regls 0.3: regularized least squares for gretl

Allin Cottrell

November 8, 2021

1 Introduction

The regls addon is essentially a front-end for functionality coded in C in the gretl `regls` plugin; to run the package you will need gretl version 2020e or higher. The plugin implements LASSO (Tibshirani, 1996)—by default via the Alternating Direction Method of Multipliers (ADMM) algorithm as set out in Boyd *et al.* (2010); Ridge regression, by default via Singular Value Decomposition; and the "elastic net" hybrid of LASSO and Ridge.

The best-known implementation of regularized regression is that provided by the glmnet package for R. Since we make several references to glmnet below we should state up front what we're talking about. The authors of glmnet are Jerome Friedman, Trevor Hastie, Rob Tibshirani, *et al.* Current information on glmnet can be found at https://glmnet.stanford.edu/; for further information on the algorithms used in the package see Friedman *et al.* (2010).[1]

This package supports LASSO, Ridge and elastic net via the `regls()` function. We begin by discussing LASSO, which is the default method. Ridge is discussed in section 8 and elastic net in section 10.

We use the LASSO parameterization employed by Boyd *et al*: the objective is

$$\min_{\hat{\beta}} \quad \frac{1}{2}\sum_{i=1}^{n}(y_i - X_i\hat{\beta})^2 + \lambda \sum_{j=1}^{k}|\hat{\beta}_j| \tag{1}$$

where n is the number of observations, k is the number of candidate regressors (the number of columns of X) and $\lambda \geq 0$ is the LASSO regularization hyperparameter. In this context $\lambda = 0$ gives plain OLS, and at the other end of the spectrum there exists a data-dependent value of λ, namely

$$\lambda_{\max} = \|X'y\|_{\infty} \tag{2}$$

which drives all elements of $\hat{\beta}$ to zero. A key control variable for our `regls` function is the scaled term $s = \lambda/\lambda_{\max}$, such that $0 \leq s \leq 1$.

The `regls` function takes three arguments: a series (the dependent variable), a list (the independent variables, not including a constant) and a bundle to contain optional parameters; and it returns a bundle, described below. Its signature is therefore

```
function bundle regls (series y, list X, bundle parms)
```

The `parms` argument may be omitted, in which case all settings assume their default values, described below.

One basic element in the `parms` bundle is a specification for λ, which may take either of two forms, as follows:

1. under the key `lfrac` ("lambda fraction"), a scalar (single s value) or vector (sequence of s values); or

[1] We should point out that glmnet supports regularized estimation of generalized linear models. At present regls only supports least squares.

2. under the key `nlambda`, the number of s values to be used (≥ 4), in which case the values will be assigned automatically.

If `nlambda` is provided instead of `lfrac`, the automatic s vector is a logarithmically declining sequence starting at 1 and finishing at 0.0001. For example, given `nlambda` = 5 the sequence will be $s = \{1, 0.1, 0.01, 0.001, 0.0001\}$.

If neither `lfrac` nor `nlambda` is specified, the default is as if `nlambda` were given as 25.

In case you wish to specify a sequence succinctly but with more control, the package contains a utility function `lambda_sequence()`, which takes up to three arguments. The first and second arguments (required) give the maximum s and the number of values, while the third (optional) argument can be used to give the minimum s (by default 0.0001). As with the `nlambda` option the values are spaced logarithmically. So if you were to do

```
parms.lfrac = lambda_sequence(1, 20, 0.001)
```

the resulting sequence would be $s = \{1, 0.69519, 0.48329, \ldots, 0.00144, 0.001\}$.

A second basic member of the parameter bundle is `stdize`, a boolean switch to toggle standardization of the data. The default is to perform standardization (corresponding to a non-zero value of this option), but if the data are already standardized on input `stdize` may be set to 0. The estimates include an intercept (which is not subject to regularization) only if `stdize` is on.

Another basic option is `verbosity`. This has a default value of 1, meaning that `regls` prints out a certain amount of information about its progress and/or results. Setting it to 0 makes `regls` run (mostly) quietly; setting it to 2 or 3 produces more output in some cases.

The further optional parameters, as well as the contents of the bundle returned by `regls`, are best explained by reference to the various modes of usage of the function, namely estimation with a single value of λ; exploration of a range of λ values using a unified training sample; and (probably most relevant in practice) search for optimal λ via cross validation.

2 Estimation with a single regularization

Suppose we have 1200 observations on some series `y` and list `X` (with $k = 100$ members) and we wish to train on the first 1000 observations, using $s = 0.2$, then predict for the remaining 200. And let's say the data are not pre-standardized. We might then do:

```
bundle parms = defbundle("lfrac", 0.2)
smpl 1 1000
bundle lb = regls(y, X, parms)
```

We'll then find the following in `lb`:

- `B`: The full vector of $k + 1$ coefficients (including an intercept).
- `nzb`: A vector holding only the non-zero coefficients.
- `nzX`: A list identifying the regressors with non-zero coefficients.
- `lmax`: The $\lambda_{\max}$ value for the standardized data.
- `lambda`: The value of $\lambda = s\,\lambda_{\max}$, see (2) above.
- `crit`: The minimized LASSO criterion, see (1) above.
- `R2`: Coefficient of determination, $1 - \sum(y - \hat{y})^2 / \sum(y - \bar{y})^2$.
- `BIC`: The Bayesian Information Criterion for the estimated model.
- `nobs`: The number of training observations used.

- lfrac: The input value of s.
- stdize: Whether regls did standardization or not.

To predict for the remainder of the observations we could then do:

```
smpl 1001 1200
series pred = lincomb(lb.nzX, lb.nzb)
```

At some points below we refer to the coefficient of determination (under the key R2) as "R^2". But note that with regularized regression, unlike OLS, this figure is not equal to the squared correlation between y and $\hat{y}$.

3 Exploring a range of regularizations

Suppose we wish to compare results from several values of λ, using all the training data. We might then revise the prior script as:

```
matrix lamseq = lambda_sequence(1, 10)
bundle parms = defbundle("lfrac", lamseq)
smpl 1 1000
bundle lb = regls(y, X, parms)
```

In this case lb.B will be a matrix holding the full coefficient vector for each s (one column per s value); lb.crit, R2 and lb.BIC will be column vectors holding the LASSO criterion, R^2 and BIC value for each s; and the bundle will contain these additional items:

- lfmin: The s value which produces the smallest BIC value.
- idxmin: The 1-based index value of lfmin in the vector passed as lfrac.

The BIC (Schwarz, 1978) is calculated by regls as $-2\ell(\hat{\beta}) + k^*(\lambda)\log n$, where $\ell(\hat{\beta})$ is the log-likelihood, based on the sum of squared residuals, and $k^*(\lambda)$ the number of non-zero coefficients for the given λ. This criterion provides a guide (though certainly not an infallible one) to the likely out-of-sample performance of a model: smaller values of BIC are better. Note that the LASSO criterion itself does not offer such a guide (it is likely to decrease monotonically along with λ), but it can be useful in comparing the effectiveness of minimization algorithms (see Appendix A).

When multiple λ values are specified, the vector nzb and list nzX refer to the non-zero coefficients and associated regressors obtained with $s =$ lfmin (the BIC minimizer). Out-of-sample predictions using this s can be obtained via lincomb(nzX, nzb).

4 Optimizing via cross validation

Searching for optimal λ over the entire training sample we run the risk of overfitting. The standard remedy is to divide the training data into "folds" and do cross validation. The algorithm is then (in pseudo-code):

```
for each s value, s(j)
  MSE(j) = 0
end
for each fold, f(i)
  set the estimation sample to the complement of f(i)
  for each s value, s(j)
    perform regularized estimation using s(j) and predict for f(i)
    MSE(j) <- MSE(j) + MSE for f(i)
  end
end
```

We then perform regularized estimation on the full training data using the s value that yields the least total MSE on the above procedure (or perhaps take an alternative approach—see below).

The basic options connected with cross validation (to be entered in the parameter bundle passed to regls) are as follows:

- xvalidate: Boolean, trigger for doing cross validation (required).
- nfolds: Integer, the number of folds (optional, default 10).
- randfolds: Boolean, whether the folds should be assigned randomly (optional, default 0).

At present the folds are either assigned at random or (by default) they are sequences of consecutive observations. It may be worth adding a facility to set the folds via a predefined series. A further point: at present the folds are by construction all the same size—the result of integer division of the number of training observations by the number of folds, which means that any "remainder" training observations are ignored. That could be generalized if it seems worthwhile.

When cross validation is specified regls will print some information on the performance of the values of s used, a snippet of which is shown below:

```
         s         MSE          se
  1.000000    1.000000    0.063336
  0.615848    0.870633    0.057432
  0.379269    0.759641    0.048581
  0.233572    0.694237    0.043515
```

The MSE value is the mean across the folds, and se is its standard error, computed as per glmnet.

While it would seem most natural to select for further prediction the s value that minimizes MSE on cross validation—call this s^*—glmnet suggests an alternative policy: select the largest s that delivers an MSE within one standard error of the minimum, which we'll call $s^\dagger$. It may be that s^* and $s^\dagger$ are the same value, but if not this policy gives the benefit of the doubt to parsimony.

After cross validation, regls by default stores the full coefficient matrix (one column per value of s, estimated on the full training data) under the key B. And the returned bundle also holds the indices of both s^* and $s^\dagger$, under the keys idxmin and idx1se respectively. One can therefore select the desired set of coefficients and obtain fitted values using the full input list X—with const prepended unless your data are pre-standardized.

```
matrix optimal_b = lb.B[,idxmin] # or idx1se, or other column
list All = const X
series fitted = lincomb(All, optimal_b)
```

Note that B has as many rows as X has members, plus one for the intercept.

But there's another method which may be more convenient: after cross validation the regls return bundle holds a single nzb vector and nzX list, such that fitted values can be obtained thus:

```
series fitted = lincomb(lb.nzX, lb.nzb)
```

By default it's the s^* (idxmin) column that's selected for forming nzb, but if you wish to use $s^\dagger$ you can arrange for that by setting use_1se to a non-zero value in the parameter bundle passed to regls, as in

```
parms.use_1se = 1
```

Further cross validation options

Some additional cross validation options are supported.

- seed: Integer, you can supply this to control the randomization when randfolds is active, hence getting exactly repeatable results.

- single_b: Boolean. If non-zero it stops regls from estimating coefficients for all s values after cross validation; only the selected "best" value (s^* or $s^\dagger$) is used. The matrix B mentioned above then holds just a single column. This may shave a little off the execution time.

5 Execution speed

According to the discussion in section 3.2.2 of Boyd *et al.* (2010): the ADMM algorithm is reliable but is known *not* to be fast (or not if accurate results are wanted). However, we have been able to accelerate ADMM to the point where execution time is unlikely to be an issue, by two main means.

- We implemented the suggestion in section 3.4.1 of Boyd *et al.* (2010): letting the penalty factor ρ vary across ADMM iterations to keep the magnitudes of the primary and dual residuals in rough balance. This turns out to be highly effective.

- We implemented automatic "farming out" of cross validation to multiple MPI processes (when MPI is available on the host machine). It's possible to prevent this by adding no_mpi to the parameter bundle with a non-zero value.

In one benchmark case we considered—with 1500 training observations, 101 covariates, 50 values of λ and 10 randomized cross validation folds—the execution time was about 13 seconds before making the changes mentioned above, and about 1.5 seconds thereafter.[2]

6 Additional ADMM controls

This section describes some additional controls over the ADMM algorithm that can be passed to the regls function via the parms bundle. Under the key admmctrl you can supply a 3-vector whose elements are, in order:

- rho: a positive real number, the initial ADMM penalty parameter. It seems that $\rho = 8.0$ works well but higher or lower values might produce faster convergence in some cases.

- reltol: the relative tolerance used in gauging whether the algorithm has converged sufficiently.

- abstol: the absolute convergence tolerance (which will be scaled by the square root of the number of candidate regressors).

We have found that reltol and abstol values of 10^{-4} and 10^{-6}, respectively, produce reasonably accurate results in a manageable number of iterations. Setting smaller values will produce greater accuracy at the cost of more iterations. Non-positive values of these terms are ignored, so one can, for example, set a single element by passing a zero vector with just the desired term set to a positive value.

7 LASSO examples

Besides the sample script supplied with the package, more examples can be found in the directories murder, wine and fat at http://ricardo.ecn.wfu.edu/pub/gretl/lasso/. Some of these scripts incorporate comparison with glmnet. The murder-rate and wine quality examples use real-world data; the fat example is an artificial case with more regressors than observations.

Note that it's necessary to run the scripts involving randomized cross validation several times to get a good idea of what's going on: in each case there seem to be a few "favoured solutions" of varying probability. Sometimes one sees regls finding the better one, sometimes glmnet.

[2] On a desktop machine with 4 Intel i7 processors, running Linux.

8 Ridge regression

While LASSO involves ℓ_1 regularization, Ridge uses ℓ_2: the penalty factor λ applies to the sum of squared coefficients, giving rise to the following objective:

$$\min_{\hat{\beta}} \quad \sum_{i=1}^{n}(y_i - X_i\hat{\beta})^2 + \lambda \sum_{j=1}^{k} \hat{\beta}_j^2 \tag{3}$$

In consequence, although a large value of λ will shrink Ridge estimates substantially relative to OLS it will not send any coefficients to exactly zero as does LASSO. If the X matrix exhibits strong collinearity, LASSO will tend to eliminate most of the collinear terms while Ridge will tend to distribute the predictive weight across the terms, yielding several small coefficients instead of one relatively substantial coefficient and a bunch of zeros.

To get the `regls` function to perform Ridge regression rather than LASSO, set a value of 1 under the key `ridge` in the `parms` bundle, as in

```
parms.ridge = 1
```

Most of the points made above with respect to LASSO carry over to Ridge. The same three modes of operation described in sections 2 to 4 (from estimation using a single value of λ to cross-validation with as many values as you like) are available.

There is an important difference, however, in respect of the calibration of λ. In the LASSO case there's an easily computed $\lambda_{\max}$ $(= \|X'y\|_\infty)$ which just suffices to force all slope coefficients to zero and so, as explained above, the user is asked to express the LASSO penalty as a fraction of this maximum. In the case of Ridge there is generally no finite λ that will drive all coefficients to zero and so no "natural" maximum to serve as a benchmark. We therefore offer the user three options for the specification of "`lfrac`," controlled by the integer-valued parameter `lambda_scale`:

- `lambda_scale` = 0: no scaling is performed. The "`lfrac`" values are taken as actual λ values (and so do not have to be bounded by 1.0 above).
- `lambda_scale` = 1 (the default): we emulate glmnet. The largest value of λ is set to 9.9×10^{35}, which will drive all coefficients to near-zero. The second-largest λ (call it λ_2) is then set to 1000 times $\|X'y\|_\infty$, and subsequent values in the sequence are scaled in relation to λ_2.
- `lambda_scale` = 2: we follow the suggestion of some practitioners, setting $\lambda_{\max}$ to the squared Frobenius norm of X, which will not drive all coefficients to near-zero but will impose substantial shrinkage in relation to OLS.

To be clear on the action of options 1 and 2 for `lambda_scale`, suppose our `lfrac` specification is

```
lfrac = {1, 0.5, 0.25, 0.125}
```

Then if `lambda_scale` = 1 this translates to

```
lam2 = 1000 * infnorm(X'y)
effective_lambda = {9.9e35, lam2, 0.5*lam2, 0.25*lam2}
```

while if `lambda_scale` = 2 it becomes

```
lam1 = tr(X'X) # Frobenius norm squared
effective_lambda = {lam1, 0.5*lam1, 0.25*lam1, 0.125*lam1}
```

Note that the relevant matrix norms are computed after standardization.

In addition to `BIC` and `R2` the return bundle from Ridge regression contains `edf` (a scalar if a single λ is specified, otherwise a column vector). This is the "effective" degrees of freedom, or number of free parameters, calculated via the SVD of the matrix of regressors:

$$\text{edf} = \sum_{i=1}^{k} \frac{\sigma_i^2}{\sigma_i^2 + \lambda} \tag{4}$$

where the σ_is are the singular values. As a measure of the "size" of a model this takes the place of the number of non-zero coefficients in LASSO.

In the case of a single λ, when estimation is performed using the default SVD method, further information is available: the return bundle contains the covariance matrix of the parameter estimates (other than the constant) under the key `vcv`. And if the `verbosity` option is set to 2 you get a printout of the model, showing standard errors, z statistics and P-values.

9 The CCD option

As stated above, the default algorithms used by regls for LSSO and Ridge are ADMM and SVD, respectively. However, you have the option—for both LASSO and Ridge—of using the Cyclical Coordinate Descent (CCD) algorithm, as employed by glmnet. This is governed by two additional keys in the `parms` bundle:

- `ccd`: boolean, default 0. Set this to 1 to use CCD.
- `ccd_toler`: a positive scalar setting the convergence tolerance for CCD. The default is 10^{-7} (as in glmnet); setting a smaller value will give greater accuracy at the expense of more iterations.

Using CCD will give results that are more directly comparable with glmnet. Beyond that, practitioners are likely to ask, how do the algorithms compare in terms of speed and accuracy? This question is addressed in detail in Appendix A. The short answer is that CCD at its default tolerance is faster but somewhat less accurate than ADMM and SVD. By tightening the CCD tolerance one can generally close the accuracy gap; this may or may not reverse the ranking in terms of speed.

10 Elastic net

As mentioned above, elastic net is a hybrid of LASSO and Ridge. It employs a combination of ℓ_1 and ℓ_2 penalties governed by a hyperparameter $0 \leq \alpha \leq 1$. The objective is

$$\min_{\hat\beta} \quad \frac{1}{2}\sum_{i=1}^{n}(y_i - X_i\hat\beta)^2 + \lambda\left(\frac{1-\alpha}{2}\sum_{j=1}^{k}\hat\beta_j^2 + \alpha\sum_{j=1}^{k}|\hat\beta_j|\right)$$

Thus $\alpha = 1$ gives LASSO, $\alpha = 0$ gives Ridge, and anything between gives a combination. It has been argued that better out-of-sample prediction can be obtained in some cases by preserving some highly collinear regressors à la Ridge, while sending some coefficients to exact zero as in LASSO, and elastic net allows for this.

In the `regls` function, elastic net is selected by specifying a fractional value under the key `alpha` in the parameters bundle. This automatically switches to the CCD algorithm (section 9), so the `ccd_toler` option becomes applicable.[3]

When elastic net is used on a sequence of λs without cross validation (see section 3) `regls` provides a BIC measure as a possible means of selecting the most promising penalty factor. This requires calculation of the effective number of parameters (degrees of freedom), for which we use the method specified in Zou and Hastie (2005).

Note that if cross validation is called for with elastic net, it is only the λ value that is optimized. To assess the efficacy of various α values one would have to perform several cross validation runs.

[3] In principle the ADMM algorithm could handle elastic net, but to date we have not implemented such support.

11 Reference: public functions

```
bundle regls (series y, list X, bundle parms)
```

Performs LASSO, Ridge or Elastic net estimation given the dependent variable `y`, the regressors `X`, and options in `parms`. Returns a bundle containing the results. Table 1 lists the parameters that can be passed via the `parms` argument.

`lfrac`	scalar or vector	λ-fraction(s)
`nlambda`	integer	number of automatic λs
`stdize`	0/1, default 1	standardize the data
`ridge`	0/1, default 0	do Ridge regression
`lambda_scale`	0, 1 or 2, default 1	see section 8
`verbosity`	0, 1, 2 or 3, default 1	printing of output
`xvalidate`	0/1, default 0	do cross validation
`nfolds`	optional integer > 1, default 10	number of folds
`randfolds`	0/1, default 0	use random folds
`use_1se`	0/1, default 0	see section 4
`seed`	optional integer	controls random folds
`single_b`	0/1, default 0	see section 4
`no_mpi`	0/1, default 0	see section 5
`admmctrl`	optional control vector	see section 6
`ccd`	0/1, default 0	see section 9
`ccd_toler`	positive scalar, default 10^{-7}	see section 9
`alpha`	$0 \leq \alpha \leq 1$ (default 1)	see section 10

Table 1: Summary of parameters for the `regls` function

```
matrix lambda_sequence (scalar lmax, int K, scalar eps[0.0001])
```

Produces a column vector holding a logarithmic sequence of `K` values running from `lmax` to `eps`. It is required that $0 <$ `lmax` ≤ 1 and $0 \leq$ `eps` $<$ `lmax`. In context such values are interpreted as instances of $s = \lambda/\lambda_{\max}$.

```
matrix regls_get_stats (const series y, const series yhat)
```

Returns a 2-vector holding $\text{MSE} = \sum(y - \hat{y})^2/n$ and $R^2 = 1 - \sum(y - \hat{y})^2/\sum(y - \bar{y})^2$.

```
scalar regls_pc_correct (const series y, const series yhat)
```

Returns the percentage of cases in which `yhat` rounded to the nearest integer equals `y`. Useful only when `y` is integer-valued.

Listing 1: Usage of `regls_multiprint`

```
set verbose off
include regls.gfn
open murder.gdt --quiet --frompkg=regls

# all available predictors w. no missing values
list X = population..LemasPctOfficDrugUn
smpl 1 800
bundle parms = defbundle("nlambda", 8, "verbosity", 0)
bundle lb = regls(murdPerPop, X, parms)
regls_multiprint(lb.B, lb.lfrac, murdPerPop, X, lb.crit)
```

```
matrix regls_foldvec (int nobs, int nf)
```

Returns a column vector of length `nobs` in which `nf` successive blocks of length `nobs/nf` take on the values 1, 2,. . . , `nf`, respectively. Useful only for composing a `folds` vector than can be passed to glmnet for comparison with gretl when consecutive folds are used in cross validation.

```
void regls_multiprint (const matrix B, const matrix lam,
                       const series y, list X,
                       const matrix crit[null])
```

Given a $k \times m$ (or $k + 1 \times m$) coefficient matrix `B`, an m-vector of λ values `lam`, a dependent variable `y` and a list `X` of k regressors, prints a summary table with m rows showing R^2, the sum of absolute values of the coefficients, and df (the number of non-zero coefficients) associated with each λ. It is presumed that `B[,j]` holds LASSO estimates produced using `lam[j]`. If the optional `crit` matrix is provided it should be an m-vector holding the criterion value associated with each λ, which will be shown as a fifth column.

As in the `regls` function, the `X` list should not contain a constant: if the `B` matrix has one more row than `X` has members, the constant is handled automatically. Listing 1 shows how the information provided in the `regls` return bundle can be used to supply the arguments `B`, `lam` and `crit`.

```
series glmnet_pred (matrix *Rb, list X)
```

Convenience function for handling results retrieved by gretl from glmnet. On entry `Rb` should hold the full coefficient vector (including any zeros) and `X` the full list of candidate regressors, and the return value is the result of `lincomb(X, Rb)`. On exit `Rb` holds only the non-zero coefficients, with row-names added based on the `X` list. This is then comparable with gretl's `nzb`.

12 Change log

Version 0.3, 2021-04-17: fix bug with Ridge verbose printout, and replace some tables with figures in the doc for ease of comprehension of comparative experiments.
Version 0.2, 2021-01-29: support a higher verbosity level for GUI use; improve printout for LASS0 coefficients, when applicable.
Version 0.1, 2020-10-09: initial release.

References

Boyd, S., N. Parikh, E. Chu, B. Peleato and J. Eckstein (2010) 'Distributed optimization and statistical learning via the Alternating Direction Method of Multipliers', *Foundations and Trends in Machine Learning* 3(1): 1–122. URL https://dl.acm.org/doi/10.1561/2200000016.

Friedman, J., T. Hastie and R. Tibshirani (2010) 'Regularization paths for generalized linear models via coordinate descent', *Journal of Statistical Software* 33(1): 1–22. URL https://www.jstatsoft.org/article/view/v033i01/v33i01.pdf.

Schwarz, G. (1978) 'Estimating the dimension of a model', *Annals of Statistics* 6: 461–464.

Tibshirani, R. (1996) 'Regression shrinkage and selection via the lasso', *Journal of the Royal Statistical Society, Series B* 58(1): 267–288. URL https://www.jstor.org/stable/2346178.

Zou, H. and T. Hastie (2005) 'Regularization and variable selection via the elastic net'. Department of Statistics, Stanford University. URL https://web.stanford.edu/~hastie/TALKS/enet_talk.pdf.

Appendix A Comparison of algorithms

This appendix reports some experiments designed to gauge the accuracy and speed of the Cyclical Coordinate Descent (CCD) algorithm as compared to the default algorithms in regls—ADMM for LASSO and SVD for Ridge.

Design of experiments

The general design of our experiments is as follows. For some selected dataset and sample range we compute the values of the minimized objective function (LASSO or Ridge) for a sequence of λ values. In each test we compare two optimization methods—call them A and B—taking A as the baseline and exploring how the difference in results between the methods behaves as we tighten the convergence tolerance for method B. We assume that for the iterative methods ADMM and CCD, tightening the tolerance (within reason) will produce more accurate results, or at least will not produce less accurate results. Therefore, if the difference diminishes as tolerance is tightened for B we can infer that A was more accurate at the initial tolerance level.

We measure the difference between sets of results via Euclidean distance.[4] To gauge the trade-off between accuracy and speed we also record the execution time for method B divided by that for A.

In the experiments reported below we used the murder rates dataset (`murder.gdt`) supplied with the `regls` package, with `murdPerPop` as dependent variable and 101 regressors.[5] The λ sequence was of length 20. In the LASSO tests we used the first 800 observations and in the Ridge tests the first 1500 (to make the test take longer and improve the resolution of the timer). We're aware that the results we show are liable to be data- and model-dependent and we offer some comments on this in the concluding section.

Ridge: SVD and CCD

The Ridge problem has an analytical solution and regls implements this using Singular Value Decomposition, which is generally regarded as the gold standard for accuracy in digital computation. Results obtained via SVD can therefore be used as a benchmark against which to assess the accuracy of the solution provided by CCD.

Figure 1 shows our findings. The distance between the two sets of results declines monotonically as the CCD tolerance is tightened, as one would expect. At its default tolerance of 10^{-7} CCD is faster than SVD (in this example by about 30 percent), but it takes longer than SVD if one wants the extra accuracy associated with a tolerance of 10^{-9} or less. Note, however, that there seems to be little point in reducing the CCD tolerance below 10^{-9}.

LASSO: ADMM and CCD

LASSO is trickier than Ridge in that there's no analytical solution to serve as a natural benchmark. In this case we take ADMM (at its default tolerance in regls) as baseline—without assuming its results are "correct"—and see what happens.

Figure 2 shows monotonic decline in difference of results as the CCD tolerance is tightened from 10^{-7} to 10^{-10}, at which point the results become practically indistinguishable. We interpret this to mean that ADMM at its default settings produces results that can be taken as "correct" for practical purposes.

Notice that with LASSO the effect of tighter tolerance on the execution time for CCD relative to the baseline is a good deal more marked than in the Ridge case. When the CCD tolerance is reduced from 10^{-7} to 10^{-9} Ridge time becomes slightly greater than SVD time, while LASSO time becomes over twice that of ADMM.

If ADMM is more accurate than CCD at their respective default tolerances, can we find tolerances for the former that produce similar accuracy to the CCD default? And if so, what happens to the speed

[4] We also tried Mean Absolute Deviation but this did not seem to contribute additional information.

[5] This dataset was referenced by Ryan Tibshirani in the LASSO context; see https://www.stat.cmu.edu/~ryantibs/datamining/lectures/17-modr2.pdf.

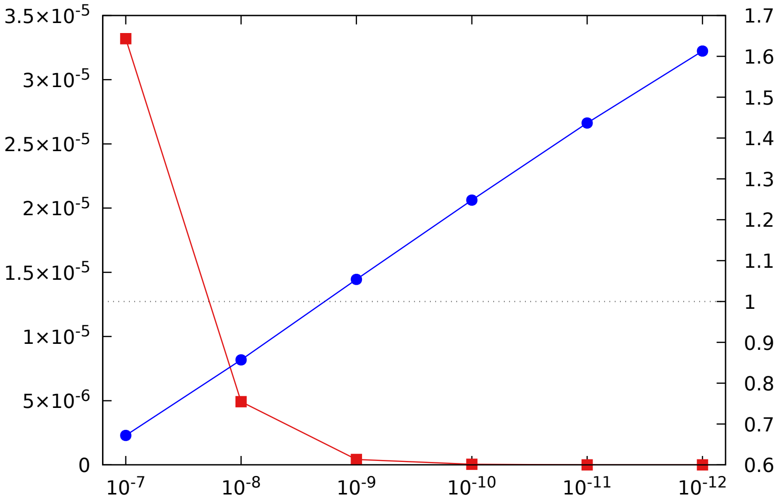

Figure 1: Ridge regression: CCD performance relative to SVD baseline. CCD tolerance on x-axis, Euclidean distance between estimates in red (left) and relative execution time in blue (right).

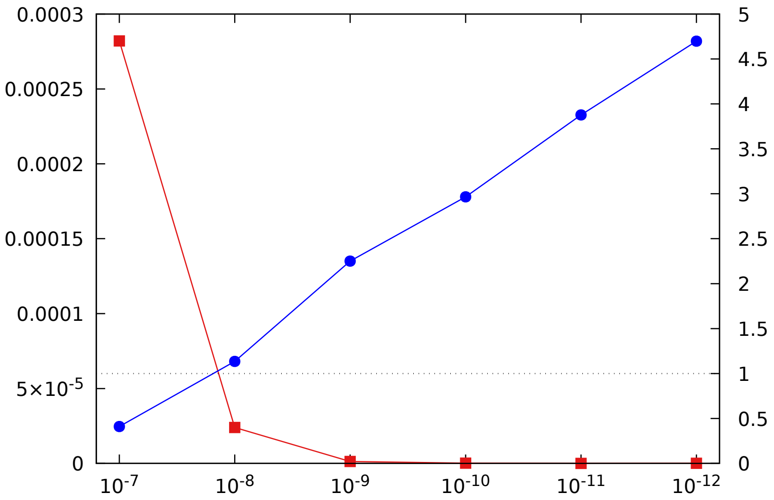

Figure 2: LASSO estimation: CCD performance relative to ADMM baseline. CCD tolerance on x-axis, Euclidean distance between estimates in red (left) and relative execution time in blue (right).

comparison?

Figure 3 shows the results of a relevant experiment. The integers, i, on the x-axis represent progressively slacker tolerance pairs, $(i \times 10^{-2}, i \times 10^{-4})$, for ADMM. (Note that the $i = 1$ already gives values 100 times greater than the ADMM default.) As expected, greater tolerances correspond to shorter execution times (blue line, right-hand scale) and increasing distance from the baseline high-accuracy ADMM results (red line, left-hand scale). CCD, at its default tolerance of 10^{-7}, enters the picture in two ways: its execution speed is by construction 1 on the right-hand scale, while its deviation from the baseline estimates is shown by the dotted red line.

In this example there is a range of the ADMM tolerances, comprising $i = 2$ and $i = 3$, over which ADMM is both faster and more accurate than CCD.

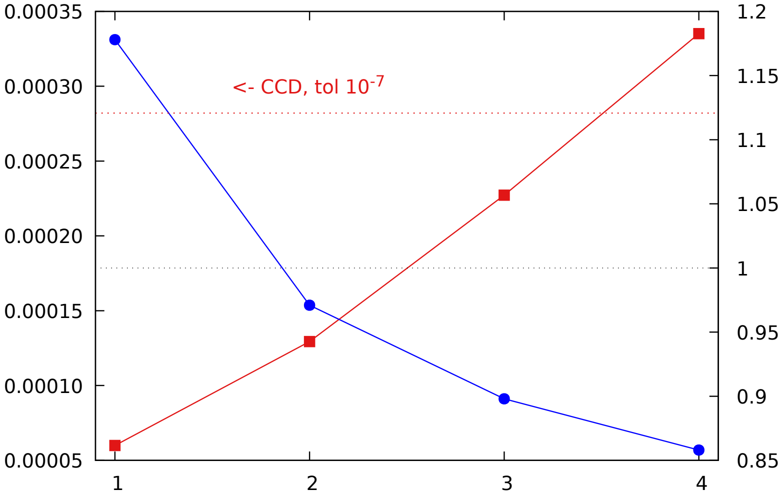

Figure 3: LASSO estimation: ADMM performance at slacker tolerances. See text for explanation of x-axis. Euclidean distance from high-accuracy estimates in red (left) and time relative to CCD in blue (right).

As noted above, such figures are likely to be data- and model-dependent, but we conjecture that ADMM tolerances of $(10^{-2}, 10^{-4})$ are conservative relative to CCD at tolerance 10^{-7} in the sense that they are likely to deliver results of equal accuracy to CCD or better.

Conclusion

The results shown above, from a single dataset, are obviously illustrative rather than definitive. On the strength of similar tests on other datasets we're able to say something about what is generally applicable and what is variable.

In all of our experiments CCD at its default tolerance is faster but less accurate than SVD for Ridge regression, and faster but less accurate than ADMM (at its default tolerance) for LASSO. And in all cases the accuracy of CCD can be increased (up to a point) by reducing its tolerance.

Two things are relatively variable (apparently depending on, among other things, the number of observations and the number of regressors, though not in any easily predictable way).

- The time taken by CCD relative to the alternatives as a function of the CCD tolerance. In some cases (unlike the example above) CCD retains its speed advantage as its tolerance is reduced. While CCD is bound to slow down some at tighter tolerance it may still be the fastest method.
- Convergence of CCD is not guaranteed. In a few LASSO trials we saw failure at, for example, a tolerance of 10^{-10}, when ADMM had converged OK and the difference statistics seemed to show room for further improvement of accuracy on the part of CCD. This suggests that CCD is not

always capable of accuracy equal to ADMM. It's possible that in other cases this could be reversed (ADMM unable to equal the accuracy of CCD), though we didn't see any such in our trials.

So here's our conclusion. (As a warning to the reader we have emphasized the words that signal our remaining uncertainty!) If you want maximally accurate results you should use SVD for Ridge and *probably* use ADMM for LASSO. You can *usually* get equal accuracy from CCD if you tighten its tolerance far enough but then CCD *may* take longer than the alternatives. On the other hand, if you reckon the accuracy of CCD at its default tolerance is good enough for practical purposes you can save time by using it. Unless, in the case of LASSO, you'd like to set the ADMM tolerances to $(10^{-2}, 10^{-4})$, in which case you *may* get somewhat more accurate results with little difference in execution time.

To go any further we would have to assess what's "good enough" accuracy (for example, with out-of-sample prediction in view). Does the extra accuracy of ADMM and SVD actually help, or is it surplus to requirements? We have something to say about that in Appendix B.

Appendix B Comparison with glmnet

Given the benchmark status of R's glmnet we have tried to ensure that our results are very close to those from glmnet unless we can demonstrate a good reason for divergence. We comment below on reasons why results may differ in certain respects.

Different conventions

It should be noted that regls and glmnet employ different conventions in some respects. This does not affect the comparison of reported coefficients or predicted values, but it can make comparison of λ values a little awkward. The LASSO objective function and definition of $\lambda_{\max}$ used by regls were stated in section 1, but to be fully explicit we should say that the X and y in equation (2) for the maximum λ are taken to be standardized values.

In glmnet the objective (in the linear Gaussian case) is

$$\min_{\hat{\beta}} \quad \frac{1}{2n}\sum_{i=1}^{n}(y_i - X_i\hat{\beta})^2 + \lambda\sum_{j=1}^{k}|\hat{\beta}_j|$$

This differs from our equation (1) in dividing the sum of squared residuals (SSR) by $2n$ rather than 2. Since glmnet is not actually using a different relative weighting of the SSR and the sum of absolute coefficient values, it follows that their "λ" must be read as n^{-1} times ours. Moreover, while we take each λ_i value to be s_i times $\lambda_{\max}$ as defined in equation (2), the λ values printed by glmnet are (in our notation)

$$\lambda_i = s_i \cdot \lambda_{\max} \cdot \hat{\sigma}_y / n$$

where $\hat{\sigma}_y$ is the ML estimate of the standard deviation of the dependent variable. To obtain the glmnet λ corresponding to a given s one can do:

```
Rlam = s * b.lmax * sdc({y}) / b.nobs
```

where b is a bundle obtained via regls on the same data, y is the dependent series and sdc({y}) gives $\hat{\sigma}_y$. The current sample range must be the same as for b to get $\hat{\sigma}_y$ right, but if glmnet was told *not* to standardize the data then this term should be omitted as it is assumed to be 1.

Cross validation methodologies

There's a substantive difference between regls and glmnet in respect of cross validation. This applies even if the CCD algorithm is selected in regls, which results in near-identical results for LASSO or Ridge coefficients when simply processing a sequence of λ values.

In regls cross validation, the entire training dataset is standardized at the outset, then each fold gets its share of the standardized data. The maximum λ is also determined using the full training set and the same λ sequence is used for each fold. In glmnet, by contrast, both standardization and calculation of the λ sequence are done per fold. For example, suppose the training data are divided into 10 folds, each comprising 10 percent of the observations. Then glmnet both standardizes and computes a λ sequence using the complementary 90 percent of the data.

Extended test of cross validation

The primary point of cross validation is to determine the value of a hyperparameter (for LASSO, λ) that is likely to give best results in genuine out-of-sample prediction. In this section we describe some experiments designed to probe the impact (if any) of certain differences noted above on the efficacy of out-of-sample prediction. We are particularly interested in

- the methodological difference between regls and glmnet noted in the previous section, and
- the "extra accuracy" of the ADMM algorithm, at its default tolerance, over CCD at its default tolerance, noted in Appendix A.

As regards the methodological difference, not a great deal can be said about this *a priori*, though one thing is clear: the more homogeneous the training data, the less details of method are going to matter. If the statistical properties of the fold-complement samples are very similar to those of the full training data then the locus of standardization (training data or fold-complement) won't make much difference. In addition, if $\lambda_{\max} = \|X'y\|_\infty$ doesn't differ much across the samples the locus of calculation of the λ sequence won't matter much either, and λ-matching—if it is required—should be relatively unproblematic.

That said, real-world datasets of interest are not necessarily very homogeneous so the details *could* matter. To investigate this we ran experiment on two rather different datasets.

- Dataset 1: murder rates and covariates for US localities (`murder.gdt`, supplied with the regls package). Comprises 2215 observations on 102 variables.
- Dataset 2: white wine quality and physico-chemical covariates. Comprises 4898 observations on 12 variables (78 after adding squares and interactions of covariates).[6]

We "leveraged" the datasets by randomizing the order of the observations at each of 2000 iterations then taking the first N observations for training and the next M for testing, with $N + M$ a subset of the full data available. For Dataset 1 $N = 1200$ and $M = 200$, and for Dataset 2 $N = 1500$, $M = 500$.

The body of the test involved cross validating with 10 folds (composed of consecutive observations since the whole dataset was randomized) then predicting for the M holdout observations using the optimal λ on the "one standard error" rule favored by glmnet. This rule selects the larger of (a) the λ^* which minimizes mean out-of-sample MSE and (b) the largest λ that lies within σ^* of λ^*, where σ^* is the standard error of the minimized mean MSE.[7] The figure of merit calculated at each iteration was the R^2 for the testing data, $1 - \sum(y - \hat{y})^2 / \sum(y - \bar{y})^2$.

This test was run (with common randomization) using three variants of cross validation, each with its default settings: the glmnet function `cv.glmnet`; regls using the CCD algorithm; and regls using the ADMM algorithm. As mentioned above, there are two distinct differences at play. In comparing glmnet with regls CCD the coefficient vectors produced for given data and given λ are near-identical, and the relevant difference lies in the details of the cross validation methodology. In comparing regls CCD and ADMM the cross validation method is exactly the same and the relevant difference lies in the "excess precision" afforded by ADMM over CCD, at their respective default tolerances, as discussed in Appendix A.

Statistics for out-of-sample R^2 from the Dataset 1 experiment are shown in Table 2. In this experiment regls CCD gave better out of sample prediction than glmnet, and ADMM did a little better again. The first difference—due to cross validation methodology—appears to be more substantial than the second.

Figure 4 gives another angle on the comparisons, plotting estimated kernel densities for the three variants.[8]

[6] See https://archive.ics.uci.edu/ml/datasets/wine+quality.

[7] The mean is taken across the folds, weighted if necessary.

[8] We truncated the plot on the left to focus on the bulk of the distributions; instances of $R^2 < 0.2$ were rare, and their

	mean	s.d.	s.e.(mean)	95% C.I.	median	min	max
glmnet	0.4724	0.1518	0.0034	0.4657 - 0.4790	0.4881	-2.6289	0.7044
CCD	0.4954	0.1545	0.0035	0.4886 - 0.5022	0.5118	-2.7911	0.6900
ADMM	0.4984	0.1608	0.0036	0.4914 - 0.5055	0.5172	-2.7831	0.6925

Table 2: Out of sample R^2, 2000 replications, Dataset 1

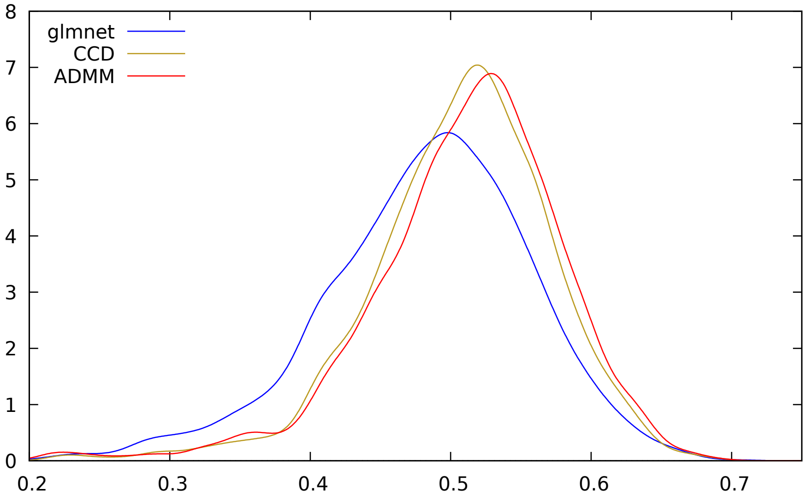

Figure 4: Estimated densities for out of sample R^2, Dataset 1

Since each mothod was given the same data at each iteration, paired-difference tests for R^2 might be considered appropriate; these are shown in Table 3, along with the correlations across the methods per iteration.

	$\|z\|$	ρ
glmnet, regls CCD	20.6	0.946
glmnet, regls ADMM	21.6	0.942
regls CCD, regls ADMM	8.4	0.996

Table 3: Paired-difference tests and correlations, out of sample R^2

From this point of view all the differences are strongly statistically significant, though the advantage of ADMM over CCD might not be considered of much practical importance.

Table 4 shows out-of-sample R^2 statistics for Dataset 2. Again ADMM has the highest mean and median, and regls CCD does better than glmnet on these citeria, but here the differences are relatively small. Kernel densities are shown in Figure 5. The displacement of the distribution between methods, in the same direction as for Dataset 1, is appreciable[9] but maybe not large enough to be of practical importance.

Dataset heterogeneity

Can we account for the difference in results between Dataset 1 and Dataset 2 by reference to the relative heterogeneity of the data? It's not obvious how such heterogeneity can best be measured, but we tried a rough and ready heuristic with focus on the dependent variable: how much do its sample statistics vary across the fold-complement samples, relative to the full training data?

frequency did not differ much by method.

[9] And statistically significant: paired-difference $|z| = 19.9$ for glmnet vs CCD and 5.8 for CCD vs ADMM.

	mean	s.d.	s.e.(mean)	95% C.I.	median	min	max
glmnet	0.2735	0.0518	0.0012	0.2712 - 0.2758	0.2775	-0.5072	0.3994
CCD	0.2763	0.0523	0.0012	0.2740 - 0.2786	0.2803	-0.5072	0.3994
ADMM	0.2774	0.0558	0.0012	0.2750 - 0.2799	0.2826	-0.5260	0.4029

Table 4: Out of sample R^2, 2000 replications, Dataset 2

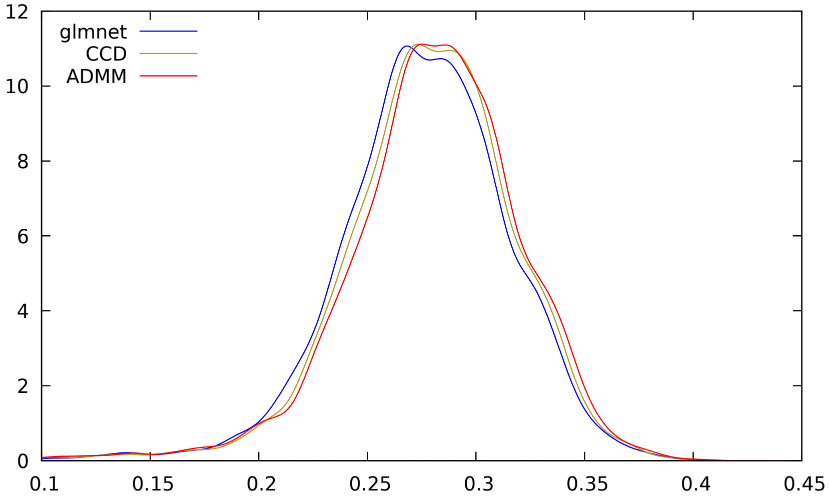

Figure 5: Estimated densities for out of sample R^2, Dataset 2

We calculated two statistics, H_μ and H_σ, for each dataset, using the randomize-and-subset procedure described above. At each of 2000 iterations, i, we summed the absolute proportional deviations of the fold-complement sample means, $\bar{y}_{ij}$, $j = 1, 2, \ldots, 10$, from the sample mean for the training data, $\bar{y}_i$. We then took the mean of these values across the iterations:

$$H_\mu = \frac{1}{2000} \sum_{i=1}^{2000} \sum_{j=1}^{10} |\bar{y}_{ij} - \bar{y}_i| / |\bar{y}_i|$$

H_σ was calculated in an exactly analogous way, with the sample standard deviations in place of the means. The values of these measures for the datasets were

	H_μ	H_σ
Dataset 1	0.12059	0.15032
Dataset 2	0.01039	0.05089

Thus it appears that—on this crude measure at least—Dataset 2 is a good deal more homogeneous than Dataset 1. This is consistent with the observation that differences in out-of-sample performance attributable to differences in the algorithms were much smaller for Dataset 2.

Conclusion

It's risky to conclude much on the strength of just two datasets—even when leveraged by randomization and subsetting. But it does seem that standardization at the level of the full training data, and employment of a single λ sequence—derived from the full training data and applied for all folds—may be conducive to best out-of-sample prediction. It also seems that the extra precision of ADMM at its default setting is helpful for out-of-sample prediction, though this effect is relatively small and may or may not be considered worthwhile.

Example script

An example script used with Dataset 1 is shown on the following page. This instance produces results for regls CCD and glmnet. Results for regls ADMM can be obtained by omitting the `ccd` setting in the `parms` bundle. To explore a different randomization one could comment out the `set seed` line, in which case the seed for the random number generator will be set from the clock on start-up.

Listing 2: Monte Carlo script for out-of-sample prediction

```
set verbose off
set R_lib on
set R_functions on
include regls.gfn
open murder.gdt --quiet --frompkg=regls

# obtain results for regls CCD and cv.glmnet

foreign language=R
  lasso_R <- function(x,y,f,nl) {
    if (! "glmnet" %in% (.packages())) {
       library(glmnet)
    }
    m <- cv.glmnet(x, y, foldid = f, family = "gaussian", alpha = 1,
      nlambda = nl, standardize = T, intercept = T)
    Rb <- as.matrix(coef(m$glmnet.fit, s = m$lambda.1se))
  }
end foreign

# all available predictors without missing values
list X = population..LemasPctOfficDrugUn
list X0 = const X # for glmnet prediction

bundle parms = _(nlambda=50, verbosity=0, ccd=1, xvalidate=1)
parms.nfolds = 10
parms.use_1se = 1

# for glmnet
matrix foldvec = regls_foldvec(1200, 10)

set seed 997361
K = 2000
matrix OSR2 = zeros(K,2)

loop i=1..K --quiet
   smpl full
   series sorter = uniform()
   dataset sortby sorter

   smpl 1 1200 # training data
   bundle lb = regls(murdPerPop, X, parms)
   matrix Rb = R.lasso_R({X}, {murdPerPop}, foldvec, 50)

   smpl 1201 1400 # testing data
   series pred = lincomb(lb.nzX, lb.nzb)
   m = regls_get_stats(murdPerPop, pred)
   OSR2[i,1] = m[2]
   series Rpred = lincomb(X0, Rb)
   m = regls_get_stats(murdPerPop, Rpred)
   OSR2[i,2] = m[2]
endloop

mwrite(OSR2, "murder_ccd.mat")
```