This post deals with statistical models. In the text below, I explain what hyper-parameters are, and as an example I run a ridge regression using the {glmnet} package. The book is still being written, so  
comments are more than welcome!

**Hyper-parameters**

Hyper-parameters are parameters of the model that cannot be directly learned from the data.  
A linear regression does not have any hyper-parameters, but a random forest for instance has several.  
You might have heard of ridge regression, lasso and elasticnet. These are  
extensions to linear models that avoid over-fitting by penalizing *large* models. These  
extensions of the linear regression have hyper-parameters that the practitioner has to tune. There  
are several ways one can tune these parameters, for example, by doing a grid-search, or a random  
search over the grid or using more elaborate methods. To introduce hyper-parameters, let’s get  
to know ridge regression, also called Tikhonov regularization.

**Ridge regression**

Ridge regression is used when the data you are working with has a lot of explanatory variables,  
or when there is a risk that a simple linear regression might overfit to the training data, because,  
for example, your explanatory variables are collinear.  
If you are training a linear model and then you notice that it generalizes very badly to new,  
unseen data, it is very likely that the linear model you trained overfits the data.  
In this case, ridge regression might prove useful. The way ridge regression works might seem  
counter-intuititive; it boils down to fitting a *worse* model to the training data, but in return,  
this worse model will generalize better to new data.

The closed form solution of the ordinary least squares estimator is defined as:

\[  
\widehat{\beta} = (X'X)^{-1}X'Y  
\]

where \(X\) is the design matrix (the matrix made up of the explanatory variables) and \(Y\) is the  
dependent variable. For ridge regression, this closed form solution changes a little bit:

\[  
\widehat{\beta} = (X'X + \lambda I\_p)^{-1}X'Y  
\]

where \(\lambda \in \mathbb{R}\) is an hyper-parameter and \(I\_p\) is the identity matrix of dimension \(p\)  
(\(p\) is the number of explanatory variables).  
This formula above is the closed form solution to the following optimisation program:

\[  
\sum\_{i=1}^n \left(y\_i – \sum\_{j=1}^px\_{ij}\beta\_j\right)^2  
\]

such that:

\[  
\sum\_{j=1}^p(\beta\_j)^2 < c  
\]

for any strictly positive \(c\).

The glmnet() function from the {glmnet} package can be used for ridge regression, by setting  
the alpha argument to 0 (setting it to 1 would do LASSO, and setting it to a number between  
0 and 1 would do elasticnet). But in order to compare linear regression and ridge regression,  
let me first divide the data into a training set and a testing set. I will be using the Housing  
data from the {Ecdat} package:

library(tidyverse)

library(Ecdat)

library(glmnet)

index <- 1:nrow(Housing)

set.seed(12345)

train\_index <- sample(index, round(0.90\*nrow(Housing)), replace = FALSE)

test\_index <- setdiff(index, train\_index)

train\_x <- Housing[train\_index, ] %>%

select(-price)

train\_y <- Housing[train\_index, ] %>%

pull(price)

test\_x <- Housing[test\_index, ] %>%

select(-price)

test\_y <- Housing[test\_index, ] %>%

pull(price)

I do the train/test split this way, because glmnet() requires a design matrix as input, and not  
a formula. Design matrices can be created using the model.matrix() function:

train\_matrix <- model.matrix(train\_y ~ ., data = train\_x)

test\_matrix <- model.matrix(test\_y ~ ., data = test\_x)

To run an unpenalized linear regression, we can set the penalty to 0:

model\_lm\_ridge <- glmnet(y = train\_y, x = train\_matrix, alpha = 0, lambda = 0)

The model above provides the same result as a linear regression. Let’s compare the coefficients between the two:

coef(model\_lm\_ridge)

## 13 x 1 sparse Matrix of class "dgCMatrix"

## s0

## (Intercept) -3247.030393

## (Intercept) .

## lotsize 3.520283

## bedrooms 1745.211187

## bathrms 14337.551325

## stories 6736.679470

## drivewayyes 5687.132236

## recroomyes 5701.831289

## fullbaseyes 5708.978557

## gashwyes 12508.524241

## aircoyes 12592.435621

## garagepl 4438.918373

## prefareayes 9085.172469

and now the coefficients of the linear regression (because I provide a design matrix, I have to use  
lm.fit() instead of lm() which requires a formula, not a matrix.)

coef(lm.fit(x = train\_matrix, y = train\_y))

## (Intercept) lotsize bedrooms bathrms stories

## -3245.146665 3.520357 1744.983863 14336.336858 6737.000410

## drivewayyes recroomyes fullbaseyes gashwyes aircoyes

## 5686.394123 5700.210775 5709.493884 12509.005265 12592.367268

## garagepl prefareayes

## 4439.029607 9085.409155

as you can see, the coefficients are the same. Let’s compute the RMSE for the unpenalized linear  
regression:

preds\_lm <- predict(model\_lm\_ridge, test\_matrix)

rmse\_lm <- sqrt(mean(preds\_lm - test\_y)^2)

The RMSE for the linear unpenalized regression is equal to 2077.4197343.

Let’s now run a ridge regression, with lambda equal to 100, and see if the RMSE is smaller:

model\_ridge <- glmnet(y = train\_y, x = train\_matrix, alpha = 0, lambda = 100)

and let’s compute the RMSE again:

preds <- predict(model\_ridge, test\_matrix)

rmse <- sqrt(mean(preds - test\_y)^2)

The RMSE for the linear penalized regression is equal to 2072.6117757, which is smaller than before.  
But which value of lambda gives smallest RMSE? To find out, one must run model over a grid of  
lambda values and pick the model with lowest RMSE. This procedure is available in the cv.glmnet()  
function, which picks the best value for lambda:

best\_model <- cv.glmnet(train\_matrix, train\_y)

# lambda that minimises the MSE

best\_model$lambda.min

## [1] 66.07936

According to cv.glmnet() the best value for lambda is 66.0793576. In the  
next section, we will implement cross validation ourselves, in order to find the hyper-parameters  
of a random forest.