# User-Specified General-to-Specific (GETS) and Indicator Saturation (ISAT) Methods[1]

Genaro Sucarrat[2]

This version: 28th September 2021

[1]This vignette is a revised and updated version of the article "User-Specified General-to-Specific and Indicator Saturation Methods", which was published in The R Journal (2020) 12:2, pages 388-401: `https://journal.r-project.org/archive/2021/RJ-2021-024/`. I am grateful to the Editor Catherine Hurley, an anonymous reviewer, Damian Clarke, Éric Dubois, Jonas Kurle, Felix Pretis, James Reade, Moritz Schwarz, Gareth Thomas, participants at the UseR! 2019 conference (Toulouse, July), the Statistics Norway seminar (May, 2018) and the Norges Bank seminar (April, 2018) for helpful comments, suggestions and questions.

[2]Department of Economics, BI Norwegian Business School, Nydalsveien 37, 0484 Oslo, Norway. Webpage: `http://www.sucarrat.net/`.

## Summary

General-to-Specific (GETS) modelling provides a comprehensive, systematic and cumulative approach to modelling that is ideally suited for conditional forecasting and counterfactual analysis, whereas Indicator Saturation (ISAT) is a powerful and flexible approach to the detection and estimation of structural breaks (e.g. changes in parameters), and to the detection of outliers. In both methods multi-path backwards elimination, single and multiple hypothesis tests on the coefficients, diagnostics tests and goodness-of-fit measures are combined to produce a parsimonious final model. In many situations a specific model or estimator is needed, a specific set of diagnostics tests may be required, or a specific fit criterion is preferred. In these situations, if the combination of estimator/model, diagnostics tests and fit criterion is not offered in a pre-programmed way by publicly available software, then the implementation of user-specified GETS and ISAT methods puts a large programming-burden on the user. To reduce this burden, the R package **gets** provides a complete set of facilities and generic functions for user-specified GETS and ISAT methods: User-specified model/estimator, user-specified diagnostics and user-specified goodness-of-fit criteria. This vignette explains and illustrates how user-specified GETS and ISAT methods can be created with the R package **gets**.

## 1 Introduction

General-to-Specific (GETS) modelling provides a comprehensive, systematic and cumulative approach to modelling that is ideally suited for scenario analysis, e.g. conditional forecasting and counterfactual analysis. To this end, well-known ingredients (tests of coefficients, multi-path backwards elimination, diagnostics tests and fit criteria) are combined to produce a parsimonious final model that passes the chosen diagnostics. GETS modelling originated at the London School of Economics (LSE) during the 1960s, and gained widespread acceptance and usage in economics during the 1980s and 1990s. The two-volume article collection by Campos et al. (2005) provides a comprehensive historical overview of key-developments in GETS modelling. Software-wise, a milestone was reached in 1999, when the data-mining experiment of Lovell (1983) was re-visited by Hoover and Perez (1999). They showed that automated GETS modelling could improve substantially upon the then prevalent modelling approaches. The study spurred numerous new studies and developments, including Indicator Saturation (ISAT) methods, see Hendry et al. (2008) and Castle et al. (2015). ISAT methods provide a powerful and flexible approach to the detection and estimation of structural breaks (e.g. changes in parameters), and to the detection of outliers.

On CRAN, there are two packages that provide GETS methods. The second, named **gets**, is simply the successor of the first, which is named **AutoSEARCH**.[3] Since October 2014 the development of **AutoSEARCH** is frozen, and all development efforts have been directed towards the **gets** package.[4] An introduction to the **gets** package is provided by Pretis et al. (2018). However, it does does not cover the user-specification capabilities of the package, some of which were not available at the time.

At the time of writing (August 2021), the publicly available softwares that provide GETS and ISAT methods are contained in Table 1. Although they offer GETS and ISAT methods for some of the most popular models in applications, in many situations a specific model or estimator will be needed, a specific set of diagnostics tests may be required, or a specific fit criterion is preferred. In these situations, if the combination of estimator/model, diagnostics tests and fit criterion is not offered in a pre-programmed way by the publicly available softwares, then the implementation of user-specified GETS and ISAT methods puts a large programming-burden on the user. To reduce this burden, generic functions and procedures that facilitate the implementation of user-specified GETS and ISAT methods for specific problems can therefore be of great benefit. The R package **gets**, since version 0.20 (September 2019), is the first software – both inside and outside the R universe – to provide a complete set of facilities and generic functions for user-specified GETS and ISAT methods: User-specified model/estimator, user-specified diagnostics and user-specified goodness-of-fit criteria. The aim of this vignette is to illustrate how user-specified GETS and ISAT methods can be implemented.

The rest of this vignette contains three sections. In the next the model selection properties

[3]Both packages were created by the me. Originally, I simply wanted to rename the first to the name of the second. This, however, is inconvenient in practice I was told, so I was instead asked by CRAN to publish a "new" package with the new name.

[4]See the Gitub page for the current development version of the package: https://github.com/gsucarrat/gets/.

| | HP1999 (MATLAB) | Autometrics (OxMetrics) | Grocer (Scilab) | genspec (STATA) | EViews | **gets** (R) |
|---|---|---|---|---|---|---|
| More than 10 paths | | Yes | Yes | Yes | Yes | Yes |
| GETS of linear regression | Yes | Yes | Yes | Yes | Yes | Yes |
| GETS of variance models | | | | | | Yes |
| GETS of logit models | | Yes | | | | Yes |
| GETS of count models | | Yes | | | | |
| GETS of probit models | | Yes | Yes | | | |
| GETS of panel models | | | Yes | Yes | | |
| GETS of MIDAS models | | | | | Yes | |
| ISAT of linear regression | | Yes | Yes | | Yes | Yes |
| User-specified GETS | | | Yes | | | Yes |
| User-specified ISAT | | | | | | Yes |
| User-specified diagnostics | | | Yes | | | Yes |
| User-specified goodness-of-fit | | | | | | Yes |
| Menu-based GUI | | Yes | | | Yes | |
| Free and open source | Yes* | | Yes | Yes* | | Yes |

Table 1: A comparison of publicly available GETS and ISAT softwares with emphasis on user-specification capabilities. HP1999, the MATLAB code of Hoover and Perez (1999). Autometrics, OxMetrics version 15, see Doornik and Hendry (2018). Grocer, version 1.8, see Dubois and Michaux (2019). genspec, version 1.2.2, see Clarke (2014). EViews, version 12, see IHS Markit (2020). **gets**, version 0.29, see Sucarrat et al. (2021), and Pretis et al. (2018).
*The modules in themselves are free and open source, but they run in non-free and closed source software environments (MATLAB and STATA, respectively).

of GETS and ISAT methods are summarised. This is followed by a section that outlines the general principles of how user-specified estimation, user-specified diagnostics and user-specified goodness-of-fit measures are implemented. Next, a section with four illustrations follows.

## 2 Model selection properties of GETS and ISAT methods

It is useful to denote a generic model for observation $t$ as

$$m\left(y_t, \boldsymbol{x}_t, \boldsymbol{\beta}\right), \qquad t = 1, 2, \ldots, n, \tag{1}$$

where $y_t$ is the dependent variable, $\boldsymbol{x}_t = (x_{1t}, x_{2t}, \ldots)'$ is a vector of covariates, $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots)'$ is a vector of parameters to be estimated and $n$ is the sample size. Two examples are the linear regression model and the logit-model:

$$\begin{aligned} y_t &= \beta_1 x_{1t} + \cdots + \beta_k x_{kt} + \epsilon_t, && (2) \\ Pr\left(y_t = 1 | \boldsymbol{x}_t\right) &= \frac{1}{1 + \exp\left(-h_t\right)} \quad \text{with} \quad h_t = \beta_1 x_{1t} + \cdots + \beta_k x_{kt}. && (3) \end{aligned}$$

Note that, in a generic model $m(y_t, \boldsymbol{x}_t, \boldsymbol{\beta})$, the dimension $\boldsymbol{\beta}$ is usually – but not necessarily – equal to the dimension of $\boldsymbol{x}_t$. Here, unless otherwise stated, they will both have dimension $k$.

In (2)–(3), a variable $x_{jt} \in \boldsymbol{x}_t$ is said to be relevant if $\beta_j \neq 0$ and irrelevant if $\beta_j = 0$. Let $k_{\text{rel}} \geq 0$ and $k_{\text{irr}} \geq 0$ denote the number of relevant and irrelevant variables, respectively, such that $k_{\text{rel}} + k_{\text{irr}} = k$. GETS modelling aims at finding a specification that contains as many relevant variables as possible, and a proportion of irrelevant variables that on average equals the significance level $\alpha$ chosen by the investigator. Put differently, if $\widehat{k}_{\text{rel}}$ and $\widehat{k}_{\text{irr}}$ are the retained number of relevant and irrelevant variables in an empirical application, respectively, then GETS modelling aims at satisfying

$$E\left(\widehat{k}_{\text{rel}} / k_{\text{rel}}\right) \rightarrow 1 \quad \text{and} \quad E\left(\widehat{k}_{\text{irr}} / k_{\text{irr}}\right) \rightarrow \alpha \quad \text{as} \quad n \rightarrow \infty, \tag{4}$$

when $k_{\text{rel}}, k_{\text{irr}} > 0$. If either $k_{\text{rel}} = 0$ or $k_{\text{irr}} = 0$, then the targets are modified in natural ways: If $k_{\text{rel}} = 0$, then the first target is $E(\widehat{k}_{\text{rel}}) = 0$, and if $k_{\text{irr}} = 0$, then the second target is $E(\widehat{k}_{\text{irr}}) = 0$. Sometimes, the irrelevance proportion $\widehat{k}_{\text{irr}}/k_{\text{irr}}$ is also referred to as *gauge*, whereas the relevance proportion $\widehat{k}_{\text{irr}}/k_{\text{irr}}$ is also referred to as *potency*.

In targeting a relevance proportion equal to 1 and an irrelevance proportion equal to $\alpha$, GETS modelling combines well-known ingredients: Multi-path backwards elimination, tests on the $\beta_j$'s (both single and multiple hypothesis tests), diagnostics tests and fit-measures (e.g. information criteria). Let $V(\widehat{\boldsymbol{\beta}})$ denote the estimated coefficient-covariance matrix. GETS modelling in the package **gets** can be described as proceeding in three steps:[5]

[5]The exact way GETS modelling is implemented across softwares varies.

1. Formulate a General Unrestricted Model (GUM), i.e. a starting model, that passes a set of chosen diagnostic tests. A regressor $x_j$ in the GUM is non-significant if the $p$-value of a two-sided $t$-test is lower than the chosen significance level $\alpha$, and each non-significant regressor constitutes the starting point of a backwards elimination path. The test-statistics of the $t$-tests are computed as $\widehat{\beta}_j / se(\widehat{\beta}_j)$, where $se(\widehat{\beta}_j)$ is the square root of the $j$th. element of the diagonal of $V(\widehat{\boldsymbol{\beta}})$.

2. Undertake backwards elimination along multiple paths by removing, one-by-one, non-significant regressors as determined by the chosen significance level $\alpha$. Each removal is checked for validity against the chosen set of diagnostic tests, and for parsimonious encompassing (i.e. a multiple hypothesis test) against the GUM. These multiple hypothesis tests on subsets of $\boldsymbol{\beta}$ are implemented as Wald-tests.

3. Multi-path backwards elimination can result in multiple terminal models. The last step of GETS modelling consists of selecting, among the terminal models, the specification with the best fit according to a fit-criterion, e.g. the Schwarz (1978) information criterion.

In ISAT methods, the vector $\boldsymbol{x}_t$ contains at least $n-1$ indicators in addition to other covariates that are considered. Accordingly, standard estimation methods are infeasible, since the number of variables in $\boldsymbol{x}_t$ is usually larger than the number of observations $n$. The solution to this problem provided by ISAT methods is to first organise $\boldsymbol{x}_t$ into $B$ blocks: $\boldsymbol{x}_t^{(1)}, \ldots, \boldsymbol{x}_t^{(B)}$. These blocks need not be mutually exclusive, so a variable or subset of variables can appear in more than one block. Next, GETS modelling is applied to each block, which leads to $B$ final models. Finally, a new round of GETS modelling is undertaken with the union of the retained variables from the $B$ blocks as covariates in a new starting model (i.e. a new GUM). The model selection properties targeted by ISAT methods are the same as those of GETS methods. Note, however, that since the starting model (the GUM) contains at least $n-1$ regressors, a tiny significance level – e.g. $\alpha = 0.001$ or smaller – is usually recommended in ISAT methods.

## 3 General principles

In the current version of the package **gets**, version 0.28, the functions that admit user-specified estimation are `arx()`, `getsFun()`, `blocksFun()` and `isat()`. The user-specification principles are the same in all four. However, if the result (i.e. a `list`) returned from the user-specified estimator does not have the same structure as that returned from the default estimator `ols()` (part of the **gets** package), then `arx()` and `isat()` may not always work as expected. This is particularly the case with respect to their extraction functions, e.g. `print()`, `coef()`, `residuals()` and `predict()`. User-specified diagnostics and goodness-of-fit functions are optional. By default, `getsFun()`, `blocksFun()` and `isat()` do not perform any diagnostics tests, whereas the default in `arx()`, `getsm()` and `getsv()` is to test the standardised residuals for autocorrelation and Autoregressive Heteroscedasticity (ARCH). This is implemented via the `diagnostics()` function (part of the **gets** package). Also by default, all four functions use the Schwarz (1978) information criterion as goodness-of-fit measure, which favours parsimony, via the `infocrit()` function (part of the **gets** package).

## 3.1 The `getsFun()` function

The recommended, most flexible and computationally most efficient approach to user-specified GETS modelling is via the `getsFun()` function. Currently, it accepts up to twenty-five arguments. For the details of all these arguments, the reader is referred to the discussion of the `getsm()` function (Section 5) in Pretis et al. (2018), and the help pages of `getsFun()` (type `?getsFun`). For the purpose of user-specified estimation, user-specified diagnostics and user-specified goodness-of-fit measures, the most important arguments are:

```
getsFun(y, x,
  user.estimator = list(name = "ols"),
  user.diagnostics = NULL,
  gof.function = list(name = "infocrit", method = "sc"),
  gof.method = c("min", "max"),
  ...)
```

The `y` is the left-hand side variable (the regressand), `x` is the regressor or design matrix, `user.estimator` controls which estimator or model to use and further arguments – if any – to be passed on to the estimator, `user.diagnostics` controls the user-specified diagnostics if any, and `gof.function` and `gof.method` control the goodness-of-fit measure used. Note that `y` and `x` should satisfy `is.vector(y) == TRUE` and `is.matrix(x) == TRUE`, respectively, and enter in “clean" ways: If either `y` or `x` are objects of class, say, `"ts"` or `"zoo"`, then `getsFun()` may not behave as expected. By default, the estimator `ols()` is used with its default arguments, which implements OLS estimation via the `qr()` function. The value `NULL` on `user.diagnostics` means no diagnostics checks are undertaken by default. The following code illustrates `getsFun()` in linear regression (the default), and reproduces the information printed while searching:

```
n <- 40 #number of observations
k <- 20 #number of Xs

set.seed(123) #for reproducibility
y <- rnorm(n) #generate Y
x <- matrix(rnorm(n*k), n, k) #create matrix of Xs

#do gets w/default estimator ols(), store output in 'result':
result <- getsFun(y, x)

#the information printed during searching:
18 path(s) to search
Searching: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

The object named `result` is a list, and the code `summary(results)` returns a summary of its contents. The most important items are:

- `paths`: A list of vectors containing the searched paths. Each vector (i.e. path) indicates the sequence of deletion of the regressors. In the example above the first path is

  ```
  $paths[[1]]
  [1]  1 15  6  7  3 14 11 16  4  2  8 12  5  9 20 19 13
  ```

  That is, regressor no. 1 was the first to be deleted, regressor no. 15 was the second, regressor no. 6 was the third, and so on. If the regressors in `x` were named, then a name-representation of the first deletion path can be obtained with `colnames(x)[ paths[[1]] ]`.

- `terminals`: A list of vectors with the distinct terminal models of the specification search. In the example above it is equal to

  ```
  $terminals
  $terminals[[1]]
  [1] 10 17 18

  $terminals[[2]]
  [1] 10 18
  ```

  That is, two terminal models. The first contains regressors 10, 17 and 18, whereas the second contains regressors 10 and 18.

- `terminals.results`: A data frame with the goodness-of-fit information of the terminal models. In the above example the entry is equal to:

  ```
  $terminals.results
          info(sc)      logl  n k
  spec 1: 2.514707 -44.76081 40 3
  spec 2: 2.529923 -46.90958 40 2
  ```

  `spec 1` is short for specification 1, i.e. terminal model 1, and `spec 2` is short for specification 2, i.e. terminal model 2. `info(sc)` indicates that the Schwarz (1978) criterion (the default) is used as goodness-of-fit measure, whereas `n` and `k` denote the number of observations and parameters, respectively.

- `best.terminal`: An integer that indicates which terminal model is the best according to the goodness-of-fit criterion used. In the example above the value is 1.

- `specific.spec`: A vector of integers that indicates which regressors that are contained in the best terminal model. In the above example it is

  ```
  $specific.spec
  [1] 10 17 18
  ```

  That is, the best terminal model contains regressors no. 10, 17 and 18.

## 3.2 User-specified estimation

User-specified estimation is carried out via the `user.estimator` argument. It must be a list containing at least one entry – a character – named `name` with the name of the estimator to be called. Optionally, the list can also contain an item named `envir`, a character, which indicates the environment in which the user-specified estimator resides. If unspecified, then the function is looked for in the usual way by R. Additional entries in the list, if any, are passed on to the estimator as arguments.

The user-specified estimator must also satisfy the following:

1. It should be of the form `myEstimator(y, x)` or `myEstimator(y, x, ...)`, where `y` is a vector, `x` is a matrix and `...` means the user-estimator can accept further arguments (optional) that are passed on to the estimator. In other words, while the name of the function is arbitrary, the first argument should be the regressand and the second a matrix (e.g. of covariates).

2. The user-defined estimator should return a `list` with a minimum of six items:

   - `n` (the number of observations)
   - `k` (the number of coefficients)
   - `df` (degrees of freedom, used in the *t*-tests)
   - `coefficients` (a vector with the coefficient estimates)
   - `vcov` (the coefficient covariance matrix)
   - `logl` (a goodness-of-fit value, e.g. the log-likelihood)

   The items need not appear in this order. However, the naming should be exactly as indicated. If also the diagnostics and/or the goodness-of-fit criterion is user-specified, then additional objects may be required, see the subsections below on user-specified diagnostics and goodness-of-fit criteria. Note also that, if the goodness-of-fit criterion is user-specified, then `logl` can in certain situations be replaced by another item that needs not be named `logl`.

3. The user-defined estimator must be able to handle `NULL` regressor-matrices, i.e. situations where either `NCOL(x)` is 0 or `is.null(x)` is `TRUE`. This is needed in situations where a terminal model is empty (i.e. no regressors are retained).

To illustrate how the requirements above can be met in practice, suppose – as an example – that we would like to use the function `lm()` for estimation rather than `ols()`. The first step is then to make an "interface" function that calls `lm()` while satisfying requirements 1 to 3:

```
lmFun <- function(y, x, ...){

  ##create list:
  result <- list()

  ##n, k and df:
    result$n <- length(y)
    if( is.null(x) || NCOL(x) == 0 ){
      result$k <- 0
    }else{
      result$k <- NCOL(x)
    }
    result$df <- result$n - result$k

    ##call lm if k > 0:
    if( result$k > 0){
      tmp <- lm(y ~ x - 1)
      result$coefficients <- coef(tmp)
      result$vcov <- vcov(tmp)
      result$logl <- as.numeric(logLik(tmp))
    }else{
      result$coefficients <- NULL
      result$vcov <- NULL
      result$logl <- sum(dnorm(y, sd = sqrt(var(y)), log = TRUE))
    }

    ##return result:
    return(result)

}
```

Next, the code

```
  getsFun(y, x, user.estimator = list(name = "lmFun"))
```

undertakes the same specification search as earlier, but uses `lmFun()` rather than `ols()`.

## 3.3 User-specified diagnostics

User-specified diagnostics is carried out via the `user.diagnostics` argument. By default, the argument is `NULL`, so no diagnostic tests are undertaken. To carry out user-specified diagnostics tests, the argument must be a list containing at least two entries: A character named `name` containing the name of the diagnostics function to be called, and an entry named `pval` that contains a vector with values between 0 and 1, i.e. the chosen significance level(s) for the diagnostics test(s).[6] If only a single test is undertaken by the diagnostics function, then

[6]A word of caution is required. Let $R^{(i)}$ denote the event of rejecting the null, under the null, for a significance level $\alpha^{(i)}$ in diagnostic test $i$. For example, if only a single test is undertaken so that $i = 1$, then $Pr(R^{(1)}) = \alpha^{(1)}$. If two tests are undertaken, however, then the probability of rejecting in one or both tests is $Pr\left(R^{(1)} \cup R^{(2)}\right)$. As rule of thumb, therefore, to control the overall error, it is recommended that the significance level of each diagnostic test is set equal to $\alpha/m$, where $\alpha$ is the overall or total significance level targeted by the user, and $m$ is the number of diagnostic tests. This is sometimes referred to as a Bonferroni correction.

pval should be of length one. If two tests are undertaken, then pval should be of length two. And so on. An example of the argument when only a single test is undertaken is:

```
user.diagnostics = list(name = "myDiagnostics", pval = 0.05))
```

That is, the name of the function is `myDiagnostics`, and the chosen significance level for the single test that is carried out is 5%. Optionally, just as when the estimator is user-specified, the list can contain an item named `envir`, a character, which indicates the environment in which the user-specified diagnostics function resides. Additional items in the list, if any, are passed on to the user-specified function as arguments.

The user-specified diagnostics function must satisfy the following:

1. It should be of the form `myDiagnostics(result)` or `myDiagnostics(result, ...)`, where `result` is the list returned from the estimator in question, e.g. that of the user-specified estimator (recall requirement 2 in the previous section above), and ... means the function can accept further arguments (optional) that are passed on.

2. It should return an $m \times 3$ matrix that contains the $p$-value(s) of the test(s) in the third column, where $m \geq 1$ is the number of tests carried out. So if only a single test is carried out, then $m = 1$ and the $p$-value should be contained in the third column. An example could look like:

   ```
               statistic df   pval
   normality          NA NA 0.0734
   ```

   Note that the row-names and column-names in the example are not required. However, they do indicate what kind of information you may wish to put there for reporting purposes, e.g. by using the function `diagnostics()` (part of the **gets** package).

To illustrate how the requirements can be met in practice, suppose we would like to ensure the residuals are normal by testing for non-normality with the Shapiro-Wilks test function `shapiro.test()`. In this context, its main argument is the residuals of the estimated model. The list returned by the user-defined estimator named `lmFun()` above, however, does not contain an item with the residuals. The first step, therefore, is to modify the estimator `lmFun()` so that the returned list also contains the residuals:

```
lmFun <- function(y, x, ...){

  ##info needed for estimation:
  result <- list()
  result$n <- length(y)
  if( is.null(x) || NCOL(x)==0 ){
    result$k <- 0
  }else{
    result$k <- NCOL(x)
  }
  result$df <- result$n - result$k
```

```
    if( result$k > 0){
      tmp <- lm(y ~ x - 1)
      result$coefficients <- coef(tmp)
      result$vcov <- vcov(tmp)
      result$logl <- as.numeric(logLik(tmp))
    }else{
      result$coefficients <- NULL
      result$vcov <- NULL
      result$logl <- sum(dnorm(y, sd=sqrt(var(y)), log=TRUE))
    }

    ##residuals:
    if( result$k > 0){
      result$residuals <- residuals(tmp)
    }else{
      result$residuals <- y
    }

    ##return result:
    return(result)

  }
```

Computationally, the only modification appears under `##residuals`. We can now make the user-specified diagnostics function:

```
  myDiagnostics <- function(x, ...){
    tmp <- shapiro.test(x$residuals) #do the test
    result <- rbind( c(tmp$statistic, NA, tmp$p.value) )
    return(result)
  }
```

The following code undertakes GETS modelling with the user-specified estimator defined above, and the user-specified diagnostics function using a 5% significance level for the latter:

```
  getsFun(y, x, user.estimator = list(name = "lmFun"),
    user.diagnostics = list(name = "myDiagnostics", pval = 0.05))
```

Note that if the chosen significance level for the diagnostics is sufficiently high, then no specification search is undertaken because the starting model does not pass the non-normality test. With the current data, for example, a little bit of trial and error reveals this is the case for a level of about `pval = 0.35`.

## 3.4 User-specified goodness-of-fit

User-specified goodness-of-fit is carried out with the `gof.function` and `gof.method` arguments. The former indicates which Goodness-of-Fit (GOF) function to use, and the second is a character that indicates whether the best model maximises (`"max"`) or minimises (`"min"`) the GOF criterion in question. The argument `gof.function` is a list with a structure similar to earlier: It must contain at least one entry, a character named `name`, with the name of the GOF function to call. An example is:

```
gof.function = list(name = "myGof"))
```

Optionally, also here the list can contain an item named `envir`, a character, which indicates the environment in which the user-specified GOF function resides. Also as earlier, additional items in the list are passed on to the user-specified GOF function as arguments. The default value, for example, `gof.function = list(name = "infocrit", method = "sc")`, means the argument `method = "sc"` is passed on to the function `infocrit()`, see the help pages of `infocrit()` (type `?infocrit`) for information on the methods available via this function. The user-specified GOF function must satisfy the following:

1. It should be of the form `myGof(result)` or `myGof(result, ...)`, where `result` is the list returned from the estimator in question, e.g. that of the user-specified estimator, and `...` means the function can accept further arguments (optional) that are passed on.

2. It should return a single numeric value, i.e. the value of the GOF measure in question.

To illustrate how the requirements can be met in practice, suppose we would like to use the adjusted $R^2$ as our GOF measure in combination with our user-defined estimator. For the moment, the user-defined estimator `lmFun()` does not contain the information necessary to compute the adjusted $R^2$. In particular, it lacks the regressand `y`. However, this is readily added:

```
lmFun <- function(y, x, ...){

  ##info needed for estimation:
  result <- list()
  result$n <- length(y)
  if( is.null(x) || NCOL(x)==0 ){
    result$k <- 0
  }else{
    result$k <- NCOL(x)
  }
  result$df <- result$n - result$k
  if( result$k > 0){
    tmp <- lm(y ~ x - 1)
    result$coefficients <- coef(tmp)
    result$vcov <- vcov(tmp)
    result$logl <- as.numeric(logLik(tmp))
  }else{
    result$coefficients <- NULL
    result$vcov <- NULL
```

```
      result$logl <- sum(dnorm(y, sd=sqrt(var(y)), log=TRUE))
    }

    ##residuals:
    if( result$k > 0){
      result$residuals <- residuals(tmp)
    }else{
      result$residuals <- y
    }

    ##info needed for r-squared:
    result$y <- y

    ##return result:
    return(result)

  }
```

The added part appears under `##info needed for r-squared`. A GOF function that returns the adjusted $R^2$ is:

```
  myGof <- function(object, ...){
    TSS <- sum((object$y - mean(object$y))^2)
    RSS <- sum(object$residuals^2)
    Rsquared <- 1 - RSS/TSS
    result <- 1 - (1 - Rsquared) * (object$n - 1)/(object$n - object$k)
    return(result)
  }
```

The following code undertakes GETS modelling with all the three user-specified functions defined so far:

```
  getsFun(y, x, user.estimator = list(name = "lmFun"),
    user.diagnostics = list(name = "myDiagnostics", pval = 0.05),
    gof.function = list(name = "myGof"), gof.method = "max")
```

Incidentally, it leads to the same final model as when the default GOF function is used.

## 3.5 The `blocksFun()` function

The `blocksFun()` function was added to the package **gets** in version 0.24 (July 2020). It enables block-based GETS modelling with user-specified estimator, diagnostics and goodness-of-fit criteria, and one of its main attractions is that it can handle more variables than observations $n$. This is not possible in `getsFun()`. Currently, `blocksFun()` accepts up to 31 arguments, most of which are the same as those of `getsFun()` (type `?blocksFun` for the details). In particular, the main principles outlined above in Sections 3.1 to 3.4 also apply to `blocksFun()`.

The following code illustrates the basic usage of `blocksFun()` in a situation where the number of regressors is larger than the number of observations:

```
n <- 40 #number of observations
k <- 60 #number of Xs
set.seed(123) #for replicability
y <- rnorm(n) #generate Y
x <- matrix(rnorm(n*k), n, k) #create matrix of Xs

#do block-based gets w/default estimator ols(), store output in 'result':
result <- blocksFun(y, x)

#the information printed during searching:
x block 1 of 2:
30 path(s) to search
Searching: 1 2 3 ... 29 30

x block 2 of 2:
30 path(s) to search
Searching: 1 2 3 ... 29 30
```

The printed information during search is from the first round of the block-based GETS modelling. The information states that the original regressor matrix $x_t$ was divided into 2 blocks, and then GETS-modelling was applied to each of the blocks. An internal algorithm is used to split $x_t$, and the algorithm can be tweaked via the arguments `blocks`, `no.of.blocks`, `max.block.size` and `ratio.threshold`, type `?blocksFun` for the details. For complete control of the block-composition, use the argument `blocks`. In the list returned by `blocksFun()`, i.e. the object named `result` in the example above, the exact composition of the blocks produced by the internal algorithm is contained in the entry named `blocks`. Just as for `getsFun()`, the final specification is contained in the entry named `specific.spec`. However, note that here the entry is now a list rather than a vector.

The argument `x` argument can also be specified as a list of matrices, for example:

```
xlist <- list(x1=x[,5:30], x2=x[,26:50])
blocksFun(y, xlist)
```

This is useful when it makes sense to group subsets of variables together. Note that, as the example indicates, one or more regressors can enter more than one matrix or “group".

## 3.6 More speed: `turbo`, `max.paths` and parallel computing

In multi-path backwards elimination search, one may frequently arrive at a specification that has already been estimated and tested. As an example, consider the following two paths:

```
$paths[[1]]
[1]  2 4 3 1 5
```

```
$paths[[2]]
[1]  4 2 3 1 5
```

In path 1, i.e. `paths[[1]]`, regressor no. 2 is the first to be deleted, regressor no. 4 is the second, and so on. In path 2 regressor no. 4 is the first to be deleted, regressor no. 2 is the second, and so on. In other words, after the deletion of the first two variables, the set of remaining variables (i.e. 3, 1 and 5) in the two paths is identical. Accordingly, knowing the result from the first path, in path 2 it is unnecessary to proceed further after having deleted the first two regressors. Setting the argument `turbo` equal to `TRUE` turns such a check on, and thus skips searches, estimations and tests that are unnecessary. The turbo comes at a small computational cost (often less than 1 second), since the check is undertaken at each deletion. This is why the default is `turbo = FALSE` in `getsFun()`. However, if the estimation time is noticeable, then turning the turbo on can reduce the search time substantially. As a rule of thumb, if each estimation takes 1 second or more, then turning the turbo on will (almost) always reduce the total search time.

Searching more paths may increase the relevance proportion or potency. Whether and to what extent this happens depends on the sample size $n$, and on the degree of multicolinearity among the regressors $\boldsymbol{x}_t$. If $n$ is sufficiently large, or if the regressors are sufficiently uncorrelated, then searching fewer paths will not reduce the relevance proportion. In many situations, therefore, one may consider reducing the number of paths to increase the speed. This is achieved via the `max.paths` argument. Setting `max.paths = 10`, for example, means a maximum of 10 paths is searched. The paths that are searched are those of the 10 most insignificant variables (i.e. those with the highest $p$-values) in the starting model.

The functions `blocksFun()` and `isat()` implement a two-round version of block-based GETS modelling. In the first round the regressors $\boldsymbol{x}_t$ are split into $B$ blocks, and then GETS modelling is undertaken on each block. This is a socalled "embarassingly parallel" problem. To make `isat` search in parallel during the first round, simply set the argument `parallel.options` equal to an integer greater than 1. The integer determines how many cores/threads to use, and the command `detectCores()` can be used to find out how many cores/threads that are available on the current machine. Remember, it is not recommended to use all the cores/threads available. Within `blocksFun()` and `isat()`, parallel-computing is implemented with the `makeCluster()` and `parApply()` functions from the package **parallel**. If the time required by `makeCluster()` to set up parallel computing is negligible relative to the total computing time (on an average computer the setup-time is about 1 second), then the total computing time may – in optimal situations – be reduced by a factor of about $m - 0.8$, where $m > 1$ is the number of cores/threads used for parallel computing.

## 4 Illustrations

### 4.1 GETS modelling of Generalised Linear Models (GLMs)

The function `glm()` enables the estimation of a large number of specifications within the class of Generalised Linear Models (GLMs). Here, it is illustrated how GETS modelling can be implemented with GLMs. To fix ideas, the illustration is in terms of the logit-model.

Let $y_t \in \{0, 1\}$ denote the regressand of the logit-model given by

$$Pr\left(y_t = 1|\boldsymbol{x}_t\right) = \frac{1}{1 + \exp\left(-h_t\right)}, \qquad h_t = \boldsymbol{\beta}'\boldsymbol{x}_t. \tag{5}$$

Consider the following set of data:

```
n <- 40 #number of observations
k <- 20 #number of Xs
set.seed(123) #for reproducibility
y <- round(runif(40)) #generate Y
x <- matrix(rnorm(n*k), n, k) #create matrix of Xs
```

In other words, one regressand $y_t \in \{0, 1\}$ which is entirely independent of the 20 regressors in $\boldsymbol{x}_t$. The following interface function enables GETS modelling of logit-models:

```
logitFun <- function(y, x, ...){

  ##create list:
  result <- list()

  ##n, k and df:
  result$n <- length(y)
  if( is.null(x) || NCOL(x)==0 ){
    result$k <- 0
  }else{
    result$k <- NCOL(x)
  }
  result$df <- result$n - result$k

  ##call glm if k > 0:
  if( result$k > 0){
    tmp <- glm(y ~ x - 1, family = binomial(link="logit"))
    result$coefficients <- coef(tmp)
    result$vcov <- vcov(tmp)
    result$logl <- as.numeric(logLik(tmp))
  }else{
    result$coefficients <- NULL
    result$vcov <- NULL
    result$logl <- result$n*log(0.5)
  }

  ##return result:
  return(result)

}
```

To undertake GETS modelling:

```
getsFun(y, x, user.estimator=list(name="logitFun"))
```

Two variables are retained, namely $x_{5t}$ and $x_{11t}$, at the default significance level of 5% (i.e. `t.pval = 0.05`). To reduce the chance of retaining irrelevant variables, the significance level can be lowered to, say, 1% by setting `t.pval = 0.01`.

To implement GETS modelling for a different GLM model, only two lines of code must be modified in the user-defined function above. The first is the line that specifies the `family`, and the second is the one that contains the log-likelihood associated with the empty model (i.e. the line `result$logl <- result$n*log(0.5)`).

## 4.2 Creating a `gets()` method (S3) for a model of class `"lm"`

The package **gets** provides the generic function `gets()`. This enables the creation of GETS methods (S3) for models of arbitrary classes (type `?S3Methods` for more info on S3 methods). Here, this is illustrated for models of class `"lm"`. Since version 0.28 (August 2021), the **gets** package provides a method `gets()` for models of class `"lm"` (type `?gets.lm` for the details). So the example outlined here is purely for illustration purposes.

Suppose, for illustratory purposes, that a method `gets()` for models of class `"lm"` does not exist, and that our aim is to be able to do the following:

```
mymodel <- lm(y ~ x)
gets(mymodel)
```

That is, to first estimate a model of class `"lm"` where `x` is a matrix of regressors, and then to conveniently undertake GETS modelling by simply applying the code `gets(.)` to the named object `mymodel`. To this end, a function named `gets.lm()` that relies on `getsFun()` must be created. In doing so, a practical aspect is how to appropriately deal with the intercept codewise. Indeed, as we will see, a notable part of the code in the user-defined function will be devoted to the intercept. The reason for this is that `lm()` includes the intercept by default. Another practical aspect is whether to use `lm()` or `ols()` whenever a model is estimated by OLS (both employ the QR decomposition). The latter is simpler codewise, so here we opt for the latter.[7] The function is:

```
gets.lm <- function(object, ...){

  ##make y:
  y <- as.vector(object$model[, 1])
  yName <- names(object$model)[1]

  ##make x:
  x <- as.matrix(object$model[, -1])
  xNames <- colnames(x)
  if(NCOL(x) == 0){
```

[7]The function `gets.lm()` that ships with **gets** since version 0.28 employs `lm()` only, not `ols()`.

```
      x <- NULL
      xNames <- NULL
    }else{
      if(is.null(xNames)){
        xNames <- paste0("X", 1:NCOL(x))
        colnames(x) <- xNames
      }
    }

    ##is there an intercept?:
    if(length(coef(object)) > 0){
      cTRUE <- names(coef(object))[1] == "(Intercept)"
      if(cTRUE){
        x <- cbind(rep(1, NROW(y)), x)
        xNames <- c("(Intercept)", xNames)
        colnames(x) <- xNames
      }
    }

    ##do gets:
    myspecific <- getsFun(y, x, ...)

    ##which are the retained regressors?:
    retainedXs <- xNames[myspecific$specific.spec]
    cat("Retained regressors:\n ", retainedXs, "\n")

    ##return result
    return(myspecific)

  }
```

Next, recall the Data Generation Process (DGP) of the first experiment:

```
  n <- 40 #number of observations
  k <- 20 #number of Xs
  set.seed(123) #for reproducibility
  y <- rnorm(n) #generate Y
  x <- matrix(rnorm(n*k), n, k) #create matrix of Xs
```

We can now do GETS modelling on models of class `"lm"` by simply applying the code `gets()` on the object in question. As an example, the following code first stores an estimated model of class `"lm"` in an object named `startmodel`, and then applies the newly defined function `gets.lm()` to it:

```
  startmodel <- lm(y ~ x)
  finallm <- gets(startmodel)
```

The information from the specification search is stored in the object called `finallm`, and during the search the following is printed:

```
18 path(s) to search
Searching: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Retained regressors:
X10 X17 X18
```

In other words, the retained regressors are no. 10, 17 and 18. Note that, due to the way the user-defined function has been put together, the intercept is excluded from deletion. For a comparison with the function `gets.lm()` available in the **gets** package since version 0.28, type `gets::gets.lm(startmodel)`.

## 4.3 Regression with ARMA error

The function `arima()` can be used to estimate a linear regression with deterministic regressors and an error-term that follows an ARMA. An example is

$$y_t = \boldsymbol{\beta}' \boldsymbol{x}_t + \epsilon_t, \qquad \epsilon_t = \phi_1 \epsilon_{t-1} + \theta_1 u_{t-1} + u_t, \qquad u_t \sim WN\left(0, \sigma_u^2\right),$$

where $\boldsymbol{x}_t$ is a vector of deterministic regressors and $WN$ is short for White Noise. The error $\epsilon_t$ is thus governed by an ARMA(1,1). Let $x_t$ denote a (deterministic) step-shift variable in which the step-shift occurs at observation 30, i.e. $x_t = 1\,(t \geq 30)$. Next, consider the DGP given by

$$y_t = 4x_t + \epsilon_t, \qquad \epsilon_t = 0.4\epsilon_{t-1} + 0.1u_{t-1} + u_t, \qquad u_t \sim N\left(0, 1\right), \qquad t = 1, \ldots, n \tag{6}$$

with $n = 60$. In other words, the series $y_t$ is non-stationary and characterised by a large location shift at $t = 30$. Figure 1 illustrates the evolution of $y_t$, which is generated with the following code:

```
set.seed(123) #for reproducibility
eps <- arima.sim(list(ar = 0.4, ma = 0.1), 60) #epsilon
x <- coredata(sim(eps, which.ones = 30)) #step-dummy at t = 30
y <- 4*x + eps #the dgp
plot(y, ylab="y", xlab="t", lwd = 2)
```

By just looking at the graph, it seems clear that there is a location shift, but it is not so clear that it in fact occurs at $t = 30$. I now illustrate how the `arima()` function can be used in combination with `getsFun()` to automatically search for where the break occurs. The idea is to do GETS modelling over a set or block of step-indicators that cover the period in which the break visually appears to be in. Specifically, the aim is to apply GETS modelling to the following starting model with 11 regressors:

$$y_t = \sum_{i=1}^{11} \beta_i \cdot 1_{\{t \geq 24+i\}} + \epsilon_t, \qquad \epsilon_t = \phi_1 \epsilon_{t-1} + \theta_1 u_{t-1} + u_t.$$

To this end, we first need to make the user-specified estimator:

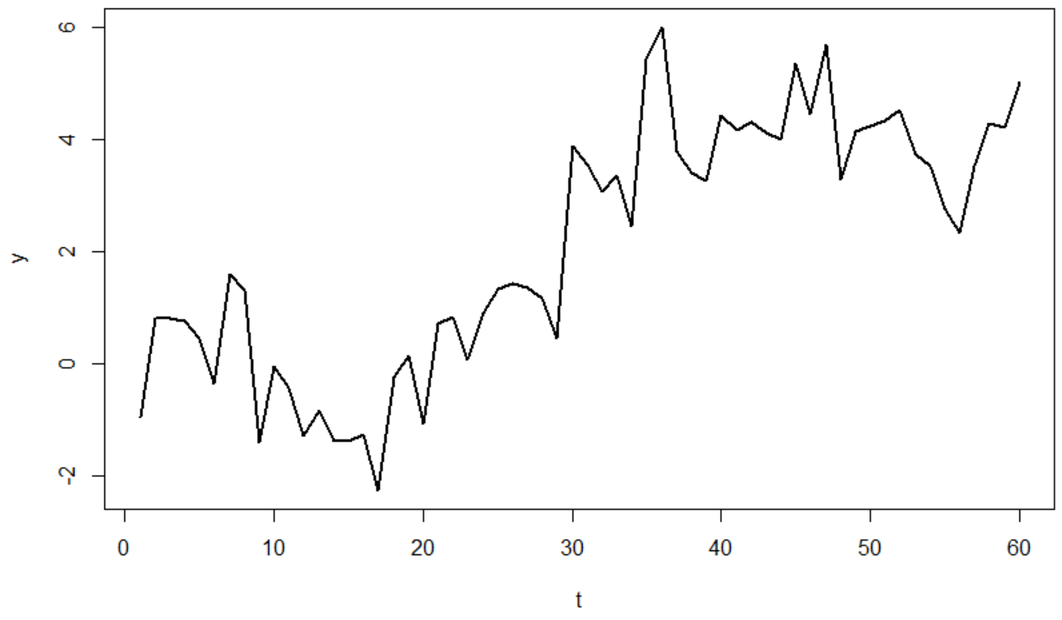

Figure 1: The graph of $y_t$ as given in (6).

```
myEstimator <- function(y, x){

  ##create list:
  result <- list()

  ##estimate model:
  if( is.null(x) || NCOL(x)==0 ){
    result$k <- 0
    tmp <- arima(y, order = c(1,0,1)) #empty model
  }else{
    result$k <- NCOL(x)
    tmp <- arima(y, order = c(1,0,1), xreg = x)
    result$coefficients <- tmp$coef[-c(1:3)]
    result$vcov <- tmp$var.coef
    result$vcov <- result$vcov[-c(1:3),-c(1:3)]
  }

  ##rename and re-organise things:
  result$n <- tmp$nobs
  result$df <- result$n - result$k
  result$logl <- tmp$loglik
```

```
    #return result:
    return(result)

  }
```

Note that the estimator has been put together such that the ARMA(1,1) specification of the error $\epsilon_t$ is fixed. As a consequence, the specification search is only over the regressors. The following code first creates the 11 step dummies, and then undertakes the GETS modelling:

```
  xregs <- coredata(sim(eps, which.ones = 25:35)) #11 step-dummies
  getsFun(y, xregs, user.estimator = list(name = "myEstimator"))
```

Two step-dummies are retained, namely those of $t = 30$ and $t = 35$.

## 4.4 Faster ISAT with large datasets

Block-based GETS modelling and ISAT methods in particular are computationally intensive. This is particularly the case when the number of observations $n$ is large in ISAT methods, since at least $n - 1$ indicators are included as regressors. Accordingly, as $n$ grows large, purpose-specific estimators can greatly reduce the computing time. One way of building such an estimator is by using tools from the package **Matrix**, see Bates and Maechler (2018). The code below illustrates this. First it loads the library, and then it creates a function named `olsFaster()` that re-produces the structure of the estimation result returned by the function `ols()` with `method = 3` (i.e. OLS with the ordinary coefficient-covariance), but with functions from **Matrix**. The code is:

```
  library(Matrix)
  olsFaster <- function(y, x){
    out <- list()
    out$n <- length(y)
    if (is.null(x)){ out$k <- 0 }else{ out$k <- NCOL(x) }
    out$df <- out$n - out$k
    if (out$k > 0) {
      x <- as(x, "dgeMatrix")
      out$xpy <- crossprod(x, y)
      out$xtx <- crossprod(x)
      out$coefficients <- as.numeric(solve(out$xtx,out$xpy))
      out$xtxinv <- solve(out$xtx)
      out$fit <- out$fit <- as.vector(x %*% out$coefficients)
    }else{ out$fit <- rep(0, out$n) }
    out$residuals <- y - out$fit
    out$residuals2 <- out$residuals^2
    out$rss <- sum(out$residuals2)
    out$sigma2 <- out$rss/out$df
    if (out$k > 0) { out$vcov <- as.matrix(out$sigma2 * out$xtxinv) }
    out$logl <- -out$n * log(2 * out$sigma2 * pi)/2 - out$rss/(2 * out$sigma2)
    return(out)
  }
```

Depending on the data and hardware/software configuration, the estimator may lead to considerably speed-improvement. In the following example, the function `system.time()` suggests a speed improvement of about 20% on the current hardware/software configuration:

```
set.seed(123) #for reproducibility
y <- rnorm(1000)
x <- matrix(rnorm(length(y)*20), length(y), 20)

#with ols():
system.time( finalmodel <- isat(y, mxreg = x, max.paths = 5) )

#with olsFaster():
system.time( finalmodel <- isat(y, mxreg = x, max.paths = 5,
  user.estimator = list(name = "olsFaster")) )
```

## References

Bates, D. and M. Maechler (2018). *Matrix: Sparse and Dense Matrix Classes and Methods*. R package version 1.2-15.

Campos, J., D. F. Hendry, and N. R. Ericsson (Eds.) (2005). *General-to-Specific Modeling. Volumes 1 and 2*. Cheltenham: Edward Elgar Publishing.

Castle, J., J. Doornik, D. F. Hendry, and F. Pretis (2015). Detecting Location Shifts During Model Selection by Step-Indicator Saturation. *Econometrics 3*, 240–264. DOI: https://doi.org/10.3390/econometrics3020240.

Clarke, D. (2014). General-to-specific modeling in Stata. *The Stata Journal 14*, 895–908. `https://www.stata-journal.com/article.html?article=st0365`.

Doornik, J. A. and D. F. Hendry (2018). *Empirical Econometric Modelling - PcGive 15*. London: Timberlake Consultants Ltd.

Dubois, É. and E. Michaux (2019). Grocer 1.8: an econometric toolbox for Scilab. `http://dubois.ensae.net/grocer.html`.

Hendry, D. F., S. Johansen, and C. Santos (2008). Automatic selection of indicators in a fully saturated regression. *Computational Statistics 23*, 317–335. DOI 10.1007/s00180-007-0054-z.

Hoover, K. D. and S. J. Perez (1999). Data Mining Reconsidered: Encompassing and the General-to-Specific Approach to Specification Search. *Econometrics Journal 2*, 167–191. Dataset and code: `http://www.csus.edu/indiv/p/perezs/Data/data.htm`.

IHS Markit (2020). *EViews Version 11*. Irvine: IHS Markit.

Lovell, M. C. (1983). Data Mining. *The Review of Economics and Statistics 65*, 1–12.

Pretis, F., J. Reade, and G. Sucarrat (2018). Automated General-to-Specific (GETS) Regression Modeling and Indicator Saturation for Outliers and Structural Breaks. *Journal of Statistical Software 86*, 1–44.

Schwarz, G. (1978). Estimating the Dimension of a Model. *The Annals of Statistics 6*, 461–464.

Sucarrat, G., J. Kurle, F. Pretis, J. Reade, and M. Schwarz (2021). *gets: General-to-Specific (GETS) Modelling and Indicator Saturation (ISAT) Methods*. R package version 0.29.