Appendix A  Object-oriented programming for statistical analysis in R

A.1. Introduction to Object-oriented programming

R has a system for object-oriented programming (OOP), which is based on generic functions. In general, OOP is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. OOP-related programming techniques may include features such as information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance. These techniques started to be commonly used in mainstream software application development in the early 1990s. Nowadays, many modern programming languages support OOP.

In R, some basic concepts concerning OOP are as follows.

✓ A **class** is a description of a thing (i.e., how objects of a certain type look like).
✓ An **object** is an instance of a class.
✓ A **generic function** is a function which dispatches methods. A generic function typically encapsulates a “generic” concept, such as plot, mean, logLik, residuals, predict, summary, and so on and so forth. The generic function does not actually do any computation.
✓ A **method** is the implementation of a generic function for an object of a particular class.

A class defines how an object is represented in the program, and an object is an instance of the class that exists at run time. Thus, the class is the abstract definition, and every time when we actually create an instance of the class, we create an object of the class. As a real world analog, TV is a class, and particular TV sets such as the ones at my home are objects of the TV class.

Once the classes are defined we probably want to perform some computations on objects, implemented by the generic function. A generic function dispatches methods to objects in such as way that the same function performs different computations depending on the classes of its arguments. We’ll see how a generic function works in section A.3.

In general, OOP provides the following advantages for programming.
- Stability of models with respect to real-world entities
- Iterative construction, which is made easier by the weak coupling between components
- The ability to reuse elements across development projects

OOP provides some further merits for statistical programming, as stated below:

- Code reusability
- Enforced organization of development process
- Architecture of software more closely resembles the statistical relationships
- Protection against errors

### A.2. A first experience with generic functions

As a first experience, we define a S3 generic function called `whoami` and some tagged methods to dispatch to:

```r
> #--- Create a generic method
> whoami <- function(x, ...) UseMethod("whoami")
```

Seemingly, define a generic function is easy. All we need to do is to define a function with argument `x` (which is an object), and the body of the function is simply `UseMethod(generic)`, where `generic` is character string. In general, the syntax of `UseMethod` is:

```
UseMethod(generic, object)
```

Here, both arguments are optional according to the help page for the function. The second argument, `object` can safely be excluded as it will be ignored anyway. If the method name argument (i.e., the first argument) is also excluded, then the name of the method to be called will be the same as the name of the generic function.

Then, it is the time to define methods for this generic function. Here, two methods are defined to identify the identity of an object (i.e., person), `teacher` and `student`, in addition to a default method which displays the message “I don’t know who I am” (but I would wish you).

```r
> #--- Define methods for the generic function
> whoami.student <- function(x) print("I am a student")
> whoami.teacher <- function(x) print("I am a teacher")
> whoami.default <- function(x) print("I don't know who I am")
```

The mechanism of method dispatch in a generic function is as follows. The first argument of a generic function is an object of a particular class. Note that there may be other arguments present in the function, but, in
S3, only the class of the first argument matters in the method dispatching (though it is possible to select the method to be called from a set of one or more arguments in the S4 paradigm.). First, the generic function checks the class that the object belongs to, and, then, a search is done to see if there is an appropriate method for that class:

- If there is a method for that class, the method is called on the object and we're done.
- If there is no method for that class, a default method, if it exists, is called and we are done too.
- If a default method doesn't exist, then an error is thrown.

In R, every object may belong to one or more classes, or none of them. So, the method resolution order (MRO) of a given object is simply a vector of named classes, if any, in its class attribute.

In the follow, we try to tag two objects (e.g., persons), \( p1 \) and \( p2 \), by their class members (which however have not been given so far).

```r
> p1 = "Vazquez"
> p2 = "Gianola"
> whoami(p1)  # No class assigned
[1] "I don't know who I am"
> whoami(p2)  # No class assigned
[1] "I don't know who I am"
```

Then, we assign attributes to objects \( att1 \) and \( att2 \), and then re-tag them.

```r
> attr(p1,'class') <- 'student'
> attr(p2,'class') <- c('teacher', 'advisor', 'staff')
> whoami(p1)      # p1 has a “student” class
[1] "I am a student"
> whoami(p2)      # Search the MRO vector for defined method
[1] "I am a teacher"
> attr(p1,'class') <- 'teacher'  # Now, change the class of p1 to “teacher”
> whoami(p1)      # Now p1 has a “teacher” class
[1] "I am a teacher"
```

Now, you see that the S3 system is very interactive, and you can change the class of an object easily.

### A.3 S3 vs. S4 systems

So far, we have had a first flavor of the S3 generic function. In R, there are two object systems, known informally as S3 and S4, and each system can be used fairly independently of the other.
S3 objects, classes and methods have been available in R from the beginning (Chambers and Hastie, 1992). They are informal and very interactive. In some sense, S3 is more a set of naming conventions than a true OOP system, but it is sufficient for most purposes. Nevertheless, many developers keep on use S3 classes/methods because they are “quick and dirty”.

S4 objects, classes and methods are much more formal and rigorous, but less interactive than the S3 system (Chambers, 1998). The S4 system is available through the methods package, attached by default since version 1.7.0. As formal object oriented programming within an interactive environment, S4 can help a lot to write clean and consistent code, checks automatically if objects conform to class definitions. S4 classes/methods are used extensively in the Bioconductor project.

The `summary` function is a typical example of the S3 generic functions (as in the `base` package). Its definition is briefly displayed as follows.

```r
> summary
function (x, ...) 
UseMethod("summary")
<environment: namespace:base>
```

The `show` function is an example of the S4 generic functions (as in the `methods` package), which is displayed differently in the definition.

```r
> show
standardGeneric for "show" defined from package "methods"
function (object)
standardGeneric("show")
<environment: 0x8d7cdc8>
Methods may be defined for arguments: object
```

Next, let us see how a generic function works using the `summary` function as an example.

```r
> # object x has a “factor” class
> x <- sample(letters[1:2], 20, replace=T)
> x<-factor(x)
> x
[1] b b a a b a b a b a b a b b b b a b b b
Levels: a b
> class(x)
[1] “factor”
> summary(x)
a  b
```
> # object y has a "numeric" class
> y <- rnorm(20)
> y
>     [1] -0.41086375  0.12307135 -1.11066349 -1.78317568  0.07674794 -0.54638233
>     [7]  1.72942745  0.07904103 -1.51675804  1.13866626 -0.16302497 -0.46831334
>     [13]  0.38975598  1.08018292  0.53280307  1.12501742  0.05243098 -0.57400977
>     [19] -0.89831398  1.53820817
> class(y)
> [1] "numeric"
> summary(y)
>    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     
> -1.78300 -0.55330  0.06459  0.01969  0.66960  1.72900

> # object lm.demo has a "lm" class
> lm.demo <- lm(y~x)
> class(lm.demo)
> [1] "lm"
> summary(lm.demo)

Call:
  lm(formula = y ~ x)

Residuals:
    Min     1Q   Median     3Q    Max
-1.7500 -0.6665 -0.1008  0.5537  1.5156

Coefficients:        
Estimate Std. Error t value  Pr(>|t|)
(Intercept) -0.3769     0.3609  -1.044    0.310
xb            0.6102     0.4477   1.363    0.190

Residual standard error: 0.9549 on 18 degrees of freedom
Multiple R-squared: 0.09355,  Adjusted R-squared: 0.04319
F-statistic: 1.858 on 1 and 18 DF,  p-value: 0.1897

The class of the y object is “numeric”, but there is no summary method for the “numeric” class. So, the `summary.default` function is called, and the five-number summary plus the mean is calculated. The class of object x is “factor”, and the `summary.factor` function is called, producing a contingency table. The lm.demo object belongs to the lm class, and result summaries from the simple linear regression were output by a call to the `summary.lm` function.
Evidently, the `summary` function performs different operations on objects of different classes. As a matter of fact, far more methods are available by a call to the summary function, and these methods are displayed as follows.

```r
> methods(summary)
[1] summary.agnes*       summary.aov            summary.aovlist
[7] summary.data.frame   summary.Date         summary.default
[10] summary.diana*       summary.dissimilarity* summary.ecdf*
[13] summary.factor       summary.fanny*       summary.find.matches
[16] summary.formula      summary.glm          summary.impute
[19] summary.infl         summary.ldBands      summary.lm
[22] summary.loess*       summary.manova       summary.matrix
[25] summary.mChoice      summary.mlm          summary.mona*
[28] summary.nls*         summary.packageStatus* summary.pam*
[31] summary.POSIXct      summary.POSIXlt       summary.ppr*
[34] summary.prcomp*      summary.princomp*      summary.shingle*
[37] summary.silhouette* summary.stepfun       summary.stl*
[40] summary.table        summary.transcan      summary.trellis*
[43] summary.tukeysmooth*
```

Non-visible functions are asterisked

In R, classes are attached to objects as an attribute:

```r
> attributes(x)
$levels
 [1] "a" "b"
$class
 [1] "factor"
```

For a generic function, a method is defined for each of the relevant classes. For example, the `summary.factor` function defines the method for the “factor” class, the code of which is displayed as follows.

```r
> summary.factor
function (object, maxsum = 100, ...)
{
  nas <- is.na(object)
  ll <- levels(object)
  if (any(nas))
    maxsum <- maxsum - 1
  tbl <- table(object)
  tt <- c(tbl)
  names(tt) <- dimnames(tbl)[[1]]
}
if (length(ll) > maxsum) {
    drop <- maxsum:length(ll)
    o <- sort.list(tt, decreasing = TRUE)
    tt <- c(tt[o[-drop]], `(Other)` = sum(tt[o[drop]]))
}
if (any(nas))
    c(tt, `NA's` = sum(nas))
else tt
}
<environment: namespace:base>

As a general rule, you are not expected to call methods directly in OOP. Instead, you should use a generic function to dispatch methods to objects of different classes.

In S3 there is no formal definition of a class. A class can be assigned to an object by using `attr`, or simply by setting the class attribute of an object to the name of the desired class:

```r
> # now, newx has a “factor” class, because x has a “factor” class
> x.new<-x
> class(x.new)
[1] “factor”
> x.new  # same as print(newx)
[1] b b a a b a b a b a a b a a b a b b b a b a
Levels: a b
```

> # change the class of x.new to “myvector”
> class(x.new)<-”myvector”
> class(x.new)
[1] “myvector”

Note that it is useless to define a class without defining any method for the class. Next, we define a new `print` method, namely `print.myvector`, for the `myvector` class.

```r
> print.newvector<-function(x,...) {
+    cat(“This is my vector, and its content is kept confidential!”
+}
```

Now, try to print the content of x.new, but you won’t be able to see its this time.

> x.new  # or, print(x.new)
This is my vector, and its content is kept confidential!
So, we see that S3 is highly interactive, and new classes can be created and methods changed very easily. This convenience, however, can be associated with misuse of generic functions, leading to unsuccessful method dispatch. In S3, because everything is just solved by naming conventions, it is easy to give a class to an object that by no means fit in that class. For example, the function `lm` returns objects of class "lm". Methods for that class of course expect a certain format for the objects, e.g., that they contain regression coefficients and so on. However, we may falsely do the following thing:

```r
> class(x.new) <- "lm"
> class(x.new)
[1] "lm"
> x.new
Error in x$call : $ operator is invalid for atomic vectors
```

See, an error message was given because the `x.new` object does not have the expected structure of an “lm” object. When this happened, the computer should detect the real problem, but it did not. Instead, S4 can handle this situation properly.

In S4, a new class is defined using `setClass`, and the method is defined using `setMethod`. Below is an example for defining a polygon class and a method on the objects of this class.

```r
> setClass("polygon", representation(x = "numeric", y = "numeric"))
[1] "polygon"

> setMethod("plot", "polygon", function(x, y, ...) {
+   xlim <- range(x@x)
+   ylim <- range(x@y)
+   plot(0, 0, type = "n", xlim = xlim,
+     ylim = ylim , ...)
+   xp <- c(x@x, x@x[1])
+   yp <- c(x@y, x@y[1])
+   lines(xp, yp)
+ })
[1] "plot"

> p <- new("polygon", x = c(1,2,3,4), y = c(1,2,3,1))
> plot(p)
```
So, in S4, when an object created as an instance of a class, the object is not only given a class name, but it also comes with required data for that class. Thus, the misuse of the class to a wrong data type can be avoided.

A.4 OOP for statistical analysis

A.4.1 Why use OOP in programming for statistical analysis

Here, we give an example that evidently justifies the use of OOP in statistical programming. Suppose we have a binomial data \( x_i \sim Binomial(n,p) \), for \( i = 1, ..., n \), where \( n \) is the number of trials and \( p \) is the success probability. The mean and variance of the binomial data are give by \( E(x_i) = p \) and \( Var(x_i) = np(1-p) \), respectively. Now, we define a function, called \( stats \), which returns the mean, standard deviation, and the number of trials.

```r
stats <- function(x) {
  n = length(x)
  p = mean(x)
  mu = n*p
  sigma = sqrt(n*p*(1-p))
  return( list(mu = mu, sigma = sigma, n = n) )
}
```

Now, we generate 100 random numbers, independently and identically, from a Bernulli distribution with \( p=0.3 \). Then, the sum of the 100 Bernulli random variables is supposed to follow Binomial(100,0.3). We
compute the mean and standard deviation of the binomial data distribution using the *stats* function.

```r
> y1 <- rbinom(100, size=1, p=.3)
$mu
[1] 33

$sigma
[1] 4.702127

$n
[1] 100

The above results look pretty well. However, the outcome could go terribly wrong if we apply this function to a wrong data type. Suppose we have a sample generated from a standard normal distribution with mean = 0.3 and sd = 1. If by chance we falsely apply the *stats* function to this normal data, then the computed mean and the standard deviation would totally be wrong.

```r
> y2<-rnorm(100,mean=0.3,sd=1)
> stats(y2)
$mu
[1] 42.52809

$sigma
[1] 4.943855

$n
[1] 100

Obviously, it does not make any sense to calculate the mean and variance for a normal data in the way as for a binomial data. The mean and standard deviation of the normal data are as follows:

```r
> mean(y2)
[1] 0.4252809

> sd(y2)
[1] 1.005184

Though we could define functions for binomial and normal data, separately. More specifically, we can define a function called *stats.binomial* for the binomial data, and a function called *stats.normal* for the normal data. Each function implements an appropriate method to compute the mean and standard deviation for its data type. However, a
A careless person can still use them in wrong ways, such as `stat.binomial(y2)` or `stat.normal(y1)`, because there is no name tag on the “face” of a data vector. The type of method misuse, however, can be avoided by following object-oriented programming.

**Defining generic functions, classes, and methods.**

For simplicity, we show how to define generic functions, classes, and methods in the S3 system. First, we need to define functions as the class constructors, and assign y1 and y2 to different classes.

```r
> as.binomial <- function(x) {
+    class(x) <- "binomial"
+    return( x )
+ }
> as.normal <- function(x) {
+    class(x) <- "normal"
+    return( x )
+ }
> y1 <- as.binomial(y1)
> class(y1)
[1] "binomial"
> y2 <- as.normal(y2)
> class(y2)
[1] "normal"
```

Then, we define a generic function, namely `stats`, and three methods associated with this generic function. The three method functions include one that computes the mean and standard deviation for the normal data, one counterpart for the binomial data, and a default method for handling unknown type of data.

```r
stats <- function(x) UseMethod("stats")
stats.binomial <- function(x) {
    n = length(x)
    p = mean(x)
    mu = n*p
    sigma = sqrt(n*p*(1-p))
    return( list(mu = mu, sigma = sigma, n = n) )
}
stats.normal <- function(x) {
```

```r
```
\begin{verbatim}
n = length(x)
mu = mean(x)
sigma = sd(x)
return( list(mu = mu, sigma = sigma, n = n) )
\}

stats.default <- function(x) stop("Unknown data type!")

Now, let us see what will be happening if we apply this generic function to the two data vector, y1 and y2.

$mu
[1] 33

$sigma
[1] 4.702127

$n
[1] 100

> stats(y2)
$mu
[1] 0.4252809

$sigma
[1] 1.005184

$n
[1] 100

Further, suppose we have a data vector y3, which is not given a known class (i.e., data type). If we apply the stats function to this data vector, the default method is applied and a warning of “Unknown data type” is given. This way, the wrong results from method misuse is avoided.

> y3 <- rnorm(100,mean=2,sd=1)
> stats(y3)
Error in stats.default(y3) : Unknown data type!

Inheritance

Among the many features of OOP, a very useful one is inheritance, which is a way to form new classes using classes that have already been defined. By the mechanism of inheritance, a new class (known as a derived class) takes over or inherits attributes and behavior of pre-existing classes (referred to as base classes or ancestor classes).
Inheritance also facilitates the reuse of existing code with little or no modification.

Let us define a new class and methods for a standard normal data by making use of those we previously defined for the normal data.

```r
as.standardNormal <- function(x){
  x <- as.normal(x)
  class(x) <- c("standardNormal", class(x))
  return( x )
}

stats.standardNormal <- function(x){
  object <- stats.normal(x)
  object$sigma <- 1
  return(object)
}

> y4<-rnorm(100)
> y4<-as.standardNormal(y4)
> stats(y4)
$mu
[1] 0.03292102

$sigma
[1] 1

$n
[1] 100

> class(y4)
[1] "standardNormal" "normal"
```

**A.5. OOP for building R statistical packages**

OOP is very useful in building a statistical package. In this section, we illustrate the usefulness of OOP in developing a package for linear regression analysis based on the following model:

\[ y = X\beta + e \]

where \( e \sim N(0, \sigma^2) \)

Here, our goal is not to implement all the bells and whistles of the `lm` function already available in R, but to show how OOP helps build a linear
regression package that has a “professional look and feel”, resembling the interface of \textit{lm}.

Here, we do not go into details of statistical theory regarding the estimations of regression coefficients, which are covered in Chapter 6. Instead, we have already had the R function for linear regression. The actual algorithm used here differs somewhat from that in Chapter 6, but both programs do the same thing and presumably produce the same results.

\begin{verbatim}
lm2Est <- function(x, y) 
{
   ## compute QR-decomposition of x
   qx <- qr(x)
   ## compute (x'x)^(-1) x'y
   coef <- solve.qr(qx, y)
   ## degrees of freedom and standard deviation of residuals
   df <- nrow(x) - ncol(x)
   sigma2 <- sum((y - x %*% coef)^2)/df
   ## compute sigma^2 * (x'x)^-1
   vcov <- sigma2 * chol2inv(qx$qr)
   colnames(vcov) <- rownames(vcov) <- colnames(x)
   list(coefficients = coef, vcov = vcov, sigma = sqrt(sigma2), df = df)
}
\end{verbatim}

The above function is then used to predict heart weight from body weight in the classic Fisher cats data from package MASS (Venables & Ripley (2002)).

\begin{verbatim}
> data(cats, package="MASS")
> lm2Est(cbind(1, cats$Bwt), cats$Hwt)

$coefficients
[1] -0.3566624 4.0340627

$vcov
[,1]       [,2]
[1,] 0.4792475 -0.1705820
[2,] -0.1705820  0.0626308

$sigma
[1] 1.452373

$df
[1] 142
\end{verbatim}

Then, we compared the above results with those obtained from a call to the \textit{lm} function.

\begin{verbatim}
> lm1 <- lm(Hwt~Bwt, data=cats)
\end{verbatim}
> lm1
Call:
lm(formula = Hwt ~ Bwt, data = cats)
Coefficients:
(Intercept) Bwt
-0.3567 4.0341
> vcov(lm1)
(Intercept) Bwt
(Intercept) 0.4792475 -0.17058197
Bwt -0.1705820 0.06263081

We see that the numerical estimates are exactly the same, but our program lacks a nice look of user interface in the following aspects:

✓ Prettier formatting of results.
✓ Add utilities for fitted model like a summary() function to test for significance of parameters.
✓ Handle categorical predictors.
✓ Use formulas for model specification.

Luckily enough, OOP can help us with issues 1 and 2, formula with 3 and 4. Following the rules of OOP in R, we need to make our main function `lm2` generic. As has been mentioned, a generic function is a standard R function with a special body, usually containing only a call to `UseMethod`:

```r
lm2 <- function(x, y) UseMethod("lm2")
```

Then, we add a default method, namely `lm2.default`.

```r
lm2.default <- function(x, y) {
  x <- as.matrix(x)
  y <- as.numeric(y)
  est <- lm2Est(x, y)
  est$fitted.values <- as.vector(x %*% est$coefficients)
  est$residuals <- y - est$fitted.values
  est$call <- match.call()
  class(est) <- "lm2"
  est
}
```

This function converts its first argument `x` to a matrix, and the second argument `y` to a numeric vector. Then, results from the linear regression analysis are stored to the `est` object, which is a list. From this list, we may extract values for parameter estimation, add fitted values, residuals
as well as the function called. Finally we set the class of the return object to class "lm2".

The print method for the "lm2" class can be defined resembling that of lm:

```r
print.lm2 <- function(x, ...) {
  cat("Call:
")
  print(x$call)
  cat("Coefficients:
")
  print(x$coefficients)
}
```

The results are displayed as follows (they almost have a “professional” looking, right?)

```r
> x = cbind(Const=1, Bwt=cats$Bwt)
> y = cats$Hw
> mod2 <- lm2(x, y)
> mod2
Call:
lm2.default(x = x, y = y)
Coefficients:
  Const   Bwt
-0.3566624 4.0340627
```

Note that we have used the standard names "coefficients", "fitted.values" and "residuals" for the elements of our class "lm2".

By following OOP, we get some generic functions for free, such as coef, fitted and resid, because their default methods are generally applicable and thus work for our class:

```r
> coef(mod1)
Const Bwt
-0.3566624 4.0340627
> fitted(mod1)
[1] 7.711463 7.711463 7.711463 8.114869 8.114869 8.114869 ...
> resid(mod1)
[1] -0.7114630 -0.3114630 1.7885370 -0.9148692 -0.8148692 ...
```

The summary function is generic and it provides result summaries concerning parameter estimates, standard deviation, t statistics, p-value, and so on and so forth. So, we define a summary method for our "lm2" class.
summary.lm2 <- function(object, ...) {
  se <- sqrt(diag(object$vcov))
  tval <- coef(object) / se
  TAB <- cbind(Estimate = coef(object),
               StdErr = se,
               t.value = tval,
               p.value = 2*pt(-abs(tval), df=object$df))
  res <- list(call=object$call,
               coefficients=TAB)
  class(res) <- "summary.lm2"
  res
}

The utility function `printCoefmat` can be used to print the coefficient matrix with appropriate rounding and some decoration:

```
print.summary.lm2 <- function(x, ...) {
  cat("Call:
")
  print(x$call)
  cat("\n")
  printCoefmat(x$coefficients, P.value=TRUE, has.Pvalue=TRUE)
}
```

So, the results is

```
> summary(mod2)
Call:
lm2.default(x = x, y = y)
Estimate Std. Err t.value p.value
Const -0.35666 0.69228 -0.5152 0.6072
Bwt 4.03406 0.25026 16.1194 <2e-16 ***
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1

S formulas refer to the unifying interface for selecting variables from a data frame for a plot, test or model. The most common formula is of type y ~ x1+x2+x3

The central object that is usually created first from a formula is the model.frame, a data frame containing only the variables appearing in the formula, together with an interpretation of the formula in the terms attribute. It tells us whether there is a response variable (always the first column of the model.frame), an intercept, and so on.
The model.frame is then used to build the design matrix for the model and get the response. The code here shows the simplest handling of formulas, which however is already sufficient for many applications.

```r
lm2.formula <- function(formula, data=list(), ...) {
  mf <- model.frame(formula=formula, data=data)
  x <- model.matrix(attr(mf, "terms"), data=mf)
  y <- model.response(mf)
  est <- lm2.default(x, y, ...)
  est$call <- match.call()
  est$formula <- formula
  est
}
```

The above function is an example for the most common exception to the rule that all methods should have the same arguments as the generic and in the same order. By convention formula methods have arguments formula and data rather than x and y.

The few lines of R code above give our model access to the wide variety of design matrices S formulas allow us to specify. E.g., to fit a model with main effects and an interaction term for body weight and sex we can use

```r
> summary(lm2(Hwt~Bwt*Sex, data=cats))
```

Call:
```
lm2.formula(formula = Hwt ~ Bwt * Sex, data = cats)
```

Estimate Std.Err t.value p.value
(Intercept) 2.98131 1.84284 1.6178 0.1079605
Bwt 2.63641 0.77590 3.3979 0.0008846 **
SexM -4.16540 2.06176 -2.0203 0.0452578 *
Bwt:SexM 1.67626 0.83733 2.0019 0.0472246 *

---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1

The last missing methods most statistical models in R are `plot` and `predict` methods. For the latter, a simple solution could be

```r
predict.lm2 <- function(object, newdata=NULL, ...) {
  if(is.null(newdata))
    y <- fitted(object)
  else{
    if(!is.null(object$formula)){
      ## model has been fitted using formula interface
      x <- model.matrix(object$formula, newdata)
    ```
} else{
    x <- newdata
}
y <- as.vector(x %*% coef(object))
return(y)
}

This works for models fitted with either the default method in which case newdata is assumed to be a matrix with the same columns as the original x matrix, or for models fitted using the formula method in which case newdata will be a data frame. Note that model.matrix() can also be used directly on a formula and a data frame rather than first creating a model.frame.