…a series of posts on 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, we will look at the **type.gaussian** option.

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

```
glmnet(x, y, family = c("gaussian", "binomial", "poisson", "multinomial",
  "cox", "mgaussian"), weights, offset = NULL, alpha = 1,
  nlambda = 100, lambda.min.ratio = ifelse(nobs < nvars, 0.01, 1e-04),
  lambda = NULL, standardize = TRUE, intercept = TRUE,
  thresh = 1e-07, dfmax = nvars + 1, pmax = min(dfmax * 2 + 20,
  nvars), exclude, penalty.factor = rep(1, nvars), lower.limits = -Inf,
  upper.limits = Inf, maxit = 1e+05, type.gaussian = ifelse(nvars <
  500, "covariance", "naive"), type.logistic = c("Newton",
  "modified.Newton"), standardize.response = FALSE,
  type.multinomial = c("ungrouped", "grouped"), relax = FALSE,
  trace.it = 0, ...)
```

type.gaussian

According to the official R documentation,

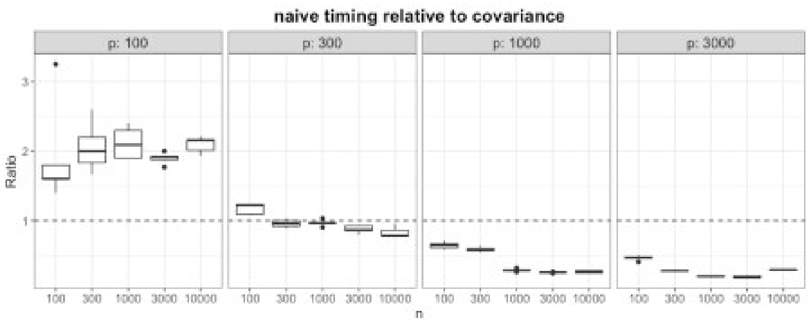
> Two algorithm types are supported for (only) `family="gaussian"`. The default when `nvar<500` is `type.gaussian="covariance"`, and saves all inner-products ever computed. This can be much faster than `type.gaussian="naive"`, which loops through nobs every time an inner-product is computed. The latter can be far more efficient for `nvar >> nobs` situations, or when `nvar > 500`.

Generally speaking there is no need for you as the user to change this option.

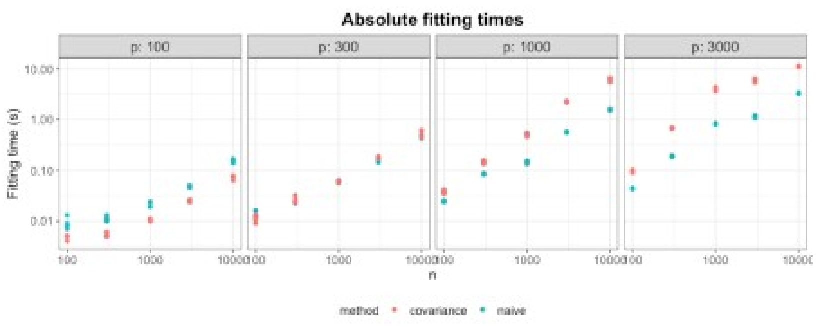
How do the fitting times compare?

I ran a timing simulation to compare the function run times of `type.gaussian="naive"` vs. `type.gaussian="covariance"` for a range of number of observations (`nobs` or n) and number of features (`nvar` or p). The results are shown below. (For the R code that generated these plots, see here.)

This first panel of boxplots shows the time taken for `type.gaussian="naive"` to complete as a fraction (or multiple) of that for `type.gaussian="covariance"` (each boxplot represents 5 simulation runs). As advertised, naive runs more slowly for small values of p but more quickly for large values of p. The difference seems to be more stark when n is larger.



This next plot shows the absolute fitting times: note the log scale on both the x and y axes.



So, what algorithms do these two options represent? What follows is based on *Statistical Learning with Sparsity* by Hastie, Tibshirani and Wainwright (free PDF here, p124 of the PDF, p113 of the book).

Let $y_i \in \mathbb{R}$ denote the response for observation i, and let $x_{ij} \in \mathbb{R}$ denote the value of feature j for observation i. Assume that the response and the features are standardized to have mean zero so that we don't have to worry about fitting an intercept term. For each value of λ in lambda, `glmnet` is minimizing the following objective function:

$$\underset{\beta}{\text{minimize}} \quad J(\beta) = \frac{1}{2n}\sum_{i=1}^n \left(y_i - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p \left(\frac{1-\alpha}{2}\beta_j^2 + \alpha|\beta_j| \right).$$

We minimize the expression above by ***cyclic coordinate descent***. That is, we cycle through the features $j = 1, \dots, p$. For each j, treat J as a function of β_j **only** (pretend that β_k for $k \neq j$ are fixed), then update β_j to the value which minimizes J. It turns out that this update is very simple:

$$\beta_j \leftarrow \frac{\mathcal{S}_{\alpha\lambda}\left(\frac{1}{n}\sum_{i=1}^n r_i^{(j)} x_{ij} \right)}{\frac{1}{n}\sum_{i=1}^n x_{ij}^2 + (1-\alpha)\lambda},$$

where $\mathcal{S}$ is the soft-thresholding operator and $r_i^{(j)} = y_i - \sum_{k \neq j} \beta_k x_{ik}$ is the *partial residual*.

Both of the modes minimize J in this way. Where they differ is in how they keep track of the quantities needed to do the update above. From here, assume that the data has been standardized. (What follows works for unstandardized data as well but just has more complicated expressions.)

`type.gaussian = "naive"`

As the features are standardized, we can write the argument in the soft-thresholding operator as

$$\frac{1}{n}\sum_{i=1}^n r_i^{(j)} x_{ij} = \frac{1}{n}\sum_{i=1}^n r_i x_{ij} + \beta_j, \qquad (1)$$

where r_i is the full residual for observation i. In this mode, we keep track of the full residuals r_i, $i = 1, \dots, n$.

- At a coordinate descent step for feature j, if the coefficient β_j doesn't change its value, no updating of r_i is needed. However, to get the LHS of (1) for the next feature ($j+1$), we need to make $O(n)$ operations to compute the sum on the RHS of (1).
- If β_j changes value, then we have to update the r_i's, then recompute the LHS of (1) for the next feature using the expression on the RHS. This also takes $O(n)$ time.

All in all, a full cycle through all p variables costs $O(np)$ operations.

`type.gaussian = "covariance"`

Ignoring the factor of $\frac{1}{n}$, note that the first term on the RHS of (1) can be written as

$$\sum_{i=1}^{n} r_i x_{ij} = \langle x_j, y \rangle - \sum_{k:|\beta_k|>0} \langle x_j, x_k \rangle \beta_k.$$

In this mode, we compute all the inner products $\langle x_j, y \rangle$ (p of them) which takes $O(np)$ operations. For each k such that $\beta_k \neq 0$, we store the current values of $\langle x_j, x_k \rangle \beta_k$ (there are p of them for each k).

- At a coordinate descent step for feature k, if $\beta_k \neq 0$ and the beginning of the step and its value changes, we need to update the $\langle x_j, x_k \rangle \beta_k$ terms with $O(p)$ operations. Then, to calculate $\sum_{i=1}^{n} r_i x_{ij}$ for the next coordinate descent step, we only need $O(q)$ operations, where q is the number of non-zero coefficients at the moment.
- As such, if no new variables become non-zero in a full cycle through the features, one full cycle takes only $O(pq)$ operations.
- If a new feature x_k enters the model for the first time (i.e. β_k becomes non-zero), then we need to compute and store $\langle x_j, x_k \rangle \beta_k$ for $j = 1, \dots, p$, which takes $O(np)$.

This form of updating avoids the $O(n)$ updating needed at every step at each feature in naive mode. While we sometimes occur $O(np)$ operations when a new variable enters the model, such events don't happen often. Also, we have $q \leq p$, so if p is small or if $n \gg p$, the $O(pq)$ operations for one full cycle pale in comparison with $O(np)$ operations for naive updating.