Functional and Object Oriented Programming

Rachel

2022-11-01

```
library(ggplot2)
library(tidyr)
```

To illustrate the difference between these two programming frameworks, we will consider a classic linear regression problem, where we have:

- A design matrix `X` containing the explanatory variables
- A vector of responses `y`

and we want to find:

- A vector of predictors `hat.y`
- A set of confidence intervals for predictors
- A plot showing all of this information

In this portfolio, I will tackle this problem from both perspectives, after a brief description of the paradigms.

Functional Programming

The basics:

There are a few features of functional programming that we introduce before moving on to the implementation via regression:

- First Class Functions: We say a programmming language has first class functions if functions are able to be passed to other functions or stored in data structures as any other variable would be (note: R does have this feature)
- Pure Functions: These are functions that do not affect the global environment. We aim to have as few side-effects on the global environment as possible but we will have to use some impure functions the majority of the time (e.g. reading/writing from/to the disk). -Closures: This feature is such that if a free variable is defined inside a function which was created by another function, it does not become part of the global environment. It is however, captured by the function itself.
- Lazy Evaluation: This feature makes it so any argument to a function is only evaluated if they are accessed. This means CPU power is not wasted computing arguments more than necessary, but it can cause some bugs (e.g. if a global variable has changed since the function was defined). This can be combatted using the `force()` function.

Linear Regression Example

To find the fitted values, we first need to find an estimate for the parameters of the model. This is usually done via least squares and we could write a function to obtain these values which we can then pass to the predictor function. However this would modify the global environment by creating a new variable in which we store the parameter estimates. Instead what we can do is write a function `get.predictorfun()` which returns a predictor function. The parameter estimates are then captured in the function returned by `get.predictorfun()` but would not be a part of the global program state.

```
get.predictorfun <- function(x, y) {
  X <- cbind(1, x)
  beta.hat <- as.vector(solve(t(X)%*%X, t(X)%*%y))
  predictor <- function(x.new) as.vector( cbind(1, x.new) %*% beta.hat)
}
```

We can see that we have created a pure function while making use of the closure and first-class properties of the language.

Before we implement the function, we need to create some training data:

```
n <- 1000
x.train <- matrix(rnorm(n), nrow = n)
beta <- c(5,-3)

y.train<- cbind(1,x.train) %*% beta + rnorm(n)
```

We can now create the predictor function. As we can see, the function itself can be accessed, but not the free variable `beta.hat`

```
predictor <-  get.predictorfun(x.train,y.train)
predictor
```

```
## function(x.new) as.vector( cbind(1, x.new) %*% beta.hat)
## <environment: 0x55e5c4294f98>
```

```
exists('beta.hat')
```

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

Now that we have a function for the fitted values, we can apply this to a dataset which was not used in fitting the predictor function.

```
x.test <- rnorm(100)
y.test <- cbind(1, x.test) %*% beta + rnorm(100)
fitted.val <- predictor(x.test)
```

Often a single prediction is not very useful, so we could instead create a function which returns a confidence interval function

```
get.CI.fun <- function(x,y,alpha){
  n <- nrow(x)

  prediction <- get.predictorfun(x,y)
  fitted <- prediction(x)

  sigma.sqr.hat <- sum((y-fitted)^2) / (n -2)

  z <- - qnorm(alpha/2)

  CI <- function(x.new){
    cbind(predictor(x.new) - z*sigma.sqr.hat, predictor(x.new) + z*sigma.sqr.hat)}
}
```

Again we use this on the training function to get a 95% confidence interval function for our model:

```
CI.fun <- get.CI.fun(x.train, y.train,alpha = 0.05)

CI.test <- CI.fun(x.test)
```

Finally, we might want to visualise all of this, so we can put it all together by creating a function which returns a plotting function:

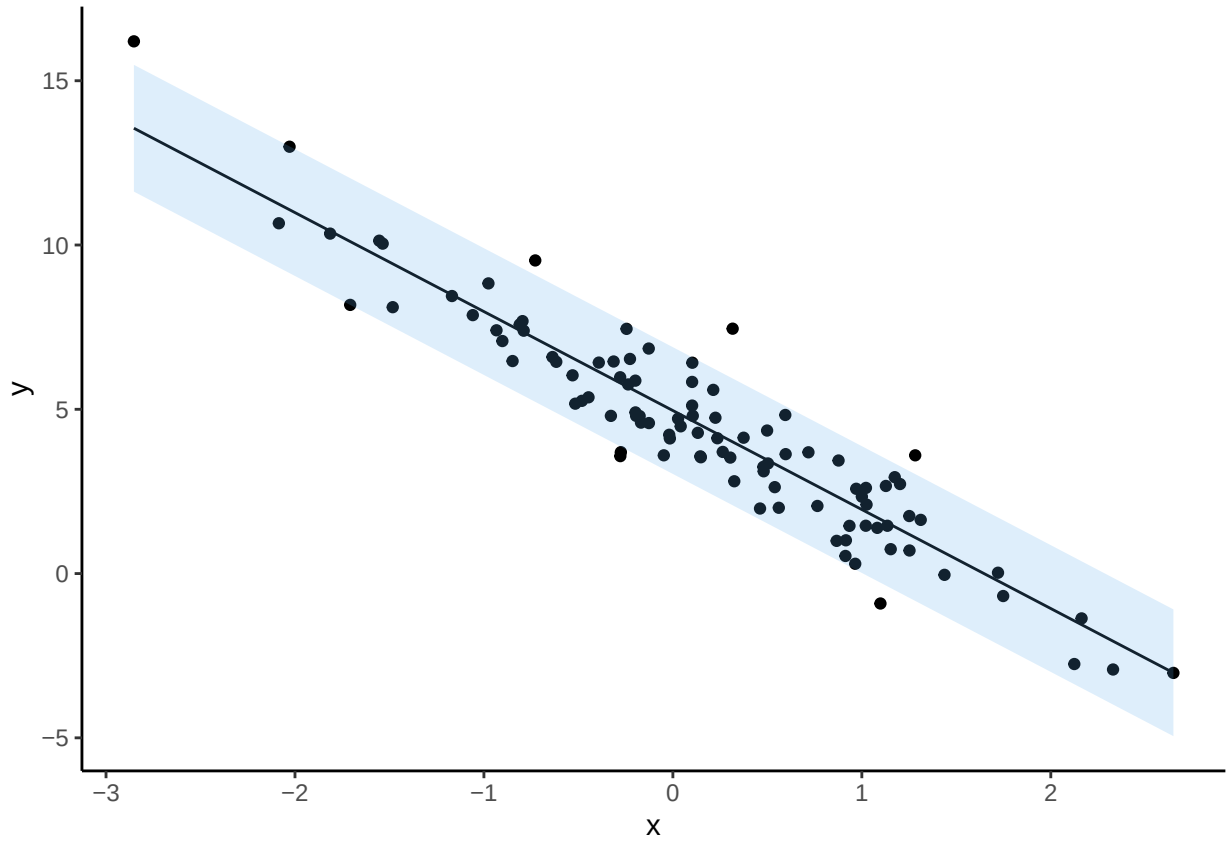
```
get.plot.fun <- function(x,y,alpha){
  predictor <- get.predictorfun(x,y)
  CI.fun <- get.CI.fun(x,y,alpha)

  plot.fun <- function(x.new, y.new){
    CI <- CI.fun(x.new)
    data <- tibble(x = x.new, y = y.new, fitted = predictor(x.new),
                   CI_low = CI[,1], CI_high = CI[,2])

    return(
      ggplot(data = data, aes(x = x, y = y)) +
      geom_point() +
      geom_line(aes(x = x, y = fitted)) +
      geom_ribbon(aes(ymin = CI_low, ymax = CI_high, x = x),
                  fill = "steelblue2", alpha = 0.2, inherit.aes = FALSE)
      +theme_classic()
    )
  }
}
```

Getting the plotting function for our training data and applying to the testing data we get

```
plot.fun <- get.plot.fun(x.train, y.train, 0.05)
plot.fun(x.test,y.test)
```



Object-Oriented Programming (OOP)

As opposed to functional programming, in this framework, we change the global environment by changing the state of objects via methods, which are a type of function. - Objects are defined as members of a class - Classes are defined by their fields, which detail the properties of the objects, and methods, which describe the behaviour of the objects - We can also define relations between classes, which allow for inheritance of properties

For this portofolio, I will only consider the S4 implementation of OOP, available in the `methods` package:

```
library(methods)
```

Linear Regression Example

For this section, we discard the idea of testing and training sets and consider only one set of regressors and responses for simplicity.

We can now define a new class for our regression object:

```
setClass("linear.regression",
   slots = c(
     response  = "numeric",
     regressor = "numeric",
     fitted  = "numeric",
```

```
      ci = "matrix",
      est.var = "numeric"
      )
)
```

Then we write a method to initialise an object of class `linear.regression`. In this intialisation, we can compute the `fitted`, `ci` and `sigma.hat` fields from the provided `response` and `regression`. Here the `ci` field is set to be a the 95% confidence interval by default.

```
setMethod("initialize", "linear.regression",
  function(.Object, response, regressor, alpha = 0.05) {
    .Object@response  <- response
    .Object@regressor <- regressor

    n <- length(response)
    X <- cbind(1,regressor)

    beta <- solve(t(X) %*% X) %*% t(X) %*% response
    fitted <-X %*% beta
    .Object@fitted <- as.numeric(fitted)

    est.var <- sum((y-fitted)^2) / (n -2)
    .Object@est.var <- est.var

    z <- - qnorm(alpha/2)
    ci <- cbind(fitted- z*est.var, fitted + z*est.var)

    .Object@ci <- ci

    return(.Object)
  }
)
```

We can now initialise a `linear.regression` object

```
n <- 1000
x<- rnorm(n)
beta <- c(5,-3)

y <- as.vector(cbind(1, x) %*% beta + rnorm(n))

model <- new("linear.regression", response = y, regressor = x)
```

To view our object, we need to define a print method:

```
setMethod("print", "linear.regression",
  function(x) {
    cat("head(x) =", head(x@regressor), "\n")
    cat("head(y) =", head(x@response), "\n\n")
    cat("Estimated regression: E[y|x] = b0 + b1 * x = ", head(x@fitted), "\n")
    cat("Confidence Interval: \n", "Lower bound= ", head(x@ci[,1]),
        "\n Upper bound= ", head(x@ci[,2]),"\n")
```

```
    cat("Estimated variance: ", x@est.var, "\n")
  }
)

print(model)
```

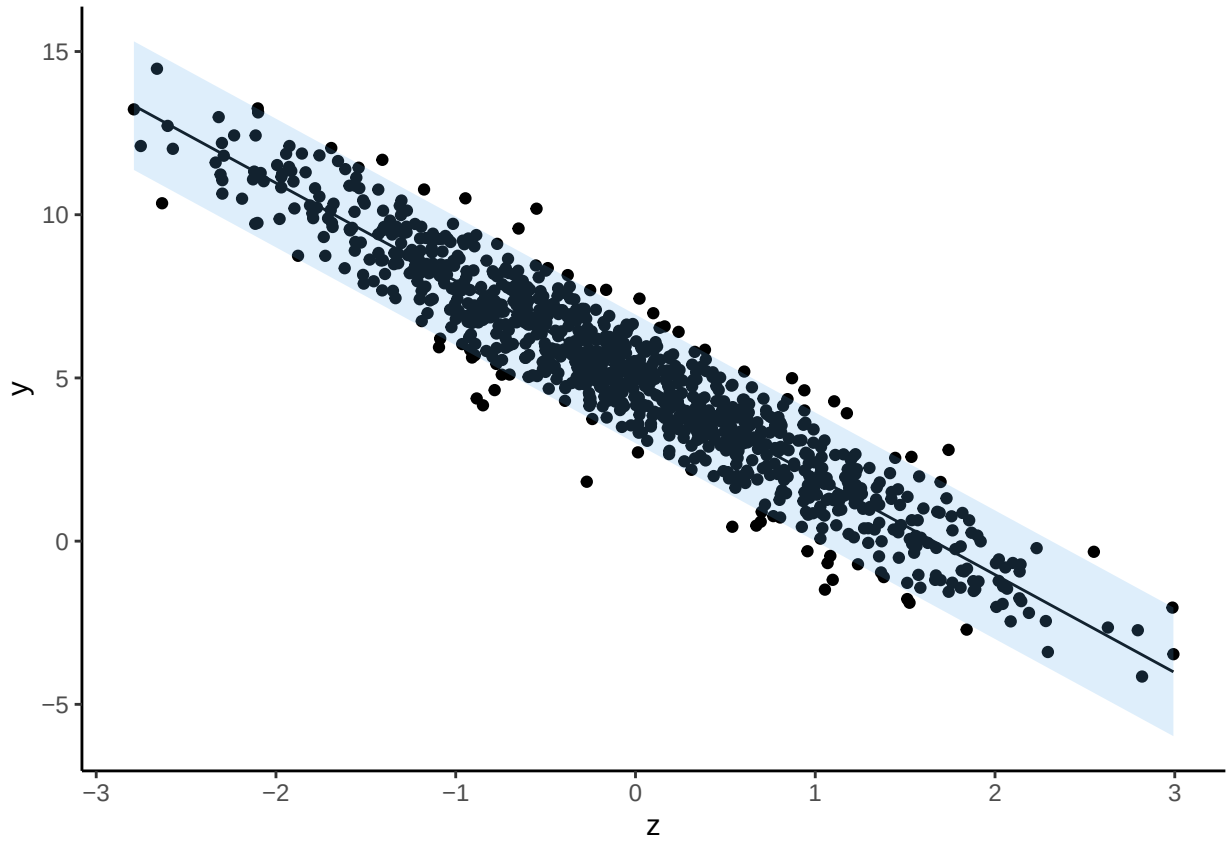
```
## head(x) = 0.3203785 0.5013752 0.435149 -0.2136702 0.1049778 -0.1067874
## head(y) = 3.505428 2.994531 1.992807 5.822393 3.964553 5.457643
##
## Estimated regression: E[y|x] = b0 + b1 * x =  4.010707 3.467767 3.666427 5.612707 4.656851 5.292088
## Confidence Interval:
##  Lower bound=  2.040695 1.497755 1.696415 3.642695 2.686838 3.322076
##  Upper bound=  5.980719 5.437779 5.636439 7.582719 6.626863 7.2621
## Estimated variance:  1.005127
```

Finally we can construct a plot method that will produce a similar plot to that of the functional programming section:

```
setMethod("plot", "linear.regression",
  function(x){
    data <- tibble(z= x@regressor, y = x@response, fitted = x@fitted,
                   CI_low = x@ci[,1], CI_high = x@ci[,2])
    ggplot(data ,aes(x = z, y = y)) +
      geom_point() +
      geom_line(aes(x = z, y = fitted)) +
      geom_ribbon(aes(ymin = CI_low, ymax = CI_high, x = z),
                  fill = "steelblue2", alpha = 0.2, inherit.aes = FALSE) +
      theme_classic()
  }
)
```

Then using the `plot` method on `model` we get

```
plot(model)
```


15
10
5
0
−5
y
−3
−2
−1
0
1
2
3
z