Statistical Computing Chapter_2

Henry Bourne

2022-10-17

Version Control

In the last chapter we mentioned version control, here we dive into exactly what version control is and how to use it. In the next section we will talk about creating packages in R, where using version control is very usefull. Git is a version control software, it allows us to keep track of the changes we've made and revert changes we've made. In combination with Github it also makes workin on code with multiple people more streamlined, it also allows for third-parties to suggest changes for your code.

Using Git

To initialise a git-repositry we go to the top-level directory we want to track and type: **git init**. If we want to add a file to be tracked we type: **git add** . If we want to commit changes we've made in the repository to our current version held by git we type: **git commit -m <message describing what we've changed>**. If we want to revert all the changes we've made back to the last commit we made we type: **git revert HEAD**. If we want to make a number of changes without affecting the default "master" branch we can create a new branch by typing: **git branch** . We can then start working on a given branch by typing **git checkout** . To merge a branch with master we first switch to master and then type: **git merge** , where it will then ask you to solve any conflicts. To ignore certain files we create a file named **.gitignore**, in the file we can write a file extension to ignore all files with that extension, eg. to ignore all jpg files we type ***.jpg**. Note that we can have multiple .gitignore files in different directories and we can negate a previous gitignore statement using: **!**.

Git with Github

Github is an online repository which we can use in conjunction with git to keep a backup of our code aswell as make our code more easily sharable and make working on others with code more streamlined. To clone a repository on Github onto your system we type: **git clone https://github.com//.git**. To check the Github online repository we type: **git remote -v**. To create a new project online from your local repository; first add the Github repository as the remote "origin" repository by typing: **git remote add origin https://github.com//.git**, we then push to it by typing: **git push -u origin master**. To fetch updates from the remote repository type: **git fetch** . To merge fetched updates into your local branch type: **git merge /**. To fetch and merge type: **git pull** . To push local commits to the remote repository type: **git push** .

We can also use git in conjunction with Github to contribute to others code. To create your own online copy of a repository you can click the **fork** button on Github. To push your fork to the main repository you can click **pull request**. To get changes from the upstream repositry (the respoitry you forked from) you can type: **git remote add upstream** . You can fetch and merge upstream changes into your branch by typing: **git fetch upstream** and then **git merge upstream/master**, after merging if you commit and then push a pull request made will automatically be updated (the pull request to the master repo on Github).

A note on licenses: when software is written in the UK, the author automatically receives copy-write protection. If you want others to be able to use your code you must include a license in your repo (in the top level

directory - the root), you do this by including an appropriately formatted (look up licenses online) LICENSE, LICENSE.txt or LICENSE.md file.

Packages in R

A suggested top-level folder structure for an R project is:

> |- R (r scripts containing functions)
>
> |- README.md (file describing what the package does)
>
> |- data/ (contains data, can have sub-directories: raw (for raw data), processed (for processed data))
>
> |- doc/ (documentation)
>
> |- output/ (results of applying R functions to the data, a sub-directory might hold figures or tables produced for visualization purposes)

And if using some C or C++ code then this would reside in a src/ folder, to load code from other files can use the source() command. To load an R project use the library() command.

A suggested top-level folder structure for an R package is:

> |- DESCRIPTION
>
> |- LICENSE.md
>
> |- NAMESPACE
>
> |- R
>
> |- man
>
> |- tests

The **DESCRIPTION** file contains information on the package name, title, authors, version amongst other things, the contents of the DESCRIPTION file used in the R package I will introduce later is written below as an example:

> Package: stattools
> Type: Package
> Title: Statistical methods
> Version: 0.1.0
> Authors@R: person("Henry", "Bourne", email = "hwbourne@gmail.com", role = c("aut", "cre"))
> Description: Package containing tools to perform various statistical methods covered in the statistical methods unit in the compass course (at The Univeristy of Bristol).
> License: MIT License
> Encoding: UTF-8
> LazyData: true
> Suggests: testthat (>= 3.0.0)
> Config/testthat/edition: 3

The **LICENSE.md** file contains the license, a good resource for this is "https://choosealicense.com/". The **NAMESPACE** file contains directives, each directive describes an R object and says whether it's exported from this package to be used by others, or it's imported from another package to be used locally. We don't tend to write these directives by hand, for example we can create a NAMESPACE file using roxygen2. The **R** directory contains all our R code for the package (in R we don't have sub-directories in this folder). The **man** directory contains documentation, we can use automatic documentation software to handle this directory for us. The **tests** directory contains R code for various tests for code in our R directory.

When building a package its essential you install the **devtools** package. Once the devtools package is installed you can do things such as add automatic documentation building in Rstudio by clicking: Build -> Configure Build Tools... -> Generate documentation with Roxygen, or you can run **devtools::document()**. In Rstudio you can also add skeleton documentation to each function.

Testing is crucial when programming and especially when creating packages that are going to be ran on different machines by different people. Install the **testthat** package and then run **usethis::use_test("")**, this will create a tests directory and populate it. You can run tests from the build pane in Rstudio or by typing **devtools::test()**.

We can check which code is and isn't being run by tests by installing the **covr** package and typing **covr::report()**.

Finally we can also test a package using "Github Actions". To tests defined in a package on Windows, macOS and Linux with the latest release of R, type: **usethis::use_github_action_check_standard()**. To run package tests on macOS and check which lines of source code have been run, then upload results to "Codecov.io", type: **usethis::use_github_action("test-coverage")**.

My package

To implement what I've learnt about packages I have written (and will continue to write over the course of the statistical computing module) a package containing functions to complete all the labs in the statistical methods module. The package can be found at "https://github.com/h-aze/compass_yr1/tree/master/labs/stattools", where you can observe the R package structure i mentioned earlier alongside the various files needed for the package including the DESCRIPTION, LICENSE.md files, documentation and R code. Over the course of the rest of the module I will be using methods learnt in the statistical computing lectures to implement solutions (from scratch, eg. not using existing r functions for creating model matrices etc.) for the labs in the statistical methods lectures (where possible). Most of this code I will write into the R package I have created so that this package can be used to quickly get solutions to all the statistical methods labs. For future portfolio chapters in statistical computing I will in most cases copy code from the package to show how I've implemented certain ideas that were discussed in the lectures related to the portfolio chapter. For the most part I will most likely include these copied functions in the context of a lab from the statistical methods course which utilizes those functions.

We do exactly this here where we utilize my package to perform the second statistical methods lab.

Lab 2: Normalized Regression and Decision making

We install my package containing all the functions needed to carry out the statistical methods in this lab (note in the spirit of reproducible programming we download the package from github (as opposed to using it locally) and we make sure the devtools package is installed).

```
library(devtools)
```

```
## Loading required package: usethis
```

```
devtools::install_github("h-aze/compass_yr1", subdir = "/labs/stattools")
```

```
## Skipping install of 'stattools' from a github remote, the SHA1 (36308210) has not changed since last
##   Use `force = TRUE` to force installation
```

Regularised Least Squares

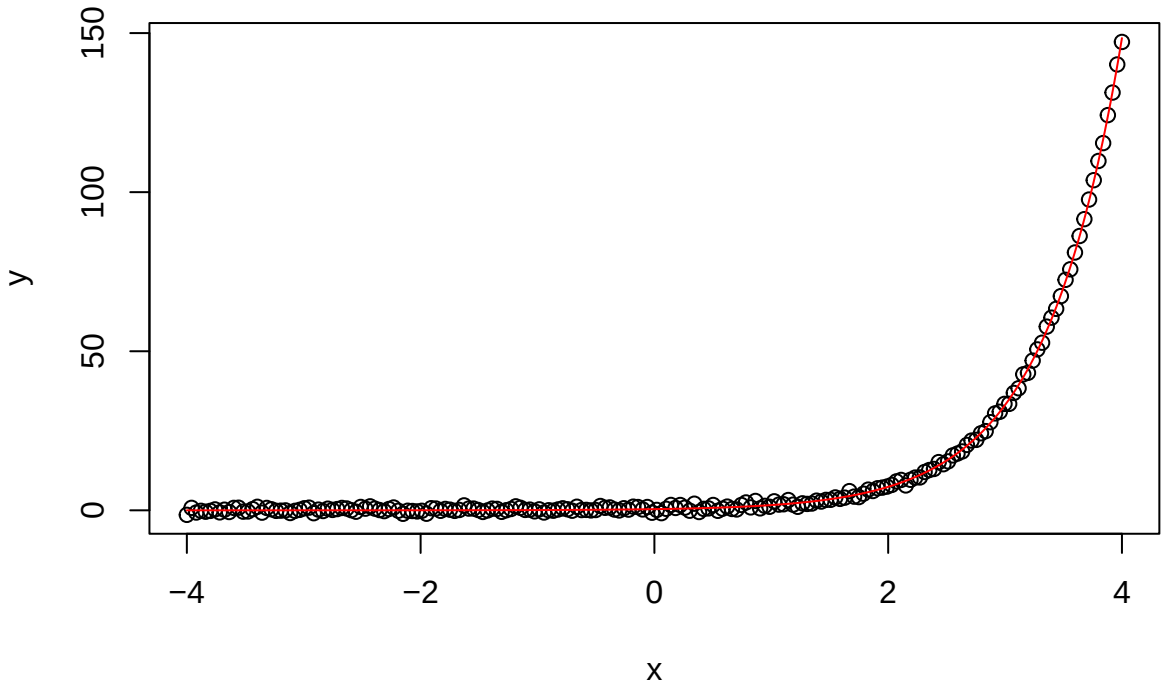
In this section of the homework we will be generating a dataset to experiment with regularized linear least squares, including an investigation into how our choice of regularization constant lamba affects our model and trying to find a lambda which minimizes the cross-validation error obtained by the model fitted.

We start by generating the dataset we will be using in this lab.

```
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 can visualize the dataset by plotting the data generated (black circles) and the underlying function (in red).

```
plot(x, y)
par(new = TRUE)
eq = function(x) {
    exp(1.5 * x - 1)
}
lines(seq(x_start, x_end, length.out = 500), eq(seq(x_start,
    x_end, length.out = 500)), type = "l", col = "red")
```



We now begin our model fitting by first performing a feature transform on x, so we can fit a more complex non-linear model but still use (regularised) linear least squares (LS-R).

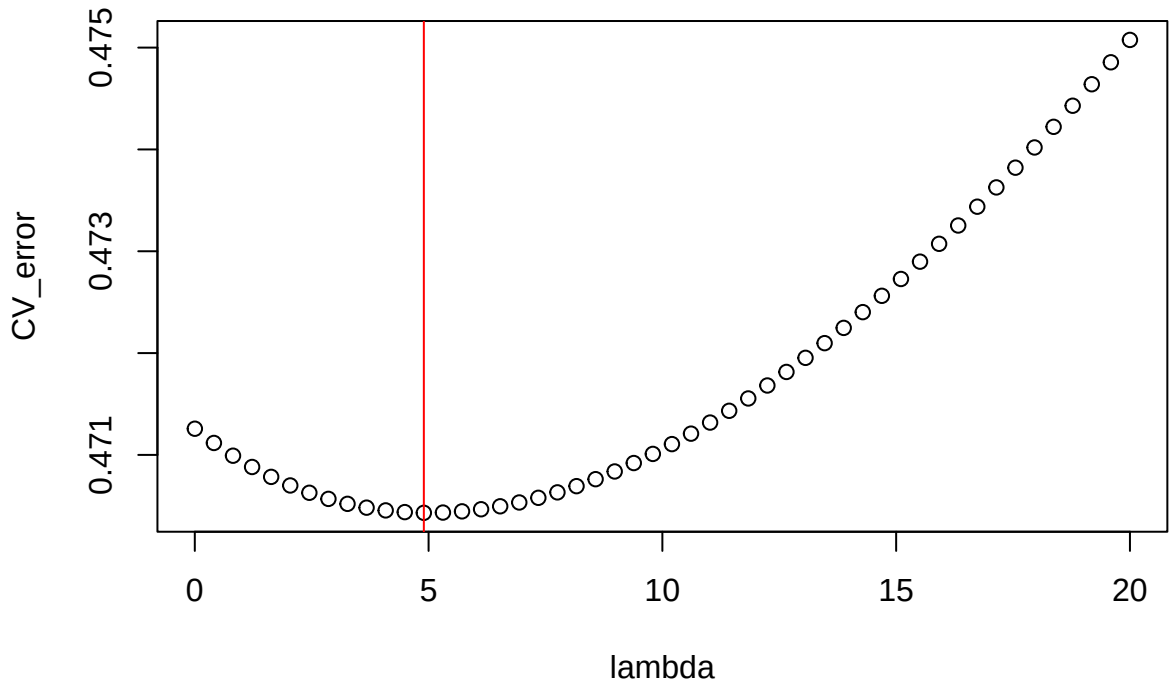
```
x_ft <- stattools::feat_trans(x, 7)
```

Next we select a sequence of lambda values in a set range and using the "regr_cross_val" function fit a model using LS-R and compute the Cross-Validation (CV) error for each value of lambda. Here we perform CV using k=200 (leave-one-out validation) and use as our error function the euclidean norm.

```
lambda <- seq(10^-3, 20, length.out = 50)
CV_error <- c()
for (i in 1:length(lambda)) {
    CV_error[i] <- stattools::regr_cross_val(x_ft, y, RM = stattools::LLS_R,
        k = 200, lambda[i])
}
```

We now plot the CV errors obtained for each value of lambda.

```
plot(lambda, CV_error)
err_min <- which.min(CV_error)
abline(v = lambda[err_min], col = "red")
```



From the plot we can see that as lambda increases the CV error initially decreases, after the red line we then observe the CV error begin to increase. What is happening here is that as lambda initially increases it is scaling the regularization term to make it larger, this in turn puts more pressure on LS-R to keep the parameters of its solution small in order to keep the regularization term smaller whilst still minimizing the loss obtained by the model on the data. This leads to a model that is more general as it can't have wildly large parameters which leads to overfitting of the data and hence better CV performance on the test data. However, as lambda gets even larger the model has to prioritize minimizing the regularization term to a larger and larger degree, leading to decreased performance of the model on the data as it can only fit an increasingly less complex model.

We now repeat the above but over a smaller range to find an optimal value of lambda.

```
lambda <- seq(1, 4, length.out = 1000)
CV_error <- c()
for (i in 1:length(lambda)) {
```

```
    CV_error[i] <- stattools::regr_cross_val(x_ft, y, RM = stattools::LLS_R,
        k = 200, lambda[i])
}
lambda.hat <- lambda[which.min(CV_error)]
lambda.hat
```

```
## [1] 4
```

Our optimal value of lambda is printed above.

Probabalistic model

Now we would like to find the predictive probability distribution,$p(\hat{y}|\mathbf{x})$, which we will do using the "marginalization trick".

Note that we can write $P(\hat{y}|x, Dw) = \int P(\hat{y}|x, w) \cdot P(w|D)dw$ and we know that this is distributed normally with mean $f(x; w_{LS-R})$. Hence, we can directly calculate the mean of $\hat{y}$ using the following function:

```
# We get the feature transformed model matrix
X <- stattools::model_matrix(x_ft)
# We calculate the parameters using LS-R and our data (with
# lambda set to the optimal value we found in previous
# section)
w.LS_R <- stattools::LLS_R(X, y, lambda.hat)

mu.predictor <- function(x) {
    X <- stattools::model_matrix(stattools::feat_trans(x, 7))
    X %*% w.LS_R
}
```

Now all we have to do is estimate the sd, we can infer $\hat{\sigma}$ by calculating $\hat{\sigma} = \frac{RSS}{n-1}$.

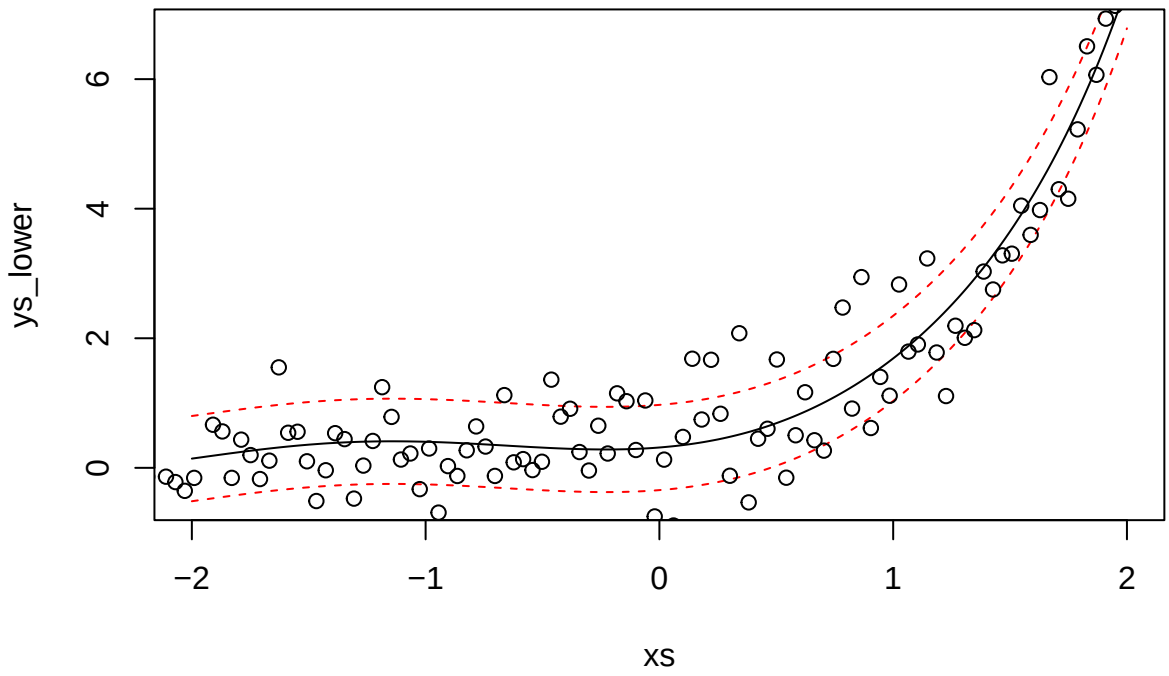
```
X <- stattools::model_matrix(stattools::feat_trans(x, 7))
y_hat <- X %*% w.LS_R
sigma.predictor <- sqrt((t(y - y_hat) %*% (y - y_hat) * (1/(length(y) -
    1)))[1])
sigma.predictor
```

```
## [1] 0.6579283
```

Now we have the predictive probability distribution we can plot the expected value of $\hat{y}$ given $\mathbf{x}$, along with its sd.

```
xs <- seq(-2, 2, length.out = 200)
ys <- mu.predictor(xs)
ys_lower <- ys - sigma.predictor
ys_upper <- ys + sigma.predictor

plot(xs, ys_lower, type = "l", col = "red", lty = 2)
lines(xs, ys, type = "l")
lines(xs, ys_upper, type = "l", col = "red", lty = 2)
points(x, y)
```



Above you can see plotted the expected value of x (in black) along with the tube of values within one standard deviation (red dotted lines), we also plot the sampled points used to train the model within the range of x for this plot (we plot for a smaller range of x to better see the tube).

Lastly, we will calculate the percentage of our samples found within one standard deviation of our expected value of $\hat{y}$.

```
y_lower <- mu.predictor(x) - sigma.predictor
y_upper <- mu.predictor(x) + sigma.predictor
cov <- 0
for (i in 1:length(y)) {
    if (y[i] >= y_lower[i] & y[i] <= y_upper[i]) {
        cov <- cov + 1
    }
}
(cov/length(y)) * 100
```

```
## [1] 67.5
```

So 66.5% of the time our samples are within one standard deviation of our expected value.

All of this was done using the help of functions (built from scratch) in my package: "stattools".