genrules technical documentation

By Andrew Wheeler, PhD

This is the technical documentation for `genrules`, a genetic algorithm to identify co-morbidities. The github repo is currently located at https://github.com/apwheele/genrules. This document describes the genetic algorithm used, as well as more specific function arguments.

Genetic Algorithm Overview

Genetic algorithms take their inspiration from how nature improves itself via evolution. In this particular example, we can consider genes to be different categorical attributes in the data, and then we evaluate those common those attributes are associated with maternal morbidities (although one could swap out any type of data in these examples).

Below are examples of how the genetic algorithm is conducted in this library.

Generating the initial population

The initial population in a genetic algorithm defines the potential outcomes in which offspring (attribute pairs) will be assessed. This algorithm generates the initial population as a complete enumeration of potential attributes based on the `k` parameter.

So pretend we have 3 variables with attributes:

- V1, {1,2}
- V2, {0,1}
- V3, {a,b,c}

If `k=1`, the initial population table would then look like this:

```
  V1   V2    V3
   1   None  None
   2   None  None
None    0    None
None    1    None
None   None  a
None   None  b
None   None  c
```

So it generates all potential singles of attributes (1 variable at a time), and sets the other attributes to None.

If k=2, it also includes all pairwise sets of attributes in the original population (as well as k=1 sets):

```
   V1   V2    V3
    1   None  None
    2   None  None
None    0    None
None    1    None
None   None  a
None   None  b
None   None  c
    1    0    None
    1    1    None
    1   None  a
    1   None  b
    1   None  c
    2    0    None
    2    1    None
    2   None  a
    2   None  b
    2   None  c
 None    0    a
 None    0    b
 None    0    c
 None    1    a
 None    1    b
 None    1    c
```

The algorithm does this for every variable pair (but only includes actual groups in the data, e.g. if `(None, 1, c)` never occurs in the supplied data, it will not be included in the subsequent population list.

With small variable sets (e.g. under 100), it is easily feasible to generate k=2 or k=3 (if slightly more patient). Thus the algorithm can be used not quite easily to examine all observed permutations for smaller k (which will often be of most interest and less likely to be spurious).

Mating and Mutations

Evolutionary algorithms rely on mating current parents in the population, as well as random mutations, and then evaluating the fitness of the offspring (here according to an arbitrary function).

Mating is done at random in the parent population. Parents who have different active chromosomes (think of these are dominant traits in the genetic analogy) always pass on their traits. So if we have these `V` variables with arbitrary traits (represented as integers):

- Parent A, active traits `{V1: 1, V2: 3}`
- Parent B, active traits `{V3: 4, V4: 0}`

The offspring of Parent A and Parent B would then have the traits:

- Offspring AB, activate traits {V1: 1, V2: 3, V3: 4, V4: 0}

In the case Parents have overlapping active traits, the offspring chooses a trait at random from the parents,

- Parent C, active traits {V1: 1, V2: 3}
- Parent D, active traits {V1: 2, V3: 0}

The potential outcomes are:

- Offspring CD1, active traits {V1: 1, V2: 3, V3: 0}
- Offspring CD2, active traits {V1: 2, V2: 3, V3: 0}

The current evolutionary algorithm randomly assigns parents and picks a random offspring this way.

Once an offspring is developed, then the offspring undergo random mutations. Random mutations are specified by the the genrules object (default 50%), but any mutation only occurs in one attribute at a time.

So if variable V1 has potential outcomes 1,2,3, Offspring AB above could then be mutated to be:

- Offspring AB, without mutation {V1: 1, V2: 3, V3: 4, V4: 0}
- Offspring AB, with mutation {V1: 3, V2: 3, V3: 4, V4: 0}

There is also an option in the genetic algorithm to turn off attributes, so when choosing to remove attributes, Offspring AB could be mutated to be:

- Offspring AB1, with remove {V1: 1, V2: 3, V3: 4} (V4 removed)
- Offspring AB2, with remove {V1: 1, V3: 4, V4: 0} (V2 removed)

etc. The algorithm defaults to adding mutations, but the example notebook provided shows how to evolve the population to both add and remove attributes in several steps.

Fitness function

The fitness function here is defined as:

```
Fitness = log2(relative_risk) + ratio_penalty*p_value -
att_penalty*number_attributes
```

log2 refers to the log base 2 function, e.g. log2(4) = 2, log2(8) = 3, etc. Relative risk is defined as:

```
Prob(Morbidity|Attributes) / Prob(Morbidity|~Attributes)
```

Where `~Attributes` means those individual not within the attribute set. The p-value is defined for the log Risk Ratio and its standard error (all equations can be seen on Wikipedia). This effectively penalizes risk ratios that are large, but have a large variance (e.g. very small groups). The algorithm also defaults to only examining subgroups that have over a particular number of observations (default 50, but can be changed).

The final penalty term slightly penalizes solutions that are more complicated, e.g. a solution with the same risk ratio (and sample size), if one have 4 attributes and the other had 2, the 4 solution would have a fitness value of `att_penalty*2` lower than the 2 solution.

Both of these penalties are set to what I believe are reasonable defaults. The ratio-penalty was derived via imagining what is a reasonable risk ratio you would trade off for a p-value of 0.5 (e.g. you would explore RR=16, even if the p-value was quite large). Here I default that to be 16.

The attribute penalty is quite small overall, and really just helps to rank order simpler rules. One can eliminate this entirely (e.g. set to 0), without altering the results by much.

These penalties become more important in smaller datasets. Currently the genrules algorithm also defaults to not providing rules with samples smaller than 50 cases. If one allows smaller groups, you will uncover many more spurious associations when evaluation large attribute sets.

genrules specific implementation

This section details the main arguments to the `genrules` class and its methods.

Class genrules

The main `genrules class object has the following arguments when it is being initialized:

- `data`, the dataframe that contains the observations
- `y_var`, string specifying which variable is the outcome morbidity being examined. This should be a 0/1 variable with no missing values, with 1 being the negative outcome.
- `x_vars`, list of strings for the variables used to assess group co-morbidities. These can be encoded however, but missing data should not be encoded as `None`.
- `w_var`, string for if you are using frequency weights, default None assumes each row in `data` is weighted with a value of 1
- `k`, integer (default 2). This creates the initial population for the genetic algo with all k potential groupings. So `k=2` means it looks at all pairwise groups, `k=3` examines all potential triplets of characteristics. Larger variable groups will take longer for higher values of k. Smaller variable groups makes more sense to up the k value and just list all possible group permutations.

- `penrat`, float (default `16`), this relates to the weight used to penalize high variance relative risks. This default says `I would trade off a relative risk of 16 with a p-value of 0.5, to one with a relative risk of 2 with a p-value of 0`. Higher values mean you are willing to accept higher relative risks for larger variances.
- `pen_var`, float (default 0.2), this penalizes rules with a larger number of attributes, smaller values means you are more likely to find more complicated rules.
- `clip_val`, float (default 0.001), relative risks can be infinite if a reference group has 0 cases in it. This limits the denominator risk for the reference group to prevent division by zeroes.
- `min_sample`, integer (default 50), rules with group sizes under this value will not be chosen for the leaderboard.
- `mut_prob`, float between 0 and 1 (default 0.5), sets the mutation probability for the evolutionary algorithm.
- `leader_tot`, integer (default 100), sets the size of the leaderboard in which to maintain potential rules
- `negative_fit`, float (default -5), this is the default negative fitness value assigned if rules have no cases or groups are too small. A negative value prevents it being added to the leaderboard ever.

The class `genrules` has several methods, methods users are likely to call directly are documented here as well (e.g. there is no documentation of `opt_stats`, as that is mainly used internally in generating the evolutionary algorithm, it is unlikely to be called directly by the user).

method `evolve`

Method `evolve` arguments (evolving the genetic algorithm)

- `rep`, integer number of times to evolve the population. 0 results in no evolutions, which can be helpful to just view all evaluated permutations.
- `set_mute`, string (default `add`), either `add` or `remove`. Sets mutations in the genetic algorithm to either add different attributes or remove removes attributes. Makes more sense to have add at first, but later rounds it may make sense to remove attributes.
- `redo_pop`, boolean (default `False`). Resets the population for the genetic algorithm to be the candidates on the leaderboard.

method `active_table`:

- `type`: string, either default (“vars”), which shows the most activate variables, or “att”, which shows most active variables and their subsequent attributes.

Future Potential Developments

There are of course endless potential improvements. Ones that I believe are feasible are:

- limiting categories for specific variables (e.g. drug dummy variables only look at 1's (taken), not any 0's in rules.
- More general EDA tools to explore common co-morbidities (see the `network` method for a start of this).
- Smarter data based defaults for the evolutionary algorithm (e.g. adding and pruning at the same time, weights for penalties based on the data).

The last will require more extensive tests with multiple data sources. I believe the current defaults work reasonably well with the nuMoM2b data, but may need to be altered for larger datasets with rare outcomes.

For much rarer outcomes, one may wish to examine a different metric besides the relative risk, or in case-control or panel data may wish to incorporate sampling weights.