Five levels of analytical automation

Robert L Grant, BayesCamp Ltd, 2019. ©

I have been thinking more about how programming that requires minimal human input is a virtue in computer science, and hence machine learning, circles. Although there's no doubt that is one of the central goals of programming a computer in general, I'm not convinced this extends to data analysis, which needs some thought, contextual knowledge and curation. The contextual knowledge can be broken down further into understanding where the data came from and its weaknesses, what question we are really trying to answer, and where the findings are going: the statistical literacy of the audience and how their decisions should be influenced by the findings.

Let's imagine a sliding scale of code autonomy. At one end is work that requires humans to do everything, most typically by operating a point-and-click graphical user interface (GUI) such as many users of Tableau, SPSS, etc are familiar with. That is completely non-reproducible in the sense that it can only be reproduced if you have a willing and suitably informed human to do it, and I think that makes it an island off the end of the sliding scale. Let's note it and then restrict consideration to data analysis that is coded in some way so that the machine can re-run at least part of it without human intervention.

There's no question that this is a good idea. You should be able to show where your results came from (an audit trail). Also, it's very common to reach what you think is the end of the analysis only to be asked to run it again on some corrected or extended data; being able to just replace the data file and press Go is a huge time saver, and helps avoid typos.

But just how far down the scale do you need to go? I suggest five levels along the way to complete autonomy; let's look at each in turn with a simple example. I use R here but the same ideas translate to every software tool and language.

Level one:

- hard-coded results mean it won't give different answers with new data
- it doesn't look for changes in the size of the input data
- it doesn't check for violations of data assumptions
- it doesn't take arguments to control algorithm parameters, random number generator (RNG) seeds, or outputs such as file names or graphics captions

```
mydata <- read.csv("datafile.csv")
myregression <- lm(outcome ~ riskfactor + confounder, data=mydata)
prediction <- 0.213 + 0.918*mydata$riskfactor + 0.301*mydata$confounder
png("results.png")
plot(mydata$riskfactor, mydata$outcome, main="Regression findings")
points(mydata$riskfactor, prediction, pch=19)
dev.off()
```

Even if you don't know R, you can see that this reads in a CSV file with a certain name, runs a linear regression using some particular variables in the data ("outcome", "riskfactor" and "confounder" — economically-trained readers might prefer to think of "confounder" as "endogenous_variable"), calculates predicted values of the outcome variable by hard-coding in the regression coefficients (presumably, after running the lm() line, the analyst printed the summary(myregression) and looked at the coefficients before typing them in), and then makes a plot of the observed and predicted values with a certain title, storing that in a PNG file with a certain name.

This is all fine for giving an audit trail. You could re-run this on the same data and get the same results, and with some effort, you could go through and change options and see how that affects the results (for example, you could include an interaction term between "riskfactor" and "confounder"). I say "with some effort", because you'd have to change the lm() line and the prediction line, and to avoid confusion, you'd probably want to add a caption to the plot too.

When the boss calls you up and tells you that – "great news" – they found some more data and want to re-run the analysis, you'd have saved some time over using a GUI, but you'd still have to make some amendments to the code, and that would cost you time, plus the risk of typos. Let's say that takes half a day to amend, run and check. The boss is satisfied, unless you made a typo and have to retract it later, in which case they are definitely not satisfied.

Level two:

- fix the hard-coding

This is low-hanging fruit, really. In the case of regression coefficients, nobody would write the prediction line because you could just have:

```
prediction <- predict(myregression)
```

and that would take care of any changes to the lm() formula or the data.

But in more complex settings, you might want to get the coefficients (or equivalents for other analyses) and calculate something a bit more bespoke from them. You should really do this by pulling out the relevant values returned by the regression function. In R, Python or Julia, most functions like this return a list (or whatever they call it) of all sorts of scalars, vectors, matrices and strings, and in there is something you need. In Stata, which I use a lot, such things go into the r() or e() lists of macros and you can copy them from there before the next estimation command replaces them. Other software might not provide flexible return values like that; if so, that software is not going to be helpful for reproducible and semi-autonomous analysis.

```
mycoef <- myregression$coefficients
prediction <- mycoef[1] + mycoef[2]*mydata$riskfactor +
mycoef[3]*mydata$confounder
```

You can get more smart-ass about it, but the principle is clear.

```
prediction <- mycoef * model.matrix(model.frame(~riskfactor+confounder,
data=mydata))
```

And the same thing applies to any hard-coding.

Level three:

- examine and respond to the size of the input data; what if there are suddenly more variables than before?

It's generally not an issue if the number of observations (rows) changes, but you ought to watch out for future situations where you need to add an extra column. Here, the balance of cost and benefit starts to become contested. If you are going to add a new predictor to a linear regression, you'd better check for missing data, multicollinearity and such before you proceed. So, is it better to allow a regression with unspecified inputs or to force the user to stop and think? On the other hand, some analytical methods can handle all sorts of inputs, and you might feel more relaxed about them (notably, those that get used in autoML are like this: random forests and boosted trees do well with minimal data processing, while regressions, artificial neural networks and clustering are sensitive to the transformations and filtering of the input data that, in the ML world, they call feature engineering; perhaps tellingly, I see very little unsupervised learning in autoML products for this reason).

In reality, coding up analysis that's flexible to columns can be quite hard work. There's often some kind of syntactical structure that requires you to type out all the constituent parts, like those formulas in R or layering of twoway graphs in Stata. If you can supply a matrix instead, you're going to have more flexibility straight away. It might be worth looking for a different package / library / command that can do what you want with the right kind of input.

Level four:

- check for violations of data assumptions

Who said the data supplied would always be numeric? Or factors? Or ISO8601 dates and times? The fact you've only received strings "Yes" and "No" in a variable so far does not mean some buffoon isn't about to start typing "n", "y", "yup", or even "see notes". You better check for that and halt the process if you find departures from your assumptions. Remember to write nice, informative error messages for yourself, and others in the future. Every programming language (Ok, maybe not those weird ones) has functionality to check some logical comparison and halt with an error message if the check fails.

Level five:

- make it a function / command / subroutine / macro that receives arguments

To give yourself (or others) real flexibility in re-running this analysis, but on different data, and with somewhat different settings, you need one function that takes the data and the settings as inputs, and returns the results. It might look like this:

```
runanalysis(data_file="Nov2016.csv", outcome_column=3,
predictor_columns=c(5,7,8), gamma=0.3, test_sample=0.2, iterations=1000,
log_file="week_31_run.txt", image_folder="week_31_images/",
regression_option="glmnet")
```

I leave the precise meaning to your imagination. The point is, you can try out a range of different settings by looping over different values of those arguments. You can run different versions of runanalysis() on different CPU cores, or groups of cores, or virtual machines. You can plug in new data, or new settings.

Finally – and I think this is very cool because I'm interested in how we communicate our work to non-quants – you can make a GUI for your new function. R has some nice tools for this and there's a book too.

Note that any of these levels can be automated to run at scheduled intervals, looking for new data at a certain location (they're not limited to CSV files, they might query databases or other APIs), or they could periodically check for the appearance of a new data file and run when they find it (when the data collection team drop it into the folder ready for analysis). That's an important part of automation but actually not the same dimension as this scale of code autonomy, I think.

Having reached level five, you can now go in two ways for near-complete autonomy, and they're not mutually exclusive.

One is to build a package that contains code like this for a whole suite of relevant analyses that your organisation uses, so that you can specify what data to use, what the variables of interest are, what algorithms to use (you might want to run a GAM, regression tree, and random forest, for example), where to store the results, which of the organisation style templates to use for the text and tabular outputs, and lists of algorithm-specific tuning parameters. There might be a top-level function that calls one of the underlying runanalysis() functions. Or one overarching GUI (and at that point, you've come close to developing your own statistics software; you might even consider trying to sell it).

Another is to take an autoML approach, where you have your code run a variety of different analyses and select one on the basis of some performance metrics. If this interests you, watch out for my series of test drives of cloud analysis front-ends, coming to this blog soon.