Accelerate Analytics – hiring re EMRs and bio.

NYC R conference is April 21-22 – sign up! Early signup discount code still works

firstR is discount code. Rstats.nyc

Max Kuhn – caret guy. Will teach a master class after the conference.

Worked in drug discovery for many years, now is w/ RStudio.

Tonight wants to explain the *formula* method, how it works, its flaws, and a new package he’ll have out ~ end of next month to improve things.

Example of a formula in lin model to predict sale prices of houses:

*Library(caret)*

*Data(“Sacramento”)*

*Mod1 <- lm(log(price) ~ type + sqft, data=Sacramento, subset=beds > 2)*

Purpose of this code chunk:

1. Subset some of the data pts (subset)
2. Create a design matrix for 2 predictor variable (but 3 model terms)
   1. “Design matrix” – numeric values that rep the terms that’ll be in your model. 2D array of data w/ samples on the rows, vars or features on the columns
   2. Here, there are 3 levels of type, so actually generates 3 binary columns in type matrix
3. Log transform the outcome var
4. Fit a linreg model

How does this happen?

How model formulas work under the hood

Lm uses the formula nd appropriate environment to translate the rels b/w vars to make a data frame

The main tools used to get the design matrix are the model.frame and model.matrix fcts

Lm under thhe hood:

Function(formula, data, subset, weights, na.action, method=”qr”, model=T, x=F, y=F, qr=T, singular.ok=T, contrasts=NULL, effset,…) {

Ret.x <- x

Ret.y <- y

Cl <- match.call() #captures what the call to lm was. This is an R expression object. The CL one is an R #expression that is the original lm call

Mf <- match.call(expand.dots=F) #this one is for the possible arguments elided w/ … up top

M <- match(c = c(“formula”...) #this line keeps the arguments that are common to data.frame and to lm

Mf <- eval(expr=mf, envir=parent.frame()) #here’s the interesting part

Mf is a call object. You can treat a call object kind of like a list. Lm expression does this, subsets it, adds and takes out arguments appropriately.

Note that lm and model.frame have *fairly* similar arguments, but not entirely.

There’s a bit of functional programming involved in changing the call

Model Frame: evaluates the call on the “parent frame”, where the data actually lives

Where the “Sacramento” data is. Not inside the mf fct call, but one level up

Because the data isn’t inside the lm fct, it’s one level up!

Mf <- eval(expr = mf, envir = parent.frame())

Recommendation: ch 10-12 at adv-r.had.co.nz. This is Hadley’s advanced R book, which goes down to this level.

About the model frame: note that:

All the columns are present – predictors and responses

The filtering defined by subset arg is executed here

Price has been logged

Type hasn’t created any dummy vars yet

Doesn’t req that all columns be numeric! Many modeling fcts do

Need a step to turn factor var *type* into levels

Note there’s a hidden “dims” attribute w/ number of rows and columns

*Terms* object is impt. Notable attributes it has include *factors* and *predvars*

Factors is a matrix of the columns in the data against each other, binary

So, now we’ve got a model frame, and now lm uses that model frame to create the actual design matrix

Mt <- attr(x=mf, which=”terms”)

X <- model.matrix(object=mt, data=mf, contrasts.arg=contrasts)

The model.matrix fct uses the data in the terms object to gen any interactions and/ or dummy vars from factors

Won’t go into the mechanics of this

Design matrix is a/b your predictor vars – doesn’t include prediction itself yet

Note that if there are *c* levels in factor vars, you need *c-1* dummy vars

Polynomial expansion – think glms. Your design matrix’ll have x and x^2 for a parabola, for example

Spline goes from single columns in design matrix to multiple columns to make nonlinear fcts of the original data

As you add more terms to the polynomial expansion (more degrees of freedom), it gets more nonlinear

Ns() = natural spline” fct. Constructs natural splines based on diff predictor vars

Ns() returns multiple elements: the basis fct spline results, and the data required to generate such results for new data

Gives back a matrix that comprises the info that need to do same transform on new data

You can call ns() as an arg in lm! This is useful for getting certain model terms saved so you can work w/ them later

Predvars attribute of terms object:

First off, predvars is a misleading name.

Summarizing the model formula method:

Model formulas can rep model terms easily. Are v expressive.

Formula/ terms framework does some elegant fctal programming

Fcts can be embedded inline to do complicated things to single vars, and these can be applied to new data

However! Sig limitations to what this framework can do, and can be v ineff.

Due to being written before large-scale modeling and ML were common. Doesn’t work well when you’re gonna have 100s of predictors

You’ll care a/b these limitations if you’re doing ML, not so much if you’re doing ANOVA or linreg

Limitations to model formula method:

1. Limits to extensibility. V verbose, many ops on many vars.
   1. You might want to, say, model principal components rather than the predictors themselves. This doesn’t work so nicely b/c of predvars – it’d require computing stuff twice
      1. Predvars aspect limits the utility of the ops.
      2. If want to scale units (obv square footage and number of bathrooms are v diff units!), can’t do that well.
   2. If there’s missing data, you can’t impute well.
      1. Max likes to impute using k nearest neighbors. Here, you’d have to save the entire data set in predvars, separately, for *every* predictor you wanted to impute
         1. E.g. knn for type and for square feet, you’d have to save it twice
2. Everything happens at once
   1. It sounds reasonable to impute missing value -> center and scale -> convert to PCA scores. Can’t sequentially describe the preprocessing steps like this inside model formula framework! This is a problem if you wanna do feature engineering
      1. Max has an idea using %>% for allowing this in his package!
3. No recycling
   1. If there’s a lengthy preprocessing step inside model formula, you *can’t* save it there and then add another technique w/o having to rerun that step
4. Doesn’t play nice w/ wide data (many-column data)
   1. Model saves the formula info v inefficiently
      1. If you’re doing random forest or rpart model, formula takes up a ton of time and memory
      2. As the number of predictors goes up, the time that’s working on the formula rather than on the data to go in it goes up exponentially
      3. If you’ve got 1000 predictors, this is a pain in the ass – and remember need to repeat all that time again if change the model
5. Variable roles 0 you can’t have diff vars do diff things in your model well
   1. Mixed models require you to specify experimental unit, which is like a label/ ID variable. When wrote mixed model package, needed to totally change how formula works to make that work nicely
   2. Can’t do diff subformulas inside a model you’re trying to make
   3. Slightly annoying if you wanna predict smthng multivariate – e.g. aggregate the mean of two vars by month is aggregate(cbind(ozone, Temp) ~ Month, data=airquality, mean)
      1. The cbind isn’t that bad, but would prefer not to need to do that
      2. You can nest matrices instead columns of dataframes – and in fact, this is often necessary if you wanna do some multivariate stuff
   4. General list of possible variable roles could be: outcomes, predictors, stratification, model performance data (eg weight by loan amt to compute expected loss), conditioning of faceting vars (for your ggplots or lattice plots), ID vars in random effects model or hierarchical model
      1. Also case weights, offsets, and error terms, which formula method can handle better than can handle doing several of the above in one formula

Recipes

We can approach the design matrix and preproc steps by specifying a *sequence of steps*

e.g.

1. price is an outcome

2. Type and sqft are predictors

3. Log transform price

4. Convert type to dummy vars

Compare to ggplot! Until you hit enter, ggplot doesn’t do anything. It lets you add stuff you wanna do until have +’d in everything you want. That kind of delayed execution is the idea here

Can extend w/ other steps – dim reduction like PCA or ICA, interactions b/w vars, simple trnasfomrations like centering and scaling, imputing missing values…

A *recipe* can be estimated then applied to any data

Example:

Rec <- recipe(price ~ type + sqft, data=Sacramento) #just needs to

Rec <- rec %>% step\_log(~ price) #taking the log of price

%>% step\_dummy(~type) #making type into dummy vars

Rec\_trained <- learn(rec, training=Sacramento, retain=TRUE)

Similar to the preproc fct in caret package. Could set training= to a diff training set if you wanted – you need to specify data when you’re first defining recipe(), but won’t actually *do* the calcs until do learn(), so you can do it on completely diff dataset if you want.

You can write custom *step\_\** fcts for whatever you want – will be a vignette on it. Should be easy to come up w/ steps you want to do.

Design\_mat <- process(rec\_trained, newdata= whatever)

Made a recipe, then apply it to new data and get a design matrix

Extending:

If need to add more preprocessing or other operations, can only estimate the new parts using smthng like

Rec <- learn(rec) after adding some new things to rec

Standardized <- rec\_trained %>% step\_center(~is\_numeric()) %>% *more stuff*

Standardized <- learn(standardized)

Won’t duplicate stuff you’ve already done – you can take the rec\_trained you made above and just add the new steps to that recipe

You can also subtract stuff from your recipe if you want

In some cases, you won’t know the names of the predictors at the time you construct a recipe

Solution to this is that you can use dplyr selectors on variable names. Such as match(“^PC[1-9]”). Can put that inside your recipe so it’ll select matching columns!

Hm, I suppose doing regex to handle diff capitalization and abbreviations might be handy for making recipes to handle data from a bunch of diff sources

Gonna implement a summary fct to keep track of what’s available to you

Currently stores everything in tibbles, though can convert to data.frame if you want

Recipes can also be created w/ diff roles manually, like for adding stratification variables. Add\_role(“zip”, role=”strata”)

Also, the sequential nature of steps means can do this w/ stuff that’s not even R operations, e.g. calling Weka for NLP or scikit-learn b/c you like Python too. Putting an entire pipeline in one fct?

Can create wrappers that do recipe instead of formula to a fct you like, so that you don’t have to use a formula and have all downsides of formulas at the last step

If individual fct doesn’t have a way to pass matrix or vector, are still limited by limitations of formula method – working on ways to get around that

There are plans to make a sparse tibble akin to existing sparse object types.

Plans to write recipe method in caret later this year – will take a while.