PS C236A/ Stat C239A

Erin Hartman

October 13, 2010

Genetic Matching

1 The Matching part of `GenMatch()`

1.1 Mahalanobis Distance

GenMatch is a generalized version of the Mahalanobis distance. The idea is that in cases in which Mahalanobis distance is not optimal, one can search the space of all distance metrics to find an optimal distance metric. The Mahalanobis distance is defined as:

$$d_m(X_i, X_j) = \left\{(X_i - X_j)^T S^{-1} (X_i - X_j)\right\}^{\frac{1}{2}}$$

where S is the sample covariance matrix of X. Notice that if S is the identity matrix, then this collapses to the Euclidean distance and if it is diagonal then it is a normalized Euclidean distance. Mahalanobis distance and GenMatch are both affinely invariant matching methods.

1.2 EPBR

The proofs for Mahalanobis distance matching are based on ellipsoidal distributed distributions (normal, t, etc.). When the Xs are distributed ellipsoidally, the property of equal percent bias reduction (EPBR). We say that a matching procedure is EPBR if:

$$\frac{\eta_1 - \eta_{2*}}{\eta_1 - \eta_2} = \gamma$$

for a scalar $0 \leq \gamma \leq 1$, where η_1 and η_2 refer to the original mean vectors, and η_{2*} refers to the matched mean vector. Note that this implies that:

$$(\eta_1 - \eta_{2*}) = \gamma\{(\eta_1 - \eta_2)\}$$

$$\Rightarrow \mathbb{E}_1((X)) - \mathbb{E}_{2*}((X)) = \gamma\{\mathbb{E}_1((X)) - \mathbb{E}_2((X))\}$$

which means that a matching method is EPBR if the bias in each coordinate x of $\mathbf{X}$ is reduced by the same percentage by the matching method. If EPBR does not hold, then it can be proven that there is some vector w for which the matching increases the bias for some linear function of $\mathbf{X}$.

However, while EPBR is a nice property, it is only of limited value if the mapping between $\mathbf{X}$ and Y is non-linear. Also, there may be a substantive reason to believe that getting good balance on some covariates is more important than others, in which case we'd rather optimize balance on some covariates rather than have EPBR hold.

1.3 GenMatch

GenMatch overcomes the problems of Mahalanobis distance by searching over the space of distance metrics by adding a weight matrix to generalize the Mahalanobis distance.

$$d_{gm}(X_i, X_j) = \left\{(X_i - X_j)^T (S^{-\frac{1}{2}})^T W (S^{-\frac{1}{2}})(X_i - X_j)\right\}^{\frac{1}{2}}$$

where W is a $k \times k$ positive definite weight matrix and $S^{1/2}$ is the Cholesky decomposition of S. It is an affinely invariance matching algorithm that uses the above distance metric, where all off-diagonal elements of W are zero. It searches only for the diagonal in order to save computational power. The diagonal elements are chosen to satisfy a certain loss criterion, with the default being to minimize the maximum discrepancy. However, the function allows for the user to pass in any loss criterion that they desire.

It is important to note here that the GenMatch distance simplifies to the Mahalanobis distance metric if W is the identity matrix. If the p-score is passed in as the first column of X, and all other columns of X are covariates orthogonalized to the pscore, then if the pscore is correct, GenMatch will approach the pscore. However, the mean squared error will be lower.

2 The Genetic part of `GenMatch()`

GenMatch is based on the optimizer `genoud` written by Mebane and Sekhon. Genetic algorithms are one way to solve the difficult optimization problem we face of choosing the optimal weights in a possibly irregular response surface, and they a type of evolutionary algorithm. Evolutionary algorithms, using the biological notion of evolution, modify a population of potential solutions so that each generation, on average, tends to have a better "fit" than the previous generation. One advantage of genetic algorithms is that they do not require derivatives to exist, as many other optimizers do, nor do they rely on continuous functions. The success of a genetic algorithm depends on a sufficiently large population of code-strings (in the case of GenMatch(), the code string is a vector of pvalues).

GenMatch uses the following "reproductions" and "mutations" to create new code-strings in successive generations:

- *cloning* makes copies of the best trial solution in the current generation
- *uniform mutation* changes one parameter in the trial solution to a random value uniformly on a specified domain
- *boundary mutation* replaces one parameter with one of the bounds of the domain
- *non-uniform mutation* shrinks one parameter towards the bounds, with the shrinkage depending on the number of generations that have been performed

- *whole non-uniform mutation* non-uniform mutation for all parameters in a trial solution
- *heuristic crossover* uses two trial solutions to produce a new solution
- *polytope crossover* computes a trial solution as a convex combination of as many trial solutions as there are parameters
- *simple crossover* computes two new trial solutions from two old trial solutions by swapping parameters at a starting at a random point
- *local minimum crossover* computes a new trial solution in two steps: preset number of quasi-newton based iterations then computes a convex combination of the input trial solutions and the quasi-newton based iterations.

It starts with a random initial population and determines the fitness of each individual, then performs these operations. Once the new population has been created, the fitness of each of the new individual is determined and the process repeats accordingly. The algorithm stops when there has not been "significant gain" in the fitness for a certain number of generations or the user tells the algorithm to stop.

There is no guarantee that these reproduction and mutations will necessarily improve the fitness of the next generation, however the long-run properties of genetic algorithms can be thought of as a Markov chain. A state of the chain is a code-string population of the size used in the algorithm (which, for GenMatch is the `pop.size` function). For finite population sizes and code-strings of finite length, and given the random reproduction and mutation operators above, the genetic algorithm is Markov chain that converges to a unique stationary distribution. Therefore, the probability that each population occurs converges to a constant, positive value. There have been proofs that have shown that the probability that each code-string is selected to reproduce is proportional to its observed fitness.

In other words, this means that, asymptotically in the population size, populations that have low fitness values have a probability approaching zero in the stationary distribution, while populations with high fitness have a probability approaching one. The critical implication of this is that *the success of a genetic algorithm, including genoud, as an optimizer depends on having a sufficiently large population of code-strings.* This is why, for publication quality work, it is important that you have a large population size, otherwise the convergent distribution will not have sharply distinguished probabilities for optimal and sub-optimal populations. This could result in sub-optimal balance for our matches.

3 The PS Cluster

The PS cluster allows for us to run code both remotely and in parallel, thus reducing the time it takes to run our very computationally intense code. Since GenMatch is a genetic search algorithm, it must calculate it's loss function a `pop.size` number of times for each generation. Mark Huberty set up the PS cluster last year for use by students, and he has provided very simple code that easily allows for set up for the code. Once you have access to the cluster, you have to login using the command:

```
ssh myLogin@pscluster.polisci.berkeley.edu
```

You also need to put your code on the cluster. I recommend using a gui application, such as Fetch for mac or winSCP for Windows. These gui applications allow you to drag and drop your files from your local computer to the cluster, and some let you use a gui interface to edit the code once it is on the cluster. More detailed directions can be found at:

`http://pscluster.berkeley.edu/`
and
`http://pscluster.berkeley.edu/docs/Cluster_README.pdf`

Once you have your code on the cluster, there are three lines that you must add to your `R` file in order to set up the cluster. The `cluster` option in `GenMatch` takes in the name of the cluster you created, by default it is called `cl` in Mark's code, in order to use the parallel code. More detailed instructions can be found in the readme file on the cluster website.

Finally, to run the code, we call the command:

```
nohup Rnotify.sh <Rcodefile.R> <number of nodes> <email address>
```

Note: if we want to save the output, then we should use the command:

```
nohup Rnotify.sh <Rcodefile.R> <# of nodes> <email address> > <outputFile.Rout>
```

An example would be:

```
nohup Rnotify.sh myCode.R 4 myName@berkeley.edu > myCode.today.Rout
```

which would run the file myCode.R (which should be located in your home directory) on four nodes. The notification email will be sent to myName@berkeley.edu, and the output will be saved to the file myCode.today.Rout (which will also be located in the home directory). Much more detailed explanations can be found in the readme file listed above. There are 18 nodes, however in general you should check who is using the nodes with the command `squeue`, and make sure not to dominate the cluster if others are using it.