WORKING WITH SPARSE MATRICES IN MATLAB (AND R)

Sometimes a researcher might find themselves needing to build an extremely large matrix that only contains a small number of non-zero values. In these cases, rather than working with a normal matrix (which may even be infeasible), it is usually more efficient to work with a sparse matrix – keeping only the row indices, column indices, and values of the non-zero entries. Working in this format (as opposed to working with the full matrix) improves computational speed. This problem comes up most commonly when researchers work with high dimensional fixed effects models, but also when working with problems that involve lots of permutations of outcomes – such as multi-object auction models.

In this post, I describe the basics for working with sparse matrices, and give two examples:

1) a high dimensional fixed effects linear regression model, and

2) a non-linear model with high dimensional fixed effects.

For example, suppose you work with the Panel Study of Income Dynamics, and want to include all N = 9000 family fixed effects somewhere in your model. As the PSID contains around 40 waves, this requires building matrix D, an NT x N = 360000 x 9000 matrix of dummy variables with a single non-zero entry in each row. This is infeasible. Instead, we can build the matrix in its `raw sparse' format, writing D = [rows, columns, values] containing the row indices, column indices, and values of only the non-zero entries. In a fixed effects model, with id_var as our id variable, this would be given by:

```
D = [1:NT,id_var,ones(NT,1)];
```

At this point, if you need to perform relatively complex matrix operations on D, such as matrix multiplication, inversion, or decomposition, the most efficient course of action is to use Matlab's inbuilt sparse matrix functionality. You can declare D a sparse matrix by using the "sparse" function:

```
D = sparse(rows, columns, values, M, N);
```

Where M and N are the intended dimensions of the matrix D. Although it is not necessary to specify the dimensions, it is good practice to do so in case the matrix has a number of all zero rows or columns on the end.

In spite of the inbuilt sparse matrix functionality, it is sometimes more efficient to work with the `raw sparse' matrix (as opposed to the sparse matrix, having called the sparse function). This is for two reasons: First, because there are some things one cannot do using the inbuilt functionality – such as the "repmat" or "permute" functions or working with 3 (or larger) dimensional arrays. Second, and most importantly, the "sparse" function can occasionally be relatively slow – especially when you need to use it within a function that will be evaluated many times (such as a likelihood or loss function). If you only need to perform simple matrix operations, such as multiplying by a vector as in fixed effects models, it is often more efficient to work in raw sparse form.

Suppose you want to find X = aD + b, where a and b are scalars. This is very simply given by:

```
X = [rows, columns, a* values + b];
```

Suppose you want to find X = Dc where c is a conformable vector. First, recognise that c(columns) gives the corresponding entries of c that each element of D will be multiplied by. Next, see that Dc^T = [rows, columns, values c(columns)]. We then just need to sum across columns, which can be done efficiently using accumarray to sum within rows:

```
Dc = accumarray(rows,values.*c(columns), M);
```

Interestingly, this actually outputs a non-sparse matrix. This is even easier in the fixed-effects case. When each row contains only a single non-zero value we can write Dc = c(id_var);

Example 1: High Dimensional Fixed Effects Regression. (Note that the algorithm described below is less efficient than other algorithms one can find, albeit much simpler).

Suppose we have a model given by $y = X\beta + D\alpha + \varepsilon$, where X is an NT x K vector of covariates of interest, β is a K x 1 vector of coefficients (with K being small), and α is our N x 1 vector of fixed effects. By applying Frisch-Waugh-Lovell we know that the unique least squares solution is given by:

$$\hat{\beta} = (X^TX)^{-1}X^T(y - D\hat{\alpha})$$
$$\hat{\alpha} = (D^TD)^{-1}D^T(y - X\hat{\beta})$$

A common way to solve this problem is using the `zig-zag' algorithm – beginning with a $\hat{\beta}_0$ and $\hat{\alpha}_0$, iteratively update the two equations until convergence is achieved. Unfortunately this requires constructing both $D\hat{\alpha}$, $D^T(y - X\hat{\beta})$, and $(D^TD)^{-1}$. Fortunately, we can use the following code, exploiting the structure of D. Call the vectors $\hat{\beta}$ and $\hat{\alpha}$ alphahat and betahat respectively. First, $D\hat{\alpha}$, which I call Dalpha:

```
Dalpha = alphahat(id_var);
```

Second, $D^T(y - X\hat{\beta})$ which I call Dy_Xbeta. Essentially, all we do is sum $(y - X\hat{\beta})$ by id_var.

```
Dy_Xbeta = accumarray( id_var, y-X*betahat, [N 1]);
```

Finally, $(D^TD)^{-1}$. As an intermediate step, consider (D^TD), which I call DD:

```
DD =  diag(accumarray( id_var, 1, [N 1]));
```

As expected, this is just a diagonal matrix with the number of observations of each id on the diagonals. Therefore we can use:

```
DD_inv = diag(1./accumarray( id_var, 1, [N 1]));
```

If the panel is balanced, this is equivalent to:

```
DD_inv = (1/T)*eye(N);
```

It is also worth recognising that weights can be included very easily here.

Example 2: High Dimensional Fixed Effects non-linear model.

Suppose we have a model defined, perhaps, as some sort of extremum:

$(\hat{\beta}, \hat{\alpha}) = \arg\min\{ F(y, X, D\alpha; \beta)\}$

Where F() gives the known loss function to be minimised. In general, there will not be a closed form solution, so researchers make use of numerical minimisation.

Evaluating the loss function at $(\hat{\beta}, \hat{\alpha})$ requires calculating $D\alpha^k$, which we established previously was just given by alphahat(id_var).

Where possible, it is always good to make use of an analytic gradient (especially in the presence of high dimensional fixed effects). This requires that we supply an expression for the Jacobian of the loss function, given by:

$$J = \begin{bmatrix} \nabla_\beta F \\ D^T \nabla_{D\alpha} F \end{bmatrix}$$

Evaluating the Jacobian requires calculating $D^T \nabla_{D\alpha} F$. This can be done efficiently by first evaluating F_Dalpha = $\nabla_{D\alpha} F$ (y, X , Dα; β)$, then summing this by id_var. Writing F_Dalpha() for the known first derivative of F with respect to Dα :

```
J_Dalpha = accumarray(id_var, F_Dalpha( y, X, Dalphahat; betahat), [N 1]);
```

Note that everything described above can also be easily transcribed into R. R even has the "Matrix" package with similar functionality to Matlab's inbuilt "sparse" functionality. For example, putting D into sparse form from raw sparse can be done using:

```
D <- sparseMatrix(i = rows, j = columns, x = entries, dims = list(M, N))
```

Samuel Altmann, DPhil Candidate in Department of Economics, Wadham College
27 April 2021