The package xplain is designed to help users interpret the results of their statistical analyses. It does so not in an abstract way as textbooks do. Textbooks do not help the user of a statistical method understand his findings *directly*. What does a result of 3.14 actually mean? This is often hard to answer with a textbook alone because the book may provide its own examples but cannot refer to the specifics of the user's case. However, as we all know, we understand things best when they are explained to us with reference to the actual problem we are working on. xplain is made to fill this gap that textbooks (and other learning materials) leave.

The basic idea behind xplain is simple: Package authors or other people interested in explaining statistics provide interpretation information for a statistical method (i.e. an R function) in the format of an XML file. With a simple syntax this interpretation information can reference the results of the user's call of the explained R function. At runtime, xplain then provides the user with textual interpretation that really relates to his/her case.

Providing xplain interpretation information can be interesting for:

* R package authors who implement a statistical method
* statisticians who develop statistical methods themselves
* college and university teachers who want to make their teaching content more accessible for their students
* everybody who enjoys teaching and explaining statistics and thinks he/she has something to contribute

xplain offers support for interpretation information in different languages and on different levels of difficulty.

This tutorial explains how xplain works and how to use it, both for providers of interpretation information and for users who work with this information. Fortunately, both, providing and using the interpretation information, is really easy. You can even provide interpretation information without any real R knowledge.

In the first step, we'll have a look at how the xplain XML files work.

**xplain XML files**

**Basic structure**

xplain XML files have a very simple structure. It looks like this:

<xml>

<xplain>

<package name = "stats">

<function name = "lm">

<title>Some title</title>

<text>Some text</text>

<result name = "df.residual">

<title>Some more info aboutthe degrees of freedom of the residuals</title>

<text>Some text related to the df.residual element of the result object of lm ()</text>

</result>

</function>

</package>

</xplain>

</xml>

This simple example assume we are providing interpretation information for the lm() function (for linear regression models) included in the stats package.

All valid xplain XML files must have at least one <xplain> element (but can have multiple ones). The xplain interpretation information is organized by *package* and *function*. Each <xplain> element can include multiple <package> elements and each <package> element multiple <function> elements. Each <function> element must be below a <package> element, <function> elements that are direct descendants of an <xplain> element are simply ignored.

The actual intrepretation information is included in the <title> and <text> elements. Titles are just plain text headers that do not include anything special like R code or references to a result object of the explained function. However, those special features can be used in the <text> elements. You will see in more detail how this is done in the next sections. <text> are the real workhorse fo the xplain, they carry most of tthe actual interpretation information.

Interpretation information can also be attached to (the elements of) the result object of the explained function using the ' tag.

xplain XML files are *not* case-sensitive, so work as you like and don't worry about capitalization.

**Defining language and level of difficulty**

xplain XML files are able to deal with multiple languages. The language of each XML element can be defined with the lang attribute, like in the follwing example:

<text lang="EN">This text is in English.</text>

<text lang="DE">Dieser Text ist in Deutsch.</text>

The value of the lang attribute is the ISO country code of the language.

Apart from the language you can also specify the level of diffulty of the explanation. This level is a number with higer values indicating higher levels of difficulty. Look at the following example:

<text level="1">This text is for beginners and is really easy to understand.</text>

<text level="2">This text goes more into the details and is for advanced learners.</text>

The numbers of the level do not have to start at 1 and do not need to be consecutive. The numbering is cumulative in the sense that if you call xplain with difficulty level 2 it will show you both the text entries for level 1 and for level 2.

Language and level information are inherited from elements further up in the XML hierarchy. To illustrate this point, let us look at the following example:

<xml>

<xplain>

<package name = "stats" lang="EN">

<function name = "lm">

<text>Text A) Some text</text>

<result name = "df.residual" level="1">

<text>Text B) Some text with diffculty 1.</text>

<text level="2">Text C) Some text with diffculty 2.</text>

<text lang="DE">Text D) Etwas Text auf Deutsch/ Some text in German.</text>

</result>

</function>

</package>

</xplain>

</xml>

In this example, text elements A, B and C are considered English texts because the <package> element to which they ultimately belong has the attribute lang="EN" (while the <function> element has no language information). They inherit the language of the hierarchically higher element. Text D, in contrast, is considered to be German. The reason is that the lang attribute of this text element overrides the innherited lang property.  
The argument is the same for the level property. The <result> element has a level attribute, so its descendants inherit this property. Text C, however, overrides this value and marks it as level 2.

If no language is specified when xplain is called then xplain tries to find the language of the user's work environment and uses this language; if it cannot be determined then it uses EN as a default.  
If no level is specified when xplain is called then all levels are shown.

**Referencing the return object of the explained function**

xplain makes it possible to reference the return object of the function that is being explained.  
Let us assume the user runs a simple regression with the lm function. Now have a look at the following extract from an xplain XML file:

<package name = "stats" lang="EN">

<function name = "lm">

<text>The residual degrees of freedom of your regression are: !%% @$df.residual %%!.</text>

</function>

</package>

In this example, we access the return object of the lm function, more specifically, the vector df.residual that holds the residual degrees of freedom of the estimated linear model. We do so by using the \*\*@\*\* symbol. This symbol is a placeholder for the return object of the lm function.

If, for example, the user works with the Anscombe dataset from the car package and runs the regression

library(car)

my.model <- lm(education ~ young + income + urban, data=Anscombe)

then the expression @$df.residual from the XML file would translate to my.model$df.residual.

From the XML extract above you see that the expression @$df.residual is embedded in an opening tag **!%%\*\* and a closing tag \*\*%%!**. These two tags are, as you will see later on, the delimiters that mark the beginning and the end of a section containing R code. Don't worry too much about that right here, we'll discuss this in more detail below.

Apart from using the @ symbol, there is yet another way to access the return object of the explained function: If you are working with a <result> element (which refers to a specific object *within* the return object of the explained function) then you can easily access this result object with the **##** symbol. Look at the following modification of the XML from above:

<package name = "stats" lang="EN">

<function name = "lm">

<result name="df.residual">

<text>The residual degrees of freedom of your regression are: !%% ## %%!.</text>

</result>

</function>

</package>

This XML extract is equivalent to the one above. Here, we refer to the current object within the return object of the lm function (and this current object is df.residual because we are in the XML element <result name="df.residual">), so ##=my.model$df.residual.

**Including R code**

With xplain it is easy to include R code into the interpretation texts.

Consider the following example:

<package name = "stats">

<function name = "lm" lang="EN">

<text>Your multiple R^2 is !%% format(summary(@)$r.squared, digits=2, nsmall=2) %%!. This

means that !%% format(summary(@)$r.squared\*100, digits=2, nsmall=2) %%! % of the total variation of

your dependent variable '!%% names(@$model)[1] %%!' can be explained by the independent variables

'!%% cat(attr(@$terms, "term.labels"), sep="\', \'") %%!'.

</text>

</function>

</package>

As you can see, we have a text element with several segments of R code included. Each segment is delimited by the opeing and closing tags !%% and %%!. xplain executes these R code segments and replaces the !%%...%%! portions of the text with the results from the code execution.

This way, you can even include more complex R structures like in the following example that works with an if construct:

<text level="1">!%% if(summary(@)$r.squared<=0.15) "Your R^2 is quite low." %%!</text>

Here, the message is only shown if the R2

of the regression does not exceed 0.15.

Two more things are noteworthy in this example:

The if condition conains a < which makes the whole line a syntactically incorrect XML expression. But don't worry, xplain takes care of this. So, if you don't mind that your XML won't pass any XML validator check, you can leave it as it is. However, if a syntactically incorrect XML causes you sleepless nights then you can convert the comparison operators <, > as well as the ampersand (&) with the usual HTML codings (so you get avalid XML at the cost of a slightly reduced readability): < becomes &lt;; > becomes &gt;; & becomes &amp;

Inside R code sections you don't need output function like print or cat (at least as long as we are talking about plain text outputs). In the example above, the expression evaluates (if the condition is met) to "Your R^2 is quite good." and this is what xplain displays on the screen

**Using placeholders with <define>**

When you use certain R code sections again and again at different places in your explanation texts, then it would be useful to define them once and then 'call' them just like functions from anywhere you like.  
For this purpose, xplain has the <define> tag. Look at the following example:

Instead of writing

<function name = "lm" lang="EN">

<title>Multiple R^2</title>

<text level="1">Your multiple R^2 is !%% format(summary(@)$r.squared, digits=2, nsmall=2) %%!.

This means that !%% format(summary(@)$r.squared\*100, digits=2, nsmall=2) %%! % of the total

variation of your dependent variable '!%% names(@$model)[1] %%!' can be explained by the

independent variables '!%% cat(attr(@$terms, "term.labels"), sep="\', \'") %%!'.</text>

<text level="1">!%% if(summary(@)$r.squared>=0.7) "Your R^2 is quite good." %%!</text>

</function>

which is complex and hard to read, you can start by defining some reusable code sections with define:

<function name = "lm" lang="EN">

<define name="r2\_2digs">!%% format(summary(@)$r.squared, digits=2, nsmall=2) %%!</define>

<define name="r2\_%2digs">!%% format(summary(@)$r.squared\*100, digits=2, nsmall=2) %%! %</define>

<define name="indep\_vars">'!%% paste(attr(@$terms, "term.labels"), collapse="\', \'") %%!'</define>

<define name="dep\_var">'!%% names(@$model)[1] %%!'</define>

<title>Multiple R^2</title>

<text level="1" lang="EN">Your multiple R^2 is !\*\* r2\_2digs \*\*!. This means that !\*\* r2\_%2digs \*\*!

of the total variation of your dependent variable !\*\* dep\_var \*\*! can be explained by the

independent variables !\*\* indep\_vars \*\*!.

</text>

</function>

With the name attribute of the define tag you assign a name to the R code expression. The expression itself is placed between the opening and closing tags of define and must be enclosed by the usual !%% and %%! markers if it is R code. (Of course, you can also put normal text into the <define>...</define> element, thereby effectively creating a kind of text modules; in fact, the apostrophes in the indep\_vars and the dep\_vars definitions above are normal text than has been inserted to make sure all the variables are enclosed in apostrophes when they are printed).

When you want to use the defined code segments in your <text>...</text> elements, you just call them by their name in the form !\*\* defined\_name \*\*!. By putting the name of the code segments into a !\*\*...\*\*! placeholder element, xplain recognizes it as a pre-defined code segment, looks up its definition and fills it into your text.

The definitions have no language or level information. If you want to create definitions for different languages or levels, just distinguish them by choosing appropriate names. <define>...</define> elements can be placed anywhere in the xplain XML document, but they must be defined before they are used for the first time.

The use of <define>...</define> elements is highly recommended as it makes your XML code more clear and readable.

**Iterating through a return object of the explained function**

Sometimes, you will want to iterate through a return object. To stay with our regression example from the last sections, imagine you wanted to give your user an explanation/interpretation on each of the regression coefficients. With xplain, this can be done very easly by using the foreach attribute of the <text>...</text> elements.

Look at the following extract from an xplain XML file (you can [download the full example](http://www.zuckarelli.de/xplain/example_lm_foreach.xml) XML file.

<function name = "lm">

<title>Interpreting the coefficients</title>

<result name = "coefficients">

<text foreach="rows"> Your coefficient '!%% names(@$coefficients)[$]%%!' is

!%% if(summary(@)$coefficients[$,1]>0) { "larger" } else { "smaller" } %%!

than zero. !%% if(names(@$coefficients)[$] != "(Intercept)")

cat("This means that the value of your dependent variable '", names(@$model)[1],

"' changes by ", format(summary(@)$coefficients[$,1], digit=2, nsmall=2),

" for any increase of 1 in your independent variable '", names(@$coefficients)[$],

"'.", sep=""); "\n" %%!

</text>

</result>

</function>

You will notice that we work with the result item coefficients. The text element has the XML attribute foreach="rows". This means, xplain will automatically go through the result item (coefficients) row by row, and will produce one output *for each row*. We cannot tell in advance how often the <text>...</text> element will be printed as we do not know the number of coefficients the user's regression will have. But we do know the user will get one output per coefficient.

The above XML produces the following output (don't worry too much about the call of xplain(), we will discuss later on in more detail how to work with the xplain() function):

library(car)

library(xplain)

xplain(call="lm(education ~ young + income + urban, data=Anscombe)",

xml="http://www.zuckarelli.de/xplain/example\_lm\_foreach.xml")

*##*

*## Call:*

*## lm(formula = education ~ young + income + urban, data = Anscombe)*

*##*

*## Coefficients:*

*## (Intercept) young income urban*

*## -286.83876 0.81734 0.08065 -0.10581*

*##*

*##*

*## Interpreting the coefficients*

*## -----------------------------*

*## Your coefficient '(Intercept)' is smaller than zero.*

*##*

*## Your coefficient 'young' is larger than zero. This means that the*

*## value of your dependent variable 'education' changes by 0.82 for*

*## any increase of 1 in your independent variable 'young'.*

*##*

*## Your coefficient 'income' is larger than zero. This means that the*

*## value of your dependent variable 'education' changes by 0.081 for*

*## any increase of 1 in your independent variable 'income'.*

*##*

*## Your coefficient 'urban' is smaller than zero. This means that the*

*## value of your dependent variable 'education' changes by -0.11 for*

*## any increase of 1 in your independent variable 'urban'.*

When you look at the XML code, you'll see that we use a mix of plain text and R code. As we have already done before, we access the explained function's return object with the @ placeholder. Nothing new so far.

However, you will also notice that we use the $ sign as an index for accessing elements of the coefficient matrix and the names() vector, for example in summary(@)$coefficients[$,1]. Here, we access the first column of the $th row of the coefficient matrix. $ is just a placeholder for the current row index, the same way as you would use the control variable in a for loop.

In this example, we run through the rows of the result item. Of course, we can do the same with columns, or with both together. In fact, xplain knows the following foreach modes:

* foreach="rows": runs through the rows of the return item (as in our example),
* foreach="columns": runs through the columns of the return item,
* foreach="rows, columns": runs first through the rows of the return item, and for each row through the columns,
* foreach="columns, rows": runs first through the columnsof the return item, and for each column through the rows,
* foreach="item": runs through the elements of the result item when the result item is a list.

If you would use foreach="rows, columns" to run both, the rows and the columns of our coefficients result item, then you could access the current element of the coefficients matrix with summary(@)$coefficients[$,$]. xplain automatically replaces the $ signs by the respective row and column indices.

**Syntax highlighting scheme for Notepad++**

Of course, like any XML file, xplain XML files can be edited with any text editor. However, using a text editor that supports syntax highlighting for XML is useful.

For the Notepad++ editor I have developed a syntax highlighting scheme that considers the specifics of xplain XML files, for example the definition of placeholders with <define> and their use in <text> tags as well as the use of R code with !%%and %%!. This scheme can be [downloaded](http://www.zuckarelli.de/xplain/xplain.xml).

To install it, please follow these instructions:

* download the highlighting scheme
* open Notepad++
* go to 'Language' | 'Define your language…'
* click 'Import'
* select the highlighting scheme XML file and click 'Open'

The highlighting scheme is readable both with a light and with a dark editor background. When using a dark editor background, don't forget to check 'Enable global background colour' in the 'Style configurator'.

**Function xplain()**

**Arguments and return values**

The function xplain() has the following arguments:

xplain(call, xml="", lang="", level = -1, filename="", sep="\n", title.char="-", before=TRUE, addfun="", addfun.args="",

addfun.title="")

Let us have a closer look at each of them:

* **call**: Function call to be explained/interpreted.
* **xml**: Path to the xplain XML file containing the interpretation/explanation information (optional). Can be either a local path or an URL. If no path is provided or the provided file does not exist then xplain() searches for a suitable XML file in various locations:
  + in the path of the package containing the function from which xplain() was called for a file of the name “package\_of\_the\_calling\_function.xml”;
  + in the same path for a file with the name “package\_of\_the\_explained\_function.xml” (the function given in the “call” argument);
  + in the path of the package containing the explained function for a file with the name “package\_of\_the\_explained\_function.xml”;
  + in the current working directory for a file with the name “package\_of\_the\_explained\_function.xml”; and
  + in the current working directory for a file with the name “explained\_function.xml”.
* **lang**: ISO country code of the language of the interpretations/explanations that shall be shown (optional). If none is specified, xplain() tries to determine the language of the user's current work environment. If that is not possible then English is taken as defualt value (same effect as lang="EN").
* **level**: Integer number indicating the complexity level of the interpretations/explanations that shall be shown (optional). level is cumulative: All interpretations/explanations with a level number up to the number provided are shown. Default: -1, i.e. all interpretations/explanations are shown.
* **filename**: File to write the xplain() output to (optional). If no filename is provided the xplain() output is shown in the console.
* **sep**: Separator used to separate the outputs from consecutive XML text elements (…) (optional). Default: “\n”.
* **title.char**: Character used for underlining titles (optional). Default: “-”.
* **before**: Indicates if the results of the call of the explained function shall be shown before the interpretations/explanations (optional). Default: TRUE, i.e. function output is shown before the interpretations/explanations.
* **addfun**: Names of additional functions that shall be called (e.g. summary()), without brackets (optional). It is assumed that these functions take the return object of the explained function as their first argument. Further arguments can be specified with addfun.args. Results of additional functions are shown right after the output of the explained function.
* **addfun.args**: Vector of arguments (beyond the return object of the explained function) for the additional functions (optional). Example: addfun.args = “trim = 0, na.rm = FALSE”. Argument must be of the same length as addfun; so addfun.args must be “” if the respective additional function does not take any additional arguments.
* **addfun.title**: Vector of titles that will be shown as headers to the outputs of the addional functions (optional).Argument must be of the same length as addfun; so addfun.args must be “” if the respective the output of the additional function shall have no title.

While xplain has quite some arguments, the only obligatory argument is call, the function call that you want to have explained. Usually, you will also specify the path to the corresponding xplain XML file. So, a very simple call of xplain() for a linear regression would look like this:

library(car)

library(xplain)

xplain(call="lm(education ~ young + income + urban, data=Anscombe)",

xml="http://www.zuckarelli.de/xplain/example\_lm.xml")

*##*

*## Call:*

*## lm(formula = education ~ young + income + urban, data = Anscombe)*

*##*

*## Coefficients:*

*## (Intercept) young income urban*

*## -286.83876 0.81734 0.08065 -0.10581*

*##*

*##*

*## Multiple R^2*

*## ------------*

*## Your multiple R^2 is 0.69. This means that 68.96 % of the total variation*

*## of your dependent variable 'education' can be explained by the*

*## independent variables 'young', 'income', 'urban'.*

If we wanted to incldue the result of another function, let's say summary() we would use the addfun argument. In this example we don't want to hand over any additional arguments (apart from our lm object, the result of the linear regression) to summary() but we want to have a nice header above the output. To achieve this, we could run the following call of xplain():

xplain(call="lm(education ~ young + income + urban, data=Anscombe)",

xml="http://www.zuckarelli.de/xplain/example\_lm.xml", addfun="summary", addfun.title="Here are some more details on the regression results:")

*##*

*## Call:*

*## lm(formula = education ~ young + income + urban, data = Anscombe)*

*##*

*## Coefficients:*

*## (Intercept) young income urban*

*## -286.83876 0.81734 0.08065 -0.10581*

*##*

*##*

*## Here are some more details on the regression results:*

*## -----------------------------------------------------*

*##*

*## Call:*

*## lm(formula = education ~ young + income + urban, data = Anscombe)*

*##*

*## Residuals:*

*## Min 1Q Median 3Q Max*

*## -60.240 -15.738 -1.156 15.883 51.380*

*##*

*## Coefficients:*

*## Estimate Std. Error t value Pr(>|t|)*

*## (Intercept) -2.868e+02 6.492e+01 -4.418 5.82e-05 \*\*\**

*## young 8.173e-01 1.598e-01 5.115 5.69e-06 \*\*\**

*## income 8.065e-02 9.299e-03 8.674 2.56e-11 \*\*\**

*## urban -1.058e-01 3.428e-02 -3.086 0.00339 \*\**

*## ---*

*## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1*

*##*

*## Residual standard error: 26.69 on 47 degrees of freedom*

*## Multiple R-squared: 0.6896, Adjusted R-squared: 0.6698*

*## F-statistic: 34.81 on 3 and 47 DF, p-value: 5.337e-12*

*##*

*##*

*## Multiple R^2*

*## ------------*

*## Your multiple R^2 is 0.69. This means that 68.96 % of the total variation*

*## of your dependent variable 'education' can be explained by the*

*## independent variables 'young', 'income', 'urban'.*

Since addfun, addfun.args and adfun.title are all vectors, we could include multiple functions here.

**Calling xplain()**

There are basically two main ways to work with xplain():

* You call xplain() directly, as we did in the previous section.
* You write a *wrapper function* that is similar to the explained function in terms of arguments but calls xplain() in the background.

Look at the following example of such a wrapper function:

lm.xplain <- function(formula, data, subset, weights, na.action,

method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,

singular.ok = TRUE, contrasts = NULL, offset, ...) {

call<-xplain.getcall("lm")

xplain(call, xml="http://www.zuckarelli.de/xplain/example\_lm.xml", level=1)

}

The function lm.xplain() has the same interface as the explained function lm(), so the user can call it exactly as he calls lm(). What the wrapper function does is that it creates a string holding the call to the explained function (as it is expected by xplain()'s call argument).

To do this very easily, the xplainpackage offers a special function, xplain.getcall(). This support function builds the call to the explained function based on a single argument, fun, which is of course "lm" in our case. The return value of xplain.getcall() is a string representing the function call that can be used directly as call argument for xplain().

This way, it very easy to write wrapper functions and to provide interpretation and explanation information for both, your own functions and functions from other packages.

**Function xplain.overview()**

If you want to get a quick overview of what information an xplain XML file provides, then analyze it with xplain.overview().

Its call looks like this:

xplain.overview(xml, show.text=FALSE, preserve.seq=FALSE)

The arguments of xplain.overview() are the following ones:

* **xml**: Filename of the xplain XML file to be analyzed.
* **show.text**: Indicates if the full interpretation/explanation texts shall be included in the result (optional). Default: FALSE.
* **preserve.seq**: Indicates if the overview results for the interpretation/explanation texts shall be shown in the same sequence as they appear in the XML file (optional). If FALSE, the results are sorted, e.g. by package, function, language and complexity level. Default: FALSE.

xplain.overview() returns a data frame that contains the summary information on the xplain XML file. It also shows the summary information in the console-

Let's look at an example:

xplain.overview(xml="http://www.zuckarelli.de/xplain/example\_lm.xml")

*## Package Function Type Language Level Result object Iteration Has R code*

*## 1 stats lm title default all FALSE*

*## 2 stats lm text default <=1 TRUE*

*## 3 stats lm text default <=1 TRUE*

*## Uses return obj.*

*## 1 FALSE*

*## 2 TRUE*

*## 3 TRUE*

The data frame returned by xplain.overview() has one row for each <title>...</title> and each <text>...</text> element in the XML file that has been analyzed.  
For each of these elements it provides some standard information like the package of the explained function, the function's name, the language and complexity level of the element, and also the element of the explained functions' result object which the XML element is related to. If the Result object column is empty this means that the respective title or text element is not related to any particular element of the return object of the explained function but directly belongs to the return object or the function as such.

Aport from this standard information, the data frame returned by xplain.overview() also shows

* if there is any iteration through the return object of the explained function (Interation),
* if the element includes any R code (Has R code), and if so,
* if the R code uses the return object (Uses return obj.).

If you use xplain.overview() with show.text=TRUE then the resulting data frame has an additional column containg the text of the <title>...</title> or <text>...</text> XML element.

Please note that (at least for the time being) the latter two pieces of information do not take into account any <define>...</define> placeholders.