modelObj: A Model Object Framework for Regression Analysis

Shannon T. Holloway

June 5, 2022

Abstract

It is becoming common practice for researchers to disseminate new statistical methods to the broader research community by developing and releasing `R` packages that implement their methods. Often, methods are developed on the framework of traditional regression. To simplify software development, researchers and developers often make choices regarding the types of regression models that can be used; hard-coding a specific regression method into the library and limiting or eliminating the ability of the user to modify regression control parameters. In many instances, there is not a fundamental reason why a new statistical method should use a specific regression method (e.g., linear vs. non-linear) and such choices artificially limit the general application of new packages. In addition, if a new method requires multiple regression steps, a developer must artificially break the method into multiple function calls, each for a specific regression step, or provide a cumbersome and/or confusing interface. We have developed a new `R` package to facilitate the use of existing and future `R` regression libraries that simplifies the development of general, non-model-specific implementations of new statistical methods.

Keywords: Regression and classification

1 Introduction

IMPACT is a joint venture of the University of North Carolina at Chapel Hill, Duke University, and North Carolina State University. The program project aims to improve the health and longevity of people by improving the clinical trial process. A key component of this research has been the development of public-use software packages that implement new statistical methods developed by the 30+ investigators. Whenever possible, these methods have been developed in `R`. The `modelObj` library was born from the need to create simple, general-use implementations of new statistical methods that do not limit the underlying regression method(s) and do not require continued upgrading as new regression methods become available.

When creating `R` packages for statistical methods developed on the framework of traditional regression or classification methods, researchers and/or software developers often make choices regarding the types of models that can be specified by the user; hard-coding the regression method into the library and limiting or eliminating the ability of the user to modify regression control parameters. However, the choice of a specific regression method may not be fundamental constraint of the new method, and such choices can limit the general application of an implementation.

In addition, a new method may require multiple regression steps. For example, `DynTxRegime` implements the Augmented Inverse Probability Weighted Estimators (AIPWE) for average treatment effects and requires multiple regression analyses. To implement this method generally without using the framework described herein would require that the procedure be artificially broken into multiple function calls, each for a specific regression step, or that the user interface to the method be cumbersome and/or confusing.

The `modelObj` library is built on the premise of a “model object." A model object contains all of the information needed to complete a standard regression analysis and subsequent prediction step: a formula object, the existing `R` regression method to be used to obtain parameter estimates (the so-called solver method), any control arguments to be passed to the regression method, the `R` method to be used to obtain predictions, and any arguments to be passed to the prediction method. This information is grouped into a single object of class `modelObj` by a call to `buildModelObj(...)`. To use a package built on the model object framework, the user creates the `modelObj` prior to calling the statistical method and passes the model object as input. The `modelObj` library provides simple functions that developers can use to implement standard regression procedures, such as `fit(...)` to obtain parameter estimates and `predict(...)` to obtain predictions.

2 Interacting with packages that implement the model object framework.

Users of packages that have been developed based on the model object framework will interface with the `modelObj` library through calls to `buildModelObj(...)`. These calls create a model object for a single regression step and are passed as input to the method. The `buildModelObj(...)` function takes as input

- **model** : an object of class formula. Any lhs variables provided will be ignored. If the fitting function specified in **solver.method** takes as input a model matrix rather than a formula object, **model** will be used to obtain the model matrix.

- **solver.method** : an object of class character; the name of the `R` regression method. For example, a user might commonly specify 'lm' or 'glm.' For classification, 'rpart' might be used. **The specified method MUST have a corresponding predict method.**

- **solver.args** : an object of class list; additional arguments to be passed to solver.method. The name of each element of the list must match a formal argument of solver.method. For example, for logistic regression using glm:

  ```
  solver.method = "glm"
  solver.args = list("family"=binomial).
  ```

  If **solver.method** takes as input a formula object, it is assumed that the function specified has formal arguments "formula" and "data." If the solver.method does not use "formula" and/or "data," **solver.args** must explicitly indicate the variable names used for these inputs. For example, list("x"="formula") if the formula object is passed to solver.method through input argument "x" or list("df"="data") if the data.frame object is passed to solver.method through input argument "df."

  If **solver.method** instead takes as input a model matrix, it is assumed that the function specified has formal arguments "x" and "y" for the design matrix and response, respectively. If the solver.method does not use "x" and/or "y," **solver.args** must explicitly indicate the variable names used for these inputs. For example, list("X"="x") if the formula object is passed to solver.method through input argument "X" or list(“Y"=“y") if the data.frame object is passed to solver.method through input argument “Y."

- **predict.method** : an object of class character; the function name of the `R` function to be used to obtain predictions. For example, ‘predict.lm’ or ‘predict.glm.’ If no function is explicitly given, the generic ‘predict’ method is assumed. Most often, this input can be omitted.

- **predict.args** : an object of class list; additional arguments to be passed to predict.method. The name of each element of the list must match a formal argument of predict.method. For example, if a logistic regression using glm was used to fit the model formula object and predictions on the scale of the response are desired,

  ```
  solver.method = "glm"
  solver.args = list("family"=binomial)
  predict.method = "predict"
  predict.args = list("type"="response").
  ```

  It is assumed that the `R` method specified in predict.method has formal arguments “object" and “newdata." If predict.method does not use these formal arguments, predict.args must explicitly indicate the variable names used for these inputs. For example, list(“x"=“object") if the object returned by solver.method is passed to predict.method through input argument “x" or list(“ndf"=“newdata") if the data.frame object is passed to predict.method through input argument “ndf."

Unless modified through **solver.args** and **predict.args**, default settings are assumed for the methods specified in **solver.method** and **predict.method**.

As a simple example,

```
> library(modelObj)
> object1 <- buildModelObj(model=~x1, solver.method='lm')
```

defines a model object for a linear model, the parameter estimates of which are to be obtained using `lm`, and predictions obtained using `predict`. The solver and prediction methods will use default settings.

As a more complex (though contrived) example, consider the following functions

```
> mylm <- function(X,Y){
+   obj <- list()
+   obj$lm <- lm.fit(x=X, y=Y)
+   obj$var <- "does something neat"
+   class(obj) = "mylm"
+   return(obj)
+ }
> predict.mylm <- function(obj,data=NULL){
+   if( is(data,"NULL") ) {
+     obj <- exp(obj$lm$fitted.values)
+   } else {
+     obj <- data %*% obj$lm$coefficients
+     obj <- exp(obj)
+   }
+   return(obj)
+ }
```

which, for the sake of argument, represent a “new" regression method that a user would like to utilize. These functions are chosen to illustrate solver methods and prediction methods that do not accept the standard formal arguments. They provide a simple illustration of how flexible the framework can be. In this circumstance, the user would define the following modeling object:

```
> object2 <- buildModelObj(model = ~x1,
+                          solver.method = mylm,
+                          solver.args = list('X' = "x", 'Y' = "y"),
+                          predict.method = predict.mylm,
+                          predict.args = list('obj' = "object",
+                                              'data' = "newdata"))
```

3 Developing packages that implement the model object framework.

The `buildModelObj()` function invoked by a user returns an object of class `modelObj`.

Developers that use this utility package should carefully document for users any required settings for **solver.method** and **predict.method**. For example, the scale of the response needed for predictions. Though the developer can access and modify the argument lists provided by users using methods `predictorArgs()` and `solverArgs()`, there is no strict variable naming convention in `R`, and some methods do not adhere to the “usual" choices. Thus, identifying the formal argument to adjust may be tricky.

Once provided an object of class modelObj, developers can **see** all of the values contained in the object but can modify only the argument lists to be passed to methods. Specifically:

- `model` Retrieves the formula object.
- `solver` Retrieves the character name of the regression method.
- `solverArgs` Retrieves the list of arguments to be passed to the regression method.
- `predictor` Retrieves the character name of the prediction method.
- `predictorArgs` Retrieves the list of arguments to be passed to the prediction method.
- `solverArgs<-` Sets the list of arguments to be passed to the regression method.
- `predictorArgs<-` Sets the list of arguments to be passed to the prediction method.

The primary utility method available for objects of class `modelObj` is `fit(...)`, which implements the regression step. The inputs for `fit(...)` are:

- **object** an object of class `modelObj`.
- **data** an object of class data.frame; the covariates to be used to obtain the fit.
- **response** an object of class numeric; the response
- `...` ignored

The `fit(...)` method constructs and executes the function call to the specified solver method using the formula object and formal arguments provided by the user in solver.args. The `fit(...)` method uses an internal naming convention for the response, and thus only the right-hand-side of the formula object is referenced.

`fit(...)` returns an S4 object of class `modelObjFit`. Developers can access members of this class using the following methods:

- `fitObject` Retrieves the value object returned by the regression method. Through this retrieve method, developers have the ability to access any defined methods for the regression function, such as `coef`, `residuals`, or `plot`.
- `predictor` Retrieves the character name of the prediction method to be used to obtain predictions.
- `predictorArgs` Retrieves the list of arguments to be passed to the prediction method when making predictions.

Note that predictor and predictorArgs only give you access to **see** what has been specified for the prediction method. Should changes need to be made to the arguments, one must apply these changes to the defining modelObj before creating the modelObjFit object.

Additional methods available for `modelObjFit` objects are

- `coef(...)` If defined for the regression method, returns the estimated coefficients.
- `plot(...)` If defined for the regression method, generates the plot of the model fitting class.
- `predict(...)` Obtains predictions from the results of the model fitting function.
- `residuals(...)` If defined for the regression method, returns the residuals.
- `show(...)` Uses the predefined show method of the regression method.
- `summary(...)` Uses the predefined summary method of the regression method.

Again, the value object returned by the regression method can be retrieved using `fitObject`; thereby providing access to any `R` methods developed for the regression method. We have chosen to implement only the most common methods (`coef()`, `residuals()`, etc.) for the `modelObjFit` object. Note that not all regression methods have these functions. If these functions are required in your implementation, additional checks must be incorporated into your code to ensure their availability.

4 Example Implementation of `modelObj`

We use a standard `R` dataset to illustrate the implementation of the model object framework. The 'pressure' data frame contains "data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters (of mercury)." The details of the datatset are not relevant for this illustration. However, the reader is referred to ?pressure for details.

```
> summary(pressure)
```

```
  temperature     pressure     
 Min.   :  0   Min.   :  0.0002  
 1st Qu.: 90   1st Qu.:  0.1800  
 Median :180   Median :  8.8000  
 Mean   :180   Mean   :124.3367  
 3rd Qu.:270   3rd Qu.:126.5000  
 Max.   :360   Max.   :806.0000  
```

It is straightforward to implement a regression step. As an example, suppose we are developing a new package called `wow`. The primary function of this package is `exampleFun()`. In this function, we want to obtain a fit and return the square of the fitted response and the estimated coefficients in a list. Our function takes the following form

```
> exampleFun <- function(modelObj, data, Y){
+ 
+     fitObj <- fit(object = modelObj, data = data, response = Y)
+ 
+     ##Test that coef() is an available method
+     cfs <- try(coef(fitObj), silent=TRUE)
+     if(class(cfs) == 'try-error'){
+       warning("Provided regression method does not have a coef method.\n")
+       cfs <- NULL
+     }
+
+     fitted <- predict(fitObj)^2
+
+     return(list("fittedSq"=fitted, "coef"=cfs))
+ }
```

To use this function, a user must create the object of class `modelObj` and provide it as input to `exampleFun()`. The user can implement a linear model as follows:

```
> ylog <- log(pressure$pressure)
> objlm <- buildModelObj(model = ~temperature,
+                        solver.method = "lm",
+                        predict.method = "predict.lm",
+                        predict.args = list("type"="response"))
> fitObjlm <- exampleFun(objlm, pressure, ylog)
> print(fitObjlm$coef)
```

```
(Intercept) temperature
-6.06814354  0.03979188
```

Or, the non-linear least squares method `nls`:

```
> objnls <- buildModelObj(model = ~exp(a + b*temperature),
+                         solver.method = "nls",
+                         solver.args = list('start'=list(a=1, b=0.1)),
+                         predict.method = "predict",
+                         predict.args = list("type" = "response"))
> fitObjnls <- exampleFun(objnls, pressure, pressure$pressure)
> print(fitObjnls$coef)
```

```
          a           b
-0.67814843  0.02052001
```

Or, even the previously defined “new" method:

```
> objectnew <- buildModelObj(model = ~temperature,
+                            solver.method = mylm,
+                            solver.args = list('X' = "x", 'Y' = "y"),
+                            predict.method = predict.mylm,
+                            predict.args = list('obj'="object",
+                                                'data'="newdata"))
```

```
> fitObjnew <- exampleFun(objectnew, pressure, ylog)
> print(fitObjnew$coef)
```

```
NULL
```

In the last example, the function returned NULL for the parameter estimates because there is no available method to retrieve the estimated parameters.

The same function, `exampleFun()` can be used to implement each of these models, and no development is required to extend the wow function to new regression methods as they become available.