Assesed_CW_1

Henry Bourne

2022-12-27

The "stattools" package

For my first piece of coursework I have created the **stattools** package. It is a package that contains various tools written from scratch (using only base r) to aid in the completion of the labs in the first statistical methods module (SM1). It contains tools to aid in the quick completion of the labs eg. one can iteratively perform cross validation with different regression methods very quickly using the in built regression methods and the CrossValidation class from *stattools*. I will start by describing the functionality of the stattools package before getting into what steps I have taken to make sure it is packaged correctly, all the functions work (on my computer aswell as different machines) and that it is versioned correctly.

The package itself can be viewed on my github "github.com/h-aze" under the "compass_yr1" repo in the "labs/stattools" subdirectory. Note that the stattools package was created to help with every lab in the SM1 module except the two labs on the "Simpletron". To create the package I used the help of the **devtools** package which among other things helps you create the package (by using the *create()* function), create documentation (using the *documentation()* function - uses roxygen2) and create and update the "NAMESPACE" and "DESCRIPTION" files. The NAMESPACE file can be geenrated by using *document()* and is created using the documentation, the "NAMESPACE" file is used to specify which functions and variables in your package should be exported and made available to the user and what your package may need to import from other packages. The DESCRIPTION file can be created using the *use_description()* function, the DESCRIPTION file gives various information about the package such as its title, description, authors and license. For the stattools package I have used the "MIT license", a good resource for information on what license to choose is "choosealicense.com". I will now start by describing the functionality of the package.

Functionality

I will begin by describing the functionality of the package, to learn more about any function type **?** to bring up the documentation relating to the function. First of all the package contains a host of functions included to help carry out regression. For example the *model_matrix()* function will create a model matrix when given data in the form of a data frame, matrix or vector. Once you have obtained the model matrix you can pass the result along with your predictor variables to a function such as *LLS()* which will carry out linear least squares regression and return the resulting parameters. Once could also use *LLS_R()* which carries out regularized linear least squares regression or *K_LLS_R()* which carries out kernel regularized linear least squares regression.

An added functionality of the stattools package is that these two regression methods involve hyperparameters, namely the value of lambda for regularization and also the kernel function for kernel regularized least squares, and by only passing the hyperparameters to these methods we will get back a function that will carry out regression with the set hyperparameters which we can then pass to other functions. This makes use of the functional programming aspect of the R language. We will look at some scenarios where this may come in handy later in this section.

Note that when carrying out kernel regularized linear least squares regression there are a number of kernel functions built in to stattools to pick from (or you can create and use your own!) such as *k_linear()* which is the linear kernel function, *k_poly()* which is the polynomial kernel function and *k_RBF()* which is the

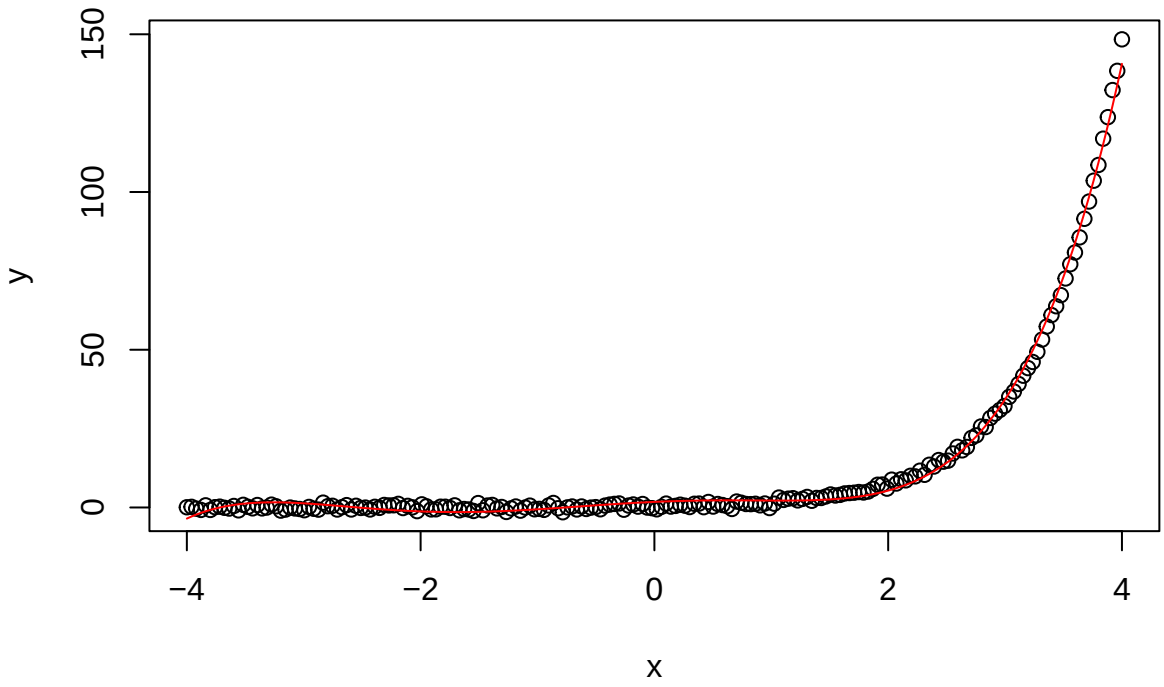
RBF kernel function. Again the kernel functions that involve hyperparameters can be called with only the hyperparameters to return a kernel function with the hyperparameters set.

We could also carry out non-linear regression using the *poly_feat_trans()* function which when given data and a degree will perform the polynomial feature transform of the given degree (again we can just supply the degree to return a polynomial feature transform function with set degree). Once we have done this we can then feed this as input to *LLS()* for example and get back a fitted non-linear model for the data. For example using all the functions we've just discussed we could do the following to fit a non-linear model to some data using a polynomial feature transform and linear least squares regression.

```
library(stattools)
# We start by creating a dataset to work on
x_start <- -4
x_end <- 4
x <- seq(x_start, x_end, length.out = 200)
e <- rnorm(200, mean = 0, sd = 0.64)
y <- exp(1.5 * x - 1) + e

# We define a polynomial feature transform of degree 4
ft <- poly_feat_trans(5)

# We find the model matrix of the feature transformed x
X <- model_matrix(ft(x))

# We now perform Linear Least Squares Regression
w_LS <- LLS(X, y)
w_LS
```

```
##            [,1]
## [1,]  1.7609513
## [2,]  1.6383735
## [3,] -1.2955547
## [4,] -0.3371817
## [5,]  0.3419273
## [6,]  0.0850681
```

```
# We now plot our data (black dots) against our fitted model (in red)
xs <- seq(-4, 4, length.out = 200)
ys <- model_matrix(ft(xs)) %*% w_LS
plot(x, y)
lines(xs, ys, col = "red")
```



We can also carry out classification using the stattools package. The *binlr_nll()* function can be used to find the negative log likelihood for a binary classification and using the *optim()* function from the stats package to minimize the negative log likelihood we can find the maximum likelihood estimator, we can then plug our estimator into the *prediction()* function along with the predictor variable for which we want a prediction to get the classification given by our MLE. For example:

```
# We create some data where if x > 5 then belongs to class 1 and otherwise belongs to
# class 0
x <- c(1:10)
y <- c(c(rep(0, 5), rep(1, 5)))

# We use the binlr_nll function in conjunction with optim to find our estimates of the
# parameters
results <- optim(par = c(0, 0), fn = binlr_nll, D = x, y = y)
results
```

```
## $par
## [1] -45.155623   8.189242
##
## $value
## [1] 0.03327149
##
## $counts
## function gradient
##      135       NA
##
## $convergence
```

```
## [1] 0
##
## $message
## NULL
```

```
# We can then use the prediction function to find our predictions
p1 <- round(prediction(5, results$par))
p1
```

```
## [1] 0
```

```
p2 <- round(prediction(6, results$par))
p2
```

```
## [1] 1
```

```
# We find that it predicts correctly that 6 (bigger than 5) belongs to class 1 and that 5
# belongs to class 0
```

The final piece of functionality of the stattools package we will talk about here is the CrossValidation class. The CrossValidation class is a class written using the reference class method of Object-Oriented Programming (OOP) in R, this class enables the calculation of the cross validation error in a flexible way where one can easily recalculate the cross validation error multiple times whilst quickly and easily being able to adjust how the cross validation is performed.

The need to recalculate the cross validation is seen many times in the SM1 labs, for example we may want to check which degree of polynomial feature transform gives the best performance and assess this using the cross validation as our measure of error. To help us do this we can initialize the class with the data, number of folds (k) and regression method we want to use, when we then use the method *getCv_error()* it will perform regression using our set regression method and return the k-fold cross validation error. In the scenario where we want to see for linear least squares regression which degree of polynomial feature transform gives the best model we would set *Regr_method=LLS* during initialization (or not set it in this case as LLS is the default) and then use the *setFeat_trans()* method with the argument *poly_feat_trans(b)* to use a polynomial feature transform of degree b, we could then repeatedly use this method in conjunction with *getCv_error()* method to calculate the cross validation error for different degrees in a quick and seamless manner. For example:

```
# We load in a dataset and create a CrossValidation object
data(longley)
D <- longley[-ncol(longley)]
y <- as.numeric(longley$Employed)
cv <- CrossValidation$new(D, y, k = as.integer(4))

# We set the regression method we would like to use, we use regularized least squares
# regression with lambda=1
cv$setRegr_method(LLS_R(1))

# We then calculate the cross validation error
err_1 <- cv$getCv_error()
err_1
```

```
## [1] 1.732158
```

```
# If we then wanted to calculate the cross validation error obtained on the same data but
# with a different regression method, lets say we wanted to investigate what the effect
# of a larger value of lambda would be, we simply:
cv$setRegr_method(LLS_R(2))
err_2 <- cv$getCv_error()
err_2
```

```
## [1] 1.305674
```

There are also lots of other methods on offer such as *setk()* which allows you to change the value of k, *setE_fun()* to change the error function that is used and *show()* to print out a list of useful information about the current state of the object.

Documentation

The stattools package is also fully documented. To create the documentation I used roxygen2 which automatically builds the documentation from "special comments" written in the code. Using roxygen2 to create the documentation is an example of literate programming as once can write source code where the same file can be both processed to produce documentation explaining what the programme does and a programme that can be executed. Literate programming is a favorable way of programming as it makes it much easier to keep code and documentation consistent as they are both produced from the same source. An example of a piece of code from the stattools package where we write a function and use special comments to create documentation is below:

```
#' Linear Least Squares (LLS) estimator
#'
#' Given model matrix, X, and targets, y, returns the LLS estimator
#' @param X, a matrix
#' @param y, a numeric or vector
#' @return w, a numeric or vector
#' @export
#' @examples
#' X <- model_matrix(c(1,2,3,4))
#' y <- c(1,4,9,16)
#' LLS(X,y)
LLS <- function(X, y) {
    w <- solve(t(X) %*% X) %*% t(X) %*% y
    w
}
```

The chunk of code where each line begins with a "#"' makes up the portion of the code that creates the documentation. Here what we do is first write the title which os what the function is, there is then a empty line followed by a description of what the function does. Next, it defines what the parameters of the function are using the "@param" command and what it returns using the "@return" command. We then sate that we want this function to be exported using the "@export" command, what this means is that this function will be available to the user when they install and use the stattools package. Finally, we define some examples of how to use this function using the "@example" command.

Below the documentation we then have the function itself which in this case is the LLS function we mentioned earlier. We can then process this piece of code to create both documentation and an executable programme, if the user wants to know more about the LLS function they can simply type *? LLS* which will build and display the documentation relating to the LLS function.

Testing

In addition to being fully documented the package also uses tests. We implement testing using the **testthat** package which we can use to create and run tests. Testing is important when creating a package as it allows us to check that the code is fully functional. Not only is this important when modifying and updating the package as integrated testing will make us aware of errors that changes we make produce in our code, but is also important in making sure that our code works across platforms on different machines. By writing a full suite of tests that cover all the functionality and scenarios we want the package to work in we can make sure that our package is always working and catch bugs early. To implement testing in our package we can use the **usethis::use_testthat()** which will set up testing. We can then use the **usethis::use_test("")**

command to create a test file with a given name, we can then write a test in this file using the **test_that()** function, eg.

```
test_that("feat_trans works when given a vector and a degree larger than 2", {
    x <- c(1, 2, 3, 4)
    b <- 3
    expect_equal(poly_feat_trans(b, x), cbind(x, x^2, x^3, deparse.level = 0))
})
```

Here we define a test for the *feat_trans()* function in the package, if **expect_equal()** evaluates to true when run then it will pass the test.

Github Actions and Versioning

I also implemented Github actions for this package, namely a R cmd check and a coverage test workflow. To implement a Github action you add a ".github" directory to the root of your repository and then a subdirectory labelled "workflows", in this subdirectory you then add ".yaml" files that define a specific workflow (or Github action). In my workflows folder I have two ".yaml" files one titled "R-CMD-check" (which runs a R cmd check) for the subdirectory in my repository where the stattools package is contained and another titled "test-coverage" that runs a coverage check on the subdirectory where my package is contained.

An R cmd check runs a whole series of tests to check for errors in the package, including checking examples in the documentation run, dependencies are properly defined in the "NAMESPACE" file and that the tests you have written all pass. To submit a package to CRAN it must be able to pass an R cmd check with the option *–cran* enabled.

To build build the package I also used the git versioning software in conjunction with Github. I can use git to keep track of changes I make to my code and use the *push* command to "push" local changes that I've committed to my Github repository. In this way I can create changes to the package and when I am happy with my changes upload them to Github, people who are then using the repository can then update their packages (by pulling from the repository).

I have set up the Github actions such that whenever I push to the repository it will carry out the workflows I defined in the two ".yaml" files, so when I push it will perform a R cmd check. I can then check the results of the Github action to check that with the updated code I haven't introduced any errors. This is an example of **continuous integration** which is a software development practice where a developer regularly integrate their code changes into a shared repository. The benefits of continuous integration are that the developer can release updates frequently and at less risk. The risk of the package not working is reduced two fold as we integrate checks on updating the repository such as the R cmd check to make sure that there aren't any errors and by having smaller updates it is likely there will be less errors and it will be easier to fix any errors that are identified.

I have also implemented a workflow which runs a coverage test on the stattools package on each push. A coverage test runs all the packages tests and checks which parts of the code are used during all the tests. This is usefull to help identify where more testing is needed and/or parts of code that aren't needed. It then uploads the results of the coverage test to "codecov.io" where you can view what code is and isn't being ran by the tests.