This post is about various function options of the glmnet function (from the package of the same name), hoping to give more detail and insight beyond R’s documentation.

In this post, instead of looking at one of the function options of glmnet, we’ll look at the predict method for a glmnet object instead. The object returned by glmnet (call it fit) has class "glmnet"; when we run predict(fit), it runs the predict method for class "glmnet" objects, i.e. predict.glmnet(fit).

For reference, here is the full signature of the predict.glmnet function/method (v3.0-2):

predict(object, newx, s = NULL, type = c("link",

"response", "coefficients", "nonzero", "class"), exact = FALSE,

newoffset, ...)

In the above, object is a fitted "glmnet" object (call it fit). Recall that every glmnet fit has a lambda sequence associated with it: this will be important in understanding what follows. (This sequence can be accessed via fit$lambda.)

For the rest of this post, we will use the following data example:

set.seed(1)

n <- 100; p <- 20

x <- matrix(rnorm(n \* p), nrow = n)

beta <- matrix(c(rep(1, 5), rep(0, 15)), ncol = 1)

y <- x %\*% beta + rnorm(n)

fit <- glmnet(x, y)

**Function option: newx**

newx is simply the new x matrix at which we want predictions for. So for example, if we want predictions for the training x matrix, we would do

predict(fit, x)

If no other arguments are passed, we will get a matrix of predictions, each column corresponding to predictions for each value of in fit$lambda. For our example, fit$lambda has length 68 and x consists of 100 rows/observations, so predict(fit, x) returns a matrix.

length(fit$lambda)

# [1] 68

dim(predict(fit, x))

# [1] 100 68

newx must be provided except when type="coefficients" or type="nonzero" (more on these types later).

**Function option: newoffset**

If the original glmnet call was fit with an offset, then an offset must be included in the predict call under the newoffset option. If not included, an error will be thrown.

set.seed(2)

offset <- rnorm(n)

fit2 <- glmnet(x, y, offset = offset)

predict(fit2, x)

# Error: No newoffset provided for prediction, yet offset used in fit of glmnet

The reverse is true, in that if the original glmnet call was NOT fit with an offset, then predict will not allow you to include an offset in the prediction, EVEN if you pass it the newoffset option. It does not throw a warning or error, but simply ignore the newoffset option. You have been warned! This is demonstrated in the code snippet below.

pred\_no\_offset <- predict(fit, x)

pred\_w\_offset <- predict(fit, x, offset = offset)

max(abs(pred\_no\_offset - pred\_w\_offset))

# [1] 0

**Function option: s and exact**

s indicates the values for which we want predictions at. If the user does not specify s, predict will give predictions for each of the values in fit$lambda.

(***Why is this option named* s and not the more intuitive lambda?** In page 5 of [this vignette](https://web.stanford.edu/~hastie/Papers/Glmnet_Vignette.pdf), the authors say they made this choice “in case later we want to allow one to specify the model size in other ways”. lambda controls the model size in the sense that the larger it is, the more coefficients will be forced to zero. There are other ways to specify model size. For example, one could imagine a function option where we specify the number of non-zero coefficients we want in the model, or where we specify the maximum norm the coefficient vector can have. None of these other options have been implemented at the moment.)

***If the user-specified s values all belong to fit$lambda,*** then predict pulls out the coefficients corresponding to those values and returns predictions. In this case, the exact option has no effect.

***If the user-specified s value does NOT belong to fit$lambda,*** things get interesting. If exact=FALSE (the default), predict uses linear interpolation to make predictions. (More accurately, it does linear interpolation of the coefficients, which translates to linear interpolation of the predictions.) As stated in the documentation: “while this is often a good approximation, it can sometimes be a bit coarse”.

As a demonstration: In the snippet below, we look at the predictions at a value of that lies between the two largest values in fit$lambda. If the function does as the documentation says, the last line should give a value of 0 (to machine precision).

b1 <- as.numeric(predict(fit, x, s = fit$lambda[1]))

b2 <- as.numeric(predict(fit, x, s = fit$lambda[2]))

b3 <- as.numeric(predict(fit, x,

s = 0.3\*fit$lambda[1] + 0.7\*fit$lambda[2]))

max(abs(b3 - (0.3\*b1 + 0.7\*b2)))

# [1] 3.885781e-16

***What happens if we have values in s that are not within the range of fit$lambda?*** First, I would recommend using exact=TRUE because extrapolation beyond the range of fit$lambda is dangerous in general. In my little experiments, it looks like predict simply returns the predictions for the value in fit$lambda that is closest to s.

If exact=TRUE, predict merges s with fit$lambda to get a single (decreasing) sequence, refits the glmnet model, then returns predictions at the values in s. If your training data is very large, this refitting could take a long time.

One note when using exact=TRUE is that you have to pass in additional arguments in order for the refitting to happen. That’s because the fitted glmnet object does not contain all the ingredients needed to do refitting. For our example, to predict for fit we need to supply x and y as well. For more complicated glmnet calls, more options have to be provided.

predict(fit, x, s = fit$lambda[68] / 2, exact = TRUE)

# Error: used coef.glmnet() or predict.glmnet() with `exact=TRUE`

# so must in addition supply original argument(s) x and y in order to

# safely rerun glmnet

predict(fit, x, s = fit$lambda[68] / 2, exact = TRUE, x = x, y = y)

# glmnet correctly returns predictions...

**Function option: type**

The type option determines the type of prediction returned. type="coefficients" returns the model coefficients for the values in s as a sparse matrix. type="nonzero" returns a list, with each element being a vector of the features which have non-zero features. For example, the code snippet below shows that for the second and third values in fit$lambda, the features that have non-zero coefficients are feature 5 and features 3 and 5 respectively.

predict(fit, type = "nonzero", s = fit$lambda[2:3])

# $`1`

# [1] 5

#

# $`2`

# [1] 3 5

For type="coefficients" and type="nonzero", the user does not have to provide a newx argument since the return value does not depend on where we want the predictions. For the rest of the possible values of type, newx is required.

For type="link" (the default) and type="response" it helps to know a little [GLM theory](https://statisticaloddsandends.wordpress.com/2019/10/31/understanding-the-components-of-a-generalized-linear-model-glm/). For a observation having values , type="link" returns , where is the coefficient vector corresponding to a value in s.

For type="response", is passed through the GLM’s inverse link function to return predictions on the y scale. For “gaussian” family it is still . For “binomial” and “poisson” families it is and respectively. For “multinomial” it returns fitted probabilities and for “cox” it returns fitted relative risk.

The final possibility, type="class", applies only to “binomial” and “multinomial” families. For each observation, it simply returns the class with the highest predicted probability.

**Bonus: The coef method**

The coef method for glmnet is actually just a special case of the predict method. This can be seen from the source code:

coef.glmnet

# function (object, s = NULL, exact = FALSE, ...)

# predict(object, s = s, type = "coefficients", exact = exact,

# ...)

#

#

**Bonus: predict.elnet, predict.lognet, …**

If you inspect the class of the object returned by a glmnet call, you will realize that it has more than one class. In the code below, we see that “gaussian” family results in an “elnet” class object. (“binomial” family returns a “lognet” object, “poisson” family returns a “fishnet” object, etc.)

class(fit)

# [1] "elnet" "glmnet"

These classes have their own predict methods as well, but they draw on this base predict.glmnet call. As an example, here is the code for predict.fishnet:

glmnet:::predict.fishnet

# function (object, newx, s = NULL, type = c("link", "response",

# "coefficients", "nonzero"), exact = FALSE, newoffset, ...)

# {

# type = match.arg(type)

# nfit = NextMethod("predict")

# switch(type, response = exp(nfit), nfit)

# }

#

#

What happens here is that predict.glmnet is first called. If type is not "response", then we simply return whatever predict.glmnet would have returned. However, if type="response", then (i) we call predict.glmnet, and (ii) the predictions are passed through the function before being returned.

This is how predict is able to give the correct return output across the different family and type options.