Statistical computing MATH10093
Computer lab 1
Solutions

Finn Lindgren

15/1/2020

Summary

In this lab session you'll gradually develop R code for estimating multiple linear models, and plotting the results. You will not hand in anything, but you should keep your code script file for later use.

1. Start `RStudio` (version 3.6.1).
2. Create a *project* for your course files:
   `File→New Project→New Directory→New Project`
   Next time you want to work on the course files, use `File→Open Project` instead. This will automatically open the script files you had open when you last worked on the course.
   NOTE: It's *highly* recommended to avoid spaces in all directory and file names! Including spaces may result in unexpected problems.
3. Go to the Learn page and open the `L01.pdf` lecture notes so you can refer to them easily during the lab. Read through the pages on formulas and functions before you continue with lab.
   (a) Create a new R script file:
   `File→New File→R Script`
   (b) Save the new R script file:
   `File→Save`, or
   Press `Ctrl+S`,
   and choose a descriptive filename (e.g. `lab_1_code.R`)
   During the lab, remember to save the script file regularly, to avoid losing any work.

   Some useful menu options, buttons, and keyboard shortcuts:

   - Configuring code syntax diagnostics:
     `Tools→Global Options→Code→Diagnostics`
     This allows you to turn on margin notes that automatically alert you to potential code problems.

- Run the current line of code (or a selected section of code) and step to the next line:
  Press the `Run` button in the top right corner of the editor window, or
  Press `Ctrl+Enter`
- Run all the code in the current script file:
  Press the `Source` button in the top right corner of the editor window, or
  Press `Ctrl+Shift+S`, or
  Press `Ctrl+Shift+Enter`
  Note: the first two options disable the normal automatic value printing; you'll only see the ouput that is explicitly fed into `print()`. The third option displays both the code and the results in the Console window.
- Optionally, follow the instructions from the lecture notes to install the `styler` package. This will enable options to reformat your script code to (usually) more readable code, by pressing the `Addins` button below the main program menu and choosing e.g. `Style active file`.

4. We will start with the simple linear model

$$y_i = \beta_0 + z_i\beta_z + e_i,$$

where y_i are observed values, z_i are observed values that we believe have a linear relationship with y_i, e_i are observation noise components, and β_0 and β_z are model parameters.

First, generate synthetic data to use when developing the code for estimating models of this type. Enter the following code into your script file and run it with e.g. `Ctrl+Enter`:

```
z <- rep(1:10, times = 10)
data <- data.frame(z = z, y = 2 * z + rnorm(length(z), sd = 4))
```

What does `rep()`, `length()`, and `rnorm()` do? Look at the help pages by running `?rep`, `?rnorm`, and `?length` in the interactive Console window, or searching for them in the `Help` pane.

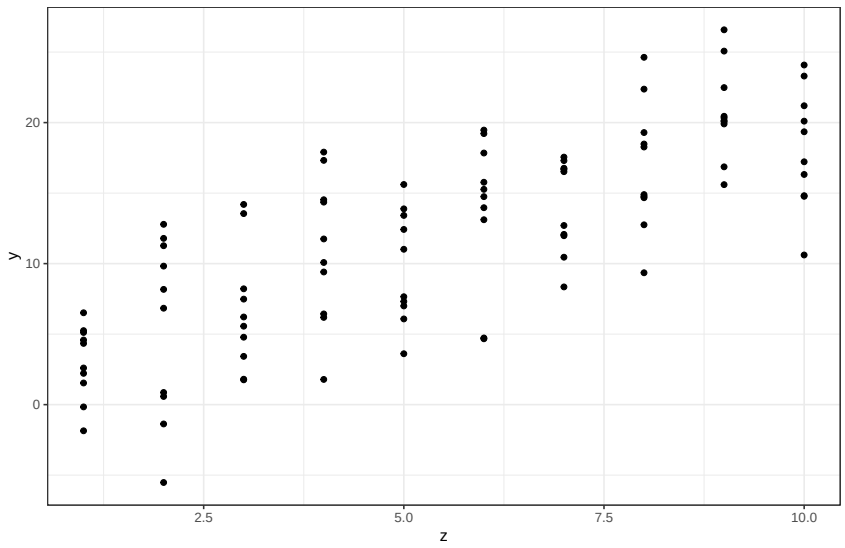
What values of β_0 and β_z did the synthetic data generation use?
Answer: $\beta_0 = 0$ and $\beta_z = 2$.

You can plot the data with "base graphics",

```
plot(data$z, data$y)
```

or with `ggplot`,

```
# Put the first two lines at the beginning of your script file!
library(ggplot2) # Load the ggplot2 package
theme_set(theme_bw()) # Change the default theme (avoids grey background in the plots)
# Now we can plot!
ggplot(data) + geom_point(aes(z, y))
```



For this simple plot, the `ggplot` approach may seem more complicated than necessary, but for more complex plots `ggplot` will make plotting much easier. In the code above, we first supply the data set, and then add information about how to plot it, using a *grammar of graphics*. See `?aes` for some more information about the parameters controlling the *aesthetics* of the *geom* that adds *points* to the plot.

5. Use the `lm()` function to estimate the model and save the estimated model in a variable called `mod`.

```
mod <- lm(y ~ z, data)
```

6. Now we want to plot the linear predictor for the model as a function of z, between $z = -10$ and $z = 20$. First, create a new `data.frame`:

```
newdata <- data.frame(z = -10:20)
```

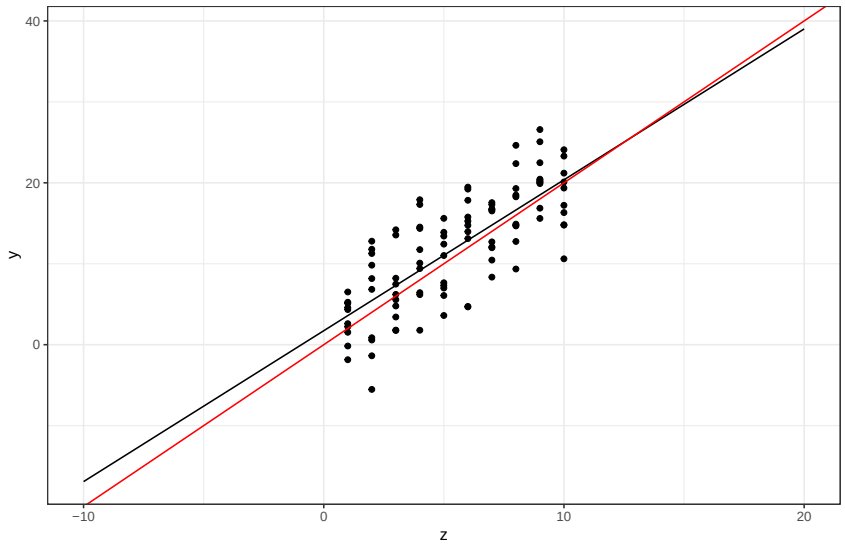
Note that we don't have any corresponding y-values!

Use `predict()`, `plot()`, and `geom_line()` to plot the linear predictor and the original data points in a single figure. Also add the true predictor with the help of `geom_abline(intercept = ..., slope = ...)` (use the `col` parameter to set a different color!). Create a new data frame and supply it directly to `geom_line` to override the original data:

```
data_pred <- cbind(newdata,
                   data.frame(fit = predict(mod, newdata)))
ggplot(...) +
  ... +
  geom_line(data = data_pred, aes(y = fit))
```

Structure the code by adding a linebreak *after* each `+` operation. The resulting plot should look similar to this:

```
# Solution
data_pred <- cbind(newdata,
                   data.frame(fit = predict(mod, newdata)))
ggplot(data) +
  geom_point(aes(z, y)) +
  geom_line(data = data_pred, aes(z, fit)) +
  geom_abline(intercept = 0, slope = 2, col = "red")
```



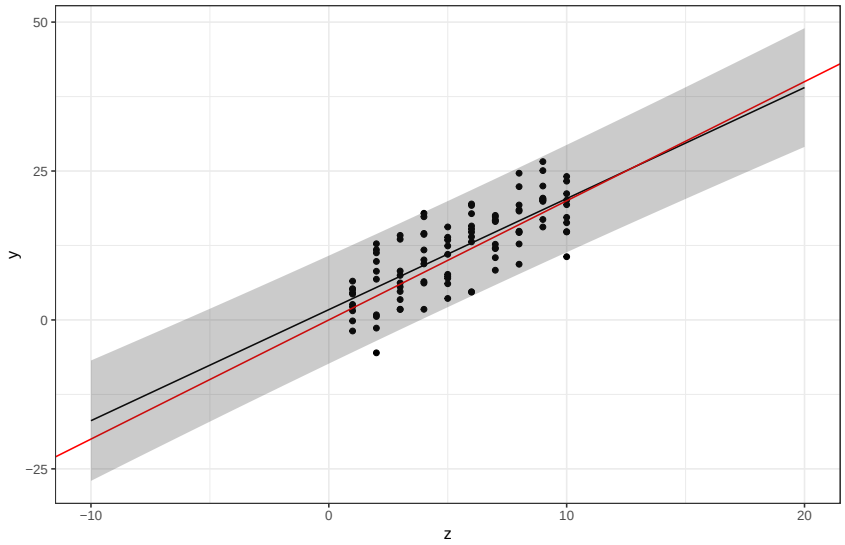
7. Now modify the plotting code to also show *prediction intervals*. Hint: use the `interval="prediction"` parameter for `predict()`, and combine the output (which is now a matrix, so there is no need to convert it to a `data.frame`) with `newdata` with the help of `cbind`. The resulting `data.frame` should have variables named `z`, `fit`, `lwr`, and `upr`, and the whole object should be called `pred`.

```
# Solution
pred <- cbind(newdata,
              predict(mod, newdata, interval = "prediction"))
str(pred)
```

```
## 'data.frame': 31 obs. of  4 variables:
##  $ z  : int  -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 ...
##  $ fit: num  -16.91 -15.05 -13.19 -11.32 -9.46 ...
##  $ lwr: num  -27 -25 -23 -21 -19 ...
##  $ upr: num  -6.808 -5.086 -3.356 -1.619 0.127 ...
```

Use the `geom_ribbon(data = ..., aes(x, ymin, ymax), alpha = 0.25)` function to add the prediction intervals to the figure. The `alpha` parameter sets the "transparency" of the ribbon. Experiment with different values between 0 and 1. The result should look similar to this:

```
# Solution
ggplot(data) +
  geom_point(aes(z, y)) +
  geom_line(data = pred, aes(z, fit)) +
  geom_abline(intercept = 0, slope = 2, col = "red") +
  geom_ribbon(data = pred, aes(x = z, ymin = lwr, ymax = upr), alpha = 0.25)
```



In "base graphics", the following would produce a similar result:

```
# Solution
plot(newdata$z, pred[, "fit"], type = "l", ylim = range(pred))
lines(newdata$z, pred[, "lwr"], lty = 2)
lines(newdata$z, pred[, "upr"], lty = 2)

points(data$z, data$y, pch = 20)
abline(0, 2, col = 2)
```

8. Let's say we want to redo the data analysis using different models (after all, we only know the true model because we generated synthetic data!).

Instead of copying the plotting code for each new model, let's create a function instead! Start with the following skeleton code, and fill in the missing bits:

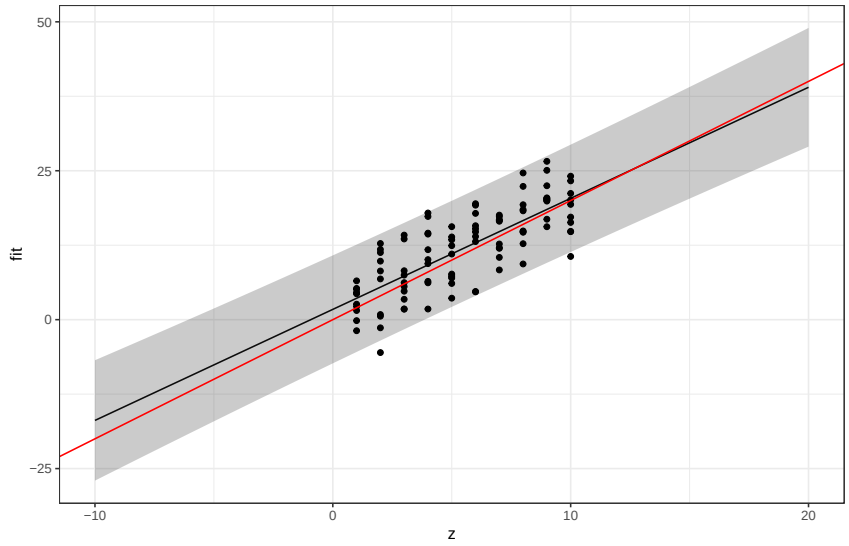
```
# x: a formula
# newdata: data for prediction
# xname: A string containing the name of the variable on the x-axis
plot_predictions <- function(x, newdata, xname = "x") {
  pred <- ???
  ggplot(pred) +
    geom_line(aes_string(xname, "???")) +
    geom_ribbon(???)
}
```

What should `xname` be set to when we call this function for our data?

```
# Solution
plot_predictions <- function(x, newdata, xname = "x") {
  pred <- cbind(newdata,
                predict(x, newdata, interval = "prediction"))
  ggplot(pred) +
    geom_line(aes_string(xname, "fit")) +
    geom_ribbon(aes_string(x = xname, ymin = "lwr", ymax = "upr"), alpha = 0.25)
}
```

You should then be able to run the following code to regenerate your previous figure:

```
plot_predictions(mod, newdata, xname = "z") +
  geom_point(data = data, aes(z, y)) +
  geom_abline(intercept = 0, slope = 2, col = "red")
```



Remark: Why can we use the `+ geom_*` technique to add to the plot drawn by `plot_predictions` even though we don't directly call `ggplot()` here? Every function returns its last computed object as output. In this case, that's a `ggplot` object, and we can use it just as if we had spelled out the call to `ggplot()`. This also means that we could in principle write a function that adds features to a plot, taking an existing `ggplot` object as *input*, making complex plotting more structured and modular.

9. You can change the plot lables and add a title by adding calls to `xlab()`, `ylab()`, and `ggtitle()` to the plot:

```
plot_predictions <- function(x, newdata, xname = "x", ???, ...) {
  ???
  ggplot(???) +
    ??? +
    xlab(xname) +
    ylab(ylab)
}
plot_predictions(mod, newdata, xname = "z", xlab = "z", ylab = "y")
```

This passes any additional parameters through to the `plot()` function.

The lab solution document has a further generalised version of this plotting function, that also adds the confidence intervals for the predictor curve to the plot.

```
# Extended version of the solution:
plot_prediction <- function(x, newdata, xname = "x",
                            xlab = NULL, ylab = NULL, ...) {
  if (is.null(xlab)) {
```

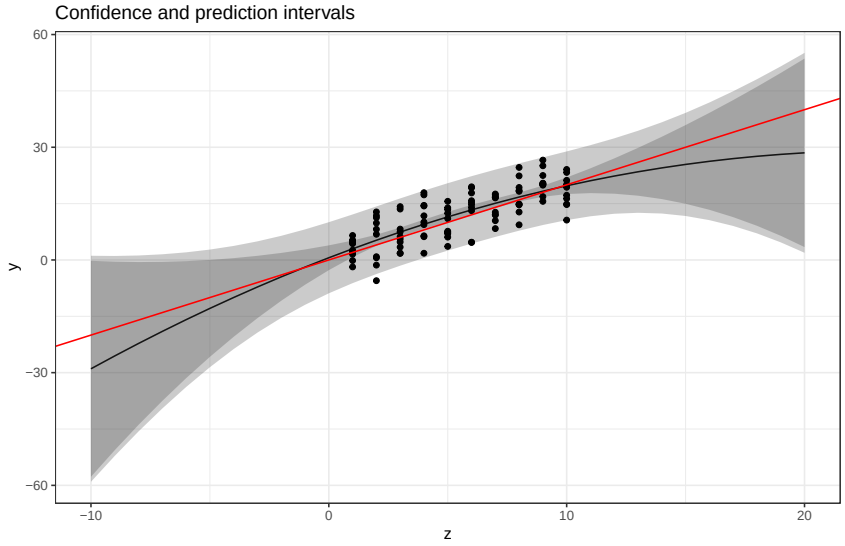
```
    xlab <- xname
  }
  if (is.null(ylab)) {
    ylab <- "Response"
  }

  pred <- cbind(newdata,
                predict(x, newdata, interval = "prediction"))
  pl <- ggplot() +
    geom_line(data = pred,
              aes_string(xname, "fit")) +
    geom_ribbon(data = pred,
                aes_string(xname, ymin = "lwr", ymax = "upr"), alpha = 0.25)

  # Also add the confidence intervals for the predictor curve
  conf <- cbind(newdata,
                predict(x, newdata, interval = "confidence"))
  pl <- pl +
    geom_ribbon(data = conf,
                aes_string(xname, ymin = "lwr", ymax = "upr"), alpha = 0.25)

  # Add lables:
  pl + xlab(xlab) + ylab(ylab)
}
```

10. Use your new function to plot the predictions for the quadratic model

```
mod <- lm(y ~ 1 + z + I(z ^ 2), data)
```

The `I()` syntax means that you don't have to create a separate data variable equal to the square of the z-values; the `lm()` function will create it for you internally. The result should look similar to this:

```
# Solution
newdata <- data.frame(z = -10:20)
plot_prediction(mod, newdata, xname = "z", ylab = "y") +
  geom_point(data = data, aes(z, y)) +
  geom_abline(intercept = 0, slope = 2, col = "red") +
  ggtitle("Confidence and prediction intervals")
```



11. Finally, let's estimate and plot predictions for four polynomial models, by first creating a list of formulas:

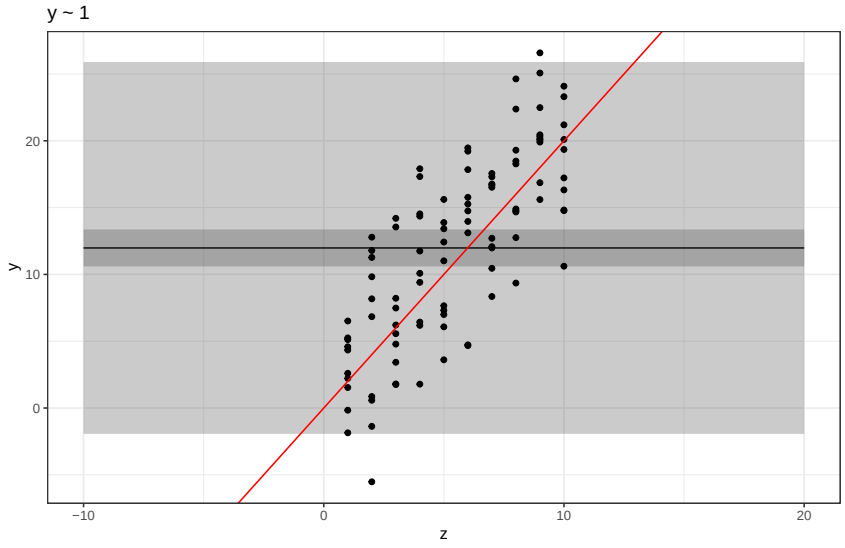
```
formulas <- c(y ~ 1,
              y ~ z,
              y ~ z + I(z^2),
              y ~ z + I(z^2) + I(z^3))
```

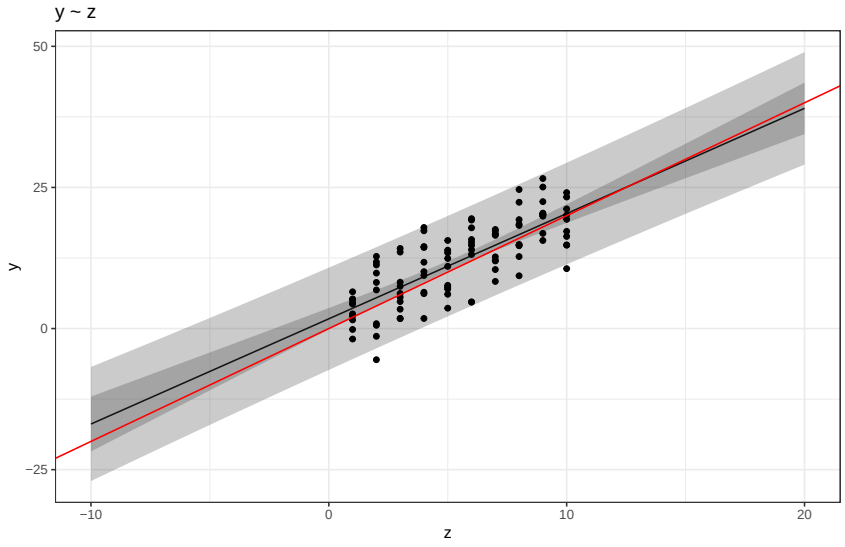
Look at the lecture notes and use the `lapply()` technique to estimate all four models and store the result in a variable called `mods`.

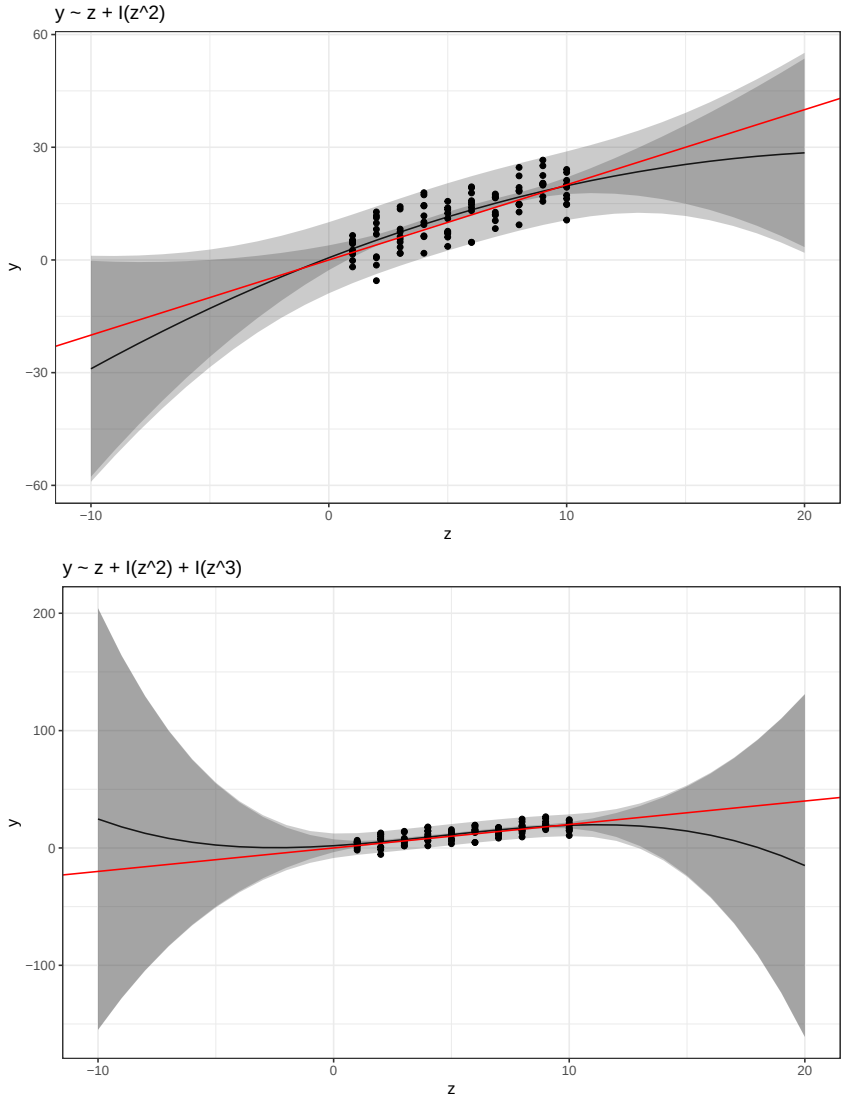
```
# Solution
mods <- lapply(formulas, function(x) lm(x, data))
```

12. You can now use something like the following code to plot all the results:

```
for (k in seq_along(formulas)) {
  pl <-
    plot_prediction(mods[[k]], newdata, xname = "z", ylab = "y") +
      geom_point(data = data, aes(z, y)) +
      geom_abline(intercept = 0, slope = 2, col = "red") +
      ggtitle(as.character(formulas[k]))
  print(pl)
}
```


y ~ 1
20
10
0
y
−10
0
10
20
z


y ~ z
50
25
0
−25
y
−10
0
10
20
z



Look at the help text for `seq_along()`! Storing the plot as an object, `pl`, and then explicitly "printing" it is sometimes needed to ensure that all the plots are generated into a report document. We will look at ways of combining multiple plots later.

Does the information displayed in the plots match your intuition about the different models?