split, or fsplit, is supplied. If provided, then the sample will be split only on the values of split.keep. The values in split.keep will be partially matched to the values of split. To enable regular expressions, you need to add an ‘@’ first. For example split.keep = c("v1", "@other|var") will keep only the value in split partially matched by "v1" or the values containing "other" or "var".

split.drop
A character vector. Only used when split, or fsplit, is supplied. If provided, then the sample will be split only on the values that are not in split.drop. The values in split.drop will be partially matched to the values of split. To enable regular expressions, you need to add an ‘@’ first. For example split.drop = c("v1", @other|var") will drop only the value in split partially matched by "v1" or the values containing "other" or "var".

cluster
Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2").cluster = \( \sim \)var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp.
You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

se Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "threeway" or "fourway"? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

ssc An object of class ssc.type obtained with the function ssc. Represents how the degree of freedom correction should be done. You must use the function ssc for this argument. The arguments and defaults of the function ssc are: adj = TRUE, fixef.K = "nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact = FALSE). See the help of the function ssc for details.

panel.id The panel identifiers. Can either be: i) a one sided formula (e.g. panel.id = ~id+time), ii) a character vector of length 2 (e.g. panel.id = c('id', 'time'), or iii) a character scalar of two variables separated by a comma (e.g. panel.id = 'id,time'). Note that you can combine variables with ^ only inside formulas (see the dedicated section in feols).

start Starting values for the coefficients in the linear part (for the non-linear part, use NL.start). Can be: i) a numeric of length 1 (e.g. start = 0, the default), ii) a numeric vector of the exact same length as the number of variables, or iii) a named vector of any length (the names will be used to initialize the appropriate coefficients).

jacobian.method (For NL models only) Character scalar. Provides the method used to numerically compute the Jacobian of the non-linear part. Can be either "simple" or "Richardson". Default is "simple". See the help of jacobian for more information.

useHessian Logical. Should the Hessian be computed in the optimization stage? Default is TRUE.

hessian.args List of arguments to be passed to function genD. Defaults is missing. Only used with the presence of NL.fml.

opt.control List of elements to be passed to the optimization method nlminb. See the help page of nlminb for more information.

nthreads The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function setFixest_nthreads.

lean Logical, default is FALSE. If TRUE then all large objects are removed from the returned result: this will save memory but will block the possibility to use many methods. It is recommended to use the arguments se or cluster to obtain the appropriate standard-errors at estimation time, since obtaining different SEs won’t be possible afterwards.

verbose Integer, default is 0. It represents the level of information that should be reported during the optimisation process. If verbose=0: nothing is reported. If
verbose=1: the value of the coefficients and the likelihood are reported. If verbose=2: 1 + information on the computing time of the null model, the fixed-effects coefficients and the hessian are reported.

**theta.init**
Positive numeric scalar. The starting value of the dispersion parameter if family="negbin". By default, the algorithm uses as a starting value the theta obtained from the model with only the intercept.

**fixef.tol**
Precision used to obtain the fixed-effects. Defaults to 1e-5. It corresponds to the maximum absolute difference allowed between two coefficients of successive iterations. Argument fixef.tol cannot be lower than 10000*.Machine$double.eps.

**fixef.iter**
Maximum number of iterations in fixed-effects algorithm (only in use for 2+ fixed-effects). Default is 10000.

**deriv.tol**
Precision used to obtain the fixed-effects derivatives. Defaults to 1e-4. It corresponds to the maximum absolute difference allowed between two coefficients of successive iterations. Argument deriv.tol cannot be lower than 10000*.Machine$double.eps.

**deriv.iter**
Maximum number of iterations in the algorithm to obtain the derivative of the fixed-effects (only in use for 2+ fixed-effects). Default is 1000.

**warn**
Logical, default is TRUE. Whether warnings should be displayed (concerns warnings relating to convergence state).

**notes**
Logical. By default, two notes are displayed: when NAs are removed (to show additional information) and when some observations are removed because of only 0 (or 0/1) outcomes in a fixed-effect setup (in Poisson/Neg. Bin./Logit models). To avoid displaying these messages, you can set notes = FALSE. You can remove these messages permanently by using setFixest_notes(FALSE).

**combine.quick**
Logical. When you combine different variables to transform them into a single fixed-effects you can do e.g. y ~ x | paste(var1, var2). The algorithm provides a shorthand to do the same operation: y ~ x | var1*var2. Because pasting variables is a costly operation, the internal algorithm may use a numerical trick to hasten the process. The cost of doing so is that you lose the labels. If you are interested in getting the value of the fixed-effects coefficients after the estimation, you should use combine.quick = FALSE. By default it is equal to FALSE if the number of observations is lower than 50,000, and to TRUE otherwise.

**mem.clean**
Logical, default is FALSE. Only to be used if the data set is large compared to the available RAM. If TRUE then intermediary objects are removed as much as possible and gc is run before each substantial C++ section in the internal code to avoid memory issues.

**only.env**
(Advanced users.) Logical, default is FALSE. If TRUE, then only the environment used to make the estimation is returned.

**only.coef**
Logical, default is FALSE. If TRUE, then only the estimated coefficients are returned. Note that the length of the vector returned is always the length of the number of coefficients to be estimated: this means that the variables found to be collinear are returned with an NA value.

**env**
(Advanced users.) A fixest environment created by a fixest estimation with only.env = TRUE. Default is missing. If provided, the data from this environment will be used to perform the estimation.
Details

This function estimates maximum likelihood models where the conditional expectations are as follows:

Gaussian likelihood:
\[ E(Y|X) = X\beta \]

Poisson and Negative Binomial likelihoods:
\[ E(Y|X) = \exp(X\beta) \]

where in the Negative Binomial there is the parameter \( \theta \) used to model the variance as \( \mu + \mu^2/\theta \), with \( \mu \) the conditional expectation. Logit likelihood:
\[ E(Y|X) = \frac{\exp(X\beta)}{1 + \exp(X\beta)} \]

When there are one or more fixed-effects, the conditional expectation can be written as:
\[ E(Y|X) = h(X\beta + \sum_k \sum_m \gamma_{km} \times C_{km}), \]

where \( h(,.) \) is the function corresponding to the likelihood function as shown before. \( C^k \) is the matrix associated to fixed-effect dimension \( k \) such that \( C^k_{im} \) is equal to 1 if observation \( i \) is of category \( m \) in the fixed-effect dimension \( k \) and 0 otherwise.

When there are non linear in parameters functions, we can schematically split the set of regressors in two:
\[ f(X,\beta) = X^1\beta^1 + g(X^2,\beta^2) \]

with first a linear term and then a non linear part expressed by the function \( g \). That is, we add a non-linear term to the linear terms (which are \( X \ast \text{beta} \) and the fixed-effects coefficients). It is always better (more efficient) to put into the argument \( \text{NL.fml} \) only the non-linear in parameter terms, and add all linear terms in the \( \text{fml} \) argument.

To estimate only a non-linear formula without even the intercept, you must exclude the intercept from the linear formula by using, e.g., \( \text{fml} = \text{z} \sim 0 \).

The over-dispersion parameter of the Negative Binomial family, \( \theta \), is capped at 10,000. If \( \theta \) reaches this high value, it means that there is no overdispersion.

Value

A \texttt{fixest} object. Note that \texttt{fixest} objects contain many elements and most of them are for internal use, they are presented here only for information. To access them, it is safer to use the user-level methods (e.g. \texttt{vcov.fixest}, \texttt{resid.fixest}, etc) or functions (like for instance \texttt{fitstat} to access any fit statistic).

\begin{itemize}
  \item \texttt{coefficients} \hspace{1cm} The named vector of coefficients.
  \item \texttt{coeftable} \hspace{1cm} The table of the coefficients with their standard errors, z-values and p-values.
  \item \texttt{loglik} \hspace{1cm} The loglikelihood.
\end{itemize}
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterations</td>
<td>Number of iterations of the algorithm.</td>
</tr>
<tr>
<td>nob</td>
<td>The number of observations.</td>
</tr>
<tr>
<td>nparams</td>
<td>The number of parameters of the model.</td>
</tr>
<tr>
<td>call</td>
<td>The call.</td>
</tr>
<tr>
<td>fml</td>
<td>The linear formula of the call.</td>
</tr>
<tr>
<td>fml_all</td>
<td>A list containing different parts of the formula. Always contain the linear formula. Then, if relevant: fixef: the fixed-effects; NL: the non linear part of the formula.</td>
</tr>
<tr>
<td>ll_null</td>
<td>Log-likelihood of the null model (i.e. with the intercept only).</td>
</tr>
<tr>
<td>pseudo_r2</td>
<td>The adjusted pseudo R2.</td>
</tr>
<tr>
<td>message</td>
<td>The convergence message from the optimization procedures.</td>
</tr>
<tr>
<td>sq.cor</td>
<td>Squared correlation between the dependent variable and the expected predictor (i.e. fitted.values) obtained by the estimation.</td>
</tr>
<tr>
<td>hessian</td>
<td>The Hessian of the parameters.</td>
</tr>
<tr>
<td>fitted.values</td>
<td>The fitted values are the expected value of the dependent variable for the fitted model: that is $E(Y</td>
</tr>
<tr>
<td>cov.iid</td>
<td>The variance-covariance matrix of the parameters.</td>
</tr>
<tr>
<td>se</td>
<td>The standard-error of the parameters.</td>
</tr>
<tr>
<td>scores</td>
<td>The matrix of the scores (first derivative for each observation).</td>
</tr>
<tr>
<td>family</td>
<td>The ML family that was used for the estimation.</td>
</tr>
<tr>
<td>residuals</td>
<td>The difference between the dependent variable and the expected predictor.</td>
</tr>
<tr>
<td>sumFE</td>
<td>The sum of the fixed-effects for each observation.</td>
</tr>
<tr>
<td>offset</td>
<td>The offset formula.</td>
</tr>
<tr>
<td>NL.fml</td>
<td>The nonlinear formula of the call.</td>
</tr>
<tr>
<td>bounds</td>
<td>Whether the coefficients were upper or lower bounded. – This can only be the case when a non-linear formula is included and the arguments 'lower' or 'upper' are provided.</td>
</tr>
<tr>
<td>isBounded</td>
<td>The logical vector that gives for each coefficient whether it was bounded or not. This can only be the case when a non-linear formula is included and the arguments 'lower' or 'upper' are provided.</td>
</tr>
<tr>
<td>fixef_vars</td>
<td>The names of each fixed-effect dimension.</td>
</tr>
<tr>
<td>fixef_id</td>
<td>The list (of length the number of fixed-effects) of the fixed-effects identifiers for each observation.</td>
</tr>
<tr>
<td>fixef_sizes</td>
<td>The size of each fixed-effect (i.e. the number of unique identifier for each fixed-effect dimension).</td>
</tr>
<tr>
<td>obs_selection</td>
<td>(When relevant.) List containing vectors of integers. It represents the sequential selection of observation vis a vis the original data set.</td>
</tr>
<tr>
<td>fixef_removed</td>
<td>In the case there were fixed-effects and some observations were removed because of only 0/1 outcome within a fixed-effect, it gives the list (for each fixed-effect dimension) of the fixed-effect identifiers that were removed.</td>
</tr>
</tbody>
</table>
theta  

In the case of a negative binomial estimation: the overdispersion parameter.

@seealso See also `summary.fixest` to see the results with the appropriate standard-errors, `fixef.fixest` to extract the fixed-effects coefficients, and the function `etable` to visualize the results of multiple estimations.

And other estimation methods: `feols`, `femlm`, `feglm`, `fepois`, `fenegbin`.

Lagging variables

To use leads/lags of variables in the estimation, you can: i) either provide the argument `panel.id`, ii) either set your data as a panel with the function `panel`, `f` and `d`.

You can provide several leads/lags/differences at once: e.g. if your formula is equal to \( f(y) \sim 1(x, -1:1) \), it means that the dependent variable is equal to the lead of \( y \), and you will have as explanatory variables the lead of \( x_1 \), \( x_1 \) and the lag of \( x_1 \). See the examples in function `l` for more details.

Interactions

You can interact a numeric variable with a "factor-like" variable by using `i(factor_var, continuous_var, ref)`, where `continuous_var` will be interacted with each value of `factor_var` and the argument `ref` is a value of `factor_var` taken as a reference (optional).

Using this specific way to create interactions leads to a different display of the interacted values in `etable`. See examples.

It is important to note that if you do not care about the standard-errors of the interactions, then you can add interactions in the fixed-effects part of the formula, it will be incomparably faster (using the syntax `factor_var[continuous_var]`, as explained in the section “Varying slopes”).

The function `i` has in fact more arguments, please see details in its associated help page.

On standard-errors

Standard-errors can be computed in different ways, you can use the arguments `se` and `ssc` in `summary.fixest` to define how to compute them. By default, in the presence of fixed-effects, standard-errors are automatically clustered.

The following vignette: On standard-errors describes in details how the standard-errors are computed in `fixest` and how you can replicate standard-errors from other software.

You can use the functions `setFixest_vcov` and `setFixest_ssc` to permanently set the way the standard-errors are computed.

Multiple estimations

Multiple estimations can be performed at once, they just have to be specified in the formula. Multiple estimations yield a `fixest_multi` object which is ‘kind of’ a list of all the results but includes specific methods to access the results in a handy way. Please have a look at the dedicated vignette: Multiple estimations.

To include multiple dependent variables, wrap them in `c()` (`list()` also works). For instance `fml = c(y1, y2) ~ x1` would estimate the model `fml = y1 ~ x1` and then the model `fml = y2 ~ x1`. 
To include multiple independent variables, you need to use the stepwise functions. There are 4 stepwise functions: \texttt{sw}, \texttt{sw0}, \texttt{csw}, \texttt{csw0}, and \texttt{mvsw}. Of course \texttt{sw} stands for stepwise, and \texttt{csw} for cumulative stepwise. Finally \texttt{mvsw} is a bit special, it stands for multiverse stepwise. Let’s explain that. Assume you have the following formula: \texttt{fml = y \sim x1 + sw(x2, x3)}. The stepwise function \texttt{sw} will estimate the following two models: \texttt{y \sim x1 + x2} and \texttt{y \sim x1 + x3}. That is, each element in \texttt{sw()} is sequentially, and separately, added to the formula. Would have you used \texttt{sw0} in lieu of \texttt{sw}, then the model \texttt{y \sim x1} would also have been estimated. The \texttt{0} in the name means that the model without any stepwise element also needs to be estimated. The prefix \texttt{c} means cumulative: each stepwise element is added to the next. That is, \texttt{fml = y \sim x1 + csw(x2, x3)} would lead to the following models \texttt{y \sim x1 + x2} and \texttt{y \sim x1 + x2 + x3}. The \texttt{0} has the same meaning and would also lead to the model without the stepwise elements to be estimated: in other words, \texttt{fml = y \sim x1 + csw0(x2, x3)} leads to the following three models: \texttt{y \sim x1}, \texttt{y \sim x1 + x2} and \texttt{y \sim x1 + x2 + x3}. Finally \texttt{mvsw} will add, in a stepwise fashion all possible combinations of the variables in its arguments. For example \texttt{mvsw(x1, x2, x3)} is equivalent to \texttt{sw0(x1, x2, x3, x1 + x2, x1 + x3, x2 + x3, x1 + x2 + x3)}. The number of models to estimate grows at a factorial rate: so be cautious!

Multiple independent variables can be combined with multiple dependent variables, as in \texttt{fml = c(y1, y2) \sim cw(x1, x2, x3)} which would lead to 6 estimations. Multiple estimations can also be combined to split samples (with the arguments \texttt{split}, \texttt{fsplit}).

You can also add fixed-effects in a stepwise fashion. Note that you cannot perform stepwise estimations on the IV part of the formula (\texttt{feols} only).

If NAs are present in the sample, to avoid too many messages, only NA removal concerning the variables common to all estimations is reported.

A note on performance. The feature of multiple estimations has been highly optimized for \texttt{feols}, in particular in the presence of fixed-effects. It is faster to estimate multiple models using the formula rather than with a loop. For non-\texttt{feols} models using the formula is roughly similar to using a loop performance-wise.

**Argument sliding**

When the data set has been set up globally using \texttt{setFixest_estimation(data = data_set)}, the argument \texttt{vcov} can be used implicitly. This means that calls such as \texttt{feols(y \sim x, "HC1")}, or \texttt{feols(y \sim x, -id)}, are valid: i) the data is automatically deduced from the global settings, and ii) the \texttt{vcov} is deduced to be the second argument.

**Piping**

Although the argument ‘data’ is placed in second position, the data can be piped to the estimation functions. For example, with R >= 4.1, \texttt{mtcars |> feols(mpg \sim cyl)} works as \texttt{feols(mpg \sim cyl, mtcars)}.

**Tricks to estimate multiple LHS**

To use multiple dependent variables in \texttt{fixest} estimations, you need to include them in a vector: like in \texttt{c(y1, y2, y3)}.

First, if names are stored in a vector, they can readily be inserted in a formula to perform multiple estimations using the dot square bracket operator. For instance if \texttt{my_lhs = c("y1", "y2")}, calling \texttt{fixest} with, say \texttt{feols(.[my_lhs] \sim x1, etc)} is equivalent to using \texttt{feols(c(y1, y2) \sim x1, etc)}. 
etc). Beware that this is a special feature unique to the *left-hand-side* of *f*ix*est* estimations (the default behavior of the DSB operator is to aggregate with sums, see `xpd`).

Second, you can use a regular expression to grep the left-hand-sides on the fly. When the `..("regex")` feature is used naked on the LHS, the variables grepped are inserted into `c()`. For example `..("Pe") ~ Sepal.Length, iris` is equivalent to `c(Petal.Length, Petal.Width) ~ Sepal.Length, iris`. Beware that this is a special feature unique to the *left-hand-side* of *f*ix*est* estimations (the default behavior of `..("regex")` is to aggregate with sums, see `xpd`).

**Dot square bracket operator in formulas**

In a formula, the dot square bracket (DSB) operator can: i) create manifold variables at once, or ii) capture values from the current environment and put them verbatim in the formula.

Say you want to include the variables `x1` to `x3` in your formula. You can use `xpd(y ~ x.[1:3])` and you’ll get `y ~ x1 + x2 + x3`.

To summon values from the environment, simply put the variable in square brackets. For example:

```
for(i in 1:3) xpd(y.[i] ~ x)
```

will create the formulas `y1 ~ x` to `y3 ~ x` depending on the value of `i`.

You can include a full variable from the environment in the same way: `for(y in c("a", "b")) xpd(.[y] ~ x)` will create the two formulas `a ~ x` and `b ~ x`.

The DSB can even be used within variable names, but then the variable must be nested in character form. For example `y ~ .["x.[1:2]_sq"]` will create `y ~ x1_sq + x2_sq`. Using the character form is important to avoid a formula parsing error. Double quotes must be used. Note that the character string that is nested will be parsed with the function `dsb`, and thus it will return a vector.

By default, the DSB operator expands vectors into sums. You can add a comma, like in `.[, x]`, to expand with commas—the content can then be used within functions. For instance: `c(x.[1:2])` will create `c(x1, x2)` (and *not* `c(x1 + x2)`).

In all *f*ix*est* estimations, this special parsing is enabled, so you don’t need to use `xpd`.

One-sided formulas can be expanded with the DSB operator: let `x = ~sepal + petal`, then `xpd(y ~ .[x])` leads to `color ~ sepal + petal`.

You can even use multiple square brackets within a single variable, but then the use of nesting is required. For example, the following `xpd(y ~ .[".[letters[1:2]]..[1:2]"])` will create `y ~ a_1 + b_2`. Remember that the nested character string is parsed with `dsb`, which explains this behavior.

When the element to be expanded i) is equal to the empty string or, ii) is of length 0, it is replaced with a neutral element, namely 1. For example, `x = "" ; xpd(y ~ .[x])` leads to `y ~ 1`.

**Author(s)**

Laurent Berge

**References**


For models with multiple fixed-effects:
Gaure, Simen, 2013, "OLS with multiple high dimensional category variables", Computational Statistics & Data Analysis 66 pp. 8–18

On the unconditional Negative Binomial model:


Examples

# This section covers only non-linear in parameters examples
# For linear relationships: use femlm or feglm instead

# Generating data for a simple example
set.seed(1)
n = 100
x = rnorm(n, 1, 5)**2
y = rnorm(n, -1, 5)**2
z1 = rpois(n, x*y) + rpois(n, 2)
base = data.frame(x, y, z1)

# Estimating a 'linear' relation:
est1_L = femlm(z1 ~ log(x) + log(y), base)
# Estimating the same 'linear' relation using a 'non-linear' call
est1_NL = feNmlm(z1 ~ 1, base, NL.fml = ~a*log(x)+b*log(y), NL.start = list(a=0, b=0))
# we compare the estimates with the function esttable (they are identical)
estable(est1_L, est1_NL)

# Now generating a non-linear relation (E(z2) = x + y + 1):
z2 = rpois(n, x + y) + rpois(n, 1)
base$z2 = z2

# Estimation using this non-linear form
est2_NL = feNmlm(z2 ~ 0, base, NL.fml = ~log(a*x + b*y),
                 NL.start = 2, lower = list(a=0, b=0))
# we can't estimate this relation linearly
# => closest we can do:
est2_L = femlm(z2 ~ log(x) + log(y), base)

# Difference between the two models:
estable(est2_L, est2_NL)

# Plotting the fits:
plot(x, z2, pch = 18)
points(x, fitted(est2_L), col = 2, pch = 1)
points(x, fitted(est2_NL), col = 4, pch = 2)
Description

Estimates OLS with any number of fixed-effects.

Usage

feols(
  fml,
  data,
  vcov,
  weights,
  offset,
  subset,
  split,
  fsplit,
  split.keep,
  split.drop,
  cluster,
  se,
  ssc,
  panel.id,
  fixef,
  fixef.rm = "none",
  fixef.tol = 1e-06,
  fixef.iter = 10000,
  collin.tol = 1e-10,
  nthreads = getFixest_nthreads(),
  lean = FALSE,
  verbose = 0,
  warn = TRUE,
  notes = getFixest_notes(),
  only.coef = FALSE,
  combine.quick,
  demeaned = FALSE,
  mem.clean = FALSE,
  only.env = FALSE,
  env,
  ...
)

feols.fit(
  y,
  X,
  fixef_df,
Arguments

**fml**

A formula representing the relation to be estimated. For example: `fml = z ~ x + y`. To include fixed-effects, insert them in this formula using a pipe: e.g. `fml = z ~ x + y | fe_1 + fe_2`. You can combine two fixed-effects with `^`: e.g. `fml = z ~ x | fe_1^fe_2`, see details. You can also use variables with varying slopes using square brackets: e.g. in `fml = z ~ y | fe_1[x] + fe_2`, see details. To add IVs, insert the endogenous vars./instruments after a pipe, like in `y ~ x | x_endo1 + x_endo2 ~ x_inst1 + x_inst2`. Note that it should always be the last element, see details. Multiple estimations can be performed at once: for multiple dep. vars, wrap them in `c()`: `ex c(y1, y2)`. For multiple indep. vars, use the stepwise functions: `ex x1 + csw(x2, x3)`. The formula `fml = c(y1, y2) ~ x1 + cw0(x2, x3)` leads to 6 estimation, see details. Square brackets starting with a dot can be used to call global variables: `y.[i] ~ x.[1:2]` will lead to `y3 ~ x1 + x2` if `i` is equal to 3 in the current environment (see details in `xpd`).

**data**

A data.frame containing the necessary variables to run the model. The variables of the non-linear right hand side of the formula are identified with this data.frame names. Can also be a matrix.

**vcov**

Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "three-way". A formula of the form: `vcov_type ~ variables` can be used.
"NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the `vcov` documentation in the vignette.

weights A formula or a numeric vector. Each observation can be weighted. The weights must be greater than 0. If equal to a formula, it should be one-sided: for example `~ var_weight`.

offset A formula or a numeric vector. An offset can be added to the estimation. If equal to a formula, it should be of the form (for example) `~0.5*x**2`. This offset is linearly added to the elements of the main formula 'fml'.

subset A vector (logical or numeric) or a one-sided formula. If provided, then the estimation will be performed only on the observations defined by this argument.

split A one sided formula representing a variable (eg `split = ~var`) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. If you also want to include the estimation for the full sample, use the argument `fsplit` instead. You can use the special operators `%keep%` and `%drop%` to select only a subset of values for which to split the sample. E.g. `split = ~var %keep% c("v1", "v2")` will split the sample only according to the values `v1` and `v2` of the variable `var`; it is equivalent to supplying the argument `split.keep = c("v1", "v2")`. By default there is partial matching on each value, you can trigger a regular expression evaluation by adding a ' @' first, as in: `~var %drop% '@^v[12]'` which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

fsplit A one sided formula representing a variable (eg `split = ~var`) or a vector. If provided, the sample is split according to the variable and one estimation is performed for each value of that variable. This argument is the same as `split` but also includes the full sample as the first estimation. You can use the special operators `%keep%` and `%drop%` to select only a subset of values for which to split the sample. E.g. `split = ~var %keep% c("v1", "v2")` will split the sample only according to the values `v1` and `v2` of the variable `var`; it is equivalent to supplying the argument `split.keep = c("v1", "v2")`. By default there is partial matching on each value, you can trigger a regular expression evaluation by adding an ' @' first, as in: `~var %drop% '@^v[12]'` which will drop values starting with "v1" or "v2" (of course you need to know regexes!).

split.keep A character vector. Only used when `split`, or `fsplit`, is supplied. If provided, then the sample will be split only on the values of `split.keep`. The values in `split.keep` will be partially matched to the values of `split`. To enable regular expressions, you need to add an '@' first. For example `split.keep = c("v1", "@other|var")` will keep only the value in `split` partially matched by "v1" or the values containing "other" or "var".

split.drop A character vector. Only used when `split`, or `fsplit`, is supplied. If provided, then the sample will be split only on the values that are not in `split.drop`. The values in `split.drop` will be partially matched to the values of `split`. To enable regular expressions, you need to add an '@' first. For example `split.drop = c("v1", "@other|var")` will drop only the value in `split` partially matched by "v1" or the values containing "other" or "var".
cluster

Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

se

Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "three-way" or "four-way"? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

ssc

An object of class ssc.type obtained with the function ssc. Represents how the degree of freedom correction should be done. You must use the function ssc for this argument. The arguments and defaults of the function ssc are: adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE). See the help of the function ssc for details.

panel.id

The panel identifiers. Can either be: i) a one sided formula (e.g. panel.id = ~id+time), ii) a character vector of length 2 (e.g. panel.id=c('id', 'time'), or iii) a character scalar of two variables separated by a comma (e.g. panel.id='id,time'). Note that you can combine variables with ^ only inside formulas (see the dedicated section infeols).

fixef

Character vector. The names of variables to be used as fixed-effects. These variables should contain the identifier of each observation (e.g., think of it as a panel identifier). Note that the recommended way to include fixed-effects is to insert them directly in the formula.

fixef.rm

Can be equal to "perfect" (default), "singleton", "both" or "none". Controls which observations are to be removed. If "perfect", then observations having a fixed-effect with perfect fit (e.g. only 0 outcomes in Poisson estimations) will be removed. If "singleton", all observations for which a fixed-effect appears only once will be removed. The meaning of "both" and "none" is direct.

fixef.tol

Precision used to obtain the fixed-effects. Defaults to 1e-5. It corresponds to the maximum absolute difference allowed between two coefficients of successive iterations. Argument fixef.tol cannot be lower than 10000*.Machine$double.eps. Note that this parameter is dynamically controlled by the algorithm.

fixef.iter

Maximum number of iterations in fixed-effects algorithm (only in use for 2+ fixed-effects). Default is 10000.

collin.tol

Numeric scalar, default is 1e-10. Threshold deciding when variables should be considered collinear and subsequently removed from the estimation. Higher values means more variables will be removed (if there is presence of collinearity). One signal of presence of collinearity is t-stats that are extremely low (for instance when t-stats < 1e-3).
**nthreads**
The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function `setFixest_nthreads`.

**lean**
Logical, default is `FALSE`. If `TRUE` then all large objects are removed from the returned result: this will save memory but will block the possibility to use many methods. It is recommended to use the arguments `se` or `cluster` to obtain the appropriate standard-errors at estimation time, since obtaining different SEs won’t be possible afterwards.

**verbose**
Integer. Higher values give more information. In particular, it can detail the number of iterations in the demeaning algorithm (the first number is the left-hand-side, the other numbers are the right-hand-side variables).

**warn**
Logical, default is `TRUE`. Whether warnings should be displayed (concerns warnings relating to convergence state).

**notes**
Logical. By default, two notes are displayed: when NAs are removed (to show additional information) and when some observations are removed because of collinearity. To avoid displaying these messages, you can set `notes = FALSE`. You can remove these messages permanently by using `setFixest_notes(FALSE)`.

**only.coef**
Logical, default is `FALSE`. If `TRUE`, then only the estimated coefficients are returned. Note that the length of the vector returned is always the length of the number of coefficients to be estimated: this means that the variables found to be collinear are returned with an NA value.

**combine.quick**
Logical. When you combine different variables to transform them into a single fixed-effects you can do e.g. `y ~ x | paste(var1, var2)`. The algorithm provides a shorthand to do the same operation: `y ~ x | var1^var2`. Because pasting variables is a costly operation, the internal algorithm may use a numerical trick to hasten the process. The cost of doing so is that you lose the labels. If you are interested in getting the value of the fixed-effects coefficients after the estimation, you should use `combine.quick = FALSE`. By default it is equal to `FALSE` if the number of observations is lower than 50,000, and to `TRUE` otherwise.

**demeaned**
Logical, default is `FALSE`. Only used in the presence of fixed-effects: should the centered variables be returned? If `TRUE`, it creates the items `y_demeaned` and `X_demeaned`.

**mem.clean**
Logical, default is `FALSE`. Only to be used if the data set is large compared to the available RAM. If `TRUE` then intermediary objects are removed as much as possible and `gc` is run before each substantial C++ section in the internal code to avoid memory issues.

**only.env**
(Advanced users.) Logical, default is `FALSE`. If `TRUE`, then only the environment used to make the estimation is returned.

**env**
(Advanced users.) A fixest environment created by a fixest estimation with `only.env = TRUE`. Default is missing. If provided, the data from this environment will be used to perform the estimation.

**...**
Not currently used.

**y**
Numeric vector/matrix/data.frame of the dependent variable(s). Multiple dependent variables will return a `fixest_multi` object.
X  Numeric matrix of the regressors.
fixef_df  Matrix/data.frame of the fixed-effects.

Details

The method used to demean each variable along the fixed-effects is based on Berge (2018), since this is the same problem to solve as for the Gaussian case in a ML setup.

Value

A fixest object. Note that fixest objects contain many elements and most of them are for internal use, they are presented here only for information. To access them, it is safer to use the user-level methods (e.g. vcov.fixest, resid.fixest, etc) or functions (like for instance fitstat to access any fit statistic).

nobs  The number of observations.
fml  The linear formula of the call.
call  The call of the function.
method  The method used to estimate the model.
family  The family used to estimate the model.
fml_all  A list containing different parts of the formula. Always contain the linear formula. Then depending on the cases: fixef: the fixed-effects, iv: the IV part of the formula.
fixef_vars  The names of each fixed-effect dimension.
fixef_id  The list (of length the number of fixed-effects) of the fixed-effects identifiers for each observation.
fixef_sizes  The size of each fixed-effect (i.e. the number of unique identifier for each fixed-effect dimension).
coefficients  The named vector of estimated coefficients.
multicol  Logical, if multicollinearity was found.
coeftable  The table of the coefficients with their standard errors, z-values and p-values.
loglik  The loglikelihood.
ssr_null  Sum of the squared residuals of the null model (containing only with the intercept).
ssr_fe_only  Sum of the squared residuals of the model estimated with fixed-effects only.
ll_null  The log-likelihood of the null model (containing only with the intercept).
ll_fe_only  The log-likelihood of the model estimated with fixed-effects only.
fitted.values  The fitted values.
linear.predictors  The linear predictors.
residuals  The residuals (y minus the fitted values).
sq.cor  Squared correlation between the dependent variable and the expected predictor (i.e. fitted.values) obtained by the estimation.
hessian
   The Hessian of the parameters.

cov.iid
   The variance-covariance matrix of the parameters.

se
   The standard-error of the parameters.

scores
   The matrix of the scores (first derivative for each observation).

residuals
   The difference between the dependent variable and the expected predictor.

sumFE
   The sum of the fixed-effects coefficients for each observation.

offset
   (When relevant.) The offset formula.

weights
   (When relevant.) The weights formula.

obs_selection
   (When relevant.) List containing vectors of integers. It represents the sequential
   selection of observation vis a vis the original data set.

collin.var
   (When relevant.) Vector containing the variables removed because of collinearity.

collin.coef
   (When relevant.) Vector of coefficients, where the values of the variables removed
   because of collinearity are NA.

collin.min_norm
   The minimal diagonal value of the Cholesky decomposition. Small values indicate
   possible presence collinearity.

y_demeaned
   Only when demeaned = TRUE: the centered dependent variable.

X_demeaned
   Only when demeaned = TRUE: the centered explanatory variable.

Combining the fixed-effects

You can combine two variables to make it a new fixed-effect using \(^{\cdot}\). The syntax is as follows: \texttt{fe\_1^{fe\_2}}. Here you created a new variable which is the combination of the two variables \texttt{fe\_1} and \texttt{fe\_2}. This is identical to doing \texttt{paste0(fe\_1, ",", fe\_2)} but more convenient.

Note that pasting is a costly operation, especially for large data sets. Thus, the internal algorithm uses a numerical trick which is fast, but the drawback is that the identity of each observation is lost (i.e. they are now equal to a meaningless number instead of being equal to \texttt{paste0(fe\_1, ",", fe\_2)}). These “identities” are useful only if you’re interested in the value of the fixed-effects (that you can extract with \texttt{fixef.fixest}). If you’re only interested in coefficients of the variables, it doesn’t matter. Anyway, you can use \texttt{combine.quick = FALSE} to tell the internal algorithm to use \texttt{paste} instead of the numerical trick. By default, the numerical trick is performed only for large data sets.

Varying slopes

You can add variables with varying slopes in the fixed-effect part of the formula. The syntax is as follows: \texttt{fixef.var[\texttt{var1, var2}].} Here the variables \texttt{var1} and \texttt{var2} will be with varying slopes (one slope per value in \texttt{fixef.var}) and the fixed-effect \texttt{fixef.var} will also be added.

To add only the variables with varying slopes and not the fixed-effect, use double square brackets: \texttt{fixef.var[[\texttt{var1, var2}].]

In other words:

- \texttt{fixef.var[\texttt{var1, var2}]} is equivalent to \texttt{fixef.var + fixef.var[[\texttt{var1}]] + fixef.var[[\texttt{var2}]]}
• `fixef_var[[var1, var2]]` is equivalent to `fixef_var[[var1]] + fixef_var[[var2]]`

In general, for convergence reasons, it is recommended to always add the fixed-effect and avoid using only the variable with varying slope (i.e. use single square brackets).

Lagging variables

To use leads/lags of variables in the estimation, you can: i) either provide the argument `panel.id`, ii) either set your data set as a panel with the function `panel_f and d`.

You can provide several leads/lags/differences at once: e.g. if your formula is equal to `f(y) ~ 1(x, -1:1)`, it means that the dependent variable is equal to the lead of `y`, and you will have as explanatory variables the lead of `x1`, `x1` and the lag of `x1`. See the examples in function `l` for more details.

Interactions

You can interact a numeric variable with a "factor-like" variable by using `i(factor_var, continuous_var, ref)`, where `continuous_var` will be interacted with each value of `factor_var` and the argument `ref` is a value of `factor_var` taken as a reference (optional).

Using this specific way to create interactions leads to a different display of the interacted values in `etable`. See examples.

It is important to note that if you do not care about the standard-errors of the interactions, then you can add interactions in the fixed-effects part of the formula, it will be incomparably faster (using the syntax `factor_var[continuous_var]`, as explained in the section “Varying slopes”).

The function `i` has in fact more arguments, please see details in its associated help page.

On standard-errors

Standard-errors can be computed in different ways, you can use the arguments `se` and `ssc` in `summary.fixest` to define how to compute them. By default, in the presence of fixed-effects, standard-errors are automatically clustered.

The following vignette: On standard-errors describes in details how the standard-errors are computed in `fixest` and how you can replicate standard-errors from other software.

You can use the functions `setFixest_vcov` and `setFixest_ssc` to permanently set the way the standard-errors are computed.

Instrumental variables

To estimate two stage least square regressions, insert the relationship between the endogenous regressor(s) and the instruments in a formula, after a pipe.

For example, `fml = y ~ x1 | x_endo ~ x_inst` will use the variables `x1` and `x_inst` in the first stage to explain `x_endo`. Then will use the fitted value of `x_endo` (which will be named `fit_x_endo`) and `x1` to explain `y`. To include several endogenous regressors, just use "+", like in: `fml = y ~ x1 | x_endo1 + x_endo2 ~ x_inst1 + x_inst2`.

Of course you can still add the fixed-effects, but the IV formula must always come last, like in `fml = y ~ x1 | fe1 + fe2 | x_endo ~ x_inst`. 
If you want to estimate a model without exogenous variables, use "1" as a placeholder: e.g. \( fml = y \sim 1 | x_{\text{endo}} + x_{\text{inst}} \).

By default, the second stage regression is returned. You can access the first stage(s) regressions either directly in the slot `iv_first_stage` (not recommended), or using the argument `stage = 1` from the function `summary.fixest`. For example `summary(iv_est, stage = 1)` will give the first stage(s). Note that using `summary` you can display both the second and first stages at the same time using, e.g., `stage = 1:2` (using `2:1` would reverse the order).

Multiple estimations

Multiple estimations can be performed at once, they just have to be specified in the formula. Multiple estimations yield a `fixest_multi` object which is ‘kind of’ a list of all the results but includes specific methods to access the results in a handy way. Please have a look at the dedicated vignette: **Multiple estimations**.

To include multiple dependent variables, wrap them in `c()` (`list()` also works). For instance `fml = c(y1, y2) \sim x1` would estimate the model `fml = y1 \sim x1` and then the model `fml = y2 \sim x1`.

To include multiple independent variables, you need to use the stepwise functions. There are 4 stepwise functions: `sw`, `sw0`, `csw`, `csw0`, and `mvsw`. Of course `sw` stands for stepwise, and `csw` for cumulative stepwise. Finally `mvsw` is a bit special, it stands for multiverse stepwise. Let’s explain that. Assume you have the following formula: `fml = y \sim x1 + sw(x2, x3)`. The stepwise function `sw` will estimate the following two models: \( y \sim x1 + x2 \) and \( y \sim x1 + x3 \). That is, each element in `sw()` is sequentially, and separately, added to the formula. Would have you used `sw0` in lieu of `sw()`, then the model `y \sim x1` would also have been estimated. The `0` in the name means that the model without any stepwise element also needs to be estimated. The prefix `c` means cumulative: each stepwise element is added to the next. That is, `fml = y \sim x1 + csw(x2, x3)` would lead to the following models: \( y \sim x1 + x2 \) and \( y \sim x1 + x2 + x3 \). The `0` has the same meaning and would also lead to the model without the stepwise elements to be estimated: in other words, `fml = y \sim x1 + csw0(x2, x3)` leads to the following three models: \( y \sim x1 \), \( y \sim x1 + x2 \) and \( y \sim x1 + x2 + x3 \). Finally `mvsw` will add, in a stepwise fashion all possible combinations of the variables in its arguments. For example `mvsw(x1, x2, x3)` is equivalent to `sw0(x1, x2, x3, x1 + x2, x1 + x3, x2 + x3, x1 + x2 + x3)`. The number of models to estimate grows at a factorial rate: so be cautious!

Multiple independent variables can be combined with multiple dependent variables, as in `fml = c(y1, y2) \sim cw(x1, x2, x3)` which would lead to 6 estimations. Multiple estimations can also be combined to split samples (with the arguments `split`, `fsplit`).

You can also add fixed-effects in a stepwise fashion. Note that you cannot perform stepwise estimations on the IV part of the formula (`feols` only).

If NAs are present in the sample, to avoid too many messages, only NA removal concerning the variables common to all estimations is reported.

A note on performance. The feature of multiple estimations has been highly optimized for `feols`, in particular in the presence of fixed-effects. It is faster to estimate multiple models using the formula rather than with a loop. For non-`feols` models using the formula is roughly similar to using a loop performance-wise.

**Tricks to estimate multiple LHS**

To use multiple dependent variables in `fixest` estimations, you need to include them in a vector: like in `c(y1, y2, y3)`.
First, if names are stored in a vector, they can readily be inserted in a formula to perform multiple estimations using the dot square bracket operator. For instance if \( \text{my_lhs} = c(\"y1\", \"y2\") \), calling \texttt{fixest} with \texttt{feols(.[\text{my_lhs}] \sim x1, etc)} is equivalent to using \texttt{feols(c(y1, y2) \sim x1, etc)}. Beware that this is a special feature unique to the left-hand-side of \texttt{fixest} estimations (the default behavior of the DSB operator is to aggregate with sums, see \texttt{xpd}).

Second, you can use a regular expression to grep the left-hand-sides on the fly. When the \texttt{..("regex")} feature is used naked on the LHS, the variables grepped are inserted into \texttt{c()}. For example \texttt{..("Pe") \sim Sepal.Length, iris} is equivalent to \texttt{c(Petal.Length, Petal.Width) \sim Sepal.Length, iris}. Beware that this is a special feature unique to the left-hand-side of \texttt{fixest} estimations (the default behavior of \texttt{..("regex")} is to aggregate with sums, see \texttt{xpd}).

**Argument sliding**

When the data set has been set up globally using \texttt{setFixest_estimation(data = data_set)}, the argument \texttt{vcov} can be used implicitly. This means that calls such as \texttt{feols(y \sim x, \"HC1\")}, or \texttt{feols(y \sim x, \~id)}, are valid: i) the data is automatically deduced from the global settings, and ii) the \texttt{vcov} is deduced to be the second argument.

**Piping**

Although the argument 'data' is placed in second position, the data can be piped to the estimation functions. For example, with \( R >= 4.1 \), \texttt{mtcars |> feols(mpg \sim cyl)} works as \texttt{feols(mpg \sim cyl, mtcars)}.

**Dot square bracket operator in formulas**

In a formula, the dot square bracket (DSB) operator can: i) create manifold variables at once, or ii) capture values from the current environment and put them verbatim in the formula.

Say you want to include the variables \( x1 \) to \( x3 \) in your formula. You can use \texttt{xpd(y \sim x.[1:3])} and you’ll get \( y \sim x1 + x2 + x3 \).

To summon values from the environment, simply put the variable in square brackets. For example: \texttt{for(i in 1:3) xpd(y.[i] \sim x)} will create the formulas \( y1 \sim x \) to \( y3 \sim x \) depending on the value of \( i \).

You can include a full variable from the environment in the same way: \texttt{for(y in c("a", "b")) xpd(.[y] \sim x)} will create the two formulas \( a \sim x \) and \( b \sim x \).

The DSB can even be used within variable names, but then the variable must be nested in character form. For example \( y \sim .[\text{"x.}[1:2:;_sq"]]\) will create \( y \sim x1\_sq + x2\_sq \). Using the character form is important to avoid a formula parsing error. Double quotes must be used. Note that the character string that is nested will be parsed with the function \texttt{dsb}, and thus it will return a vector.

By default, the DSB operator expands vectors into sums. You can add a comma, like in \texttt{.[, x]}, to expand with commas—the content can then be used within functions. For instance: \texttt{c(x.[, 1:2])} will create \( c(x1, x2) \) (and not \( c(x1 + x2) \)).

In all \texttt{fixest} estimations, this special parsing is enabled, so you don’t need to use \texttt{xpd}.

One-sided formulas can be expanded with the DSB operator: let \( x = \sim\text{sepal + petal} \), then \texttt{xpd(y \sim .[x])} leads to \texttt{color \sim sepal + petal}.

You can even use multiple square brackets within a single variable, but then the use of nesting is required. For example, the following \texttt{xpd(y \sim .[\text{"letters}[1:2:;_;[1:2:;"]\})} will create \( y \)
\[ \sim a_1 + b_2. \] Remember that the nested character string is parsed with \texttt{dsb}, which explains this behavior.

When the element to be expanded i) is equal to the empty string or, ii) is of length 0, it is replaced with a neutral element, namely 1. For example, \[ x = "" ; xpd(y \sim .[x]) \] leads to \[ y \sim 1. \]

**Author(s)**

Laurent Berge

**References**

Berge, Laurent, 2018, "Efficient estimation of maximum likelihood models with multiple fixed-effects: the \texttt{R} package \texttt{FENmlm}." CREA Discussion Papers, 13 ().

For models with multiple fixed-effects:

Gaure, Simen, 2013, "OLS with multiple high dimensional category variables", Computational Statistics \\ & Data Analysis 66 pp. 8–18

**See Also**

See also \texttt{summary.fixest} to see the results with the appropriate standard-errors, \texttt{fixef.fixest} to extract the fixed-effects coefficients, and the function \texttt{etable} to visualize the results of multiple estimations. For plotting coefficients: see \texttt{coefplot}.

And other estimation methods: \texttt{femlm}, \texttt{feglm}, \texttt{fepois}, \texttt{fenegbin}, \texttt{feNmlm}.

**Examples**

```r
# # Basic estimation
#
res = feols(Sepal.Length ~ Sepal.Width + Petal.Length, iris)
# You can specify clustered standard-errors in \texttt{summary}:
summary(res, cluster = \sim Species)
#
# # Just one set of fixed-effects:
#
res = feols(Sepal.Length ~ Sepal.Width + Petal.Length | Species, iris)
# By default, the SEs are clustered according to the first fixed-effect
summary(res)
#
# # Varying slopes:
#
res = feols(Sepal.Length ~ Petal.Length | Species[Sepal.Width], iris)
summary(res)
```
# Combining the FEs:

```r
base = iris
base$fe_2 = rep(1:10, 15)
res_comb = feols(Sepal.Length ~ Petal.Length | Species^fe_2, base)
summary(res_comb)
fixef(res_comb)[[1]]
```

# Using leads/lags:

```r
data(base_did)
# We need to set up the panel with the arg. panel.id
est1 = feols(y ~ l(x1, 0:1), base_did, panel.id = ~id+period)
est2 = feols(f(y) ~ l(x1, -1:1), base_did, panel.id = ~id+period)
etable(est1, est2, order = "f", drop="Int")
```

# Using interactions:

```r
data(base_did)
# We interact the variable 'period' with the variable 'treat'
est_did = feols(y ~ x1 + i(period, treat, 5) | id+period, base_did)
# Now we can plot the result of the interaction with coefplot
table(est_did)
# You have many more example in coefplot help
```

# Instrumental variables

```r
# To estimate Two stage least squares,
# insert a formula describing the endo. vars./instr. relation after a pipe:
base = iris
names(base) = c("y", "x1", "x2", "x3", "fe1")
base$x_inst1 = 0.2 * base$x1 + 0.7 * base$x2 + rpois(150, 2)
base$x_inst2 = 0.2 * base$x2 + 0.7 * base$x3 + rpois(150, 3)
base$x_endo1 = 0.5 * base$y + 0.5 * base$x3 + rnorm(150, sd = 2)
base$x_endo2 = 1.5 * base$y + 0.5 * base$x3 + 3 * base$x_inst1 + rnorm(150, sd = 5)

# Using 2 controls, 1 endogenous var. and 1 instrument
res_iv = feols(y ~ x1 + x2 | x_endo1 ~ x_inst1, base)
# The second stage is the default
summary(res_iv)
```

# To show the first stage:
summary(res_iv, stage = 1)

# To show both the first and second stages:
summary(res_iv, stage = 1:2)

# Adding a fixed-effect => IV formula always last!
res_iv_fe = feols(y ~ x1 + x2 | fe1 | x_endo1 ~ x_inst1, base)

# With two endogenous regressors
res_iv2 = feols(y ~ x1 + x2 | x_endo1 + x_endo2 ~ x_inst1 + x_inst2, base)

# Now there's two first stages => a fixest_multi object is returned
sum_res_iv2 = summary(res_iv2, stage = 1)

# You can navigate through it by subsetting:
sum_res_iv2[iv = 1]

# The stage argument also works in etable:
etable(res_iv, res_iv_fe, res_iv2, order = "endo")

etable(res_iv, res_iv_fe, res_iv2, stage = 1:2, order = c("endo", "inst"),
       group = list(control = "!endo|inst"))

# Multiple estimations:
#
# 6 estimations
est_mult = feols(c(Ozone, Solar.R) ~ Wind + Temp + csw0(Wind:Temp, Day), airquality)

# We can display the results for the first lhs:
etable(est_mult[lhs = 1])

# And now the second (access can be made by name)
etable(est_mult[lhs = "Solar.R"])

# Now we focus on the two last right hand sides
# (note that .N can be used to specify the last item)
etable(est_mult[rhs = 2:.N])

# Combining with split
est_split = feols(c(Ozone, Solar.R) ~ sw(poly(Wind, 2), poly(Temp, 2)),
                  airquality, split = ~ Month)

# You can display everything at once with the print method
est_split

# Different way of displaying the results with "compact"
summary(est_split, "compact")

# You can still select which sample/LHS/RHS to display
est_split[sample = 1:2, lhs = 1, rhs = 1]
# Split sample estimations

base = setNames(iris, c("y", "x1", "x2", "x3", "species"))

est = feols(y ~ x.[1:3], base, split = ~species)
etable(est)

# You can select specific values with the %keep% and %drop% operators
# By default, partial matching is enabled. It should refer to a single variable.
est = feols(y ~ x.[1:3], base, split = ~species %keep% c("set", "vers"))etable(est)

# You can supply regular expression by using an @ first.
# regex can match several values.
est = feols(y ~ x.[1:3], base, split = ~species %keep% c("@set|vers"))etable(est)

# Argument sliding

# When the data set is set up globally, you can use the vcov argument implicitly
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))

no_sliding = feols(y ~ x1 + x2, base, ~species)

# With sliding
setFixest_estimation(data = base)

# ~species is implicitly deduced to be equal to 'vcov'
sliding = feols(y ~ x1 + x2, ~species)

etable(no_sliding, sliding)

# Resetting the global options
setFixest_estimation(data = NULL)

# Formula expansions

# By default, the features of the xpd function are enabled in
# all fixest estimations
# Here's a few examples
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))

# dot square bracket operator
feols(y ~ x.[1:3], base)
# fetching variables via regular expressions: ..("regex")
feols(y ~ ..("1|2"), base)

# NOTA: it also works for multiple LHS
mult1 = feols(x.[1:2] ~ y + species, base)
mult2 = feols(.,("y|3") ~ x.[1:2] + species, base)
etable(mult1, mult2)

# Use .[, stuff] to include variables in functions:
feols(y ~ csw(x.[,1:3]), base)

# Same for .., "regex")
feols(y ~ csw(.,"x"), base)

---

### fitstat

**Computes fit statistics of fixest objects**

**Description**

Computes various fit statistics for fixest estimations.

**Usage**

```r
fitstat(
  x,
  type,
  simplify = FALSE,
  verbose = TRUE,
  show_types = FALSE,
  frame = parent.frame(),
  ...
)
```

**Arguments**

- **x**
  - A fixest estimation.

- **type**
  - Character vector or one sided formula. The type of fit statistic to be computed. The classic ones are: n, rmse, r2, pr2, f, wald, ivf, ivwald. You have the full list in the details section or use `show_types = TRUE`. Further, you can register your own types with `fitstat_register`.

- **simplify**
  - Logical, default is FALSE. By default a list is returned whose names are the selected types. If `simplify = TRUE` and only one type is selected, then the element is directly returned (ie will not be nested in a list).
verbose Logical, default is TRUE. If TRUE, an object of class fixest_fitstat is returned (so its associated print method will be triggered). If FALSE a simple list is returned instead.

show_types Logical, default is FALSE. If TRUE, only prompts all available types.

frame An environment in which to evaluate variables, default is parent.frame(). Only used if the argument type is a formula and some values in the formula have to be extended with the dot square bracket operator. Mostly for internal use.

... Other elements to be passed to other methods and may be used to compute the statistics (for example you can pass on arguments to compute the VCOV when using type = "g" or type = "wald").

Value

By default an object of class fixest_fitstat is returned. Using verbose = FALSE returns a simple a list. Finally, if only one type is selected, simplify = TRUE leads to the selected type to be returned.

Registering your own types

You can register custom fit statistics with the function fitstat_register.

Available types

The types are case sensitive, please use lower case only. The types available are:

n, ll, aic, bic, rmse: The number of observations, the log-likelihood, the AIC, the BIC and the root mean squared error, respectively.

my: Mean of the dependent variable.

g: The degrees of freedom used to compute the t-test (it influences the p-values of the coefficients). When the VCOV is clustered, this value is equal to the minimum cluster size, otherwise, it is equal to the sample size minus the number of variables.

r2, ar2, wr2, awr2, pr2, apr2, wpr2, awpr2: All r2 that can be obtained with the function r2. The a stands for 'adjusted', the w for 'within' and the p for 'pseudo'. Note that the order of the letters a, w and p does not matter. The pseudo R2s are McFadden’s R2s (ratios of log-likelihoods).

theta: The over-dispersion parameter in Negative Binomial models. Low values mean high overdispersion.

f, wf: The F-tests of nullity of the coefficients. The w stands for 'within'. These types return the following values: stat, p, df1 and df2. If you want to display only one of these, use their name after a dot: e.g. f.stat will give the statistic of the F-test, or wf.p will give the p-values of the F-test on the projected model (i.e. projected onto the fixed-effects).

wald: Wald test of joint nullity of the coefficients. This test always excludes the intercept and the fixed-effects. These type returns the following values: stat, p, df1, df2 and vcov. The element vcov reports the way the VCOV matrix was computed since it directly influences this statistic.
ivf, ivf1, ivf2, ivfall: These statistics are specific to IV estimations. They report either the IV F-test (namely the Cragg-Donald F statistic in the presence of only one endogenous regressor) of the first stage (ivf or ivf1), of the second stage (ivf2) or of both (ivfall). The F-test of the first stage is commonly named weak instrument test. The value of ivfall is only useful in etable when both the 1st and 2nd stages are displayed (it leads to the 1st stage F-test(s) to be displayed on the 1st stage estimation(s), and the 2nd stage one on the 2nd stage estimation – otherwise, ivf1 would also be displayed on the 2nd stage estimation). These types return the following values: stat, p, df1 and df2.

iwald, iwald1, iwald2, iwaldall: These statistics are specific to IV estimations. They report either the IV Wald-test of the first stage (iwald or iwald1), of the second stage (iwald2) or of both (iwaldall). The Wald-test of the first stage is commonly named weak instrument test. Note that if the estimation was done with a robust VCOV and there is only one endogenous regressor, this is equivalent to the Kleibergen-Paap statistic. The value of iwaldall is only useful in etable when both the 1st and 2nd stages are displayed (it leads to the 1st stage Wald-test(s) to be displayed on the 1st stage estimation(s), and the 2nd stage one on the 2nd stage estimation – otherwise, iwald1 would also be displayed on the 2nd stage estimation). These types return the following values: stat, p, df1, df2, vcov.

cd: The Cragg-Donald test for weak instruments.
kpr: The Kleibergen-Paap test for weak instruments.
wh: This statistic is specific to IV estimations. Wu-Hausman endogeneity test. H0 is the absence of endogeneity of the instrumented variables. It returns the following values: stat, p, df1, df2.
sargan: Sargan test of overidentifying restrictions. H0: the instruments are not correlated with the second stage residuals. It returns the following values: stat, p, df.

Examples

data(trade)
gravity = feols(log(Euros) ~ log(dist_km) | Destination + Origin, trade)

# Extracting the 'working' number of observations used to compute the pvalues
fitstat(gravity, "g", simplify = TRUE)

# Some fit statistics
fitstat(gravity, ~ rmse + r2 + wald + wf)

# You can use them in etable
etable(gravity, fitstat = ~ rmse + r2 + wald + wf)

# For wald and wf, you could show the pvalue instead:
etable(gravity, fitstat = ~ rmse + r2 + wald.p + wf.p)
# Now let's display some statistics that are not built-in
# => we use fitstat_register to create them

# We need: a) type name, b) the function to be applied
# c) (optional) an alias

fitstat_register("tstand", function(x) tstat(x, se = "stand")[1], "t-stat (regular)"
fitstat_register("thc", function(x) tstat(x, se = "heter")[1], "t-stat (HC1)"
fitstat_register("t1w", function(x) tstat(x, se = "clus")[1], "t-stat (clustered)"
fitstat_register("t2w", function(x) tstat(x, se = "twow")[1], "t-stat (2-way)"

# Now we can use these keywords in fitstat:
etable(gravity, fitstat = ~ . + tstand + thc + t1w + t2w)

# Note that the custom stats we created are can easily lead
# to errors, but that's another story!

---

**fitstat_register**  
*Register custom fit statistics*

**Description**

Enables the registration of custom fit statistics that can be easily summoned with the function *fitstat*.

**Usage**

`fitstat_register(type, fun, alias = NULL, subtypes = NULL)`

**Arguments**

- **type**: A character scalar giving the type-name.
- **fun**: A function to be applied to a `fixest` estimation. It must return either a scalar, or a list of unitary elements. If the number of elements returned is greater than 1, then each element must be named! If the fit statistic is not valid for a given estimation, a plain `NA` value should be returned.
- **alias**: A (named) character vector. An alias to be used in lieu of the type name in the display methods (ie when used in `print.fixest_fitstat` or `etable`). If the function returns several values, i.e. sub-types, you can give an alias to these sub-types. The syntax is `c("type" = "alias", "subtype_i" = "alias_i")`, with "type" (resp. "subtype") the value of the argument type resp. (subtypes). You can also give an alias encompassing the type and sub-type with the syntax `c("type.subtype_i" = "alias")`. 
subtypes A character vector giving the name of each element returned by the function
fun. This is only used when the function returns more than one value. Note that
you can use the shortcut "test" when the sub-types are "stat", "p" and "df"; and
"test2" when these are "stat", "p", "df1" and "df2".

Details

If there are several components to the computed statistics (i.e. the function returns several elements),
then using the argument subtypes, giving the names of each of these components, is mandatory.
This is to ensure that the statistic can be used as any other built-in statistic (and there are too many
edge cases impeding automatic deduction).

Author(s)

Laurent Berge

Examples

# An estimation
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")
est = feols(y ~ x1 + x2 | species, base)

#
# single valued tests
#
#
# say you want to add the coefficient of variation of the dependent variable
cv = function(est){
y = model.matrix(est, type = "lhs")
sd(y)/mean(y)
}

# Now we register the routine
fitstat_register("cvy", cv, "Coef. of Variation (dep. var.)")

# now we can summon the registered routine with its type ("cvy")
fitstat(est, "cvy")

#
# Multi valued tests
#
#
# Let's say you want a Wald test with an heteroskedasticity robust variance

# First we create the function
hc_wald = function(est){
w = wald(est, keep = "!Intercept", print = FALSE, se = "hetero")
head(w, 4)
}
# This test returns a vector of 4 elements: stat, p, df1 and df2
# Now we register the routine
fitstat_register("hc_wald", hc_wald, "Wald (HC1)", "test2")

# You can access the statistic, as before
fitstat(est, "hc_wald")

# But you can also access the sub elements
fitstat(est, "hc_wald.p")

---

`fitted.fixest` extracts fitted values from a `fixest` fit

### Description

This function extracts the fitted values from a model estimated with `femlm`, `feols` or `feglm`. The fitted values that are returned are the expected predictor.

### Usage

```
## S3 method for class 'fixest'
fitted(object, type = c("response", "link"), na.rm = TRUE, ...)

## S3 method for class 'fixest'
fitted.values(object, type = c("response", "link"), na.rm = TRUE, ...)
```

### Arguments

- **object**: A `fixest` object. Obtained using the functions `femlm`, `feols` or `feglm`.
- **type**: Character either equal to "response" (default) or "link". If `type="response"`, then the output is at the level of the response variable, i.e. it is the expected predictor $E(Y|X)$. If "link", then the output is at the level of the explanatory variables, i.e. the linear predictor $X \cdot \beta$.
- **na.rm**: Logical, default is TRUE. If FALSE the number of observation returned will be the number of observations in the original data set, otherwise it will be the number of observations used in the estimation.
- **...**: Not currently used.

### Details

This function returns the expected predictor of a fixest fit. The likelihood functions are detailed in `femlm` help page.
Value

It returns a numeric vector of length the number of observations used to estimate the model.

If `type = "response"`, the value returned is the expected predictor, i.e. the expected value of the dependent variable for the fitted model: \( E(Y|X) \). If `type = "link"`, the value returned is the linear predictor of the fitted model, that is \( X \cdot \beta \) (remind that \( E(Y|X) = f(X \cdot \beta) \)).

Author(s)

Laurent Berge

See Also

See also the main estimation functions `femlm`, `feols` or `feglm`, `resid.fixest`, `predict.fixest`, `summary.fixest`, `vcov.fixest`, `fixef.fixest`.

Examples

```r
# simple estimation on iris data, using "Species" fixed-effects
res_poisson = femlm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width | Species, iris)

# we extract the fitted values
y_fitted_poisson = fitted(res_poisson)

# Same estimation but in OLS (Gaussian family)
res_gaussian = femlm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width | Species, iris, family = "gaussian")

y_fitted_gaussian = fitted(res_gaussian)

# comparison of the fit for the two families
plot(iris$Sepal.Length, y_fitted_poisson)
points(iris$Sepal.Length, y_fitted_gaussian, col = 2, pch = 2)
```

---

**fixef.fixest**

*Extract the Fixed-Effects from a fixest estimation.*

Description

This function retrieves the fixed effects from a fixest estimation. It is useful only when there are one or more fixed-effect dimensions.
## S3 method for class 'fixest'

```r
fixef(
  object,
  notes = getFixest_notes(),
  sorted = TRUE,
  nthreads = getFixest_nthreads(),
  fixef.tol = 1e-05,
  fixef.iter = 10000,
  ...
)
```

### Arguments

- **object**
  A fixest estimation (e.g. obtained using `feols` or `feglm`).

- **notes**
  Logical. Whether to display a note when the fixed-effects coefficients are not regular.

- **sorted**
  Logical, default is `TRUE`. Whether to order the fixed-effects by their names. If `FALSE`, then the order used in the demeaning algorithm is used.

- **nthreads**
  The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function `setFixest_nthreads`.

- **fixef.tol**
  Precision used to obtain the fixed-effects. Defaults to `1e-5`. It corresponds to the maximum absolute difference allowed between two coefficients of successive iterations. Argument `fixef.tol` cannot be lower than `10000*.Machine$double.eps`. Note that this parameter is dynamically controlled by the algorithm.

- **fixef.iter**
  Maximum number of iterations in fixed-effects algorithm (only in use for 2+ fixed-effects). Default is 10000.

- **...**
  Not currently used.

### Details

If the fixed-effect coefficients are not regular, then several reference points need to be set: this means that the fixed-effects coefficients cannot be directly interpreted. If this is the case, then a warning is raised.

### Value

A list containing the vectors of the fixed effects.

If there is more than 1 fixed-effect, then the attribute “references” is created. This is a vector of length the number of fixed-effects, each element contains the number of coefficients set as references. By construction, the elements of the first fixed-effect dimension are never set as references. In the presence of regular fixed-effects, there should be `Q-1` references (with `Q` the number of fixed-effects).
fixest_startup_msg

Description

Package startup messages can be very annoying, although sometimes they can be necessary. Use this function to prevent fixest's package startup message from popping when loading. This will be specific to your current project.

Usage

fixest_startup_msg(x)

Arguments

x Logical, no default. If FALSE, the package startup message is removed.
Details

Note that this function is introduced to cope with the first \texttt{fixest} startup message (in version 0.9.0). This function works only with R \textgreater= 4.0.0. There are no startup messages for R < 4.0.0.

Description

This function extracts the formula from a \texttt{fixest} estimation (obtained with \texttt{femlm}, \texttt{feols} or \texttt{feglm}). If the estimation was done with fixed-effects, they are added in the formula after a pipe ("|”). If the estimation was done with a non linear in parameters part, then this will be added in the formula in between \texttt{I}().

Usage

\begin{verbatim}
## S3 method for class 'fixest'
formula(x, type = c("full", "linear", "iv", "NL"), ...)
\end{verbatim}

Arguments

\begin{itemize}
\item \texttt{x} An object of class \texttt{fixest}. Typically the result of a \texttt{femlm}, \texttt{feols} or \texttt{feglm} estimation.
\item \texttt{type} A character scalar. Default is \texttt{type = "full"} which gives back a formula containing the linear part of the model along with the fixed-effects (if any) and the IV part (if any). If \texttt{type = "linear"} then only the linear formula is returned. If \texttt{type = "NL"} then only the non linear in parameters part is returned.
\item ... Not currently used.
\end{itemize}

Value

It returns a formula.

Author(s)

Laurent Berge

See Also

See also the main estimation functions \texttt{femlm}, \texttt{feols} or \texttt{feglm}, \texttt{model.matrix.fixest}, \texttt{update.fixest}, \texttt{summary.fixest}, \texttt{vcov.fixest}. 
Examples

# simple estimation on iris data, using "Species" fixed-effects
res = femlm(Sepal.Length ~ Sepal.Width + Petal.Length +
            Petal.Width | Species, iris)

# formula with the fixed-effect variable
formula(res)

# linear part without the fixed-effects
formula(res, "linear")

---

hatvalues.fixest  
Hat values for fixest objects

Description

Computes the hat values for feols or feglm estimations. Only works when there are no fixed-effects.

Usage

## S3 method for class 'fixest'
hatvalues(model, ...)

Arguments

model  
A fixest object. For instance from feols or feglm.

...  
Not currently used.

Details

Hat values are not available for fenegbin, feglm and feNmlm estimations.

When there are fixed-effects, the hat values of the reduced form are different from the hat values of the full model. And we cannot get costlessly the hat values of the full model from the reduced form. It would require to reestimate the model with the fixed-effects as regular variables.

Value

Returns a vector of the same length as the number of observations used in the estimation.
Examples

```r
est = feols(Petal.Length ~ Petal.Width + Sepal.Width, iris)
head(hatvalues(est))
```

**Description**

Treat a variable as a factor, or interacts a variable with a factor. Values to be dropped/kept from the factor can be easily set. Note that to interact fixed-effects, this function should not be used: instead use directly the syntax `fe1^fe2`.

**Usage**

```
i(factor_var, var, ref, keep, bin, ref2, keep2, bin2, ...)
```

**Arguments**

- `factor_var`: A vector (of any type) that will be treated as a factor. You can set references (i.e. exclude values for which to create dummies) with the `ref` argument.
- `var`: A variable of the same length as `factor_var`. This variable will be interacted with the factor in `factor_var`. It can be numeric or factor-like. To force a numeric variable to be treated as a factor, you can add the `i.` prefix to a variable name. For instance take a numeric variable `x_num`: `i(x_fact, x_num)` will treat `x_num` as numeric while `i(x_fact, i.x_num)` will treat `x_num` as a factor (it's a shortcut to `as.factor(x_num)`).
- `ref`: A vector of values to be taken as references from `factor_var`. Can also be a logical: if `TRUE`, then the first value of `factor_var` will be removed. If `ref` is a character vector, partial matching is applied to values; use `@` as the first character to enable regular expression matching. See examples.
- `keep`: A vector of values to be kept from `factor_var` (all others are dropped). By default they should be values from `factor_var` and if `keep` is a character vector partial matching is applied. Use `@` as the first character to enable regular expression matching instead.
- `bin`: A list of values to be grouped, a vector, a formula, or the special values "bin::digit" or "cut::values". To create a new value from old values, use `bin = list("new_value"=old_values) with old_values a vector of existing values. You can use .() for list(). It accepts regular expressions, but they must start with an @, like in `bin="@Aug|Dec"`. It accepts one-sided formulas which must contain the variable x, e.g. `bin=list("<2" =~x < 2)`. The names of the list are the new names. If the new name is missing, the first value matched becomes the new name. In the name, adding "@d", with d
a digit, will relocate the value in position d: useful to change the position of factors. Use "@" as first item to make subsequent items be located first in the factor. Feeding in a vector is like using a list without name and only a single element. If the vector is numeric, you can use the special value "bin::digit" to group every digit element. For example if x represents years, using bin="bin::2" creates bins of two years. With any data, using "!bin::digit" groups every digit consecutive values starting from the first value. Using "!!bin::digit" is the same but starting from the last value. With numeric vectors you can: a) use "cut::n" to cut the vector into n equal parts, b) use "cut::a]b[" to create the following bins: [min, a], ]a, b[, [b, max]. The latter syntax is a sequence of number/quartile (q0 to q4)/percentile (p0 to p100) followed by an open or closed square bracket. You can add custom bin names by adding them in the character vector after 'cut::values'. See details and examples. Dot square bracket expansion (see dsb) is enabled.

**ref2**
A vector of values to be dropped from var. By default they should be values from var and if ref2 is a character vector partial matching is applied. Use "@" as the first character to enable regular expression matching instead.

**keep2**
A vector of values to be kept from var (all others are dropped). By default they should be values from var and if keep2 is a character vector partial matching is applied. Use "@" as the first character to enable regular expression matching instead.

**bin2**
A list or vector defining the binning of the second variable. See help for the argument bin for details (or look at the help of the function bin). You can use .() for list().

... Not currently used.

**Details**
To interact fixed-effects, this function should not be used: instead use directly the syntax fe1^fe2 in the fixed-effects part of the formula. Please see the details and examples in the help page of feols.

**Value**
It returns a matrix with number of rows the length of factor_var. If there is no interacted variable or it is interacted with a numeric variable, the number of columns is equal to the number of cases contained in factor_var minus the reference(s). If the interacted variable is a factor, the number of columns is the number of combined cases between factor_var and var.

**Author(s)**
Laurent Berge

**See Also**
iplot to plot interactions or factors created with i(), feols for OLS estimation with multiple fixed-effects.

See the function bin for binning variables.
Examples

# Simple illustration
#
x = rep(letters[1:4], 3)[1:10]
y = rep(1:4, c(1, 2, 3, 4))

# interaction
data.frame(x, y, i(x, y, ref = TRUE))

# without interaction
data.frame(x, i(x, "b"))

# you can interact factors too
z = rep(c("e", "f", "g"), c(5, 3, 2))
data.frame(x, z, i(x, z))

# to force a numeric variable to be treated as a factor: use i.
data.frame(x, y, i(x, 1.y))

# Binning
data.frame(x, i(x, bin = list(ab = c("a", "b"))))

# Same as before but using .() for list() and a regular expression
# note that to trigger a regex, you need to use an @ first
data.frame(x, i(x, bin = .(ab = @a|b)))

# In fixest estimations
#
data(base_did)
# We interact the variable 'period' with the variable 'treat'
est_did = feols(y ~ x1 + i(period, treat, 5) | id + period, base_did)

# => plot only interactions with iplot
iplot(est_did)

# Using i() for factors
est_bis = feols(y ~ x1 + i(period, keep = 3:6) + i(period, treat, 5) | id, base_did)

# we plot the second set of variables created with i()
# => we need to use keep (otherwise only the first one is represented)
coefplot(est_bis, keep = "trea")

# => special treatment in etable
etable(est_bis, dict = c("6" = "six"))

# Interact two factors
# We use the i. prefix to consider week as a factor
data(airquality)
aq = airquality
aq$week = aq$Day %/% 7 + 1

# Interacting Month and week:
res_2F = feols(Ozone ~ Solar.R + i(Month, i.week), aq)

# Same but dropping the 5th Month and 1st week
res_2F_bis = feols(Ozone ~ Solar.R + i(Month, i.week, ref = 5, ref2 = 1), aq)
etable(res_2F, res_2F_bis)

# Binning

data(airquality)
feols(Ozone ~ i(Month, bin = "bin::2"), airquality)
feols(Ozone ~ i(Month, bin = list(summer = 7:9)), airquality)

---

**lag.formula**

*Lags a variable using a formula*

**Description**

Lags a variable using panel id + time identifiers in a formula.

**Usage**

```r
## S3 method for class 'formula'
lag(
  x,
  k = 1,
  data,
  time.step = NULL,
  fill = NA,
  duplicate.method = c("none", "first"),
  ...
)

lag_fml(
```
Arguments

x A formula of the type \texttt{var ~ id + time} where \texttt{var} is the variable to be lagged, \texttt{id} is a variable representing the panel id, and \texttt{time} is the time variable of the panel.

k An integer giving the number of lags. Default is 1. For leads, just use a negative number.

data Optional, the \texttt{data.frame} in which to evaluate the formula. If not provided, variables will be fetched in the current environment.

time.step The method to compute the lags, default is \texttt{NULL} (which means automatically set). Can be equal to: \texttt{"unitary"}, \texttt{"consecutive"}, \texttt{"within.consecutive"}, or to a number. If \texttt{"unitary"}, then the largest common divisor between consecutive time periods is used (typically if the time variable represents years, it will be 1). This method can apply only to integer (or convertible to integer) variables. If \texttt{"consecutive"}, then the time variable can be of any type: two successive time periods represent a lag of 1. If \texttt{"within.consecutive"} then \textbf{within a given id}, two successive time periods represent a lag of 1. Finally, if the time variable is numeric, you can provide your own numeric time step.

fill Scalar. How to fill the observations without defined lead/lag values. Default is \texttt{NA}.

duplicate.method If several observations have the same id and time values, then the notion of lag is not defined for them. If \texttt{duplicate.method = "none"} (default) and duplicate values are found, this leads to an error. You can use \texttt{duplicate.method = "first"} so that the first occurrence of identical id/time observations will be used as lag.

Value

It returns a vector of the same type and length as the variable to be lagged in the formula.

Functions

- \texttt{lag.fml()}: Lags a variable using a formula syntax

Author(s)

Laurent Berge
See Also

Alternatively, the function `panel` changes a `data.frame` into a panel from which the functions `l` and `f` (creating leads and lags) can be called. Otherwise you can set the panel 'live' during the estimation using the argument `panel.id` (see for example in the function `feols`).

Examples

# simple example with an unbalanced panel
base = data.frame(id = rep(1:2, each = 4),
                  time = c(1, 2, 3, 4, 1, 4, 6, 9), x = 1:8)

base$lag1 = lag(x~id+time, 1, base) # lag 1
base$lead1 = lag(x~id+time, -1, base) # lead 1
base$lag2_fill0 = lag(x~id+time, 2, base, fill = 0)
# with time.step = "consecutive"
base$lag1_consecutive = lag(x~id+time, 1, base, time.step = "consecutive")
# => works for indiv. 2 because 9 (resp. 6) is consecutive to 6 (resp. 4)
base$lag1_within.consecutive = lag(x~id+time, 1, base, time.step = "within")
# => now two consecutive years within each indiv is one lag

print(base)

# Argument time.step = "consecutive" is
# mostly useful when the time variable is not a number:
# e.g. c("1991q1", "1991q2", "1991q3") etc

# with duplicates
base_dup = data.frame(id = rep(1:2, each = 4),
                      time = c(1, 1, 1, 2, 1, 2, 2, 3), x = 1:8)

# Error because of duplicate values for (id, time)
try(lag(x~id+time, 1, base_dup))

# Error is bypassed, lag corresponds to first occurrence of (id, time)
lag(x~id+time, 1, base_dup, duplicate.method = "first")

# Playing with time steps
base = data.frame(id = rep(1:2, each = 4),
                  time = c(1, 2, 3, 4, 1, 4, 6, 9), x = 1:8)

# time step: 0.5 (here equivalent to lag of 1)
lag(x~id+time, 2, base, time.step = 0.5)

# Error: wrong time step
try(lag(x~id+time, 2, base, time.step = 7))

# Adding NAs + unsorted IDs
base = data.frame(id = rep(1:2, each = 4),
                  time = c(4, NA, 3, 1, 2, NA, 1, 3), x = 1:8)
```
base$lag1 = lag(x~id+time, 1, base)
base$lag1_within = lag(x~id+time, 1, base, time.step = "w")
base_bis = base[order(base$id, base$time),]
print(base_bis)

# You can create variables without specifying the data within data.table:
if(require("data.table")){
  base = data.table(id = rep(1:2, each = 3), year = 1990 + rep(1:3, 2), x = 1:6)
  base[, x.l1 := lag(x~id+year, 1)]
}
```

---

**logLik.fixest**

Extraction the log-likelihood

**Description**

This function extracts the log-likelihood from a `fixest` estimation.

**Usage**

```r
## S3 method for class 'fixest'
logLik(object, ...)
```

**Arguments**

- `object`: A `fixest` object. Obtained using the functions `femlm`, `feols` or `feglm`.
- `...`: Not currently used.

**Details**

This function extracts the log-likelihood based on the model fit. You can have more information on the likelihoods in the details of the function `femlm`.

**Value**

It returns a numeric scalar.

**Author(s)**

Laurent Berge

**See Also**

See also the main estimation functions `femlm`, `feols` or `feglm`. Other statistics functions: `AIC.fixest`, `BIC.fixest`. 
Examples

```r
# simple estimation on iris data with "Species" fixed-effects
res = femlm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width | Species, iris)

nobs(res)
logLik(res)
```

---

**model.matrix.fixest**  
*Design matrix of a fixest object*

**Description**

This function creates the left-hand-side or the right-hand-side(s) of a `femlm`, `feols` or `feglm` estimation.

**Usage**

```r
## S3 method for class 'fixest'
matrix(  
  object,
  data,
  type = "rhs",
  na.rm = TRUE,
  subset = FALSE,
  as.matrix = FALSE,
  as.df = FALSE,
  collin.rm = TRUE,
  ...
)
```

**Arguments**

- `object`  
  A fixest object. Obtained using the functions `femlm`, `feols` or `feglm`.

- `data`  
  If missing (default) then the original data is obtained by evaluating the call. Otherwise, it should be a data.frame.

- `type`  
  Character vector or one sided formula, default is "rhs". Contains the type of matrix/data.frame to be returned. Possible values are: "lhs", "rhs", "fixef", "iv.rhs1" (1st stage RHS), "iv.rhs2" (2nd stage RHS), "iv.endo" (endogenous vars.), "iv.exo" (exoogenous vars), "iv.inst" (instruments).

- `na.rm`  
  Default is TRUE. Should observations with NAs be removed from the matrix?
model.matrix.fixest

subset Logical or character vector. Default is FALSE. If TRUE, then the matrix created will be restricted only to the variables contained in the argument data, which can then contain a subset of the variables used in the estimation. If a character vector, then only the variables matching the elements of the vector via regular expressions will be created.

as.matrix Logical scalar, default is FALSE. Whether to coerce the result to a matrix.

as.df Logical scalar, default is FALSE. Whether to coerce the result to a data.frame.

collin.rm Logical scalar, default is TRUE. Whether to remove variables that were found to be collinear during the estimation. Beware: it does not perform a collinearity check.

... Not currently used.

Value

It returns either a vector, a matrix or a data.frame. It returns a vector for the dependent variable ("lhs"), a data.frame for the fixed-effects ("fixef") and a matrix for any other type.

Author(s)

Laurent Berge

See Also

See also the main estimation functions femlm, feols or feglm. formula.fixest, update.fixest, summary.fixest, vcov.fixest.

Examples

base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

est = feols(y ~ poly(x1, 2) + x2, base)
head(model.matrix(est))

# Illustration of subset

# subset => character vector
head(model.matrix(est, subset = "x1"))

# subset => TRUE, only works with data argument!!
head(model.matrix(est, data = base[, "x1", drop = FALSE], subset = TRUE))
models

Extracts the models tree from a fixest_multi object

Description

Extracts the meta information on all the models contained in a fixest_multi estimation.

Usage

models(x, simplify = FALSE)

Arguments

- x: A fixest_multi object, obtained from a fixest estimation leading to multiple results.
- simplify: Logical, default is FALSE. The default behavior is to display all the meta information, even if they are identical across models. By using simplify = TRUE, only the information with some variation is kept.

Value

It returns a data.frame whose first column (named id) is the index of the models and the other columns contain the information specific to each model (e.g. which sample, which RHS, which dependent variable, etc).

Examples

# a multiple estimation
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
est = feols(y ~ csw(x.[, 1:3]), base, fsplit = ~species)

# All the meta information
models(est)

# Illustration: Why use simplify
est_sub = est[sample = 2]
models(est_sub)
models(est_sub, simplify = TRUE)
nobs.fixest

Expects the number of observations form a fixest object

Description

This function simply extracts the number of observations form a fixest object, obtained using the functions femlm, feols or feglm.

Usage

```r
## S3 method for class 'fixest'
nobs(object, ...)
```

Arguments

- `object` A fixest object. Obtained using the functions femlm, feols or feglm.
- `...` Not currently used.

Value

It returns an integer.

Author(s)

Laurent Berge

See Also

See also the main estimation functions femlm, feols or feglm. Use summary.fixest to see the results with the appropriate standard-errors, fixef.fixest to extract the fixed-effects coefficients, and the function etable to visualize the results of multiple estimations.

Examples

```r
# simple estimation on iris data with "Species" fixed-effects
res = femlm(Sepal.Length ~ Sepal.Width + Petal.Length +
            Petal.Width | Species, iris)

nobs(res)
logLik(res)
```
n_unik

Prints the number of unique elements in a data set

Description

This utility tool displays the number of unique elements in one or multiple data.frames as well as their number of NA values.

Usage

n_unik(x)

## S3 method for class 'vec_n_unik'
print(x, ...)

## S3 method for class 'list_n_unik'
print(x, ...)

Arguments

x

A formula, with data set names on the LHS and variables on the RHS, like data1 + data2 ~ var1 + var2. The following special variables are admitted: "." to get default values, ".N" for the number of observations, ".U" for the number of unique rows, ".NA" for the number of rows with at least one NA. Variables can be combined with "^", e.g. df~id*period; use id%^period to also include the terms on both sides. Note that using : and * is equivalent to ^ and %^%. Sub select with id[cond], when doing so id is automatically included. Conditions can be chained, as in id[cond1, cond2]. Use NA(x, y) in conditions instead of is.na(x) | is.na(y). Use the !! operator to have both a condition and its opposite. To compare the keys in two data sets, use data1: data2. If not a formula, x can be: a vector (displays the # of unique values); a data.frame (default values are displayed), or a "sum" of data sets like in x = data1 + data2, in that case it is equivalent to data1 + data2 ~ ..

Value

It returns a vector containing the number of unique values per element. If several data sets were provided, a list is returned, as long as the number of data sets, each element being a vector of unique values.

Special values and functions

In the formula, you can use the following special values: ".", ".N", ".U", and ".NA".

"." Accesses the default values. If there is only one data set and the data set is not a data.table, then the default is to display the number of observations and the number of unique rows. If
The data is a data.table, the number of unique items in the key(s) is displayed instead of the number of unique rows (if the table has keys of course). If there are two or more data sets, then the default is to display the unique items for: a) the variables common across all data sets, if there’s less than 4, and b) if no variable is shown in a), the number of variables common across at least two data sets, provided there are less than 5. If the data sets are data tables, the keys are also displayed on top of the common variables. In any case, the number of observations is always displayed.

".N" Displays the number of observations.

".U" Displays the number of unique rows.

".NA" Displays the number of rows with at least one NA.

The NA function

The special function NA is an equivalent to is.na but can handle several variables. For instance, NA(x, y) is equivalent to is.na(x) | is.na(y). You can add as many variables as you want as arguments. If no argument is provided, as in NA(), it is identical to having all the variables of the data set as argument.

Combining variables

Use the "hat", "^", operator to combine several variables. For example id^period will display the number of unique values of id x period combinations.

Use the "super hat". "%^%", operator to also include the terms on both sides. For example, instead of writing id + period + id^period, you can simply write id%^%period.

Alternatively, you can use : for ^ and * for %%^.

Sub-selections

To show the number of unique values for sub samples, simply use []. For example, id[x > 10] will display the number of unique id for which x > 10.

Simple square brackets lead to the inclusion of both the variable and its subset. For example id[x > 10] is equivalent to id + id[x > 10]. To include only the sub selection, use double square brackets, as in id[[x > 10]].

You can add multiple sub selections at once, only separate them with a comma. For example id[x > 10, NA(y)] is equivalent to id[x > 10] + id[NA(y)].

Use the double negative operator, i.e. !!, to include both a condition and its opposite at once. For example id[!!x > 10] is equivalent to id[x > 10, !x > 10]. Double negative operators can be chained, like in id[!!cond1 & !!cond2], then the cardinal product of all double negatived conditions is returned.

Author(s)

Laurent Berge
Examples

data = base_did
data$x1.L1 = round(lag(x1~id+period, 1, data))

# By default, just the formatted number of observations
n_unik(data)

# Or the number of unique elements of a vector
n_unik(data$id)

# number of unique id values and id x period pairs
n_unik(data ~.N + id + id*period)

# use the %^% operator to include the terms on the two sides at once
# => same as id*period
n_unik(data ~.N + id %^% period)

# using sub selection with []
n_unik(data ~.N + period[!NA(x1.L1)])

# to show only the sub selection: [[]]
n_unik(data ~.N + period[[!NA(x1.L1)]]))

# you can have multiple values in [],
# just separate them with a comma
n_unik(data ~.N + period[!NA(x1.L1), x1 > 7])

# to have both a condition and its opposite,
# use the !! operator
n_unik(data ~.N[!!NA(x1.L1)])

# the !! operator works within condition chains
n_unik(data ~.N[!!NA(x1.L1) & !!x1 > 7])

# Conditions can be distributed
n_unik(data ~(id + period)[x1 > 7])

#
# Several data sets
#

# Typical use case: merging
# Let's create two data sets and merge them

data(base_did)
base_main = base_did
base_extra = sample_df(base_main[, c("id", "period")], 100)
base_extra$id[1:10] = 111:120
base_extra$z = rnorm(100)
# You can use db1:db2 to compare the common keys in two data sets
n_unik(base_main:base_extra)

tmp = merge(base_main, base_extra, all.x = TRUE, by = c("id", "period"))

# You can show unique values for any variable, as before
n_unik(tmp + base_main + base_extra ~ id[!!NA(z)] + id\text{"period"})

\begin{description}
\item[obs] Extracts the observations used for the estimation
\end{description}

\section*{Description}
This function extracts the observations used in \texttt{fixest} estimation.

\section*{Usage}
\texttt{obs(x)}

\section*{Arguments}
\begin{itemize}
\item \texttt{x} A \texttt{fixest} object.
\end{itemize}

\section*{Value}
It returns a simple vector of integers.

\section*{Examples}

base = iris
names(base) = c("y", "x1", "x2", "x3", "species")
base$y[1:5] = NA

# Split sample estimations
est_split = feols(y ~ x1, base, split = ~species)
(obs_setosa = obs(est_split[[1]]))
(obs_versi = obs(est_split[sample = "versi", drop = TRUE]))
est_versi = feols(y ~ x1, base, subset = obs_versi)
etable(est_split, est_versi)
osize  

**Formatted object size**

### Description

Tools that returns a formatted object size, where the appropriate unit is automatically chosen.

### Usage

```r
osize(x)
```

```r
## S3 method for class 'osize'
print(x, ...)
```

### Arguments

- **x**: Any R object.
- **...**: Not currently used.

### Value

Returns a character scalar.

### Author(s)

Laurent Berge

### Examples

```r
osize(iris)
data(trade)
osize(trade)
```

---

panel  

**Constructs a fixest panel data base**

### Description

Constructs a fixest panel data base out of a data.frame which allows to use leads and lags in fixest estimations and to create new variables from leads and lags if the data.frame was also a `data.table::data.table`. 
Usage

`panel(data, panel.id, time.step = NULL, duplicate.method = c("none", "first"))`

Arguments

data A data.frame.
panel.id The panel identifiers. Can either be: i) a one sided formula (e.g. `panel.id = ~id+time`), ii) a character vector of length 2 (e.g. `panel.id=c('id', 'time')`), or iii) a character scalar of two variables separated by a comma (e.g. `panel.id='id,time'`). Note that you can combine variables with `^` only inside formulas (see the dedicated section in `feols`).
time.step The method to compute the lags, default is `NULL` (which means automatically set). Can be equal to: "unitary", "consecutive", "within.consecutive", or to a number. If "unitary", then the largest common divisor between consecutive time periods is used (typically if the time variable represents years, it will be 1). This method can apply only to integer (or convertible to integer) variables. If "consecutive", then the time variable can be of any type: two successive time periods represent a lag of 1. If "within.consecutive" then **within a given id**, two successive time periods represent a lag of 1. Finally, if the time variable is numeric, you can provide your own numeric time step.
duplicate.method If several observations have the same id and time values, then the notion of lag is not defined for them. If `duplicate.method = "none"` (default) and duplicate values are found, this leads to an error. You can use `duplicate.method = "first"` so that the first occurrence of identical id/time observations will be used as lag.

Details

This function allows you to use leads and lags in a fixest estimation without having to provide the argument `panel.id`. It also offers more options on how to set the panel (with the additional arguments `time.step` and `duplicate.method`).

When the initial data set was also a data.table, not all operations are supported and some may dissolve the fixest_panel. This is the case when creating subselections of the initial data with additional attributes (e.g. `pdt[x>0,.(x, y, z)]` would dissolve the fixest_panel, meaning only a data.table would be the result of the call).

If the initial data set was also a data.table, then you can create new variables from lags and leads using the functions `l` and `f`. See the example.

Value

It returns a data base identical to the one given in input, but with an additional attribute: “panel_info”. This attribute contains vectors used to efficiently create lags/leads of the data. When the data is subselected, some bookeeping is performed on the attribute “panel_info”.

Author(s)

Laurent Berge
See Also

The estimation methods `feols`, `fepois` and `feglm`.

The functions `l` and `f` to create lags and leads within `fixest_panel` objects.

Examples

data(base_did)

# Setting a data set as a panel...

pdat = panel(base_did, ~id+period)

# ...then using the functions l and f

est1 = feols(y~l(x1, 0:1), pdat)
est2 = feols(f(y)~l(x1, -1:1), pdat)
est3 = feols(l(y)~l(x1, 0:3), pdat)
etable(est1, est2, est3, order = c("f", "^x"), drop="Int")

# or using the argument panel.id

feols(f(y)~l(x1, -1:1), base_did, panel.id = ~id+period)

# You can use panel.id in various ways:
pdat = panel(base_did, ~id+period)
# is identical to:
pdat = panel(base_did, c("id", "period"))
# and also to:
pdat = panel(base_did, "id,period")

# l() and f() can also be used within a data.table:
if(require("data.table")){
  pdat_dt = panel(as.data.table(base_did), ~id+period)
  # Now since pdat_dt is also a data.table
  # you can create lags/leads directly
  pdat_dt[, x1_l1 := l(x1)]
  pdat_dt[, c("x1_l1_fill0", "y_f2") := .(l(x1, fill = 0), f(y, 2))]
}

plot.fixest.fixef

Displaying the most notable fixed-effects

Description

This function plots the 5 fixed-effects with the highest and lowest values, for each of the fixed-effect dimension. It takes as an argument the fixed-effects obtained from the function `fixef.fixest` after an estimation using `femlm`, `feols` or `feglm`. 
### Usage

```r
## S3 method for class 'fixef.fixest'
plot(x, n = 5, ...)
```

### Arguments

- **x**: An object obtained from the function `fixef.fixest`.
- **n**: The number of fixed-effects to be drawn. Defaults to 5.
- **...**: Not currently used.

Note that the fixed-effect coefficients might NOT be interpretable. This function is useful only for fully regular panels. If the data are not regular in the fixed-effect coefficients, this means that several ‘reference points’ are set to obtain the fixed-effects, thereby impeding their interpretation. In this case a warning is raised.

### Author(s)

Laurent Berge

### See Also

- `fixef.fixest` to extract clustered coefficients. See also the main estimation function `femlm`, `feols` or `feglm`. Use `summary.fixest` to see the results with the appropriate standard-errors, the function `etable` to visualize the results of multiple estimations.

### Examples

```r
data(trade)

# We estimate the effect of distance on trade
# => we account for 3 fixed-effects
est_pois = femlm(Euros ~ log(dist_km)|Origin+Destination+Product, trade)

# obtaining the fixed-effects coefficients
fe_trade = fixef(est_pois)

# plotting them
plot(fe_trade)
```
predict.fixest

Predict method for fixest fits

Description

This function obtains prediction from a fitted model estimated with `femlm`, `feols` or `feglm`.

Usage

```r
## S3 method for class 'fixest'
predict(
  object,
  newdata,
  type = c("response", "link"),
  se.fit = FALSE,
  interval = "none",
  level = 0.95,
  fixef = FALSE,
  vs.coef = FALSE,
  sample = c("estimation", "original"),
  vcov = NULL,
  ssc = NULL,
  ...)
```

Arguments

- **object**: A `fixest` object. Obtained using the functions `femlm`, `feols` or `feglm`.
- **newdata**: A data.frame containing the variables used to make the prediction. If not provided, the fitted expected (or linear if `type = "link"`) predictors are returned.
- **type**: Character either equal to "response" (default) or "link". If `type = "response"`, then the output is at the level of the response variable, i.e. it is the expected predictor $E(Y|X)$. If "link", then the output is at the level of the explanatory variables, i.e. the linear predictor $X \cdot \beta$.
- **se.fit**: Logical, default is FALSE. If TRUE, the standard-error of the predicted value is computed and returned in a column named `se.fit`. This feature is only available for OLS models not containing fixed-effects.
- **interval**: Either "none" (default), "confidence" or "prediction". What type of confidence interval to compute. Note that this feature is only available for OLS models not containing fixed-effects (GLM/ML models are not covered).
- **level**: A numeric scalar in between 0.5 and 1, defaults to 0.95. Only used when the argument 'interval' is requested, it corresponds to the width of the confidence interval.
**predict.fixest**

- **fixef** Logical scalar, default is FALSE. If TRUE, a data.frame is returned, with each column representing the fixed-effects coefficients for each observation in newdata — with as many columns as fixed-effects. Note that when there are variables with varying slopes, the slope coefficients are returned (i.e. they are not multiplied by the variable).

- **vs.coef** Logical scalar, default is FALSE. Only used when fixef = TRUE and when variables with varying slopes are present. If TRUE, the coefficients of the variables with varying slopes are returned instead of the coefficient multiplied by the value of the variables (default).

- **sample** Either "estimation" (default) or "original". This argument is only used when arg. 'newdata' is missing, and is ignored otherwise. If equal to "estimation", the vector returned matches the sample used for the estimation. If equal to "original", it matches the original data set (the observations not used for the estimation being filled with NAs).

- **vcov** Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the vcov documentation in the vignette.

- **ssc** An object of class ssc.type obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE). See the help of the function `ssc` for details.

- **...** Not currently used.

**Value**

It returns a numeric vector of length equal to the number of observations in argument newdata. If newdata is missing, it returns a vector of the same length as the estimation sample, except if sample = "original", in which case the length of the vector will match the one of the original data set (which can, but also cannot, be the estimation sample). If fixef = TRUE, a data.frame is returned. If se.fit = TRUE or interval != "none", the object returned is a data.frame with the following columns: fit, se.fit, and, if CIs are requested, ci_low and ci_high.

**Author(s)**

Laurent Berge

**See Also**

See also the main estimation functions `femlm`, `feols` or `feglm`. `update.fixest`, `summary.fixest`, `vcov.fixest`, `fixef.fixest`. 
Examples

# Estimation on iris data
res = fepois(Sepal.Length ~ Petal.Length | Species, iris)

# what would be the prediction if the data was all setosa?
newdata = data.frame(Petal.Length = iris$Petal.Length, Species = "setosa")
pred_setosa = predict(res, newdata = newdata)

# Let's look at it graphically
plot(c(1, 7), c(3, 11), type = "n", xlab = "Petal.Length", ylab = "Sepal.Length")
newdata = iris[order(iris$Petal.Length), ]
newdata$Species = "setosa"
lines(newdata$Petal.Length, predict(res, newdata))

# versicolor
newdata$Species = "versicolor"
lines(newdata$Petal.Length, predict(res, newdata), col=2)

# virginica
newdata$Species = "virginica"
lines(newdata$Petal.Length, predict(res, newdata), col=3)

# The original data
points(iris$Petal.Length, iris$Sepal.Length, col = iris$Species, pch = 18)
legend("topleft", lty = 1, col = 1:3, legend = levels(iris$Species))

# Getting the fixed-effect coefficients for each obs.
#
data(trade)
est_trade = fepois(Euros ~ log(dist_km) | Destination^Product + Origin^Product + Year, trade)
obs_fe = predict(est_trade, fixef = TRUE)
head(obs_fe)

# can we check we get the right sum of fixed-effects
head(cbind(rowSums(obs_fe), est_trade$sumFE))

# Standard-error of the prediction
#
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
est = feols(y ~ x1 + species, base)
head(predict(est, se.fit = TRUE))

# regular confidence interval
head(predict(est, interval = "conf"))

# adding the residual to the CI
head(predict(est, interval = "predi"))

# You can change the type of SE on the fly
head(predict(est, interval = "conf", vcov = ~species))

---

**print.fixest**

*A print facility for fixest objects.*

**Description**

This function is very similar to usual summary functions as it provides the table of coefficients along with other information on the fit of the estimation. The type of output can be customized by the user (using function `setFixest_print`).

**Usage**

```r
## S3 method for class 'fixest'
print(x, n, type = "table", fitstat = NULL, ...)
```

```r
setFixest_print(type = "table", fitstat = NULL)
```

```r
getFixest_print()
```

**Arguments**

- **x**
  A fixest object. Obtained using the methods `femlm, feols` or `feglm`.

- **n**
  Integer, number of coefficients to display. By default, only the first 8 coefficients are displayed if `x` does not come from `summary.fixest`.

- **type**
  Either "table" (default) to display the coefficients table or "coef" to display only the coefficients.

- **fitstat**
  A formula or a character vector representing which fit statistic to display. The types must be valid types of the function `fitstat`. The default fit statistics depend on the type of estimation (OLS, GLM, IV, with/without fixed-effect). Providing the argument `fitstat` overrides the default fit statistics, you can however use the point "." to summon them back. Ex 1: `fitstat = ~ . + ll` adds the log-likelihood to the default values. Ex 2: `fitstat = ~ ll + pr2` only displays the log-likelihood and the pseudo-R2.

- **...**
  Other arguments to be passed to `vcov.fixest`.
Details

It is possible to set the default values for the arguments type and fitstat by using the function setFixest_print.

Author(s)

Laurent Berge

See Also

See also the main estimation functions femlm, feols or feglm. Use summary.fixest to see the results with the appropriate standard-errors, fixef.fixest to extract the fixed-effects coefficients, and the function etable to visualize the results of multiple estimations.

Examples

```r
# Load trade data
data(trade)

# We estimate the effect of distance on trade
# => we account for 3 fixed-effects (FEs)
est_pois = fepois(Euros ~ log(dist_km)|Origin+Destination+Product, trade)

# displaying the results
# (by default SEs are clustered if FEs are used)
print(est_pois)

# By default the coefficient table is displayed.
# If the user wished to display only the coefficients, use option type:
print(est_pois, type = "coef")

# To permanently display coef. only, use setFixest_print:
setFixest_print(type = "coef")
est_pois
# back to default:
setFixest_print(type = "table")

# fitstat

# We modify which fit statistic to display
print(est_pois, fitstat = ~ . + lr)

# We add the LR test to the default (represented by the ".")

# to show only the LR stat:
print(est_pois, fitstat = ~ . + lr.stat)

# To modify the defaults:
setFixest_print(fitstat = ~ . + lr.stat + rmse)
```
print.fixest_fitstat

est_pois

# Back to default (NULL == default)
setFixest_print(fitstat = NULL)

---

print.fixest_fitstat  Print method for fit statistics of fixest estimations

### Description
Displays a brief summary of selected fit statistics from the function `fitstat`.

### Usage

```r
## S3 method for class 'fixest_fitstat'
print(x, na.rm = FALSE, ...)
```

### Arguments

- **x**: An object resulting from the `fitstat` function.
- **na.rm**: Logical, default is `FALSE`. If `TRUE`, the statistics that are missing are not displayed.
- **...**: Not currently used.

### Examples

```r
data(trade)
gravity = feols(log(Euros) ~ log(dist_km) | Destination + Origin, trade)

# Extracting the 'working' number of observations used to compute the pvalues
fitstat(gravity, "g", simplify = TRUE)

# Some fit statistics
fitstat(gravity, ~ rmse + r2 + wald + wf)

# You can use them in etable
etable(gravity, fitstat = ~ rmse + r2 + wald + wf)

# For wald and wf, you could show the pvalue instead:
etable(gravity, fitstat = ~ rmse + r2 + wald.p + wf.p)

# Now let's display some statistics that are not built-in
# => we use fitstat_register to create them

# We need: a) type name, b) the function to be applied
# c) (optional) an alias
```
fitstat_register("tstand", function(x) tstat(x, se = "stand")[1], "t-stat (regular)")
fitstat_register("thc", function(x) tstat(x, se = "heter")[1], "t-stat (HC1)")
fitstat_register("t1w", function(x) tstat(x, se = "clus")[1], "t-stat (clustered)")
fitstat_register("t2w", function(x) tstat(x, se = "twow")[1], "t-stat (2-way)")

# Now we can use these keywords in fitstat:
etable(gravity, fitstat = ~ . + tstand + thc + t1w + t2w)

# Note that the custom stats we created are can easily lead
# to errors, but that's another story!

print.fixest_multi

Print method for fixest_multi objects

Description
Displays summary information on fixest_multi objects in the R console.

Usage
## S3 method for class 'fixest_multi'
print(x, ...)

Arguments
x A fixest_multi object, obtained from a fixest estimation leading to multiple results.
...
Other arguments to be passed to summary.fixest_multi.

See Also
The main fixest estimation functions: feols, fepois, fenegbin, feglm, feNmlm. Tools for multiple fixestimations: summary.fixest_multi, print.fixest_multi, as.list.fixest_multi, sub-sub-.fixest_multi, sub-.fixest_multi.

Examples

base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# Multiple estimation
res = feols(y ~ csw(x1, x2, x3), base, split = ~species)

# Let’s print all that
res
Description

Reports different R2s for fixest estimations (e.g. \texttt{feglm} or \texttt{feols}).

Usage

\begin{verbatim}
r2(x, type = "all", full_names = FALSE)
\end{verbatim}

Arguments

- **x**: A \texttt{fixest} object, e.g. obtained with function \texttt{feglm} or \texttt{feols}.
- **type**: A character vector representing the R2 to compute. The R2 codes are of the form: "wapr2" with letters "w" (within), "a" (adjusted) and "p" (pseudo) possibly missing. E.g. to get the regular R2: use \texttt{type = "r2"}, the within adjusted R2: use \texttt{type = "war2"}, the pseudo R2: use \texttt{type = "pr2"}, etc. Use \texttt{"cor2"} for the squared correlation. By default, all R2s are computed.
- **full_names**: Logical scalar, default is \texttt{FALSE}. If \texttt{TRUE} then names of the vector in output will have full names instead of keywords (e.g. \texttt{Squared Correlation} instead of \texttt{cor2}, etc).

Details

The pseudo R2s are the McFaddens R2s, that is the ratio of log-likelihoods.

For R2s with no theoretical justification, like e.g. regular R2s for maximum likelihood models – or within R2s for models without fixed-effects, NA is returned. The single measure to possibly compare all kinds of models is the squared correlation between the dependent variable and the expected predictor.

The pseudo-R2 is also returned in the OLS case, it corresponds to the pseudo-R2 of the equivalent GLM model with a Gaussian family.

For the adjusted within-R2s, the adjustment factor is \( \frac{(n - nb\_fe)}{(n - nb\_fe - K)} \) with \( n \) the number of observations, \( nb\_fe \) the number of fixed-effects and \( K \) the number of variables.

Value

Returns a named vector.

Author(s)

Laurent Berge
Examples

```r
# Load trade data
data(trade)

# We estimate the effect of distance on trade (with 3 fixed-effects)
est = feols(log(Euros) ~ log(dist_km) | Origin+Destination+Product, trade)

# Squared correlation:
r2(est, "cor2")

# "regular" r2:
r2(est, "r2")

# pseudo r2 (equivalent to GLM with Gaussian family)
r2(est, "pr2")

# adjusted within r2
r2(est, "war2")

# all four at once
r2(est, c("cor2", "r2", "pr2", "war2"))

# same with full names instead of codes
r2(est, c("cor2", "r2", "pr2", "war2"), full_names = TRUE)
```

---

**ref**

*Refactors a variable*

**Description**

Takes a variable of any type, transforms it into factors, and modifies the values of the factors. Useful in estimations when you want to set some value of a vector as a reference.

**Usage**

```r
ref(x, ref)
```

**Arguments**

- **x**
  - A vector of any type (must be atomic though).
- **ref**
  - A vector or a list, or special binning values (explained later). If a vector, it must correspond to (partially matched) values of the vector `x`. The vector `x` which will be transformed into a factor and these values will be placed first in the levels. That’s the main usage of this function. You can also bin on-the-fly the values of `x`, using the same syntax as the function `bin`. To create a new value from old values, use `ref = list("new_value"=old_values)` with `old_values` a vector
of existing values. You can use .() for list(). It accepts regular expressions, but they must start with an "@", like in ref="@Aug|Dec". It accepts one-sided formulas which must contain the variable x, e.g. ref=list("<2" = ~x < 2). The names of the list are the new names. If the new name is missing, the first value matched becomes the new name. In the name, adding "@d", with d a digit, will relocate the value in position d: useful to change the position of factors. If the vector x is numeric, you can use the special value "bin::digit" to group every digit element. For example if x represents years, using ref="bin::2" creates bins of two years. With any data, using "!bin::digit" groups every digit consecutive values starting from the first value. Using "!!bin::digit" is the same but starting from the last value. With numeric vectors you can: a) use "cut::n" to cut the vector into n equal parts, b) use "cut::a]b[" to create the following bins: [min, a], a, b[, [b, max]. The latter syntax is a sequence of number/quartile (q0 to q4)/percentile (p0 to p100) followed by an open or closed square bracket. You can add custom bin names by adding them in the character vector after 'cut::values'. See details and examples. Dot square bracket expansion (see dsb) is enabled.

Value

It returns a factor of the same length as x, where levels have been modified according to the argument ref.

"Cutting" a numeric vector

Numeric vectors can be cut easily into: a) equal parts, b) user-specified bins.

Use "cut::n" to cut the vector into n (roughly) equal parts. Percentiles are used to partition the data, hence some data distributions can lead to create less than n parts (for example if P0 is the same as P50).

The user can specify custom bins with the following syntax: "cut::a]b[". Here the numbers a, b, c, etc, are a sequence of increasing numbers, each followed by an open or closed square bracket. The numbers can be specified as either plain numbers (e.g. "cut::5]12["), quartiles (e.g. "cut::q1]q3["), or percentiles (e.g. "cut::p10]p15]p90["). Values of different types can be mixed: "cut::5]q2[p80[" is valid provided the median (q2) is indeed greater than 5, otherwise an error is thrown.

The square bracket right of each number tells whether the numbers should be included or excluded from the current bin. For example, say x ranges from 0 to 100, then "cut::5]" will create two bins: one from 0 to 5 and a second from 6 to 100. With "cut::5[" the bins would have been 0-4 and 5-100.

A factor is always returned. The labels always report the min and max values in each bin.

To have user-specified bin labels, just add them in the character vector following 'cut::values'. You don’t need to provide all of them, and NA values fall back to the default label. For example, bin = c("cut::4", "Q1", NA, "Q3") will modify only the first and third label that will be displayed as "Q1" and "Q3".
bin vs ref

The functions `bin` and `ref` are able to do the same thing, then why use one instead of the other? Here are the differences:

- `ref` always returns a factor. This is in contrast with `bin` which returns, when possible, a vector of the same type as the vector in input.
- `ref` always places the values modified in the first place of the factor levels. On the other hand, `bin` tries not to modify the ordering of the levels. It is possible to make `bin` mimic the behavior of `ref` by adding an "@" as the first element of the list in the argument `bin`.
- When a vector (and not a list) is given in input, `ref` will place each element of the vector in the first place of the factor levels. The behavior of `bin` is totally different, `bin` will transform all the values in the vector into a single value in x (i.e. it's binning).

Author(s)

Laurent Berge

See Also

To bin the values of a vector: `bin`.

Examples

data(airquality)

# A vector of months
month_num = airquality$Month
month_lab = c("may", "june", "july", "august", "september")
month_fact = factor(month_num, labels = month_lab)
table(month_num)
table(month_fact)

# Main use

# Without argument: equivalent to as.factor
ref(month_num)

# Main usage: to set a level first:
# (Note that partial matching is enabled.)
table(ref(month_fact, "aug"))

# You can rename the level on-the-fly
# (Northern hemisphere specific!)
table(ref(month_fact, .("Hot month"="aug",
                       "Late summer" = "sept")))

# Main use is in estimations:
a = feols(Petal.Width ~ Petal.Length + Species, iris)

# We change the reference
b = feols(Petal.Width ~ Petal.Length + ref(Species, "vers"), iris)

tetable(a, b)

#
# Binning
#

# You can also bin factor values on the fly
# Using @ first means a regular expression will be used to match the values.
# Note that the value created is placed first.
# To avoid that behavior => use the function "bin"
table(ref(month_fact, .((summer = "@jul|aug|sep")))

# Please refer to the example in the bin help page for more example.
# The syntax is the same.

#
# Precise relocation
#

# You can place a factor at the location you want
# by adding "@digit" in the name first:
table(ref(month_num, .("@5")=5)))

# Same with renaming
table(ref(month_num, .("@5 five")=5)))

---

**Description**

Simple function that replicates fixest objects while (optionally) computing different standard-errors. Useful mostly in combination with `etable` or `coefplot`.

**Usage**

```r
## S3 method for class 'fixest'
rep(x, times = 1, each = 1, vcov, ...)

## S3 method for class 'fixest_list'
rep(x, times = 1, each = 1, vcov, ...)
```
Arguments

x Either a `fixest` object, either a list of `fixest` objects created with `.l()`. times Integer vector giving the number of repetitions of the vector of elements. By default times = 1. It must be either of length 1, either of the same length as the argument x.
each Integer scalar indicating the repetition of each element. Default is 1.
vcov A list containing the types of standard-error to be computed, default is missing. If not missing, it must be of the same length as times, each, or the final vector. Note that if the arguments times and each are missing, then times becomes equal to the length of vcov. To see how to summon a VCOV, see the dedicated section in the vignette.

Details

To apply `rep.fixest` on a list of `fixest` objects, it is absolutely necessary to use `.l()` and not `list()`. Value

Returns a list of the appropriate length. Each element of the list is a `fixest` object.

Examples

```r
# Let's show results with different standard-errors
est = feols(Ozone ~ Solar.R + Wind + Temp, data = airquality)
my_vcov = list(~ Month, ~ Day, ~ Day + Month)
etable(rep(est, vcov = my_vcov))
coefplot(rep(est, vcov = my_vcov), drop = "Int")

# To rep multiple objects, you need to use .l()
#
est_bis = feols(Ozone ~ Solar.R + Wind + Temp | Month, airquality)
etable(rep(.l(est, est_bis), vcov = my_vcov))

# using each
etable(rep(.l(est, est_bis), each = 3, vcov = my_vcov))
```
Description

This function extracts residuals from a fitted model estimated with femlm, feols or feglm.

Usage

```r
## S3 method for class 'fixest'
resid(
  object,
  type = c("response", "deviance", "pearson", "working"),
  na.rm = TRUE,
  ...
)
## S3 method for class 'fixest'
residuals(
  object,
  type = c("response", "deviance", "pearson", "working"),
  na.rm = TRUE,
  ...
)
```

Arguments

- `object`: A fixest object. Obtained using the functions femlm, feols or feglm.
- `type`: A character scalar, either "response" (default), "deviance", "pearson", or "working". Note that the "working" corresponds to the residuals from the weighted least square and only applies to feglm models.
- `na.rm`: Logical, default is TRUE. Whether to remove the observations with NAs from the original data set. If FALSE, then the vector returned is always of the same length as the original data set.
- `...`: Not currently used.

Value

It returns a numeric vector of the length the number of observations used for the estimation (if na.rm = TRUE) or of the length of the original data set (if na.rm = FALSE).

Author(s)

Laurent Berge
See Also

See also the main estimation functions `femlm`, `feols` or `feglm`, `fitted.fixest`, `predict.fixest`, `summary.fixest`, `vcov.fixest`, `fixef.fixest`.

Examples

```r
# simple estimation on iris data, using "Species" fixed-effects
res_poisson = femlm(Sepal.Length ~ Sepal.Width + Petal.Length +
                     Petal.Width | Species, iris)

# we plot the residuals
plot(resid(res_poisson))
```

---

**resid.fixest_multi**  
*Extracts the residuals from a fixest_multi object*

**Description**

Utility to extract the residuals from multiple fixest estimations. If possible, all the residuals are coerced into a matrix.

**Usage**

```r
## S3 method for class 'fixest_multi'
resid(
  object,
  type = c("response", "deviance", "pearson", "working"),
  na.rm = FALSE,
  ...
)

## S3 method for class 'fixest_multi'
residuals(
  object,
  type = c("response", "deviance", "pearson", "working"),
  na.rm = FALSE,
  ...
)
```

**Arguments**

- `object`  
  A fixest_multi object.

- `type`  
  A character scalar, either "response" (default), "deviance", "pearson", or "working". Note that the "working" corresponds to the residuals from the weighted least square and only applies to `feglm` models.
**sample_df**

Sample data set randomly.

Description

This function is useful to check a data set. It gives a random number of rows of the input data set.

Usage

```
sample_df(x, n = 10, previous = FALSE)
```

Arguments

- **x**: A data set: either a vector, a matrix or a data frame.
- **n**: The number of random rows/elements to sample randomly.
- **previous**: Logical scalar. Whether the results of the previous draw should be returned.

Value

A data base (resp vector) with \( n \) rows (resp elements).

Author(s)

Laurent Berge
Examples

```r
sample_df(iris)

sample_df(iris, previous = TRUE)
```

Description

You can set the default values of most arguments of `coefplot` with this function.

Usage

```r
setFixest_coefplot(
  style,
  horiz = FALSE,
  dict = getFixest_dict(),
  keep,
  ci.width = "1%",
  ci_level = 0.95,
  pt.pch = 20,
  pt.bg = NULL,
  cex = 1,
  pt.cex = cex,
  col = 1:8,
  pt.col = col,
  ci.col = col,
  lwd = 1,
  pt.lwd = lwd,
  ci.lwd = lwd,
  ci.lty = 1,
  grid = TRUE,
  grid.par = list(lty = 3, col = "gray"),
  zero = TRUE,
  zero.par = list(col = "black", lwd = 1),
  pt.join = FALSE,
  pt.join.par = list(col = pt.col, lwd = lwd),
  ci.join = FALSE,
  ci.join.par = list(lwd = lwd, col = col, lty = 2),
  ci.fill = FALSE,
  ci.fill.par = list(col = "lightgray", alpha = 0.5),
  ref.line = "auto",
  ref.line.par = list(col = "black", lty = 2),
  lab.cex,
)
setFixest_coefplot

```r
lab.min.cex = 0.85,
lab.max.mar = 0.25,
lab.fit = "auto",
xlim.add,
ylim.add,
sep,
bg,
group = "auto",
group.par = list(lwd = 2, line = 3, tcl = 0.75),
main = "Effect on __depvar__",
value.lab = "Estimate and __ci__ Conf. Int.",
ylab = NULL,
xlab = NULL,
sub = NULL,
reset = FALSE)
```

`getFixest_coefplot()`

**Arguments**

- **style**: A character scalar giving the style of the plot to be used. You can set styles with the function `setFixest_coefplot`, setting all the default values of the function. If missing, then it switches to either "default" or "iplot", depending on the calling function.
- **horiz**: A logical scalar, default is `FALSE`. Whether to display the confidence intervals horizontally instead of vertically.
- **dict**: A named character vector or a logical scalar. It changes the original variable names to the ones contained in the dictionary. E.g. to change the variables named `a` and `b3` to (resp.) "$\log(a)$" and to "$\text{bonus}^3$", use `dict=c(a="$\log(a)$",b3="$\text{bonus}^3$")`. By default, it is equal to `getFixest_dict()`, a default dictionary which can be set with `setFixest_dict`. You can use `dict = FALSE` to disable it. By default `dict` modifies the entries in the global dictionary, to disable this behavior, use "reset" as the first element (ex: `dict=c("reset", mpg="Miles per gallon")`).
- **keep**: Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see `base::regex` help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument `dict`). Example: you have the variable `x1` to `x55` and want to display only `x1` to `x9`, then you could use `keep = "x[[:digit:]]\$"`. If the first character is an exclamation mark, the effect is reversed (e.g. `keep = "!Intercept"` means: every variable that does not contain "Intercept" is kept). See details.
- **ci.width**: The width of the extremities of the confidence intervals. Default is `0.1`.
- **ci_level**: Scalar between 0 and 1: the level of the CI. By default it is equal to 0.95.
- **pt.pch**: The patch of the coefficient estimates. Default is 1 (circle).
- **pt.bg**: The background color of the point estimate (when the `pt.pch` is in 21 to 25). Defaults to `NULL`. 
- **cex**: Numeric, default is 1. Expansion factor for the points.
- **pt.cex**: The size of the coefficient estimates. Default is the other argument cex.
- **col**: The color of the points and the confidence intervals. Default is 1 ("black"). Note that you can set the colors separately for each of them with pt.col and ci.col.
- **pt.col**: The color of the coefficient estimates. Default is equal to the other argument col.
- **ci.col**: The color of the confidence intervals. Default is equal to the other argument col.
- **lwd**: General line width. Default is 1.
- **pt.lwd**: The line width of the coefficient estimates. Default is equal to the other argument lwd.
- **ci.lwd**: The line width of the confidence intervals. Default is equal to the other argument lwd.
- **ci.lty**: The line type of the confidence intervals. Default is 1.
- **grid**: Logical, default is TRUE. Whether a grid should be displayed. You can set the display of the grid with the argument grid.par.
- **grid.par**: List. Parameters of the grid. The default values are: lty = 3 and col = "gray". You can add any graphical parameter that will be passed to graphics::abline. You also have two additional arguments: use horiz = FALSE to disable the horizontal lines, and use vert = FALSE to disable the vertical lines. Eg: grid.par = list(vert = FALSE, col = "red", lwd = 2).
- **zero**: Logical, default is TRUE. Whether the 0-line should be emphasized. You can set the parameters of that line with the argument zero.par.
- **zero.par**: List. Parameters of the zero-line. The default values are col = "black" and lwd = 1. You can add any graphical parameter that will be passed to graphics::abline. Example: zero.par = list(col = "darkblue", lwd = 3).
- **pt.join**: Logical, default is FALSE. If TRUE, then the coefficient estimates are joined with a line.
- **pt.join.par**: List. Parameters of the line joining the coefficients. The default values are: col = pt.col and lwd = lwd. You can add any graphical parameter that will be passed to lines. Eg: pt.join.par = list(lty = 2).
- **ci.join**: Logical default to FALSE. Whether to join the extremities of the confidence intervals. If TRUE, then you can set the graphical parameters with the argument ci.join.par.
- **ci.join.par**: A list of parameters to be passed to graphics::lines. Only used if ci.join=TRUE. By default it is equal to list(lwd = lwd, col = col, lty = 2).
- **ci.fill**: Logical default to FALSE. Whether to fill the confidence intervals with a color. If TRUE, then you can set the graphical parameters with the argument ci.fill.par.
- **ci.fill.par**: A list of parameters to be passed to graphics::polygon. Only used if ci.fill=TRUE. By default it is equal to list(col = "lightgray", alpha = 0.5). Note that alpha is a special parameter that adds transparency to the color (ranges from 0 to 1).
ref.line Logical or numeric, default is "auto", whose behavior depends on the situation. It is TRUE only if: i) interactions are plotted, ii) the x values are numeric and iii) a reference is found. If TRUE, then a vertical line is drawn at the level of the reference value. Otherwise, if numeric a vertical line will be drawn at that specific value.

ref.line.par List. Parameters of the vertical line on the reference. The default values are: col = "black" and lty = 2. You can add any graphical parameter that will be passed to graphics::abline. Eg: ref.line.par = list(lty = 1, lwd = 3).

lab.cex The size of the labels of the coefficients. Default is missing. It is automatically set by an internal algorithm which can go as low as lab.min.cex (another argument).

lab.min.cex The minimum size of the coefficients labels, as set by the internal algorithm. Default is 0.85.

lab.max.mar The maximum size the left margin can take when trying to fit the coefficient labels into it (only when horiz = TRUE). This is used in the internal algorithm fitting the coefficient labels. Default is 0.25.

lab.fit The method to fit the coefficient labels into the plotting region (only when horiz = FALSE). Can be "auto" (the default), "simple", "multi" or "tilted". If "simple", then the classic axis is drawn. If "multi", then the coefficient labels are fit horizontally across several lines, such that they don’t collide. If "tilted", then the labels are tilted. If "auto", an automatic choice between the three is made.

xlim.add A numeric vector of length 1 or 2. It represents an extension factor of xlim, in percentage. Eg: xlim.add = c(0, 0.5) extends xlim of 50% on the right. If of length 1, positive values represent the right, and negative values the left (Eg: xlim.add = -0.5 is equivalent to xlim.add = c(0.5, 0)).

ylim.add A numeric vector of length 1 or 2. It represents an extension factor of ylim, in percentage. Eg: ylim.add = c(0, 0.5) extends ylim of 50% on the top. If of length 1, positive values represent the top, and negative values the bottom (Eg: ylim.add = -0.5 is equivalent to ylim.add = c(0.5, 0)).

sep The distance between two estimates – only when argument object is a list of estimation results.

bg Background color for the plot. By default it is white.

group A list, default is missing. Each element of the list reports the coefficients to be grouped while the name of the element is the group name. Each element of the list can be either: i) a character vector of length 1, ii) of length 2, or ii) a numeric vector. If equal to: i) then it is interpreted as a pattern: all element fitting the regular expression will be grouped (note that you can use the special character "^^" to clean the beginning of the names, see example), if ii) it corresponds to the first and last elements to be grouped, if iii) it corresponds to the coefficients numbers to be grouped. If equal to a character vector, you can use a percentage to tell the algorithm to look at the coefficients before aliasing (e.g. "%varname"). Example of valid uses: group=list(group_name="pattern"), group=list(group_name=c("var_start", "var_end"), group=list(group_name=1:2)). See details.

group.par A list of parameters controlling the display of the group. The parameters controlling the line are: lwd, tcl (length of the tick), line_adj (adjustment of the
position, default is 0), tick (whether to add the ticks), lwd.ticks, col.ticks.

Then the parameters controlling the text: text.adj (adjustment of the position, default is 0), text.cex, text.font, text.col.

main
The title of the plot. Default is "Effect on __depvar__". You can use the special variable __depvar__ to set the title (useful when you set the plot default with setFixest_coefplot).

value.lab
The label to appear on the side of the coefficient values. If horiz = FALSE, the label appears in the y-axis. If horiz = TRUE, then it appears on the x-axis. The default is equal to "Estimate and __ci__ Conf. Int.", with __ci__ a special variable giving the value of the confidence interval.

ylab
The label of the y-axis, default is NULL. Note that if horiz = FALSE, it overrides the value of the argument value.lab.

xlab
The label of the x-axis, default is NULL. Note that if horiz = TRUE, it overrides the value of the argument value.lab.

sub
A subtitle, default is NULL.

reset
Logical, default is TRUE. If TRUE, then the arguments that are not set during the call are reset to their “factory”-default values. If FALSE, on the other hand, arguments that have already been modified are not changed.

Value

Doesn’t return anything.

See Also
coeffplot

Examples

# coefplot has many arguments, which makes it highly flexible.
# If you don’t like the default style of coefplot. No worries,
# you can set *your* default by using the function
# setFixest_coefplot()

# Estimation
est = feols(Petal.Length ~ Petal.Width + Sepal.Length + Sepal.Width | Species, iris)

# Plot with default style
coeffplot(est)

# Now we permanently change some arguments
dict = c("Petal.Length"="Length (Petal)", "Petal.Width"="Width (Petal)",
"Sepal.Length"="Length (Sepal)", "Sepal.Width"="Width (Sepal)"

setFixest_coefplot(ci.col = 2, pt.col = "darkblue", ci.lwd = 3,
pt.cex = 2, pt.pch = 15, ci.width = 0, dict = dict)
# Tadaaa!
coefplot(est)

# To reset to the default settings:
setFixest_coefplot("all", reset = TRUE)
coefplot(est)

---

**setFixest_dict**  
*Sets/gets the dictionary relabeling the variables*

### Description

Sets/gets the default dictionary used in the function `etable`, `did_means` and `coefplot`. The dictionaries are used to relabel variables (usually towards a fancier, more explicit formatting) when exporting them into a LaTeX table or displaying in graphs. By setting the dictionary with `setFixest_dict`, you can avoid providing the argument `dict`.

### Usage

```r
setFixest_dict(dict = NULL, ..., reset = FALSE)
getFixest_dict()
```

### Arguments

- **dict**
  A named character vector or a character scalar. E.g. to change my variable named "a" and "b" to (resp.) "$\log(a)$" and "$\text{bonus}^3\$", then use `dict = c(a="$\log(a)$", b3="$\text{bonus}^3\$")`. Alternatively you can feed a character scalar containing the dictionary in the form "variable 1: definition
variable 2: definition". In that case the function `as.dict` will be applied to get a proper dictionary. This dictionary is used in LaTeX tables or in graphs by the function `coefplot`. If you want to separate LaTeX rendering from rendering in graphs, use an ampersand first to make the variable specific to `coefplot`.

- **...**
  You can add arguments of the form: `variable_name = "Definition"`. This is an alternative to using a named vector in the argument `dict`.

- **reset**
  Logical, default is `FALSE`. If `TRUE`, then the dictionary is reset. Note that the default dictionary always relabels the variable "(Intercept)" in to "Constant". To overwrite it, you need to add "(Intercept)" explicitly in your dictionary.

### Details

By default the dictionary only grows. This means that successive calls with not erase the previous definitions unless the argument `reset` has been set to `TRUE`.

The default dictionary is equivalent to having `setFixest_dict("(Intercept)" = "Constant")`. To change this default, you need to provide a new definition to "(Intercept)" explicitly.
Author(s)
Laurent Berge

Examples

data(trade)
est = feols(log(Euros) ~ log(dist_km)|Origin+Destination+Product, trade)
# we export the result & rename some variables
etable(est, dict = c("log(Euros)"="Euros (ln)", Origin="Country of Origin"))

# If you export many tables, it can be more convenient to use setFixest_dict:
setFixest_dict(c("log(Euros)"="Euros (ln)", Origin="Country of Origin"))
etable(est) # variables are properly relabeled

# The dictionary only 'grows'
# Here you get the previous two variables + the new one that are relabeled
# Btw you set the dictionary directly using the argument names:
setFixest_dict(Destination = "Country of Destination")
etable(est)

# Another way to set a dictionary: with a character string:
# See the help page of as.dict
dict = "log(dist_km): Distance (ln); Product: Type of Good"
setFixest_dict(dict)
etable(est)

# And now we reset:
setFixest_dict(reset = TRUE)
etable(est)

---

Description
This function sets globally the default arguments of fixest estimations.

Usage

setFixest_estimation(
  data = NULL,
  panel.id = NULL,
  fixef.rm = "perfect",
  fixef.tol = 1e-06,
  fixef.iter = 10000,
  collin.tol = 1e-10,
  lean = FALSE,
setFixest_estimation

verbose = 0,
warn = TRUE,
combine.quick = NULL,
demeaned = FALSE,
mem.clean = FALSE,
glm.iter = 25,
glm.tol = 1e-08,
reset = FALSE
}

getFixest_estimation()

Arguments

data A data.frame containing the necessary variables to run the model. The variables of
the non-linear right hand side of the formula are identified with this
data.frame names. Can also be a matrix.

panel.id The panel identifiers. Can either be: i) a one sided formula (e.g. panel.id =
~id+time), ii) a character vector of length 2 (e.g. panel.id=c('id', 'time'),
or iii) a character scalar of two variables separated by a comma (e.g. panel.id='id,time').
Note that you can combine variables with ^ only inside formulas (see the dedi-
cated section in feols).

fixef.rm Can be equal to "perfect" (default), "singleton", "both" or "none". Controls
which observations are to be removed. If "perfect", then observations having a
fixed-effect with perfect fit (e.g. only 0 outcomes in Poisson estimations) will be
removed. If "singleton", all observations for which a fixed-effect appears only
once will be removed. The meaning of "both" and "none" is direct.

fixef.tol Precision used to obtain the fixed-effects. Defaults to 1e-5. It corresponds to the
maximum absolute difference allowed between two coefficients of successive it-
erations. Argument fixef.tol cannot be lower than 10000*.Machine$double.eps.
Note that this parameter is dynamically controlled by the algorithm.

fixef.iter Maximum number of iterations in fixed-effects algorithm (only in use for 2+
fixed-effects). Default is 10000.

collin.tol Numeric scalar, default is 1e-10. Threshold deciding when variables should be
considered collinear and subsequently removed from the estimation. Higher
values means more variables will be removed (if there is presence of collinear-
ity). One signal of presence of collinearity is t-stats that are extremely low (for
instance when t-stats < 1e-3).

lean Logical, default is FALSE. If TRUE then all large objects are removed from the
returned result: this will save memory but will block the possibility to use many
methods. It is recommended to use the arguments se or cluster to obtain
the appropriate standard-errors at estimation time, since obtaining different SEs
won't be possible afterwards.

verbose Integer. Higher values give more information. In particular, it can detail the
number of iterations in the demeaning algorithm (the first number is the left-
hand-side, the other numbers are the right-hand-side variables).
setFixest_estimation

**warn** Logical, default is **TRUE**. Whether warnings should be displayed (concerns warnings relating to convergence state).

**combine.quick** Logical. When you combine different variables to transform them into a single fixed-effects you can do e.g. \( y \sim x | \text{paste}(\text{var1}, \text{var2}) \). The algorithm provides a shorthand to do the same operation: \( y \sim x | \text{var1}^\text{var2} \). Because pasting variables is a costly operation, the internal algorithm may use a numerical trick to hasten the process. The cost of doing so is that you lose the labels. If you are interested in getting the value of the fixed-effects coefficients after the estimation, you should use `combine.quick = FALSE`. By default it is equal to **FALSE** if the number of observations is lower than 50,000, and to **TRUE** otherwise.

**demeaned** Logical, default is **FALSE**. Only used in the presence of fixed-effects: should the centered variables be returned? If **TRUE**, it creates the items \( y_{\text{demeaned}} \) and \( X_{\text{demeaned}} \).

**mem.clean** Logical, default is **FALSE**. Only to be used if the data set is large compared to the available RAM. If **TRUE** then intermediary objects are removed as much as possible and `gc` is run before each substantial C++ section in the internal code to avoid memory issues.

**glm.iter** Number of iterations of the glm algorithm. Default is 25.

**glm.tol** Tolerance level for the glm algorithm. Default is \( 1 \times 10^{-8} \).

**reset** Logical scalar, default is **FALSE**. Whether to reset all values.

**Value**

The function `getFixest_estimation` returns the currently set global defaults.

**Examples**

```r
# Example: removing singletons is **FALSE** by default

# => changing this default

# Let's create data with singletons
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")
base$fe_singletons = as.character(base$species)

res = feols(y ~ x1 + x2 | fe_singletons, base)
res_noSingle = feols(y ~ x1 + x2 | fe_singletons, base, fixef.rm = "single")

# New defaults
setFixest_estimation(fixef.rm = "single")
res_newDefault = feols(y ~ x1 + x2 | fe_singletons, base)

etable(res, res_noSingle, res_newDefault)
```
# Resetting the defaults
setFixest_estimation(reset = TRUE)

setFixest_fml

Sets/gets formula macros

Description
You can set formula macros globally with `setFixest_fml`. These macros can then be used in `fixest` estimations or when using the function `xpd`.

Usage

```r
setFixest_fml(..., reset = FALSE)

getFixest_fml()
```

Arguments

- `...` Definition of the macro variables. Each argument name corresponds to the name of the macro variable. It is required that each macro variable name starts with two dots (e.g. `..ctrl`). The value of each argument must be a one-sided formula or a character vector, it is the definition of the macro variable. Example of a valid call: `setFixest_fml(..ctrl = ~ var1 + var2)`. In the function `xpd`, the default macro variables are taken from `getFixest_fml`, any variable in `...` will replace these values. You can enclose values in `[]`, if so they will be evaluated from the current environment. For example `..ctrl = ~ x.[1:2] + .[z]` will lead to `~x1 + x2 + var` if `z` is equal to `"var"`.

- `reset` A logical scalar, defaults to `FALSE`. If `TRUE`, all macro variables are first reset (i.e. deleted).

Details

In `xpd`, the default macro variables are taken from `getFixest_fml`. Any value in the `...` argument of `xpd` will replace these default values.

The definitions of the macro variables will replace in verbatim the macro variables. Therefore, you can include multipart formulas if you wish but then beware of the order the the macros variable in the formula. For example, using the `airquality` data, say you want to set as controls the variable Temp and Day fixed-effects, you can do `setFixest_fml(..ctrl = ~Temp | Day)`, but then `feols(Ozone ~ Wind + ..ctrl, airquality)` will be quite different from `feols(Ozone ~ ..ctrl + Wind, airquality)`, so beware!
Value

The function `getFixest_fml()` returns a list of character strings, the names corresponding to the macro variable names, the character strings corresponding to their definition.

See Also

`xpd` to make use of formula macros.

Examples

```r
# Small examples with airquality data
data(airquality)
# we set two macro variables
setFixest_fml(..ctrl = ~ Temp + Day,
             ..ctrl_long = ~ poly(Temp, 2) + poly(Day, 2))

# Using the macro in lm with xpd:
lm(xpd(Ozone ~ Wind + ..ctrl), airquality)
lm(xpd(Ozone ~ Wind + ..ctrl_long), airquality)

# You can use the macros without xpd() in fixest estimations
a = feols(Ozone ~ Wind + ..ctrl, airquality)
b = feols(Ozone ~ Wind + ..ctrl_long, airquality)
etable(a, b, keep = "Int|Win")

# Using .[
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
i = 2:3
z = "species"
lm(xpd(y ~ x.[2:3] + [z]), base)

# No xpd() needed in feols
feols(y ~ x.[2:3] + [z], base)

# Auto completion with '..' suffix
#
# You can trigger variables autocompletion with the '..' suffix
# You need to provide the argument data
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
xpd(y ~ x.., data = base)

# In fixest estimations, this is automatically taken care of
feols(y ~ x.., data = base)

# You can use xpd for stepwise estimations
```
# Note that for stepwise estimations in fixest, you can use the stepwise functions: sw, sw0, csw, csw0
# see help in feols or in the dedicated vignette

# we want to look at the effect of x1 on y controlling for different variables

base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# We first create a matrix with all possible combinations of variables
my_args = lapply(names(base)[-1:2], function(x) c("", x))
(all_combs = as.matrix(do.call("expand.grid", my_args)))

res_all = list()
for(i in 1:nrow(all_combs)){
    res_all[[i]] = feols(xpd(y ~ x1 + ..v, ..v = all_combs[i, ]), base)
}
etable(res_all)
coefplot(res_all, group = list(Species = "^^species"))

# You can use macros to grep variables in your data set

# Example 1: setting a macro variable globally

data(longley)
setFixest_fml(\..many_vars = grep("GNP|ployed", names(longley), value = TRUE))
feols(Armed.Forces ~ Population + \..many_vars, longley)

# Example 2: using \..("regex") or regex("regex") to grep the variables "live"

feols(Armed.Forces ~ Population + \..("GNP|ployed"), longley)

# Example 3: same as Ex.2 but without using a fixest estimation

# Here we need to use xpd():
lm(xpd(Armed.Forces ~ Population + regex("GNP|ployed"), data = longley), longley)

# Stepwise estimation with regex: use a comma after the parenthesis
feols(Armed.Forces ~ Population + sw(regex("GNP|ployed")), longley)

# Multiple LHS
etable(feols(\..("GNP|ployed") ~ Population, longley))

# lhs and rhs arguments

# to create a one sided formula from a character vector
vars = letters[1:5]
xpd(rhs = vars)

# Alternatively, to replace the RHS
xpd(y ~ 1, rhs = vars)

# To create a two sided formula
xpd(lhs = "y", rhs = vars)

#
# argument 'add'
#

xpd(~x1, add = ~ x2 + x3)

# also works with character vectors
xpd(~x1, add = c("x2", "x3"))

# only adds to the RHS
xpd(y ~ x, add = ~ bon + jour)

#
# Dot square bracket operator
#

# The basic use id to add variables in the formula
x = c("x1", "x2")
xpd(y ~ .[x])

# Alternatively, one-sided formulas can be used and their content will be inserted verbatim
x = ~x1 + x2
xpd(y ~ .[x])

# You can create multiple variables at once
xpd(y ~ x.[1:5] + z.[2:3])

# You can summon variables from the environment to complete variables names
var = "a"
xpd(y ~ x.[var])

# ... the variables can be multiple
vars = LETTERS[1:3]
xpd(y ~ x.[vars])

# You can have "complex" variable names but they must be nested in character form
xpd(y ~ .["x.[vars]_sq"])

# DSB can be used within regular expressions
re = c("GNP", "Pop")
xpd(Unemployed ~ regex(".[re]"), data = longley)
# => equivalent to regex("GNP|Pop")
# Use \.[,var] (NOTE THE COMMA!) to expand with commas
# !! can break the formula if misused
vars = c("wage", "unemp")
xpd(c(y.[,1:3]) ~ csw(.[,vars]))

# Example of use of [.] within a loop
res_all = list()
for(p in 1:3){
  res_all[[p]] = feols(Ozone ~ Wind + poly(Temp, .[p]), airquality)
}
etable(res_all)

# The former can be compactly estimated with:
res_compact = feols(Ozone ~ Wind + sw(.[, "poly(Temp, .[1:3])"], airquality)
etable(res_compact)

# How does it work?
# 1) .[, stuff] evaluates stuff and, if a vector, aggregates it with commas
# Comma aggregation is done thanks to the comma placed after the square bracket
# If .[stuff], then aggregation is with sums.
# 2) stuff is evaluated, and if it is a character string, it is evaluated with
# the function dsb which expands values in []
# Wrapping up:
# 2) evaluation of dsb("poly(Temp, .[1:3])") leads to the vector:
# c("poly(Temp, 1)", "poly(Temp, 2)", "poly(Temp, 3)")
# 1) .[, c("poly(Temp, 1)", "poly(Temp, 2)", "poly(Temp, 3)")] leads to
# poly(Temp, 1), poly(Temp, 2), poly(Temp, 3)
# Hence sw(.[, "poly(Temp, .[1:3])"])) becomes:
# sw(poly(Temp, 1), poly(Temp, 2), poly(Temp, 3))

# In non-fixest functions: guessing the data allows to use regex
#
# When used in non-fixest functions, the algorithm tries to "guess" the data
# so that ..("regex") can be directly evaluated without passing the argument 'data'
data(longley)
lm(xpd(Armed.Forces ~ Population + ..("GNP|ployed")), longley)

# same for the auto completion with '..'
lm(xpd(Armed.Forces ~ Population + GN..), longley)
**setFixest_notes**  
Sets/gets whether to display notes in fixest estimation functions

**Description**  
Sets/gets the default values of whether notes (informing for NA and observations removed) should be displayed in fixest estimation functions.

**Usage**  
`setFixest_notes(x)`  
`getFixest_notes()`  

**Arguments**  

- `x`  
  A logical. If FALSE, then notes are permanently removed.

**Author(s)**  
Laurent Berge

**Examples**

```r  
# Change default with  
setFixest_notes(FALSE)  
feols(Ozone ~ Solar.R, airquality)  

# Back to default which is TRUE  
setFixest_notes(TRUE)  
feols(Ozone ~ Solar.R, airquality)  
```

---

**setFixest_nthreads**  
Sets/gets the number of threads to use in fixest functions

**Description**  
Sets/gets the default number of threads to used in fixest estimation functions. The default is the maximum number of threads minus two.

**Usage**  
`setFixest_nthreads(nthreads, save = FALSE)`  
`getFixest_nthreads()`
Arguments

- **nthreads**: The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. If missing, the default is to use 50% of all threads.

- **save**: Either a logical or equal to "reset". Default is FALSE. If TRUE then the value is set permanently at the project level, this means that if you restart R, you will still obtain the previously saved defaults. This is done by writing in the ".Renviron" file, located in the project’s working directory, hence we must have write permission there for this to work, and only works with Rstudio. If equal to "reset", the default at the project level is erased. Since there is writing in a file involved, permission is asked to the user.

Author(s)

Laurent Berge

Examples

```r
# Gets the current number of threads
(nthreads_origin = getFixest_nthreads())

# To set multi-threading off:
setFixest_nthreads(1)

# To set it back to default at startup:
setFixest_nthreads()

# And back to the original value
setFixest_nthreads(nthreads_origin)
```

Description

This function defines or extracts the default type of standard-errors to be computed in `fixest summary` and `vcov`.

Usage

```r
setFixest_vcov(
  no_FE = "iid",
  one_FE = "cluster",
  two_FE = "cluster",
)```
```r
panel = "cluster",
all = NULL,
reset = FALSE
)

getFixest_vcov()

Arguments

- **no_FE**
  Character scalar equal to either: "iid" (default), or "hetero". The type of standard-errors to use by default for estimations without fixed-effects.

- **one_FE**
  Character scalar equal to either: "iid", "hetero", "cluster" (default). The type of standard-errors to use by default for estimations with one fixed-effect.

- **two_FE**
  Character scalar equal to either: "iid", "hetero", "cluster" (default), or "twoway". The type of standard-errors to use by default for estimations with two or more fixed-effects.

- **panel**
  Character scalar equal to either: "iid", "hetero", "cluster" (default), or "driscoll_kraay". The type of standard-errors to use by default for estimations with the argument `panel.id` set up. Note that `panel` has precedence over the presence of fixed-effects.

- **all**
  Character scalar equal to either: "iid", or "hetero". By default is is NULL. If provided, it sets all the SEs to that value.

- **reset**
  Logical, default is FALSE. Whether to reset to the default values.

Value

The function `getFixest_vcov()` returns a list with three elements containing the default for estimations i) without, ii) with one, or iii) with two or more fixed-effects.

Examples

```r
# By default:
# - no fixed-effect (FE): standard
# - one or more FEs: cluster
# - panel: cluster on panel id

data(base_did)
est_no_FE = feols(y ~ x1, base_did)
est_one_FE = feols(y ~ x1 | id, base_did)
est_two_FE = feols(y ~ x1 | id + period, base_did)
est_panel = feols(y ~ x1 | id + period, base_did, panel.id = ~id + period)

etable(est_no_FE, est_one_FE, est_two_FE)

# Changing the default standard-errors
setFixest_vcov(no_FE = "hetero", one_FE = "iid",
               two_FE = "twoway", panel = "drisc")
etable(est_no_FE, est_one_FE, est_two_FE, est_panel)
```
# Resetting the defaults
setFixest_vcov(reset = TRUE)

---

sigma.fixest  |  Residual standard deviation of fixest estimations

## Description

Extract the estimated standard deviation of the errors from fixest estimations.

## Usage

```r
## S3 method for class 'fixest'
sigma(object, ...)
```

## Arguments

- `object`: A fixest object.
- `...`: Not currently used.

## Value

Returns a numeric scalar.

## See Also

- `feols`, `fepois`, `feglm`, `fenegbin`, `feNmlm`.

## Examples

```r
est = feols(Petal.Length ~ Petal.Width, iris)
sigma(est)
```
**ssc**

*Governs the small sample correction in fixest VCOVs*

**Description**

Provides how the small sample correction should be calculated in `vcov.fixest/summary.fixest`.

**Usage**

```r
ssc(
  adj = TRUE,
  fixef.K = "nested",
  cluster.adj = TRUE,
  cluster.df = "min",
  t.df = "min",
  fixef.force_exact = FALSE
)

dof(
  adj = TRUE,
  fixef.K = "nested",
  cluster.adj = TRUE,
  cluster.df = "min",
  t.df = "min",
  fixef.force_exact = FALSE
)

setFixest_ssc(ssc.type = ssc())

getFixest_ssc()
```

**Arguments**

- **adj**
  - Logical scalar, defaults to TRUE. Whether to apply a small sample adjustment of the form \((n - 1) / (n - K)\), with \(K\) the number of estimated parameters. If FALSE, then no adjustment is made.

- **fixef.K**
  - Character scalar equal to "nested" (default), "none" or "full". In the small sample adjustment, how to account for the fixed-effects parameters. If "none", the fixed-effects parameters are discarded, meaning the number of parameters \(K\) is only equal to the number of variables. If "full", then the number of parameters is equal to the number of variables plus the number of fixed-effects. Finally, if "nested", then the number of parameters is equal to the number of variables plus the number of fixed-effects that are not nested in the clusters used to cluster the standard-errors.

- **cluster.adj**
  - Logical scalar, default is TRUE. How to make the small sample correction when clustering the standard-errors? If TRUE a \(G/(G-1)\) correction is performed with \(G\) the number of cluster values.
cluster.df Either "conventional" or "min" (default). Only relevant when the variance-covariance matrix is two-way clustered (or higher). It governs how the small sample adjustment for the clusters is to be performed. [Sorry for the jargon that follows.] By default a unique adjustment is made, of the form $\frac{G_{\min}}{(G_{\min}-1)}$ with $G_{\min}$ the smallest $G_i$. If cluster.df="conventional" then the i-th "sandwich" matrix is adjusted with $\frac{G_i}{(G_i-1)}$ with $G_i$ the number of unique clusters.

t.df Either "conventional", "min" (default) or an integer scalar. Only relevant when the variance-covariance matrix is clustered. It governs how the p-values should be computed. By default, the degrees of freedom of the Student t distribution is equal to the minimum size of the clusters with which the VCOV has been clustered minus one. If t.df="conventional", then the degrees of freedom of the Student t distribution is equal to the number of observations minus the number of estimated variables. You can also pass a number to manually specify the DoF of the t-distribution.

fixef.force_exact Logical, default is FALSE. If there are 2 or more fixed-effects, these fixed-effects they can be irregular, meaning they can provide the same information. If so, the "real" number of parameters should be lower than the total number of fixed-effects. If fixef.force_exact = TRUE, then fixef.fixest is first run to determine the exact number of parameters among the fixed-effects. Mostly, panels of the type individual-firm require fixef.force_exact = TRUE (but it adds computational costs).

ssc.type An object of class ssc.type obtained with the function ssc.

Details

The following vignette: On standard-errors, describes in details how the standard-errors are computed in fixest and how you can replicate standard-errors from other software.

Value

It returns a ssc.type object.

Functions

- dof(): This function is deprecated and will be removed at some point (in 6 months from August 2021). Exactly the same as ssc.

Author(s)

Laurent Berge

See Also

summary.fixest, vcov.fixest
Examples

# Equivalence with lm/glm standard-errors
#
# LM
# In the absence of fixed-effects,
# by default, the standard-errors are computed in the same way
res = feols(Petal.Length ~ Petal.Width + Species, iris)
res_lm = lm(Petal.Length ~ Petal.Width + Species, iris)
vcov(res) / vcov(res_lm)

# GLM
# By default, there is no small sample adjustment in glm, as opposed to feglm.
# To get the same SEs, we need to use ssc(adj = FALSE)
res_pois = fepois(round(Petal.Length) ~ Petal.Width + Species, iris)
res_glm = glm(round(Petal.Length) ~ Petal.Width + Species, iris, family = poisson())
vcov(res_pois, ssc = ssc(adj = FALSE)) / vcov(res_glm)

# Same example with the Gamma
res_gamma = feglm(round(Petal.Length) ~ Petal.Width + Species, iris, family = Gamma())
res_glm_gamma = glm(round(Petal.Length) ~ Petal.Width + Species, iris, family = Gamma())
vcov(res_gamma, ssc = ssc(adj = FALSE)) / vcov(res_glm_gamma)

# Fixed-effects corrections
#
# We create "irregular" FEs
base = data.frame(x = rnorm(10))
base$y = base$x + rnorm(10)
base$fe1 = rep(1:3, c(4, 3, 3))
base$fe2 = rep(1:5, each = 2)
est = feols(y ~ x | fe1 + fe2, base)

# fe1: 3 FEs
# fe2: 5 FEs

# Clustered standard-errors: by fe1
#
# Default: fixef.K = "nested"
# \Rightarrow adjustment K = 1 + 5 (i.e. x + fe2)
summary(est)
attributes(vcov(est, attr = TRUE))[[c("ssc", "dof.K")]]
# fixef.K = FALSE
# => adjustment K = 1 (i.e. only x)
summary(est, ssc = ssc(fixef.K = "none"))
attr(vcov(est, ssc = ssc(fixef.K = "none"), attr = TRUE), "dof.K")

# fixef.K = TRUE
# => adjustment K = 1 + 3 + 5 - 1 (i.e. x + fe1 + fe2 - 1 restriction)
summary(est, ssc = ssc(fixef.K = "full"))
attr(vcov(est, ssc = ssc(fixef.K = "full"), attr = TRUE), "dof.K")

# fixef.K = TRUE & fixef.force_exact = TRUE
# => adjustment K = 1 + 3 + 5 - 2 (i.e. x + fe1 + fe2 - 2 restrictions)
summary(est, ssc = ssc(fixef.K = "full", fixef.force_exact = TRUE))
attr(vcov(est, ssc = ssc(fixef.K = "full", fixef.force_exact = TRUE), attr = TRUE), "dof.K")

# There are two restrictions:
attr(fixef(est), "references")

# To permanently set the default ssc:
#
# eg no small sample adjustment:
setFixest_ssc(ssc(adj = FALSE))

# Factory default
setFixest_ssc()

---

### stepwise

**Stepwise estimation tools**

---

**Description**

Functions to perform stepwise estimations.

**Usage**

```
sw(...)
csw(...)
sw0(...)
csw0(...)
```

**Arguments**

```
...
```

Represents formula variables to be added in a stepwise fashion to an estimation.
Details

To include multiple independent variables, you need to use the stepwise functions. There are 4 stepwise functions: sw, sw0, csw, csw0. Let’s explain that.

Assume you have the following formula: \( f_{\text{ml}} = y \sim x_1 + \text{sw}(x_2, x_3) \). The stepwise function \( \text{sw} \) will estimate the following two models: \( y \sim x_1 + x_2 \) and \( y \sim x_1 + x_3 \). That is, each element in \( \text{sw}() \) is sequentially, and separately, added to the formula. Would have you used \( \text{sw0} \) in lieu of \( \text{sw} \), then the model \( y \sim x_1 \) would also have been estimated. The \( 0 \) in the name implies that the model without any stepwise element will also be estimated.

Finally, the prefix \( c \) means cumulative: each stepwise element is added to the next. That is, \( f_{\text{ml}} = y \sim x_1 + \text{csw}(x_2, x_3) \) would lead to the following models \( y \sim x_1 + x_2 \) and \( y \sim x_1 + x_2 + x_3 \). The \( 0 \) has the same meaning and would also lead to the model without the stepwise elements to be estimated: in other words, \( f_{\text{ml}} = y \sim x_1 + \text{csw0}(x_2, x_3) \) leads to the following three models: \( y \sim x_1 \), \( y \sim x_1 + x_2 \) and \( y \sim x_1 + x_2 + x_3 \).

Examples

```r
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# Regular stepwise
feols(y ~ sw(x1, x2, x3), base)

# Cumulative stepwise
feols(y ~ csw(x1, x2, x3), base)

# Using the 0
feols(y ~ x1 + x2 + sw0(x3), base)
```

Description

This function describes the style of data.frames created with the function `etable`.

Usage

```r
style.df(  
    depvar.title = "Dependent Var.":",  
    fixef.title = "Fixed-Effects:",  
    fixef.line = "-",  
    fixef.prefix = "",  
    fixef.suffix = "",  
    slopes.title = "Varying Slopes:",  
    slopes.line = "-"),
```
slopes.format = "__var__ (__slope__)",
stats.title = "_",
stats.line = "_",
yesNo = c("Yes", "No"),
headers.sep = TRUE,
signif.code = c("***" = 0.001, "**" = 0.01, "+" = 0.05, "." = 0.1),
interaction.combine = " x ",
i.equal = " = ",
default = FALSE
)

Arguments

depvar.title Character scalar. Default is "Dependent Var.:". The row name of the dependent variables.

fixef.title Character scalar. Default is "Fixed-Effects:". The header preceding the fixed-effects. If equal to the empty string, then this line is removed.

fixef.line A single character. Default is "-". A character that will be used to create a line of separation for the fixed-effects header. Used only if fixef.title is not the empty string.

fixef.prefix Character scalar. Default is "". A prefix to appear before each fixed-effect name.

fixef.suffix Character scalar. Default is "". A suffix to appear after each fixed-effect name.

slopes.title Character scalar. Default is "Varying Slopes:". The header preceding the variables with varying slopes. If equal to the empty string, then this line is removed.

slopes.line Character scalar. Default is "-". A character that will be used to create a line of separation for the variables with varying slopes header. Used only if slopes.line is not the empty string.

slopes.format Character scalar. Default is "__var__ (__slope__)". The format of the name of the varying slopes. The values __var__ and __slope__ are special characters that will be replaced by the value of the variable name and slope name, respectively.

stats.title Character scalar. Default is "_.". The header preceding the statistics section. If equal to the empty string, then this line is removed. If equal to single character (like in the default), then this character will be expanded to take the full column width.

stats.line Character scalar. Default is "_.". A character that will be used to create a line of separation for the statistics header. Used only if stats.title is not the empty string.

yesNo Character vector of length 1 or 2. Default is c("Yes", "No"). Used to inform on the presence or absence of fixed-effects in the estimation. If of length 1, then automatically the second value is considered as the empty string.

headers.sep Logical, default is TRUE. Whether to add a line of separation between the headers and the coefficients.
signif.code

Named numeric vector, used to provide the significance codes with respect to the p-value of the coefficients. Default is \texttt{c("***"=0.001, "**"=0.01, "+"=0.05, "."=0.10)}. To suppress the significance codes, use \texttt{signif.code=NA} or \texttt{signif.code=NULL}.

Can also be equal to "letters", then the default becomes \texttt{c("a"=0.01, "b"=0.05, "+"=0.10)}.

interaction.combine

Character scalar, defaults to " x ". When the estimation contains interactions, then the variables names (after aliasing) are combined with this argument. For example: if \texttt{dict = c(x1="Wind", x2="Rain")} and you have the following interaction \texttt{x1:x2}, then it will be renamed (by default) \texttt{Wind x Rain} – using \texttt{interaction.combine = "+"} would lead to \texttt{Wind*Rain}.

i.equal

Character scalar, defaults to " = ". Only affects factor variables created with the function \texttt{i}, tells how the variable should be linked to its value. For example if you have the Species factor from the iris data set, by default the display of the variable is \texttt{Species = Setosa}, etc. If \texttt{i.equal = ": "} the display becomes \texttt{Species: Setosa}.

default

Logical, default is \texttt{FALSE}. If \texttt{TRUE}, all the values not provided by the user are set to their default.

Details

@inheritParams etable

The title elements (\texttt{depvar.title}, \texttt{fixef.title}, \texttt{slopes.title} and \texttt{stats.title}) will be the row names of the returned data.frame. Therefore keep in mind that any two of them should not be identical (since identical row names are forbidden in data.frames).

Value

It returns an object of class \texttt{fixest_style_df}.

Examples

# Multiple estimations => see details in feols
aq = airquality
est = feols(c(Ozone, Solar.R) ~ Wind + csw(Temp, Temp^2, Temp^3) | Month + Day, data = aq)

# Default result
etable(est)

# Playing a bit with the styles
etable(est, style.df = style.df(fixef.title = "", fixef.suffix = " FE", stats.line = " ", yesNo = "yes"))
Style definitions for LaTeX tables

Description

This function describes the style of LaTeX tables to be exported with the function `etab`.e.

Usage

```r
style.tex(
  main = "base",
  depvar.title,
  model.title,
  model.format,
  line.top,
  line.bottom,
  var.title,
  fixef.title,
  fixef.prefix,
  fixef.suffix,
  fixef.where,
  slopes.title,
  slopes.format,
  fixef_sizes.prefix,
  fixef_sizes.suffix,
  stats.title,
  notes.intro,
  notes.tpt.intro,
  tablefoot,
  tablefoot.value,
  yesNo,
  tabular = "normal",
  depvar.style,
  no_border,
  caption.after,
  rules_width,
  signif.code,
  tpt,
  arraystretch,
  adjustbox = NULL,
  fontsize,
  interaction.combine = "$\times$",
  i.equal = "$==$"
)
```
Arguments

main
Either "base", "aer" or "qje". Defines the basic style to start from. The styles "aer" and "qje" are almost identical and only differ on the top/bottom lines.

depvar.title
A character scalar. The title of the line of the dependent variables (defaults to "Dependent variable(s):" if main = "base" (the 's' appears only if just one variable) and to "" if main = "aer").

model.title
A character scalar. The title of the line of the models (defaults to "Model:" if main = "base" and to "" if main = "aer").

model.format
A character scalar. The value to appear on top of each column. It defaults to "(1)". Note that 1, i, I, a and A are special characters: if found, their values will be automatically incremented across columns.

line.top
A character scalar equal to "simple", "double", or anything else. The line at the top of the table (defaults to "double" if main = "base" and to "simple" if main = "aer"). "simple" is equivalent to "\toprule", and "double" to "\tabularnewline \midrule \midrule".

line.bottom
A character scalar equal to "simple", "double", or anything else. The line at the bottom of the table (defaults to "double" if main = "base" and to "simple" if main = "aer"). "simple" is equivalent to "\bottomrule", and "double" to "\midrule \midrule & \tabularnewline".

var.title
A character scalar. The title line appearing before the variables (defaults to "\midrule \emph{Variables}" if main = "base" and to "\midrule" if main = "aer"). Note that the behavior of var.title = " " (a space) is different from var.title = "" (the empty string): in the first case you will get an empty row, while in the second case you get no empty row. To get a line without an empty row, use "\midrule" (and not "\midrule ")--the space!.

fixef.title
A character scalar. The title line appearing before the fixed-effects (defaults to "\midrule \emph{Fixed-effects}" if main = "base" and to "" if main = "aer"). Note that the behavior of fixef.title = " " (a space) is different from fixef.title = "" (the empty string): in the first case you will get an empty row, while in the second case you get no empty row. To get a line without an empty row, use "\midrule" (and not "\midrule ")--the space!.

fixef.prefix
A prefix to add to the fixed-effects names. Defaults to "" (i.e. no prefix).

fixef.suffix
A suffix to add to the fixed-effects names. Defaults to "" if main = "base" and to "fixed-effects" if main = "aer").

fixef.where
Either "var" or "stats". Where to place the fixed-effects lines? Defaults to "var", i.e. just after the variables, if main = "base") and to "stats", i.e. just after the statistics, if main = "aer").

slopes.title
A character scalar. The title line appearing before the variables with varying slopes (defaults to "\midrule \emph{Varying Slopes}" if main = "base" and to "" if main = "aer"). Note that the behavior of slopes.title = " " (a space) is different from slopes.title = "" (the empty string): in the first case you will get an empty row, while in the second case you get no empty row. To get a line without an empty row, use "\midrule" (and not "\midrule ")--the space!.

slopes.prefix
A prefix to add to the varying-slopes names. Defaults to "" (i.e. no prefix).

slopes.suffix
A suffix to add to the varying-slopes names. Defaults to "" if main = "base" and to "varying-effects" if main = "aer").
slopes.format Character scalar representing the format of the slope variable name. There are two special characters: "\texttt{\textvar}\text{" and "\texttt{\textslope}\text{"}, placeholders for the variable and slope names. Defaults to "\texttt{\_\_\textvar\_ (\_\_\textslope\_\_)}" if main = "base" and to "\texttt{\_\_\textvar\_ $\times \_\_\textslope\_\_}\text{" if main = "aer"}).

fixef_sizes.prefix A prefix to add to the fixed-effects names. Defaults to "# \text{".

fixef_sizes.suffix A suffix to add to the fixed-effects names. Defaults to "\text{" (i.e. no suffix).

stats.title A character scalar. The title line appearing before the statistics (defaults to "\midrule \texttt{Fit statistics}\text{" if main = "base" and to "\text{" if main = "aer"}). Note that the behavior of stats.title = "\text{" (a space) is different from stats.title = "\text{" (the empty string): in the first case you will get an empty row, while in the second case you get no empty row. To get a line without an empty row, use "\midrule" (and not "\midrule \text{"!–the space!).

notes.intro A character scalar. Some tex code appearing just before the notes, defaults to "\par \texttt{\raggedright} \text{".

notes.tpt.intro Character scalar. Only used if tpt = TRUE, it is some tex code that is passed before any threeparttable item (can be used for, typically, the font size). Default is the empty string.

tablefoot A logical scalar. Whether or not to display a footer within the table. Defaults to TRUE if main = "aer") and FALSE if main = "aer")

tablefoot.value A character scalar. The notes to be displayed in the footer. Defaults to "default" if main = "base", which leads to custom footers informing on the type of standard-error and significance codes, depending on the estimations.

yesNo A character vector of length 1 or 2. Defaults to "\texttt{Yes}" if main = "base" and to "$\checkmark$" if main = "aer" (from package amssymb). This is the message displayed when a given fixed-effect is (or is not) included in a regression. If yesNo is of length 1, then the second element is the empty string.

tabular (Tex only.) Character scalar equal to "normal" (default), "*" or "\texttt{\textchi}\text{". Represents the type of tabular environment to use: either tabular, tabular* or tabularx.

depvar.style Character scalar equal to either "\text{" (default), "*" (italic), "**" (bold), "***" (italic-bold). How the name of the dependent variable should be displayed.

no_border Logical, default is FALSE. Whether to remove any side border to the table (typically adds @{\ to the sides of the tabular).

caption.after Character scalar. Tex code that will be placed right after the caption. Defaults to "\" for main = "base" and "\medskip" for main = "aer".

rules_width Character vector of length 1 or 2. This vector gives the width of the booktabs rules: the first element the heavy-width, the second element the light-width. NA values mean no modification. If of length 1, only the heavy rules are modified. The width are in Latex units (ex: "0.1 em", etc).

signif.code Named numeric vector, used to provide the significance codes with respect to the p-value of the coefficients. Default is c("***"=0.01, "**"=0.05, "*"=0.10). To suppress the significance codes, use signif.code=NA or signif.code=NULL.
Can also be equal to "letters", then the default becomes \c{a=0.01, b=0.05, c=0.10}.

**tpt**  
(Tex only.) Logical scalar, default is FALSE. Whether to use the threeparttable environment. If so, the notes will be integrated into the tablenotes environment.

**arraystretch**  
(Tex only.) A numeric scalar, default is NULL. If provided, the command \renewcommand{arraystretch}{x} is inserted, replacing x by the value of arraystretch. The changes are specific to the current table and do not affect the rest of the document.

**adjustbox**  
(Tex only.) A logical, numeric or character scalar, default is NULL. If not NULL, the table is inserted within the adjustbox environment. By default the options are width = 1\textwidth, center (if TRUE). A numeric value changes the value before \textwidth. You can also add a character of the form "x tw" or "x th" with x a number and where tw (th) stands for text-width (text-height). Finally any other character value is passed verbatim as an adjustbox option.

**fontsize**  
(Tex only.) A character scalar, default is NULL. Can be equal to tiny, scriptsize, footnotesize, small, normalsize, large, or Large. The change affect the table only (and not the rest of the document).

**interaction.combine**  
Character scalar, defaults to "$\times$". When the estimation contains interactions, then the variables names (after aliasing) are combined with this argument. For example: if \texttt{dict = c(x1="Wind", x2="Rain"}) and you have the following interaction x1:x2, then it will be renamed (by default) Wind $\times$ Rain – using interaction.c\texttt{ombine = "*"} would lead to Wind*Rain.

**i.equal**  
Character scalar, defaults to "$=\$". Only affects factor variables created with the function \texttt{i}, tells how the variable should be linked to its value. For example if you have the Species factor from the iris data set, by default the display of the variable is Species $=$ Setosa, etc. If \texttt{i.equal = ": "} the display becomes Species: Setosa.

**Details**

The \checkmark command, used in the "aer" style (in argument yesNo), is in the amssymb package.

The commands \toprule, \midrule and \bottomrule are in the booktabs package. You can set the width of the top/bottom rules with \setlength\heavyrulewidth\{wd\}, and of the midrule with \setlength\lightrulewidth\{wd\}.

Note that all titles (depvar.title, depvar.title, etc) are not escaped, so they must be valid Latex expressions.

**Value**

Returns a list containing the style parameters.

**See Also**

etable
Examples

```r
# Multiple estimations => see details in feols
aq = airquality
est = feols(c(Ozone, Solar.R) ~
    Wind + csw(Temp, Temp^2, Temp^3) | Month + Day,
    data = aq)

# Playing a bit with the styles
etable(est, tex = TRUE)
etable(est, tex = TRUE, style.tex = style.tex("aer"))
etable(est, tex = TRUE, style.tex = style.tex("aer",
    var.title = \textit{Expl. Vars.},
    model.format = "[i]",
    yesNo = "x",
    tabular = "*"))
```

**summary.fixest**

Summary of a fixest object. Computes different types of standard errors.

**Description**

This function is similar to `print.fixest`. It provides the table of coefficients along with other information on the fit of the estimation. It can compute different types of standard errors. The new variance covariance matrix is an object returned.

**Usage**

```r
## S3 method for class 'fixest'
summary(
    object,
    vcov = NULL,
    cluster = NULL,
    ssc = NULL,
    .vcov = NULL,
    stage = NULL,
    lean = FALSE,
    agg = NULL,
    forceCovariance = FALSE,
    se = NULL,
    keepBounded = FALSE,
    n = 1000,
    nthreads = getFixest_nthreads(),
    ...
)
```
### S3 method for class 'fixest_list'

```r
summary(
  object,
  se,
  cluster,
  ssc = getFixest_ssc(),
  .vcov,
  stage = 2,
  lean = FALSE,
  n,
  ...
)
```

#### Arguments

- **object**
  A `fixest` object. Obtained using the functions `femlm`, `feols` or `feglm`.

- **vcov**
  Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the `vcov` documentation in the vignette.

- **cluster**
  Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over `var1` and `var2` contained in the data.frame `base` used for the estimation. All the following cluster arguments are valid and do the same thing: `cluster = base[, c("var1", "var2")], cluster = c("var1", "var2").cluster = ~var1+var2`. If the two variables were used as fixed-effects in the estimation, you can leave it blank with `vcov = "twoway"` (assuming `var1` [resp. `var2`] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using `^` with the following syntax: `cluster = ~var1^var2` or `cluster = "var1^var2"`.

- **ssc**
  An object of class `ssc.type` obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: `adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE)`. See the help of the function `ssc` for details.

- **.vcov**
  A user provided covariance matrix or a function computing this matrix. If a matrix, it must be a square matrix of the same number of rows as the number of variables estimated. If a function, it must return the previously mentioned matrix.

- **stage**
  Can be equal to 2 (default), 1, 1:2 or 2:1. Only used if the object is an IV estimation: defines the stage to which `summary` should be applied. If `stage = 1`
and there are multiple endogenous regressors or if stage is of length 2, then an object of class `fixest_multi` is returned.

**lean**

Logical, default is `FALSE`. Used to reduce the (memory) size of the summary object. If `TRUE`, then all objects of length N (the number of observations) are removed from the result. Note that some `fixest` methods may consequently not work when applied to the summary.

**agg**

A character scalar describing the variable names to be aggregated, it is pattern-based. For `sunab` estimations, the following keywords work: "att", "period", "cohort" and `FALSE` (to have full disaggregation). All variables that match the pattern will be aggregated. It must be of the form "(root)", the parentheses must be there and the resulting variable name will be "root". You can add another root with parentheses: "(root1)regex(root2)", in which case the resulting name is "root1::root2". To name the resulting variable differently you can pass a named vector: `c("name" = "pattern")` or `c("name" = "pattern(root2)")`. It’s a bit intricate sorry, please see the examples.

**forceCovariance**

(Advanced users.) Logical, default is `FALSE`. In the peculiar case where the obtained Hessian is not invertible (usually because of collinearity of some variables), use this option to force the covariance matrix, by using a generalized inverse of the Hessian. This can be useful to spot where possible problems come from.

**se**

Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "threeway" or "fourway"? By default if there are clusters in the estimation: `se = "cluster"`, otherwise `se = "iid"`. Note that this argument is deprecated, you should use `vcov` instead.

**keepBounded**

(Advanced users – `feNmlm` with non-linear part and bounded coefficients only.) Logical, default is `FALSE`. If `TRUE`, then the bounded coefficients (if any) are treated as unrestricted coefficients and their S.E. is computed (otherwise it is not).

**n**

Integer, default is 1000. Number of coefficients to display when the print method is used.

**nthreads**

The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function `setFixest_nthreads`.

**...**

Only used if the argument `.vocv` is provided and is a function: extra arguments to be passed to that function.

**Value**

It returns a `fixest` object with:

- `cov.scaled`: The new variance-covariance matrix (computed according to the argument `se`).
- `se`: The new standard-errors (computed according to the argument `se`).
- `coeftable`: The table of coefficients with the new standard errors.
Compatibility with sandwich package

The VCOVs from sandwich can be used with feols, feglm and fepois estimations. If you want to have a sandwich VCOV when using summary.fixest, you can use the argument vcov to specify the VCOV function to use (see examples). Note that if you do so and you use a formula in the cluster argument, an innocuous warning can pop up if you used several non-numeric fixed-effects in the estimation (this is due to the function expand.model.frame used in sandwich).

Author(s)
Laurent Berge

See Also
See also the main estimation functions femlm, feols or feglm. Use fixef.fixest to extract the fixed-effects coefficients, and the function etable to visualize the results of multiple estimations.

Examples

# Load trade data
data(trade)

# We estimate the effect of distance on trade (with 3 fixed-effects)
est_pois = fepois(Euros ~ log(dist_km)|Origin+Destination+Product, trade)

# Comparing different types of standard errors
sum_standard = summary(est_pois, vcov = "iid")
sum_hetero = summary(est_pois, vcov = "hetero")
sum_oneway = summary(est_pois, vcov = "cluster")
sum_twoway = summary(est_pois, vcov = "twoway")

etable(sum_standard, sum_hetero, sum_oneway, sum_twoway)

# Alternative ways to cluster the SE:
summary(est_pois, vcov = cluster ~ Product + Origin)
summary(est_pois, vcov = ~Product + Origin)
summary(est_pois, cluster = ~Product + Origin)

# You can interact the clustering variables "live" using the var1 ^ var2 syntax."
summary(est_pois, vcov = ~Destination^Product)

# Newey-West and Driscoll-Kraay SEs
#
data(base_did)
# Simple estimation on a panel
est = feols(y ~ x1, base_did)

# --
# Newey-West
# Use the syntax NW ~ unit + time
summary(est, NW ~ id + period)

# Now take a lag of 3:
summary(est, NW(3) ~ id + period)

# --
# Driscoll-Kraay
# Use the syntax DK ~ time
summary(est, DK ~ period)

# Now take a lag of 3:
summary(est, DK(3) ~ period)

#--
# Implicit deductions
# When the estimation is done with a panel.id, you don't need to
# specify these values.
est_panel = feols(y ~ x1, base_did, panel.id = ~id + period)

# Both methods, NM and DK, now work automatically
summary(est_panel, "NW")
summary(est_panel, "DK")

# VCOVs robust to spatial correlation
#
data(quakes)
est_geo = feols(depth ~ mag, quakes)

# --
# Conley
# Use the syntax: conley(cutoff) ~ lat + lon
# with lat/lon the latitude/longitude variable names in the data set
summary(est_geo, conley(100) ~ lat + long)

# Change the cutoff, and how the distance is computed
summary(est_geo, conley(200, distance = "spherical") ~ lat + long)

# --
# Implicit deduction
# By default the latitude and longitude are directly fetched in the data based
# on pattern matching. So you don't have to specify them.
# Further an automatic cutoff is computed by default.

# The following works
summary(est_geo, "conley")
# Compatibility with sandwich
#

# You can use the VCVs from sandwich by using the argument .vcov:
library(sandwich)
summary(est_pois, .vcov = vcovCL, cluster = trade[, c("Destination", "Product")])

## S3 method for class 'fixef.fixest'
summary(object, n = 5, ...)

### Arguments

- **object**: An object returned by the function `fixef.fixest`.
- **n**: Positive integer, defaults to 5. The n first fixed-effects for each fixed-effect dimension are reported.
- **...**: Not currently used.

### Value

It prints the number of fixed-effect coefficients per fixed-effect dimension, as well as the number of fixed-effects used as references for each dimension, and the mean and variance of the fixed-effect coefficients. Finally, it reports the first 5 (arg. n) elements of each fixed-effect.

### Author(s)

Laurent Berge

### See Also

- `femlm`, `fixef.fixest`, `plot.fixest.fixef`. 
Summary information for fixest_multi objects. In particular, this is used to specify the type of standard-errors to be computed.

Usage

```r
## S3 method for class 'fixest_multi'
summary(
  object,
  type = "short",
  vcov = NULL,
  se = NULL,
  cluster = NULL,
  ssc = NULL,
  .vcov = NULL,
  stage = 2,
  lean = FALSE,
  n = 1000,
  ...
)
```

Arguments

- `object` A fixest_multi object, obtained from a fixest estimation leading to multiple results.
type
A character either equal to "short", "long", "compact", "se_compact" or "se_long". If short, only the table of coefficients is displayed for each estimation. If long, then the full results are displayed for each estimation. If compact, a data.frame is returned with one line per model and the formatted coefficients + standard-errors in the columns. If se_compact, a data.frame is returned with one line per model, one numeric column for each coefficient and one numeric column for each standard-error. If "se_long", same as "se_compact" but the data is in a long format instead of wide.

vcov, .vcov
Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: vcov_type ~ variables. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from vcov_cluster, vcov_NW, NW, vcov_DK, DK, vcov_conley and conley. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the vcov documentation in the vignette.

se
Character scalar. Which kind of standard error should be computed: "standard", "hetero", "cluster", "twoway", "threeway" or "fourway"? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.

cluster
Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

ssc
An object of class ssc.type obtained with the function ssc. Represents how the degree of freedom correction should be done. You must use the function ssc for this argument. The arguments and defaults of the function ssc are: adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE). See the help of the function ssc for details.

stage
Can be equal to 2 (default), 1, 1:2 or 2:1. Only used if the object is an IV estimation: defines the stage to which summary should be applied. If stage = 1 and there are multiple endogenous regressors or if stage is of length 2, then an object of class fixest_multi is returned.

lean
Logical, default is FALSE. Used to reduce the (memory) size of the summary object. If TRUE, then all objects of length N (the number of observations) are removed from the result. Note that some fixest methods may consequently not work when applied to the summary.

n
Integer, default is 1000. Number of coefficients to display when the print method is used.

... Not currently used.
sunab

Value

It returns either an object of class fixest_multi (if type equals short or long), either a data.frame (if type equals compact or se_compact).

See Also

The main fixest estimation functions: feols, fepois, fenegbin, feglm, feNmlm. Tools for multiple fixest estimations: summary.fixest_multi, print.fixest_multi, as.list.fixest_multi, sub-sub-.fixest_multi, sub-.fixest_multi.

Examples

```r
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# Multiple estimation
res = feols(y ~ csw(x1, x2, x3), base, split = ~species)

# By default, the type is "short"
# You can still use the arguments from summary.fixest
summary(res, se = "hetero")
summary(res, type = "long")
summary(res, type = "compact")
summary(res, type = "se_compact")
summary(res, type = "se_long")
```

---

Sun and Abraham interactions

Description

User-level method to implement staggered difference-in-difference estimations a la Sun and Abraham (Journal of Econometrics, forthcoming).

Usage

```r
sunab(
    cohort,
    period,
    ref.c = NULL,
    ref.p = -1,
    bin,
```
sunab

```rin.rel, 
bin.c, 
bin.p, 
at = FALSE, 
no_agg = FALSE
)

sunab_att(cohort, period, ref.c = NULL, ref.p = -1)
```

**Arguments**

- **cohort**
  A vector representing the cohort. It should represent the period at which the treatment has been received (and thus be fixed for each unit).

- **period**
  A vector representing the period. It can be either a relative time period (with negative values representing the before the treatment and positive values after the treatment), or a regular time period. In the latter case, the relative time period will be created from the cohort information (which represents the time at which the treatment has been received).

- **ref.c**
  A vector of references for the cohort. By default the never treated cohorts are taken as reference and the always treated are excluded from the estimation. You can add more references with this argument, which means that dummies will not be created for them (but they will remain in the estimation).

- **ref.p**
  A vector of references for the (relative!) period. By default the first relative period (RP) before the treatment, i.e. -1, is taken as reference. You can instead use your own references (i.e. RPs for which dummies will not be created – but these observations remain in the sample). Please note that you will need at least two references. You can use the special variables `.F` and `.L` to access the first and the last relative periods.

- **bin**
  A list of values to be grouped, a vector, or the special value "bin::digit". The binning will be applied to both the cohort and the period (to bin them separately, see `bin.c` and `bin.p`). To create a new value from old values, use `bin = list("new_value"=old_values)` with `old_values` a vector of existing values. It accepts regular expressions, but they must start with an "@", like in `bin="@Aug|Dec"`. The names of the list are the new names. If the new name is missing, the first value matched becomes the new name. Feeding in a vector is like using a list without name and only a single element. If the vector is numeric, you can use the special value "bin::digit" to group every digit element. For example if `x` represent years, using `bin="bin::2"` create bins of two years. Using "!bin::digit" groups every digit consecutive values starting from the first value. Using "!!bin::digit" is the same but starting from the last value. In both cases, `x` is not required to be numeric.

- **bin.rel**
  A list or a vector defining which values to bin. Only applies to the relative periods and not the cohorts. Please refer to the help of the argument `bin` to understand the different ways to do the binning (or look at the help of `bin`).

- **bin.c**
  A list or a vector defining which values to bin. Only applies to the cohort. Please refer to the help of the argument `bin` to understand the different ways to do the binning (or look at the help of `bin`).
sunab

bin.p  A list or a vector defining which values to bin. Only applies to the period. Please refer to the help of the argument bin to understand the different ways to do the binning (or look at the help of bin).

att  Logical, default is FALSE. If TRUE: then the total average treatment effect for the treated is computed (instead of the ATT for each relative period).

no_agg  Logical, default is FALSE. If TRUE: then there is no aggregation, leading to the estimation of all cohort x time to treatment coefficients.

Details

This function creates a matrix of cohort x relative_period interactions, and if used within a fixest estimation, the coefficients will automatically be aggregated to obtain the ATT for each relative period. In practice, the coefficients are aggregated with the aggregate.fixest function whose argument agg is automatically set to the appropriate value.

The SA method requires relative periods (negative/positive for before/after the treatment). Either the user can compute the RP (relative periods) by his/her own, either the RPs are computed on the fly from the periods and the cohorts (which then should represent the treatment period).

The never treated, which are the cohorts displaying only negative RPs are used as references (i.e. no dummy will be constructed for them). On the other hand, the always treated are removed from the estimation, by means of adding NAs for each of their observations.

If the RPs have to be constructed on the fly, any cohort that is not present in the period is considered as never treated. This means that if the period ranges from 1995 to 2005, cohort = 1994 will be considered as never treated, although it should be considered as always treated: so be careful.

If you construct your own relative periods, the controls cohorts should have only negative RPs.

Value

If not used within a fixest estimation, this function will return a matrix of interacted coefficients.

Binning

You can bin periods with the arguments bin, bin.c, bin.p and/or bin.rel.

The argument bin applies both to the original periods and cohorts (the cohorts will also be binned!). This argument only works when the period represent "calendar" periods (not relative ones!).

Alternatively you can bin the periods with bin.p (either "calendar" or relative); or the cohorts with bin.c.

The argument bin.rel applies only to the relative periods (hence not to the cohorts) once they have been created.

To understand how binning works, please have a look at the help and examples of the function bin. Binning can be done in many different ways: just remember that it is not because it is possible that it does makes sense!

Author(s)

Laurent Berge
Examples

# Simple DiD example
data(base_stagg)
head(base_stagg)

# Note that the year_treated is set to 1000 for the never treated
table(base_stagg$year_treated)
table(base_stagg$time_to_treatment)

# The DiD estimation
res_sunab = feols(y ~ x1 + sunab(year_treated, year) | id + year, base_stagg)
etable(res_sunab)

# By default the reference periods are the first year and the year before the treatment
# i.e. ref.p = c(-1, .F); where .F is a shortcut for the first period.
# Say you want to set as references the first three periods on top of -1
res_sunab_3ref = feols(y ~ x1 + sunab(year_treated, year, ref.p = c(.F + 0:2, -1)) | id + year, base_stagg)

# Display the two results
iplot(list(res_sunab, res_sunab_3ref))

# ... + show all refs
iplot(list(res_sunab, res_sunab_3ref), ref = "all")

# ATT
# To get the total ATT, you can use summary with the agg argument:
summary(res_sunab, agg = "ATT")

# You can also look at the total effect per cohort
summary(res_sunab, agg = "cohort")

# Binning
# Binning can be done in many different ways

# binning the cohort
est_bin.c = feols(y ~ x1 + sunab(year_treated, year, bin.c = 3:2) | id + year, base_stagg)

# binning the period
est_bin.p = feols(y ~ x1 + sunab(year_treated, year, bin.p = 3:1) | id + year, base_stagg)

# binning both the cohort and the period
est_bin = feols(y ~ x1 + sunab(year_treated, year, bin = 3:1) | id + year, base_stagg)

# binning the relative period, grouping every two years
est_bin.rel = feols(y ~ x1 + sunab(year_treated, year, bin.rel = "bin::2") | id + year, base_stagg)

etable(est_bin.c, est_bin.p, est_bin, est_bin.rel, keep = "year")

terms.fixest

Extract the terms

Description

This function extracts the terms of a fixest estimation, excluding the fixed-effects part.

Usage

## S3 method for class 'fixest'
terms(x, ...)

Arguments

x A fixest object. Obtained using the functions femlm, feols or feglm.
...

Not currently used.

Value

An object of class c("terms", "formula") which contains the terms representation of a symbolic model.

Examples

# simple estimation on iris data, using "Species" fixed-effects
res = feols(Sepal.Length ~ Sepal.Width*Petal.Length + Petal.Width | Species, iris)

# Terms of the linear part
terms(res)
to_integer  

Fast transform of any type of vector(s) into an integer vector

Description

Tool to transform any type of vector, or even combination of vectors, into an integer vector ranging from 1 to the number of unique values. This actually creates an unique identifier vector.

Usage

```r
to_integer(
  ..., 
  sorted = FALSE, 
  add_items = FALSE, 
  items.list = FALSE, 
  multi.df = FALSE, 
  multi.join = "_", 
  internal = FALSE
)
```

Arguments

- `...`: Vectors of any type, to be transformed in integer.
- `sorted`: Logical, default is `FALSE`. Whether the integer vector should make reference to sorted values?
- `add_items`: Logical, default is `FALSE`. Whether to add the unique values of the original vector(s). If requested, an attribute `items` is created containing the values (alternatively, they can appear in a list if `items.list=TRUE`).
- `items.list`: Logical, default is `FALSE`. Only used if `add_items=TRUE`. If `TRUE`, then a list of length 2 is returned with `x` the integer vector and `items` the vector of items.
- `multi.df`: Logical, default is `FALSE`. If `TRUE` then a data.frame listing the unique elements is returned in the form of a data.frame. Ignored if `add_items = FALSE`.
- `multi.join`: Character scalar used to join the items of multiple vectors. The default is "_". Ignored if `add_items = FALSE`.
- `internal`: Logical, default is `FALSE`. For programming only. If this function is used within another function, setting `internal = TRUE` is needed to make the evaluation of `...` valid. End users of `to_integer` should not care.

Value

Reruns a vector of the same length as the input vectors. If `add_items=TRUE` and `items.list=TRUE`, a list of two elements is returned: `x` being the integer vector and `items` being the unique values to which the values in `x` make reference.
Examples

x1 = iris$Species
x2 = as.integer(iris$Sepal.Length)

# transforms the species vector into integers
to_integer(x1)

# To obtain the "items":
to_integer(x1, add_items = TRUE)
# same but in list form
to_integer(x1, add_items = TRUE, items.list = TRUE)

# transforms x2 into an integer vector from 1 to 4
to_integer(x2, add_items = TRUE)

# To have the sorted items:
to_integer(x2, add_items = TRUE, sorted = TRUE)

# The result can safely be used as an index
res = to_integer(x2, add_items = TRUE, sorted = TRUE, items.list = TRUE)
all(res$items[res$x] == x2)

# Multiple vectors

to_integer(x1, x2, add_items = TRUE)

# You can use multi.join to handle the join of the items:
to_integer(x1, x2, add_items = TRUE, multi.join = "; ")

Description

This data reports trade information between countries of the European Union (EU15).

Usage

data(trade)
Format

trade is a data frame with 38,325 observations and 6 variables named Destination, Origin, Product, Year, dist_km and Euros.

- Origin: 2-digits codes of the countries of origin of the trade flow.
- Destination: 2-digits codes of the countries of destination of the trade flow.
- Products: Number representing the product categories (from 1 to 20).
- Year: Years from 2007 to 2016
- dist_km: Geographic distance in km between the centers of the countries of origin and destination.
- Euros: The total amount in euros of the trade flow for the specific year/product category/origin-destination country pair.

Source

This data has been extrated from Eurostat on October 2017.

unpanel

Dissolves a fixest panel

Description

Transforms a fixest_panel object into a regular data.frame.

Usage

unpanel(x)

Arguments

x A fixest_panel object (obtained from function panel).

Value

Returns a data set of the exact same dimension. Only the attribute 'panel_info' is erased.

Author(s)

Laurent Berge

See Also

Alternatively, the function panel changes a data.frame into a panel from which the functions l and f (creating leads and lags) can be called. Otherwise you can set the panel 'live' during the estimation using the argument panel.id (see for example in the function feols).
**update.fixest**

Updates a fixest estimation

**Description**

Updates and re-estimates a fixest model (estimated with `femlm`, `feols` or `feglm`). This function updates the formulas and use previous starting values to estimate a new fixest model. The data is obtained from the original call.

**Usage**

```r
## S3 method for class 'fixest'
update(object, fml.update, nframes = 1, evaluate = TRUE, ...)
```

**Arguments**

- `object` A fixest object. Obtained using the functions `femlm`, `feols` or `feglm`.
- `fml.update` Changes to be made to the original argument `fml`. See more information on `update.formula`. You can add/withdraw both variables and fixed-effects. E.g. `. ~ . + x2 | . + z2` would add the variable `x2` and the cluster `z2` to the former estimation.
- `nframes` (Advanced users.) Defaults to 1. Number of frames up the stack where to perform the evaluation of the updated call. By default, this is the parent frame.
- `evaluate` Logical, default is TRUE. If FALSE, only the updated call is returned.
- `...` Other arguments to be passed to the functions `femlm`, `feols` or `feglm`.

**Value**

It returns a fixest object (see details in `femlm`, `feols` or `feglm`).

**Examples**

```r
data(base_did)

# Setting a data set as a panel
pdat = panel(base_did, ~id+period)

# ... allows you to use leads and lags in estimations
feols(y~l(x1, 0:1), pdat)

# Now unpanel => returns the initial data set
class(pdat) ; dim(pdat)
new_base = unpanel(pdat)
class(new_base) ; dim(new_base)
```
Author(s)

Laurent Berge

See Also

See also the main estimation functions \texttt{femlm}, \texttt{feols} or \texttt{feglm}. \texttt{predict.fixest}, \texttt{summary.fixest}, \texttt{vcov.fixest}, \texttt{fixef.fixest}.

Examples

# Example using trade data
data(trade)

# main estimation
est_pois = fepois(Euros ~ log(dist_km) | Origin + Destination, trade)

# we add the variable log(Year)
est_2 = update(est_pois, . ~ . + log(Year))

# we add another fixed-effect: "Product"
est_3 = update(est_2, . ~ . | . + Product)

# we remove the fixed-effect "Origin" and the variable log(dist_km)
est_4 = update(est_3, . ~ . - log(dist_km) | . - Origin)

# Quick look at the 4 estimations
etable(est_pois, est_2, est_3, est_4)

vcov.fixest

\textit{Computes the variance/covariance of a fixest object}

Description

This function extracts the variance-covariance of estimated parameters from a model estimated with \texttt{femlm}, \texttt{feols} or \texttt{feglm}.

Usage

## S3 method for class 'fixest'
vcov(
  object,
  vcov = NULL,
  se = NULL,
  cluster,
  ssc = NULL,
  attr = FALSE,
)
vcov.fixest

```r
forceCovariance = FALSE,
keepBounded = FALSE,
nthreads = getFixest_nthreads(),
...
```

### Arguments

**object**
A `fixest` object. Obtained using the functions `femlm`, `feols` or `feglm`.

**vcov**
Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form: `vcov_type ~ variables`. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from `vcov_cluster`, `vcov_NW`, `NW`, `vcov_DK`, `DK`, `vcov_conley` and `conley`. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the `vcov` documentation in the vignette.

**se**
Character scalar. Which kind of standard error should be computed: “standard”, “hetero”, “cluster”, “twoway”, “threeway” or “fourway”? By default if there are clusters in the estimation: `se = "cluster"`, otherwise `se = "iid"`. Note that this argument is deprecated, you should use `vcov` instead.

**cluster**
Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over `var1` and `var2` contained in the data.frame `base` used for the estimation. All the following cluster arguments are valid and do the same thing: `cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2`. If the two variables were used as fixed-effects in the estimation, you can leave it blank with `vcov = "twoway"` (assuming `var1` [resp. `var2`] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using `^` with the following syntax: `cluster = ~var1^var2` or `cluster = "var1^var2"`.

**ssc**
An object of class `ssc.type` obtained with the function `ssc`. Represents how the degree of freedom correction should be done. You must use the function `ssc` for this argument. The arguments and defaults of the function `ssc` are: `adj = TRUE, fixef.K="nested", cluster.adj = TRUE, cluster.df = "min", t.df = "min", fixef.force_exact=FALSE`). See the help of the function `ssc` for details.

**attr**
Logical, defaults to `FALSE`. Whether to include the attributes describing how the VCOV was computed.

**forceCovariance**
(Advanced users.) Logical, default is `FALSE`. In the peculiar case where the obtained Hessian is not invertible (usually because of collinearity of some variables), use this option to force the covariance matrix, by using a generalized inverse of the Hessian. This can be useful to spot where possible problems come from.

**keepBounded**
(Advanced users – `femlm` with non-linear part and bounded coefficients only.) Logical, default is `FALSE`. If `TRUE`, then the bounded coefficients (if any) are
treated as unrestricted coefficients and their S.E. is computed (otherwise it is not).

**nthreads**  The number of threads. Can be: a) an integer lower than, or equal to, the maximum number of threads; b) 0: meaning all available threads will be used; c) a number strictly between 0 and 1 which represents the fraction of all threads to use. The default is to use 50% of all threads. You can set permanently the number of threads used within this package using the function `setFixest_nthreads`.

...  Other arguments to be passed to `summary.fixest`.
   The computation of the VCOV matrix is first done in `summary.fixest`.

**Details**

For an explanation on how the standard-errors are computed and what is the exact meaning of the arguments, please have a look at the dedicated vignette: *On standard-errors.*

**Value**

It returns a $K \times K$ square matrix where $K$ is the number of variables of the fitted model. If attr = `TRUE`, this matrix has an attribute “type” specifying how this variance/covariance matrix has been computed.

**Author(s)**

Laurent Berge

**See Also**

You can also compute VCOVs with the following functions: `vcov_cluster`, `vcov_hac`, `vcov_conley`. See also the main estimation functions `femlm`, `feols` or `feglm`. `summary.fixest`, `confint.fixest`, `resid.fixest`, `predict.fixest`, `fixef.fixest`.

**Examples**

```r
# Load panel data
data(base_did)

# Simple estimation on a panel
est = feols(y ~ x1, base_did)

# ======== #
# IID VCOV #
# ======== #

# By default the VCOV assumes iid errors:
se(vcov(est))

# You can make the call for an iid VCOV explicitly:
se(vcov(est, "iid"))
```
# Heteroskedasticity-robust VCOV

# By default the VCOV assumes iid errors:
se(vcov(est, "hetero"))

# => note that it also accepts vcov = "White" and vcov = "HC1" as aliases.

# =============== #
# Clustered VCOVs #
# =============== #

# To cluster the VCOV, you can use a formula of the form cluster ~ var1 + var2 etc
# Let’s cluster by the panel ID:
se(vcov(est, cluster ~ id))

# Alternative ways:

# -> cluster is implicitly assumed when a one-sided formula is provided
se(vcov(est, ~ id))

# -> using the argument cluster instead of vcov
se(vcov(est, cluster = ~ id))

# For two-/three-way clustering, just add more variables:
se(vcov(est, ~ id + period))

# -------------------|  
# Implicit deduction  |  
# -------------------|
# When the estimation contains FEs, the dimension on which to cluster  
# is directly inferred from the FEs used in the estimation, so you don’t need  
# to explicitly add them.

est_fe = feols(y ~ x1 | id + period, base_did)

# Clustered along "id"
se(vcov(est_fe, "cluster"))

# Clustered along "id" and "period"
se(vcov(est_fe, "twoway"))

# =========== #
# Panel VCOVs #
# =========== #

# ---------------------|  
# Newey West (NW) VCOV  |  
# ---------------------|
# To obtain NW VCOVs, use a formula of the form NW ~ id + period  
se(vcov(est, NW ~ id + period))
# If you want to change the lag:
se(vcov(est, NW(3) ~ id + period))

# Alternative way:

# -> using the vcov_NW function
se(vcov(est, vcov_NW(unit = "id", time = "period", lag = 3)))

# -------------------------|
# Driscoll-Kraay (DK) VCOV |
# -------------------------|
# To obtain DK VCOVs, use a formula of the form DK ~ period
se(vcov(est, DK ~ period))

# If you want to change the lag:
se(vcov(est, DK(3) ~ period))

# Alternative way:

# -> using the vcov_DK function
se(vcov(est, vcov_DK(time = "period", lag = 3)))

# -------------------|
# Implicit deduction |
# -------------------|
# When the estimation contains a panel identifier, you don’t need
# to re-write them later on

est_panel = feols(y ~ x1, base_did, panel.id = ~id + period)

# Both methods, NM and DK, now work automatically
se(vcov(est_panel, "NW"))
se(vcov(est_panel, "DK"))

# VCOVs robust to spatial correlation #

data(quakes)
est_geo = feols(depth ~ mag, quakes)

# Conley VCOV |
# -------------------|
# To obtain a Conley VCOV, use a formula of the form conley(cutoff) ~ lat + lon
# with lat/lon the latitude/longitude variable names in the data set
se(vcov(est_geo, conley(100) ~ lat + lon))

# Alternative way:
vcov_cluster

# -> using the vcov_DK function
se(vcov(est_geo, vcov_conley(lat = "lat", lon = "long", cutoff = 100)))

# -------------------|
# Implicit deduction |
# -------------------|
# By default the latitude and longitude are directly fetched in the data based
# on pattern matching. So you don't have to specify them.
# Further, an automatic cutoff is deduced by default.
# The following works:
se(vcov(est_geo, "conley"))

# ============= #
# Small Sample Corrections #
# ============= #

# You can change the way the small sample corrections are done with the argument ssc.
# The argument ssc must be created by the ssc function
se(vcov(est, ssc = ssc(adj = FALSE)))

# You can add directly the call to ssc in the vcov formula.
# You need to add it like a variable:
se(vcov(est, iid ~ ssc(adj = FALSE)))
se(vcov(est, DK ~ period + ssc(adj = FALSE)))

vcov_cluster

Clustered VCOV

Description

Computes the clustered VCOV of fixest objects.

Usage

vcov_cluster(x, cluster = NULL, ssc = NULL)

Arguments

x A fixest object.

cluster Either i) a character vector giving the names of the variables onto which to
cluster, or ii) a formula giving those names, or iii) a vector/list/data.frame giving
the hard values of the clusters. Note that in cases i) and ii) the variables are
fetched directly in the data set used for the estimation.

ssc An object returned by the function ssc. It specifies how to perform the small
sample correction.
Value

If the first argument is a `fixest` object, then a VCOV is returned (i.e. a symmetric matrix).

If the first argument is not a `fixest` object, then a) implicitly the arguments are shifted to the left (i.e. `vcov_cluster(~var1 + var2)` is equivalent to `vcov_cluster(cluster = ~var1 + var2)`) and b) a VCOV-request is returned and NOT a VCOV. That VCOV-request can then be used in the argument `vcov` of various `fixest` functions (e.g. `vcov.fixest` or even in the estimation calls).

Author(s)

Laurent Berge

References


Examples

```r
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")
base$clu = rep(1:5, 30)

est = feols(y ~ x1, base)

# VCOV: using a formula giving the name of the clusters
vcov_cluster(est, ~species + clu)

# works as well with a character vector
vcov_cluster(est, c("species", "clu"))

# you can also combine the two with "^`
vcov_cluster(est, ~species^clu)

# Using VCOV requests
#
# per se: pretty useless...
vcov_cluster(~species)

# ...but VCOV-requests can be used at estimation time:
# it may be more explicit than...
feols(y ~ x1, base, vcov = vcov_cluster("species"))

# ...the equivalent, built-in way:
feols(y ~ x1, base, vcov = ~species)

# The argument vcov does not accept hard values,
# so you can feed them with a VCOV-request:
feols(y ~ x1, base, vcov = vcov_cluster(rep(1:5, 30)))
```
Description

Compute VCOVs robust to spatial correlation, a la Conley (1999).

Usage

vcov_conley(
  x, lat = NULL, lon = NULL, cutoff = NULL,
  pixel = 0, distance = "triangular",
  ssc = NULL
)

conley(cutoff = NULL, pixel = NULL, distance = NULL)

Arguments

x A fixest object.

lat A character scalar or a one sided formula giving the name of the variable representing the latitude. The latitude must lie in [-90, 90], [0, 180] or [-180, 0].

lon A character scalar or a one sided formula giving the name of the variable representing the longitude. The longitude must be in [-180, 180], [0, 360] or [-360, 0].

cutoff The distance cutoff, in km. You can express the cutoff in miles by writing the number in character form and adding "mi" as a suffix: cutoff = "100mi" would be 100 miles. If missing, a rule of thumb is used to deduce the cutoff.

pixel A positive numeric scalar, default is 0. If a positive number, the coordinates of each observation are pooled into pixel x pixel km squares. This lowers the precision but can (depending on the cases) greatly improve computational speed at a low precision cost. Note that if the cutoff was expressed in miles, then pixel will also be in miles.

distance How to compute the distance between points. It can be equal to "triangular" (default) or "spherical". The latter case corresponds to the great circle distance and is more precise than triangular but is a bit more intensive computationally.

ssc An object returned by the function ssc. It specifies how to perform the small sample correction.
Details

This function computes VCOVs that are robust to spatial correlations by assuming a correlation between the units that are at a geographic distance lower than a given cutoff.

The kernel is uniform.

If the cutoff is not provided, an estimation of it is given. This cutoff ensures that a minimum of units lie within it and is robust to sub-sampling. This automatic cutoff is only here for convenience, the most appropriate cutoff shall depend on the application and shall be provided by the user.

The function conley does not compute VCOVs directly but is meant to be used in the argument vcov of fixest functions (e.g. in \texttt{vcov.fixest} or even in the estimation calls).

Value

If the first argument is a \texttt{fixest} object, then a VCOV is returned (i.e. a symmetric matrix).

If the first argument is not a \texttt{fixest} object, then a) implicitly the arguments are shifted to the left (i.e. \texttt{vcov\_conley("lat", "long")} is equivalent to \texttt{vcov\_conley(lat = "lat", lon = "long")}) and b) a VCOV-request is returned and NOT a VCOV. That VCOV-request can then be used in the argument \texttt{vcov} of various \texttt{fixest} functions (e.g. \texttt{vcov\_fixest} or even in the estimation calls).

References


Examples

```r
data(quakes)

# We use conley() in the vcov argument of the estimation
feols(depth ~ mag, quakes, conley(100))

# Post estimation
est = feols(depth ~ mag, quakes)
vcov_conley(est, cutoff = 100)
```

---

### vcov_hac

**HAC VCOVs**

**Description**

Set of functions to compute the VCOVs robust to different forms correlation in panel or time series settings.
Usage

vcov_DK(x, time = NULL, lag = NULL, ssc = NULL)

vcov_NW(x, unit = NULL, time = NULL, lag = NULL, ssc = NULL)

NW(lag = NULL)

eweyywest(lag = NULL)

DK(lag = NULL)

driscoll_kraay(lag = NULL)

Arguments

x A fixest object.
time A character scalar or a one sided formula giving the name of the variable representing the time.
lag An integer scalar, default is NULL. If NULL, then the default lag is equal to $n_t^{0.25}$ with $n_t$ the number of time periods (as of Newey and West 1987) for panel Newey-West and Driscoll-Kraay. The default for the time series Newey-West is computed via bwNeweyWest which implements the Newey and West 1994 method.
ssc An object returned by the function ssc. It specifies how to perform the small sample correction.
unit A character scalar or a one sided formula giving the name of the variable representing the units of the panel.

Details

There are currently three VCOV types: Newey-West applied to time series, Newey-West applied to a panel setting (when the argument 'unit' is not missing), and Driscoll-Kraay.

The functions on this page without the prefix "vcov_" do not compute VCOVs directly but are meant to be used in the argument vcov of fixest functions (e.g. in vcov.fixest or even in the estimation calls).

Note that for Driscoll-Kraay VCOVs, to ensure its properties the number of periods should be long enough (a minimum of 20 periods or so).

Value

If the first argument is a fixest object, then a VCOV is returned (i.e. a symmetric matrix).

If the first argument is not a fixest object, then a) implicitly the arguments are shifted to the left (i.e. vcov_DK(~year) is equivalent to vcov_DK(time = ~year)) and b) a VCOV-request is returned and NOT a VCOV. That VCOV-request can then be used in the argument vcov of various fixest functions (e.g. vcov.fixest or even in the estimation calls).
**Lag selection**

The default lag selection depends on whether the VCOV applies to a panel or a time series.

For panels, i.e. panel Newey-West or Driscoll-Kraay VCOV, the default lag is \( n_t^{0.25} \) with \( n_t \) the number of time periods. This is based on Newey and West 1987.

For time series Newey-West, the default lag is found thanks to the `bwNeweyWest` function from the `sandwich` package. It is based on Newey and West 1994.

**References**


**Examples**

```r
data(base_did)

# For the estimation
#
# Panel Newey-West, lag = 2
feols(y ~ x1, base_did, NW(2) ~ id + period)

# Driscoll-Kraay
feols(y ~ x1, base_did, DK ~ period)

# If the estimation is made with a panel.id, the dimensions are automatically deduced:
est = feols(y ~ x1, base_did, "NW", panel.id = ~id + period)
est

# Post estimation
#
# If missing, the unit and time are automatically deduced from the panel.id used in the estimation
vcov_NW(est, lag = 2)
```
wald test of nullity of coefficients

Description
Wald test used to test the joint nullity of a set of coefficients.

Usage
wald(x, keep = NULL, drop = NULL, print = TRUE, vcov, se, cluster, ...)

Arguments

x  A fixest object. Obtained using the methods femlm, feols or feglm.

keep  Character vector. This element is used to display only a subset of variables. This should be a vector of regular expressions (see base::regex help for more info). Each variable satisfying any of the regular expressions will be kept. This argument is applied post aliasing (see argument dict). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use keep = "x[[:digit:]]{2}". If the first character is an exclamation mark, the effect is reversed (e.g. keep = "!Intercept" means: every variable that does not contain “Intercept” is kept). See details.

drop  Character vector. This element is used if some variables are not to be displayed. This should be a vector of regular expressions (see base::regex help for more info). Each variable satisfying any of the regular expressions will be discarded. This argument is applied post aliasing (see argument dict). Example: you have the variable x1 to x55 and want to display only x1 to x9, then you could use drop = "x[[:digit:]]{2}". If the first character is an exclamation mark, the effect is reversed (e.g. drop = "!Intercept" means: every variable that does not contain “Intercept” is dropped). See details.

print  Logical, default is TRUE. If TRUE, then a verbose description of the test is prompted on the R console. Otherwise only a named vector containing the test statistics is returned.

vcov  Versatile argument to specify the VCOV. In general, it is either a character scalar equal to a VCOV type, either a formula of the form vcov_type ~ variables. The VCOV types implemented are: "iid", "hetero" (or "HC1"), "cluster", "twoway", "NW" (or "newey_west"), "DK" (or "driscoll_kraay"), and "conley". It also accepts object from vcov_cluster, vcov_NW, NW, vcov_DK, DK, vcov_conley and conley. It also accepts covariance matrices computed externally. Finally it accepts functions to compute the covariances. See the vcov documentation in the vignette.

se  Character scalar. Which kind of standard error should be computed: "standard", "hetero" , "cluster", "twoway", "threeway" or "fourway"? By default if there are clusters in the estimation: se = "cluster", otherwise se = "iid". Note that this argument is deprecated, you should use vcov instead.
cluster 

Tells how to cluster the standard-errors (if clustering is requested). Can be either a list of vectors, a character vector of variable names, a formula or an integer vector. Assume we want to perform 2-way clustering over var1 and var2 contained in the data.frame base used for the estimation. All the following cluster arguments are valid and do the same thing: cluster = base[, c("var1", "var2")], cluster = c("var1", "var2"), cluster = ~var1+var2. If the two variables were used as fixed-effects in the estimation, you can leave it blank with vcov = "twoway" (assuming var1 [resp. var2] was the 1st [resp. 2nd] fixed-effect). You can interact two variables using ^ with the following syntax: cluster = ~var1^var2 or cluster = "var1^var2".

... Any other element to be passed to summary.fixest.

Details

The type of VCOV matrix plays a crucial role in this test. Use the arguments se and cluster to change the type of VCOV for the test.

Value

A named vector containing the following elements is returned: stat, p, df1, and df2. They correspond to the test statistic, the p-value, the first and second degrees of freedoms.

If no valid coefficient is found, the value NA is returned.

Examples

data(airquality)
est = feols(Ozone ~ Solar.R + Wind + poly(Temp, 3), airquality)

# Testing the joint nullity of the Temp polynomial
wald(est, "poly")

# Same but with clustered SEs
wald(est, "poly", cluster = "Month")

# Now: all vars but the polynomial and the intercept
wald(est, drop = "Intepoly")

# Toy example: testing pre-trends
#
data(base_did)
est_did = feols(y ~ x1 + i(period, treat, 5) | id + period, base_did)

# The graph of the coefficients
coefplot(est_did)

# The pre-trend test
weights.fixest

Extracts the weights from a fixest object

Description

Simply extracts the weights used to estimate a fixest model.

Usage

## S3 method for class 'fixest'
weights(object, ...)

Arguments

object  A fixest object.
...
Not currently used.

Value

Returns a vector of the same length as the number of observations in the original data set. Ignored observations due to NA or perfect fit are re-introduced and their weights set to NA.

See Also

feols, fepois, feglm, fenegbin, feNmlm.

Examples

```r
est = feols(Petal.Length ~ Petal.Width, iris, weights = ~as.integer(Sepal.Length) - 3.99)
weights(est)
```
Expands formula macros

Creation macros within formulas and expand them with character vectors or other formulas.

Usage

```r
xpd(fml, ..., add = NULL, lhs, rhs, data = NULL, frame = parent.frame())
```

Arguments

- `fml` A formula containing macros variables. Each macro variable must start with two dots. The macro variables can be set globally using `setFixest_fml`, or can be defined in `...`. Special macros of the form `..("regex")` can be used to fetch, through a regular expression, variables directly in a character vector or column names given in the argument `data` (note that the algorithm tries to "guess" the argument data when nested in function calls [see example]). You can negate the regex by starting with a "!". Square brackets have a special meaning: Values in them are evaluated and parsed accordingly. Example: `y~x.[1:2] + z.[i]` will lead to `y~x1+x2+z3` if `i==3`. You can trigger the auto-completion of variables by using the '. ' suffix, like in `y ~ x. ..` which would include `x1` and `x2`, etc. See examples.

- `...` Definition of the macro variables. Each argument name corresponds to the name of the macro variable. It is required that each macro variable name starts with two dots (e.g. `..ctrl`). The value of each argument must be a one-sided formula or a character vector, it is the definition of the macro variable. Example of a valid call: `setFixest_fml(..ctrl = ~ var1 + var2)`. In the function `xpd`, the default macro variables are taken from `getFixest_fml`, any variable in `...` will replace these values. You can enclose values in `.[]`, if so they will be evaluated from the current environment. For example `..ctrl1 = ~ x.[1:2] + [z]` will lead to `~x1 + x2 + var` if `z` is equal to "var".

- `add` Either a character scalar or a one-sided formula. The elements will be added to the right-hand-side of the formula, before any macro expansion is applied.

- `lhs` If present then a formula will be constructed with `lhs` as the full left-hand-side. The value of `lhs` can be a one-sided formula, a call, or a character vector. Note that the macro variables won't be applied. You can use it in combination with the argument `rhs`. Note that if `fml` is not missing, its LHS will be replaced by `lhs`.

- `rhs` If present, then a formula will be constructed with `rhs` as the full right-hand-side. The value of `rhs` can be a one-sided formula, a call, or a character vector. Note that the macro variables won't be applied. You can use it in combination with the argument `lhs`. Note that if `fml` is not missing, its RHS will be replaced by `rhs`. 

Either a character vector or a data.frame. This argument will only be used if a macro of the type ..("regex") is used in the formula of the argument fml. If so, any variable name from data that matches the regular expression will be added to the formula.

frame
The environment containing the values to be expanded with the dot square bracket operator. Default is parent.frame().

Details
In xpd, the default macro variables are taken from getFixest_fml. Any value in the ... argument of xpd will replace these default values.

The definitions of the macro variables will replace in verbatim the macro variables. Therefore, you can include multi-part formulas if you wish but then beware of the order of the macros variable in the formula. For example, using the airquality data, say you want to set as controls the variable Temp and Day fixed-effects, you can do setFixest_fml(.ctrl = ~Temp | Day), but then feols(Ozone ~ Wind + ..ctrl, airquality) will be quite different from feols(Ozone ~ ..ctrl + Wind, airquality), so beware!

Value
It returns a formula where all macros have been expanded.

Dot square bracket operator in formulas
In a formula, the dot square bracket (DSB) operator can: i) create manifold variables at once, or ii) capture values from the current environment and put them verbatim in the formula.

Say you want to include the variables x1 to x3 in your formula. You can use xpd(y ~ x.[1:3]) and you'll get y ~ x1 + x2 + x3.

To summon values from the environment, simply put the variable in square brackets. For example: for(i in 1:3) xpd(y.[i] ~ x) will create the formulas y1 ~ x to y3 ~ x depending on the value of i.

You can include a full variable from the environment in the same way: for(y in c("a", "b")) xpd(.[y] ~ x) will create the two formulas a ~ x and b ~ x.

The DSB can even be used within variable names, but then the variable must be nested in character form. For example y ~ .["x.[1:2]_sq"] will create y ~ x1_sq + x2_sq. Using the character form is important to avoid a formula parsing error. Double quotes must be used. Note that the character string that is nested will be parsed with the function dsb, and thus it will return a vector.

By default, the DSB operator expands vectors into sums. You can add a comma, like in .[, x], to expand with commas—the content can then be used within functions. For instance: c(x.[, 1:2]) will create c(x1, x2) (and not c(x1 + x2)).

In all fixest estimations, this special parsing is enabled, so you don't need to use xpd.

One-sided formulas can be expanded with the DSB operator: let x = ~sepal + petal, then xpd(y ~ .[x]) leads to color ~ sepal + petal.

You can even use multiple square brackets within a single variable, but then the use of nesting is required. For example, the following xpd(y ~ .["[letters[1:2]][.[1:2]]"]) will create y
\[ a_1 + b_2. \] Remember that the nested character string is parsed with \texttt{dsb}, which explains this behavior.

When the element to be expanded i) is equal to the empty string or, ii) is of length 0, it is replaced with a neutral element, namely 1. For example, \[ x = \"\" ; \ xpd(y \sim .[x]) \] leads to \[ y \sim 1. \]

**Regular expressions**

You can catch several variable names at once by using regular expressions. To use regular expressions, you need to enclose it in the dot-dot or the regex function: \texttt{..("regex")} or \texttt{regex("regex")}. For example, \texttt{regex("Sepal")} will catch both the variables \texttt{Sepal.Length} and \texttt{Sepal.Width} from the \texttt{iris} data set. In a \texttt{fixest} estimation, the variables names from which the regex will be applied come from the data set. If you use \texttt{xpd}, you need to provide either a data set or a vector of names in the argument \texttt{data}.

By default the variables are aggregated with a sum. For example in a data set with the variables \texttt{x1} to \texttt{x10}, \texttt{regex("x(1|2)\")} will yield \texttt{x1 + x2 + x10}. You can instead ask for “comma” aggregation by using a comma first, just before the regular expression: \[ y \sim \text{sw}(\text{regex(\"x(1|2)\")}) \] would lead to \[ y \sim \text{sw}(x1, x2, x10). \]

Note that the dot square bracket operator (DSB, see before) is applied before the regular expression is evaluated. This means that \texttt{regex("x.[3:4]_sq")} will lead, after evaluation of the DSB, to \texttt{regex("x3_sq|x4_sq").} It is a handy way to insert range of numbers in a regular expression.

**Author(s)**

Laurent Berge

**See Also**

\texttt{setFixest_fml} to set formula macros, and \texttt{dsb} to modify character strings with the DSB operator.

**Examples**

```r
# Small examples with airquality data
data(airquality)
# we set two macro variables
setFixest_fml(..ctrl = ~ Temp + Day,
               ..ctrl_long = ~ poly(Temp, 2) + poly(Day, 2))

# Using the macro in \texttt{lm} with \texttt{xpd}:
lm(xpd(Ozone ~ Wind + ..ctrl), airquality)
lm(xpd(Ozone ~ Wind + ..ctrl_long), airquality)

# You can use the macros without \texttt{xpd()} in \texttt{fixest} estimations
a = feols(Ozone ~ Wind + ..ctrl, airquality)
b = feols(Ozone ~ Wind + ..ctrl_long, airquality)
etable(a, b, keep = "Int|Win")

# Using .[]
```
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
i = 2:3
z = "species"

lm(xpd(y ~ x.[2:3] + .[z]), base)

# No xpd() needed in feols
feols(y ~ x.[2:3] + .[z], base)

# Auto completion with '.-' suffix
#
# You can trigger variables autocompletion with the '.-' suffix
# You need to provide the argument data
base = setNames(iris, c("y", "x1", "x2", "x3", "species"))
xpd(y ~ x.., data = base)

# In fixest estimations, this is automatically taken care of
feols(y ~ x.., data = base)

# You can use xpd for stepwise estimations
#
# Note that for stepwise estimations in fixest, you can use
# the stepwise functions: sw, sw0, csw, csw0
# -> see help in feols or in the dedicated vignette

# we want to look at the effect of x1 on y
# controlling for different variables

base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# We first create a matrix with all possible combinations of variables
my_args = lapply(names(base)[-((1:2)]), function(x) c("", x))
(all_combs = as.matrix(do.call("expand.grid", my_args)))

res_all = list()
for(i in 1:nrow(all_combs)){
  res_all[[i]] = feols(xpd(y ~ x1 + ..v, ..v = all_combs[i, ]), base)
}
etable(res_all)
coefplot(res_all, group = list(Species = "^^species"))

# You can use macros to grep variables in your data set
#
# Example 1: setting a macro variable globally
data(longley)
setFixest_fml(.many_vars = grep("GNP\|ployed", names(longley), value = TRUE))
feols(Armed.Forces ~ Population + .many_vars, longley)

# Example 2: using ..("regex") or regex("regex") to grep the variables "live"
feols(Armed.Forces ~ Population + ..("GNP\|ployed"), longley)

# Example 3: same as Ex.2 but without using a fixest estimation
# Here we need to use xpd():
lm(xpd(Armed.Forces ~ Population + regex("GNP\|ployed"), data = longley), longley)

# Stepwise estimation with regex: use a comma after the parenthesis
feols(Armed.Forces ~ Population + sw(regex("GNP\|ployed")), longley)

# Multiple LHS
etable(feols(..("GNP\|ployed") ~ Population, longley))

# # lhs and rhs arguments
#
#
# to create a one sided formula from a character vector
vars = letters[1:5]
xpd(rhs = vars)

# Alternatively, to replace the RHS
xpd(y ~ 1, rhs = vars)

# To create a two sided formula
xpd(lhs = "y", rhs = vars)

# # argument 'add'
#
xpd(~x1, add = ~ x2 + x3)

# also works with character vectors
xpd(~x1, add = c("x2", "x3"))

# only adds to the RHS
xpd(y ~ x, add = -bon + jour)

# # Dot square bracket operator
#
# The basic use id to add variables in the formula
x = c("x1", "x2")
xpd(y ~ .[x])
# Alternatively, one-sided formulas can be used and their content will be inserted verbatim
x = ~x1 + x2
xpd(y ~ .[x])

# You can create multiple variables at once
xpd(y ~ x.[1:5] + z.[2:3])

# You can summon variables from the environment to complete variables names
var = "a"
xpd(y ~ x.[var])

# ... the variables can be multiple
vars = LETTERS[1:3]
xpd(y ~ x.[vars])

# You can have "complex" variable names but they must be nested in character form
xpd(y ~ .["x.[vars]_sq"])

# DSB can be used within regular expressions
re = c("GNP", "Pop")
xpd(Unemployed ~ regex(".\[re\]")), data = longley)

# => equivalent to regex("GNP\|Pop")

# Use .[,var] (NOTE THE COMMA!) to expand with commas
# !! can break the formula if missused
vars = c("wage", "unemp")
xpd(c(y.[,1:3]) ~ csw(.,[,vars]))

# Example of use of .[] within a loop
res_all = list()
for(p in 1:3){
  res_all[[p]] = feols(Ozone ~ Wind + poly(Temp, .[p]), airquality)
}
table(res_all)

# The former can be compactly estimated with:
res_compact = feols(Ozone ~ Wind + sw(.,[poly(Temp, .[1:3]])]), airquality)
table(res_compact)

# How does it work?
# 1) .[, stuff] evaluates stuff and, if a vector, aggregates it with commas
# Comma aggregation is done thanks to the comma placed after the square bracket
# If .[stuff], then aggregation is with sums.
# 2) stuff is evaluated, and if it is a character string, it is evaluated with
# the function dsb which expands values in .[]
#
# Wrapping up:
# 2) evaluation of dsb("poly(Temp, .[1:3])") leads to the vector:
# c("poly(Temp, 1)", "poly(Temp, 2)", "poly(Temp, 3")
# 1) [, c("poly(Temp, 1)", "poly(Temp, 2)", "poly(Temp, 3")]) leads to
# poly(Temp, 1), poly(Temp, 2), poly(Temp, 3)
#
# Hence sw(.[, "poly(Temp, [1:3])"]) becomes:
# sw(poly(Temp, 1), poly(Temp, 2), poly(Temp, 3))
#
# In non-fixest functions: guessing the data allows to use regex
#
# When used in non-fixest functions, the algorithm tries to "guess" the data
# so that ..("regex") can be directly evaluated without passing the argument 'data'
data(longley)
lm(xpd(Armed.Forces ~ Population + ..("GNP\plo\ed")), longley)

# same for the auto completion with '..'
lm(xpd(Armed.Forces ~ Population + GN..), longley)

---

Subsets a fixest_multi object

Description

Subsets a fixest_multi object using different keys.

Usage

```r
## S3 method for class 'fixest_multi'
x[i, sample, lhs, rhs, fixef, iv, I, reorder = TRUE, drop = FALSE]
```

Arguments

- `x`: A fixest_multi object, obtained from a fixest estimation leading to multiple results.
- `i`: An integer vector. Represents the estimations to extract.
- `sample`: An integer vector, a logical scalar, or a character vector. It represents the sample identifiers for which the results should be extracted. Only valid when the fixest estimation was a split sample. You can use .N to refer to the last element. If logical, all elements are selected in both cases, but FALSE leads `sample` to become the rightmost key (just try it out).
- `lhs`: An integer vector, a logical scalar, or a character vector. It represents the left-hand-sides identifiers for which the results should be extracted. Only valid when the fixest estimation contained multiple left-hand-sides. You can use .N to refer to the last element. If logical, all elements are selected in both cases, but FALSE leads `lhs` to become the rightmost key (just try it out).
rhs An integer vector or a logical scalar. It represents the right-hand-sides identifiers for which the results should be extracted. Only valid when the fixest estimation contained multiple right-hand-sides. You can use .N to refer to the last element. If logical, all elements are selected in both cases, but FALSE leads rhs to become the rightmost key (just try it out).

fixef An integer vector or a logical scalar. It represents the fixed-effects identifiers for which the results should be extracted. Only valid when the fixest estimation contained fixed-effects in a stepwise fashion. You can use .N to refer to the last element. If logical, all elements are selected in both cases, but FALSE leads fixef to become the rightmost key (just try it out).

iv An integer vector or a logical scalar. It represents the stages of the IV. Note that the length can be greater than 2 when there are multiple endogenous regressors (the first stage corresponding to multiple estimations). Note that the order of the stages depends on the stage argument from summary.fixest. If logical, all elements are selected in both cases, but FALSE leads iv to become the rightmost key (just try it out).

I An integer vector. Represents the root element to extract.

reorder Logical, default is TRUE. Indicates whether reordering of the results should be performed depending on the user input.

drop Logical, default is FALSE. If the result contains only one estimation, then if drop = TRUE it will be transformed into a fixest object (instead of fixest_multi).

Details

The order with we we use the keys matter. Every time a key sample, lhs, rhs, fixef or iv is used, a reordering is performed to consider the leftmost-side key to be the new root.

Use logical keys to easily reorder. For example, say the object res contains a multiple estimation with multiple left-hand-sides, right-hand-sides and fixed-effects. By default the results are ordered as follows: lhs, fixef, rhs. If you use res[lhs = FALSE], then the new order is: fixef, rhs, lhs. With res[rhs = TRUE, lhs = FALSE] it becomes: rhs, fixef, lhs. In both cases you keep all estimations.

Value

It returns a fixest_multi object. If there is only one estimation left in the object, then the result is simplified into a fixest object.

See Also

The main fixest estimation functions: feols, fepois, fenegbin, feglm, feNmlm. Tools for multiple fixest estimations: summary.fixest_multi, print.fixest_multi, as.list.fixest_multi, sub-sub-.fixest_multi, sub-.fixest_multi.

Examples

# Estimation with multiple samples/LHS/RHS
aq = airquality[airquality$Month %in% 5:6, ]
```
est_split = feols(c(Ozone, Solar.R) ~ sw(poly(Wind, 2), poly(Temp, 2)),
aq, split = ~ Month)

# By default: sample is the root
etable(est_split)

# Let's reorder, by considering lhs the root
etable(est_split[lhs = 1:.N])

# Selecting only one LHS and RHS
etable(est_split[lhs = "Ozone", rhs = 1])

# Taking the first root (here sample = 5)
etable(est_split[i = 1])

# The first and last estimations
etable(est_split[i = c(1, .N)])
```

---

Method to subselect from a `fixest_panel`

### Description

Subselection from a `fixest_panel` which has been created with the function `panel`. Also allows to create lag/lead variables with functions `l/f` if the `fixest_panel` is also a `data.table::data.table`.

### Usage

```
## S3 method for class 'fixest_panel'
x[i, j, ...]
```

### Arguments

- **x**: A `fixest_panel` object, created with the function `panel`.
- **i**: Row subselection. Allows `data.table::data.table` style selection (provided the data is also a data.table).
- **j**: Variable selection. Allows `data.table::data.table` style selection/variable creation (provided the data is also a data.table).
- **...**: Other arguments to be passed to `[.data.frame` or `data.table::data.table` (or whatever the class of the initial data).

### Details

If the original data was also a `data.table`, some calls to `[.fixest_panel` may dissolve the `fixest_panel` object and return a regular `data.table`. This is the case for subselections with additional arguments. If so, a note is displayed on the console.
It returns a `fixest_panel` data base, with the attributes allowing to create lags/leads properly book-kept.

**Author(s)**

Laurent Berge

**See Also**

Alternatively, the function `panel` changes a `data.frame` into a panel from which the functions \( l \) and \( f \) (creating leads and lags) can be called. Otherwise you can set the panel 'live' during the estimation using the argument `panel.id` (see for example in the function `feols`).

**Examples**

```r
data(base_did)

# Creating a fixest_panel object
pdat = panel(base_did, ~id+period)

# Subselections of fixest_panel objects bookkeeps the leads/lags engine
pdat_small = pdat[!pdat$period %in% c(2, 4), ]
a = feols(y~l(x1, 0:1), pdat_small)

# we obtain the same results, had we created the lags "on the fly"
base_small = base_did[base_did$period %in% c(2, 4), ]
b = feols(y~l(x1, 0:1), base_small, panel.id = ~id+period)
etable(a, b)

# Using data.table to create new lead/lag variables
if(require("data.table")){
    pdat_dt = panel(as.data.table(base_did), ~id+period)

    # Variable creation
    pdat_dt[, x_l1 := l(x1)]
    pdat_dt[, c("x_l1", "x_f1_2") := .(1(x1), f(x1)**2)]

    # Estimation on a subset of the data
    # (the lead/lags work appropriately)
    feols(y~l(x1, 0:1), pdat_dt[!period %in% c(2, 4)])
}
```
Description

Extracts single elements from multiple fixest estimations.

Usage

```r
## S3 method for class 'fixest_multi'
x[[i]]
```

Arguments

- `x`: A `fixest_multi` object, obtained from a fixest estimation leading to multiple results.
- `i`: An integer scalar. The identifier of the estimation to extract.

Value

A fixest object is returned.

See Also

The main fixest estimation functions: `feols`, `fepois`, `fenegbin`, `feglm`, `feNmlm`. Tools for multiple fixest estimations: `summary.fixest_multi`, `print.fixest_multi`, `as.list.fixest_multi`, `sub-sub-.fixest_multi`, `sub-.fixest_multi`.

Examples

```r
base = iris
names(base) = c("y", "x1", "x2", "x3", "species")

# Multiple estimation
res = feols(y ~ csw(x1, x2, x3), base, split = ~species)

# The first estimation
res[[1]]

# The second one, etc
res[[2]]
```
## Index

### * datasets
- base_did, 9
- base_stagg, 10
- trade, 231
- .l(rep.fixest), 183
- [.fixest_multi, 254
- [.fixest_panel, 256
- [[.fixest_multi, 258
- aggregate.fixest, 4, 227
- AIC, 6, 11
- AIC.fixest, 6, 11, 159
- as.dict, 7, 193
- as.list.fixest_multi, 8, 8, 178, 225, 255, 258
- base::regex, 17–20, 24, 25, 33, 34, 36, 37, 40, 74, 75, 189, 243
- base_did, 9
- base_stagg, 10
- BIC.fixest, 7, 10, 159
- bin, 11, 13, 154, 180, 182, 226, 227
- bread.fixest, 15
- bwNeweyWest, 243, 244
- check_conv_feols, 16
- coef.fixest, 17
- coef.fixest_multi, 19
- coefficients.fixest(coef.fixest), 17
- coefficients.fixest_multi(coef.fixest_multi), 19
- coefplot, 21, 138, 183, 188, 192, 193
- coeftable, 32
- coeftable.default, 33
- coeftable.fixest, 33, 35
- coeftable.fixest_multi, 39
- collinearity, 42
- confint.fixest, 18, 43, 236
- confint.fixest_multi, 45
- conley, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 245
- conley(vcov_conley), 241
- csw0(stepwise), 209
- csw0(stepwise), 209
- d, 101, 112, 124, 135
- d(f), 91
- data.table::data.table, 91, 168, 256
- degrees_freedom, 47
- degrees_freedom_iid(degrees_freedom), 47
- demean, 49
- deviance.fixest, 52
- did_means, 53, 193
- DK, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 245
- DK(vcov_hac), 242
- dof(ssc), 206
- discoll_kraay(vcov_hac), 242
- dsb, 12, 56, 103, 104, 114, 126, 137, 138, 154, 181, 249, 250
- est_env, 65
- estfun.fixest, 64
- esttable(etable), 67
- esttex(etable), 67
- expand.model.frame, 220
- extralines_register, 83, 89
- f, 91, 101, 112, 124, 135, 169, 170, 256
- family, 96
- fdim, 92
- feglm, 6–8, 11, 17, 18, 29, 42, 43, 53, 72, 84, 93, 110, 115, 116, 124, 138, 147–152, 159–161, 163, 170–173,
pvalue (coeftable), 32
pvalue.default (coeftable.default), 33
pvalue.fixest (coeftable.fixest), 35
pvalue.fixest_multi
(coeftable.fixest_multi), 39
r2, 74, 143, 179
ref, 13, 180, 182
rep.fixest, 183
rep.fixest_list (rep.fixest), 183
resid.fixest, 99, 110, 122, 133, 148, 185, 236
resid.fixest_multi, 186
residuals.fixest (resid.fixest), 185
residuals.fixest_multi
(resid.fixest_multi), 186
sample_df, 187
se (coeftable), 32
se.default (coeftable.default), 33
se.fixest (coeftable.fixest), 35
se.fixest_multi
(coeftable.fixest_multi), 39
se.matrix (coeftable.default), 33
setFixest_coefplot, 24, 28, 29, 188, 189, 192
setFixest_dict, 7, 8, 24, 75, 189, 193
setFixest_estimation, 103, 113, 125, 137, 194
setFixest_etable (etable), 67
setFixest_fml, 197, 230
setFixest_notes, 202
setFixest_nthreads, 98, 109, 120, 132, 149, 202, 219, 236
setFixest_print (print.fixest), 175
setFixest_ssc, 102, 112, 124, 135
setFixest_ssc (ssc), 206
setFixest_vcov, 102, 112, 124, 135, 203
sigma.fixest, 205
ssc, 36, 44, 46, 48, 73, 97, 109, 120, 131, 173, 206, 207, 218, 224, 235, 239, 241, 243
stepwise, 209
style.df, 76, 210
style.tex, 76, 83, 213
summary, 203
summary.fixest.fixef, 222
summary.fixest_check_conv (check_conv_feols), 16
summary.fixest_list (summary.fixest), 217
summary.fixest_multi (summary.fixest), 217
summary.fixest_multi, 8, 178, 223, 225, 255, 258
sunab, 4, 73, 219, 225
sunab.att (sunab), 225
sw (stepwise), 209
sw0 (stepwise), 209
terms.fixest, 229
to_integer, 230
trade, 231
tstat (coeftable), 32
tstat.default (coeftable.default), 33
tstat.fixest (coeftable.fixest), 35
tstat.fixest_multi (coeftable.fixest_multi), 39
unpanel, 232
update.fixest, 151, 161, 173, 233
update.formula, 233
vcov, 203
vcov_cluster, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 236, 239, 245
vcov_conley, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 236, 241, 245
vcov_DK, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 245
vcov_OK, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 245
vcov_OK (vcov_hac), 242
vcov_hac, 236, 242
vcov_NW, 44, 46, 47, 72, 96, 107, 118, 130, 173, 218, 224, 235, 245
vcov_NW (vcov_hac), 242
wald, 245
weights.fixest, 247
xpd, 96, 103, 107, 114, 117, 126, 129, 137, 197, 198, 248