20 Feature Selection using Genetic Algorithms

Contents

20.1 Genetic Algorithms

Genetic algorithms (GAs) mimic Darwinian forces of natural selection to find optimal values of some function (Mitchell, 1998). An initial set of candidate solutions are created and their corresponding *fitness* values are calculated (where larger values are better). This set of solutions is referred to as a population and each solution as an *individual*. The individuals with the best fitness values are combined randomly to produce offsprings which make up the next population. To do so, individual are selected and undergo cross-over (mimicking genetic reproduction) and also are subject to random mutations. This process is repeated again and again and many generations are produced (i.e. iterations of the search procedure) that should create better and better solutions.

For feature selection, the individuals are subsets of predictors that are encoded as binary; a feature is either included or not in the subset. The fitness values are some measure of model performance, such as the RMSE or classification accuracy. One issue with using GAs for feature selection is that the optimization process can be very aggressive and their is potential for the GA to overfit to the predictors (much like the previous discussion for RFE).

20.2 Internal and External Performance Estimates

The genetic algorithm code in `caret` conducts the search of the feature space repeatedly within resampling iterations. First, the training data are split be whatever resampling method was specified in the control function. For example, if 10-fold cross-validation is selected, the entire genetic algorithm is

conducted 10 separate times. For the first fold, nine tenths of the data are used in the search while the remaining tenth is used to estimate the external performance since these data points were not used in the search.

During the genetic algorithm, a measure of fitness is needed to guide the search. This is the internal measure of performance. During the search, the data that are available are the instances selected by the top-level resampling (e.g. the nine tenths mentioned above). A common approach is to conduct another resampling procedure. Another option is to use a holdout set of samples to determine the internal estimate of performance (see the holdout argument of the control function). While this is faster, it is more likely to cause overfitting of the features and should only be used when a large amount of training data are available. Yet another idea is to use a penalized metric (such as the AIC statistic) but this may not exist for some metrics (e.g. the area under the ROC curve).

The internal estimates of performance will eventually overfit the subsets to the data. However, since the external estimate is not used by the search, it is able to make better assessments of overfitting. After resampling, this function determines the optimal number of generations for the GA.

Finally, the entire data set is used in the last execution of the genetic algorithm search and the final model is built on the predictor subset that is associated with the optimal number of generations determined by resampling (although the update function can be used to manually set the number of generations).

20.3 Basic Syntax

The most basic usage of the function is:

```
obj <- gafs(x = predictors,
            y = outcome,
            iters = 100)
```

where

- `x` : a data frame or matrix of predictor values
- `y` : a factor or numeric vector of outcomes
- `iters` : the number of generations for the GA

This isn't very specific. All of the action is in the control function. That can be used to specify the model to be fit, how predictions are made and summarized as well as the genetic operations.

Suppose that we want to fit a linear regression model. To do this, we can use `train` as an interface and pass arguments to that function through `gafs` :

```
ctrl <- gafsControl(functions = caretGA)
obj <- gafs(x = predictors,
            y = outcome,
            iters = 100,
            gafsControl = ctrl,
            ## Now pass options to `train`

            method = "lm")
```

Other options, such as `preProcess`, can be passed in as well.

Some important options to `gafsControl` are:

- `method`, `number`, `repeats`, `index`, `indexOut`, etc: options similar to those for `train` top control resampling.
- `metric`: this is similar to `train`'s option but, in this case, the value should be a named vector with values for the internal and external metrics. If none are specified, the first value returned by the summary functions (see details below) are used and a warning is issued. A similar two-element vector for the option `maximize` is also required. See the last example here for an illustration.
- `holdout`: this is a number between `[0, 1)` that can be used to hold out samples for computing the internal fitness value. Note that this is independent of the external resampling step. Suppose 10-fold CV is being used. Within a resampling iteration, `holdout` can be used to sample an additional proportion of the 90% resampled data to use for estimating fitness. This may not be a good idea unless you have a very large training set and want to avoid an internal resampling procedure to estimate fitness.
- `allowParallel` and `genParallel`: these are logicals to control where parallel processing should be used (if at all). The former will parallelize the external resampling while the latter parallelizes the fitness calculations within a generation. `allowParallel` will almost always be more advantageous.

There are a few built-in sets of functions to use with `gafs`: `caretGA`, `rfGA`, and `treebagGA`. The first is a simple interface to `train`. When using this, as shown above, arguments can be passed to `train` using the `...` structure and the resampling estimates of performance can be used as the internal fitness value. The functions provided by `rfGA` and `treebagGA` avoid using `train` and their internal estimates of fitness come from using the out-of-bag estimates generated from the model.

The GA implementation in `caret` uses the underlying code from the `GA` package (Scrucca, 2013).

20.4 Example

Using the example from the previous page where there are five real predictors and 40 noise predictors:

```
library(mlbench)
n <- 100
p <- 40
sigma <- 1
set.seed(1)
sim <- mlbench.friedman1(n, sd = sigma)
colnames(sim$x) <- c(paste("real", 1:5, sep = ""),
                     paste("bogus", 1:5, sep = ""))
bogus <- matrix(rnorm(n * p), nrow = n)
colnames(bogus) <- paste("bogus", 5+(1:ncol(bogus)), sep = "")
x <- cbind(sim$x, bogus)
y <- sim$y
normalization <- preProcess(x)
x <- predict(normalization, x)
x <- as.data.frame(x)
```

We'll fit a random forest model and use the out-of-bag RMSE estimate as the internal performance metric and use the same repeated 10-fold cross-validation process used with the search. To do this, we'll use the built-in `rfGA` object for this purpose. The default GA operators will be used and conduct 200 generations of the algorithm.

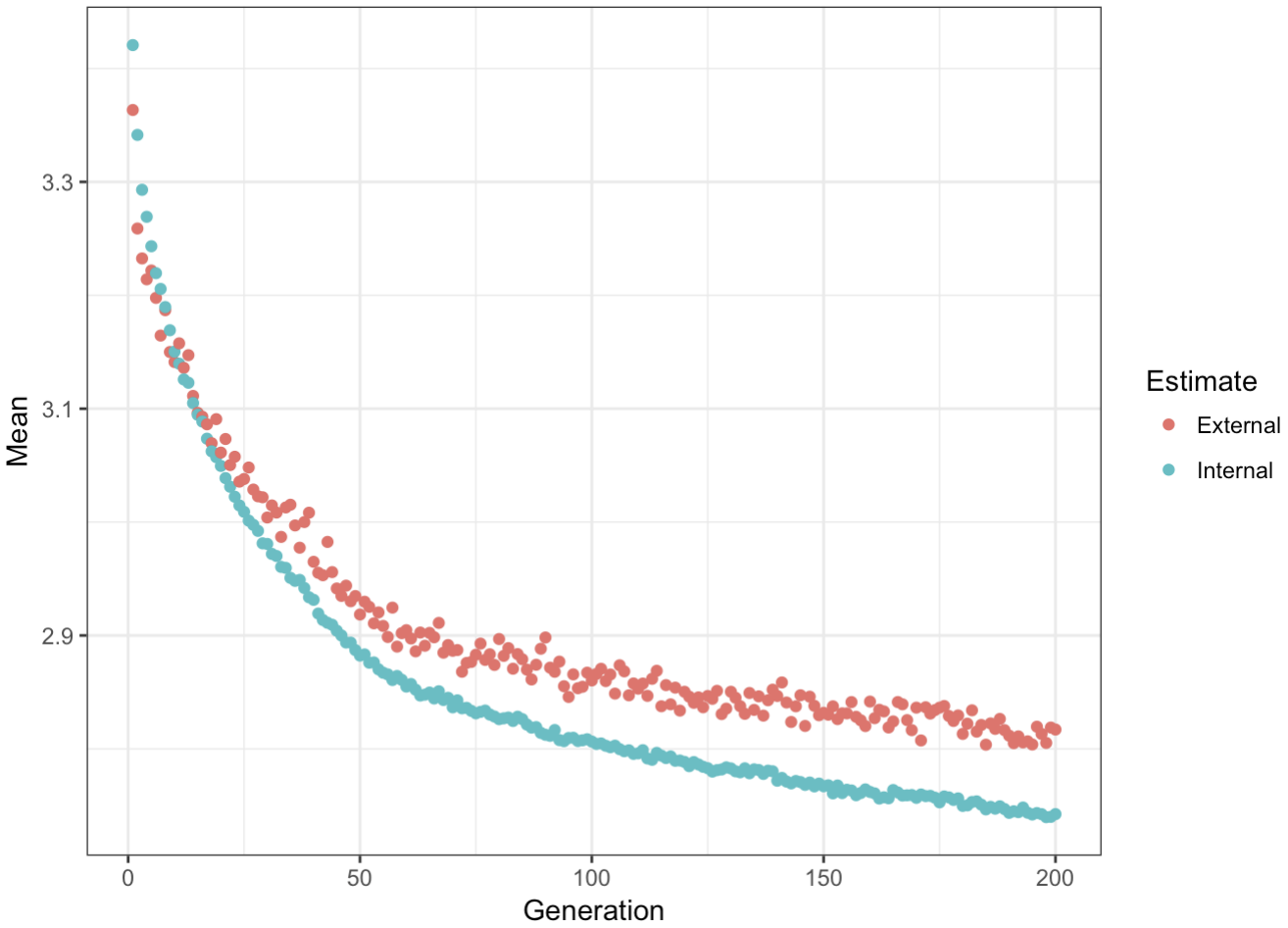
```
ga_ctrl <- gafsControl(functions = rfGA,
                       method = "repeatedcv",
                       repeats = 5)

## Use the same random number seed as the RFE process
## so that the same CV folds are used for the external
## resampling.
set.seed(10)
rf_ga <- gafs(x = x, y = y,
              iters = 200,
              gafsControl = ga_ctrl)
rf_ga
```

```
##
## Genetic Algorithm Feature Selection
##
## 100 samples
## 50 predictors
##
## Maximum generations: 200
## Population per generation: 50
## Crossover probability: 0.8
## Mutation probability: 0.1
## Elitism: 0
##
## Internal performance values: RMSE, Rsquared
## Subset selection driven to minimize internal RMSE
##
## External performance values: RMSE, Rsquared
## Best iteration chose by minimizing external RMSE
## External resampling method: Cross-Validated (10 fold, repeated 5 times)
##
## During resampling:
##   * the top 5 selected variables (out of a possible 50):
##     real1 (100%), real2 (100%), real4 (100%), real5 (100%), bogus17 (88%)
##   * on average, 8.3 variables were selected (min = 6, max = 12)
##
## In the final search using the entire training set:
##    * 12 features selected at iteration 185 including:
##      real1, real2, real3, real4, real5 ...
##    * external performance at this iteration is
##
##        RMSE    Rsquared
##      2.8037      0.7564
```

With 5 repeats of 10-fold cross-validation, the GA was executed 50 times. The average external performance is calculated across resamples and these results are used to determine the optimal number of iterations for the final GA to avoid over-fitting. Across the resamples, an average of 8.3 predictors were selected at the end of each of the algorithms.

The `plot` function is used to monitor the average of the internal out-of-bag RMSE estimates as well as the average of the external performance estimates calculated from the 50 out-of-sample predictions. By default, this function uses `ggplot2` package. A black and white theme can be "added" to the output object:

```
plot(rf_ga) + theme_bw()
```



Based on these results, the generation associated with the best external RMSE estimate was 2.8.

Using the entire training set, the final GA is conducted and, at generation 185, there were 12 that were selected: real1, real2, real3, real4, real5, bogus6, bogus7, bogus13, bogus14, bogus17, bogus32, bogus43. The random forest model with these predictors is created using the entire training set is trained and this is the model that is used when `predict.gafs` is executed.

Note: the correlation between the internal and external fitness values is somewhat atypical for most real-world problems. This is a function of the nature of the simulations (a small number of uncorrelated informative predictors) and that the OOB error estimate from random forest is a product of hundreds of trees. Your mileage may vary.

20.5 Customizing the Search

20.5.1 The `fit` Function

This function builds the model based on a proposed current subset. The arguments for the function must be:

- `x` : the current training set of predictor data with the appropriate subset of variables
- `y` : the current outcome data (either a numeric or factor vector)
- `lev` : a character vector with the class levels (or `NULL` for regression problems)
- `last` : a logical that is `TRUE` when the final GA search is conducted on the entire data set
- `...` : optional arguments to pass to the fit function in the call to `gafs`

The function should return a model object that can be used to generate predictions. For random forest, the fit function is simple:

```
rfGA$fit
```

```
## function (x, y, lev = NULL, last = FALSE, ...)
## {
##     loadNamespace("randomForest")
##     randomForest::randomForest(x, y, ...)
## }
## <environment: namespace:caret>
```

20.5.2 The `pred` Function

This function returns a vector of predictions (numeric or factors) from the current model . The input arguments must be

- `object` : the model generated by the `fit` function
- `x` : the current set of predictor set for the held-back samples

For random forests, the function is a simple wrapper for the predict function:

```
rfGA$pred
```

```
## function (object, x)
## {
##     tmp <- predict(object, x)
##     if (is.factor(object$y)) {
##         out <- cbind(data.frame(pred = tmp), as.data.frame(predict(object,
##             x, type = "prob")))
##     }
##     else out <- tmp
##     out
## }
## <environment: namespace:caret>
```

For classification, it is probably a good idea to ensure that the resulting factor variables of predictions has the same levels as the input data.

20.5.3 The `fitness_intern` Function

The `fitness_intern` function takes the fitted model and computes one or more performance metrics. The inputs to this function are:

- `object` : the model generated by the `fit` function
- `x` : the current set of predictor set. If the option `gafsControl$holdout` is zero, these values will be from the current resample (i.e. the same data used to fit the model). Otherwise, the predictor values are from the hold-out set created by `gafsControl$holdout` .
- `y` : outcome values. See the note for the `x` argument to understand which data are presented to the function.
- `maximize` : a logical from `gafsControl` that indicates whether the metric should be maximized or minimized
- `p` : the total number of possible predictors

The output should be a **named** numeric vector of performance values.

In many cases, some resampled measure of performance is used. In the example above using random forest, the OOB error was used. In other cases, the resampled performance from `train` can be used and, if `gafsControl$holdout` is not zero, a static hold-out set can be used. This depends on the data and problem at hand.

The example function for random forest is:

```
rfGA$fitness_intern
```

```
## function (object, x, y, maximize, p)
## rfStats(object)
## <environment: namespace:caret>
```

20.5.4 The `fitness_extern` Function

The `fitness_extern` function takes the observed and predicted values form the external resampling process and computes one or more performance metrics. The input arguments are:

- `data` : a data frame or predictions generated by the `fit` function. For regression, the predicted values in a column called `pred` . For classification, `pred` is a factor vector. Class probabilities are usually attached as columns whose names are the class levels (see the random forest example for the `fit` function above)
- `lev` : a character vector with the class levels (or `NULL` for regression problems)

The output should be a **named** numeric vector of performance values.

The example function for random forest is:

```
rfGA$fitness_extern
```

```
## function (data, lev = NULL, model = NULL)
## {
##     if (is.character(data$obs))
##         data$obs <- factor(data$obs, levels = lev)
##     postResample(data[, "pred"], data[, "obs"])
## }
## <environment: namespace:caret>
```

Two functions in `caret` that can be used as the summary function are `defaultSummary` and `twoClassSummary` (for classification problems with two classes).

20.5.5 The `initial` Function

This function creates an initial generation. Inputs are:

- `vars` : the number of possible predictors
- `popSize` : the population size for each generation

- `...` : not currently used

The output should be a binary 0/1 matrix where there are `vars` columns corresponding to the predictors and `popSize` rows for the individuals in the population.

The default function populates the rows randomly with subset sizes varying between 10% and 90% of number of possible predictors. For example:

```
set.seed(128)
starting <- rfGA$initial(vars = 12, popSize = 8)
starting
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## [1,]    0    1    0    0    1    0    0    0    0     0     0     0
## [2,]    0    0    0    0    0    0    1    0    0     0     1     0
## [3,]    0    0    1    1    1    0    0    0    1     0     1     0
## [4,]    0    1    0    0    1    0    0    1    0     1     0     1
## [5,]    0    1    1    1    0    0    1    0    0     1     0     0
## [6,]    1    1    1    1    1    1    0    1    1     1     1     1
## [7,]    1    1    1    1    1    0    0    1    0     1     1     1
## [8,]    1    0    1    1    0    1    1    1    0     1     1     1
```

```
apply(starting, 1, mean)
```

```
## [1] 0.1666667 0.1666667 0.4166667 0.4166667 0.4166667 0.9166667 0.7500000
## [8] 0.7500000
```

`gafs` has an argument called `suggestions` that is similar to the one in the `ga` function where the initial population can be seeded with specific subsets.

20.5.6 The `selection` Function

This function conducts the genetic selection. Inputs are:

- `population` : the indicators for the current population
- `fitness` : the corresponding fitness values for the population. Note that if the internal performance value is to be minimized, these are the negatives of the actual values

- `r , q` : tuning parameters for specific selection functions. See `gafs_lrSelection` and `gafs_nlrSelection`
- `...` : not currently used

The output should be a list with named elements.

- `population` : the indicators for the selected individuals
- `fitness` : the fitness values for the selected individuals

The default function is a version of the `GA` package's ga_lrSelection function.

20.5.7 The `crossover` Function

This function conducts the genetic crossover. Inputs are:

- `population` : the indicators for the current population
- `fitness` : the corresponding fitness values for the population. Note that if the internal performance value is to be minimized, these are the negatives of the actual values
- `parents` : a matrix with two rows containing indicators for the parent individuals.
- `...` : not currently used

The default function is a version of the `GA` package's `ga_spCrossover` function. Another function that is a version of that package's uniform cross-over function is also available.

. The output should be a list with named elements.

- `children` : from `?ga_spCrossover` : "a matrix of dimension 2 times the number of decision variables containing the generated offsprings""
- `fitness` : "a vector of length 2 containing the fitness values for the offsprings. A value `NA` is returned if an offspring is different (which is usually the case) from the two parents.""

20.5.8 The `mutation` Function

This function conducts the genetic mutation. Inputs are:

- `population` : the indicators for the current population
- `parents` : a vector of indices for where the mutation should occur.
- `...` : not currently used

The default function is a version of the `GA` package's `gabin_raMutation` function.

. The output should the mutated population.

20.5.9 The `selectIter` Function

This function determines the optimal number of generations based on the resampling output. Inputs for the function are:

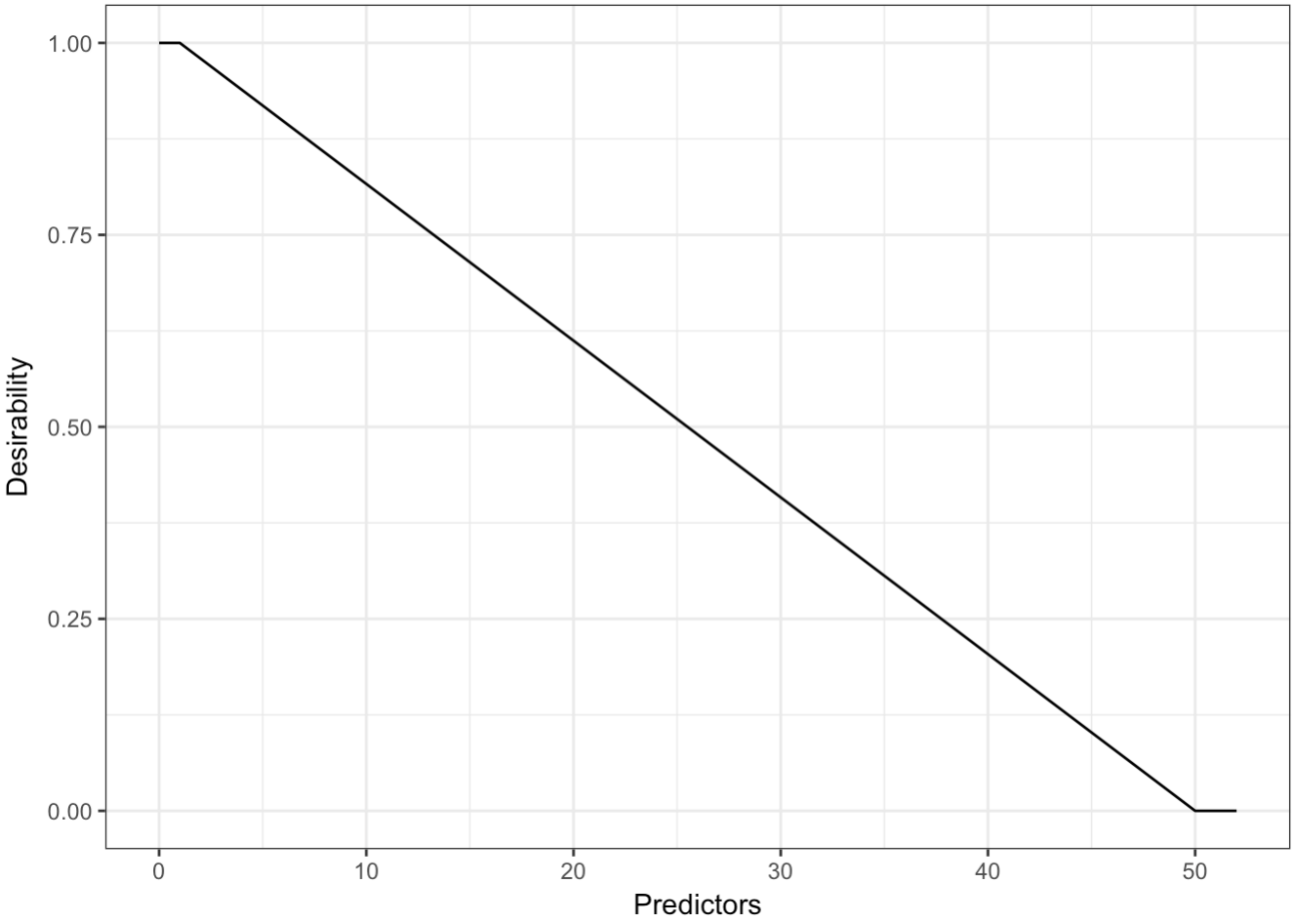
- `x` : a matrix with columns for the performance metrics averaged over resamples
- `metric` : a character string of the performance measure to optimize (e.g. RMSE, Accuracy)
- `maximize` : a single logical for whether the metric should be maximized

This function should return an integer corresponding to the optimal subset size.

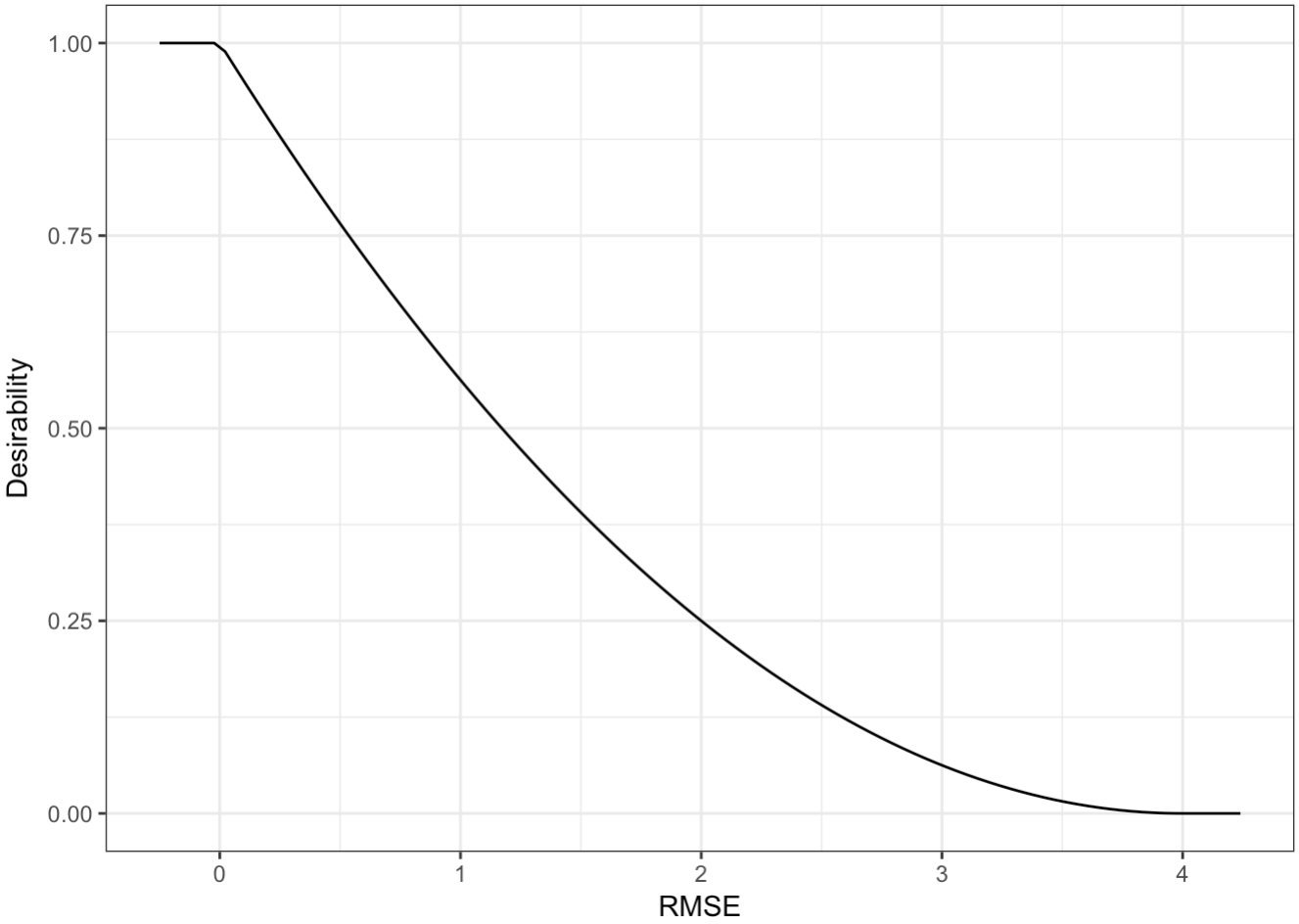
20.6 The Example Revisited

The previous GA included some of the non-informative predictors. We can cheat a little and try to bias the search to get the right solution.

We can try to encourage the algorithm to choose fewer predictors, we can penalize the the RMSE estimate. Normally, a metric like the Akaike information criterion (AIC) statistic would be used. However, with a random forest model, there is no real notion of model degrees of freedom. As an alternative, we can use desirability functions to penalize the RMSE. To do this, two functions are created that translate the number of predictors and the RMSE values to a measure of "desirability". For the number of predictors, the most desirable property would be a single predictor and the worst situation would be if the model required all 50 predictors. That desirability function is visualized as:



For the RMSE, the best case would be zero. Many poor models have values around four. To give the RMSE value more weight in the overall desirability calculation, we use a scale parameter value of 2. This desirability function is:



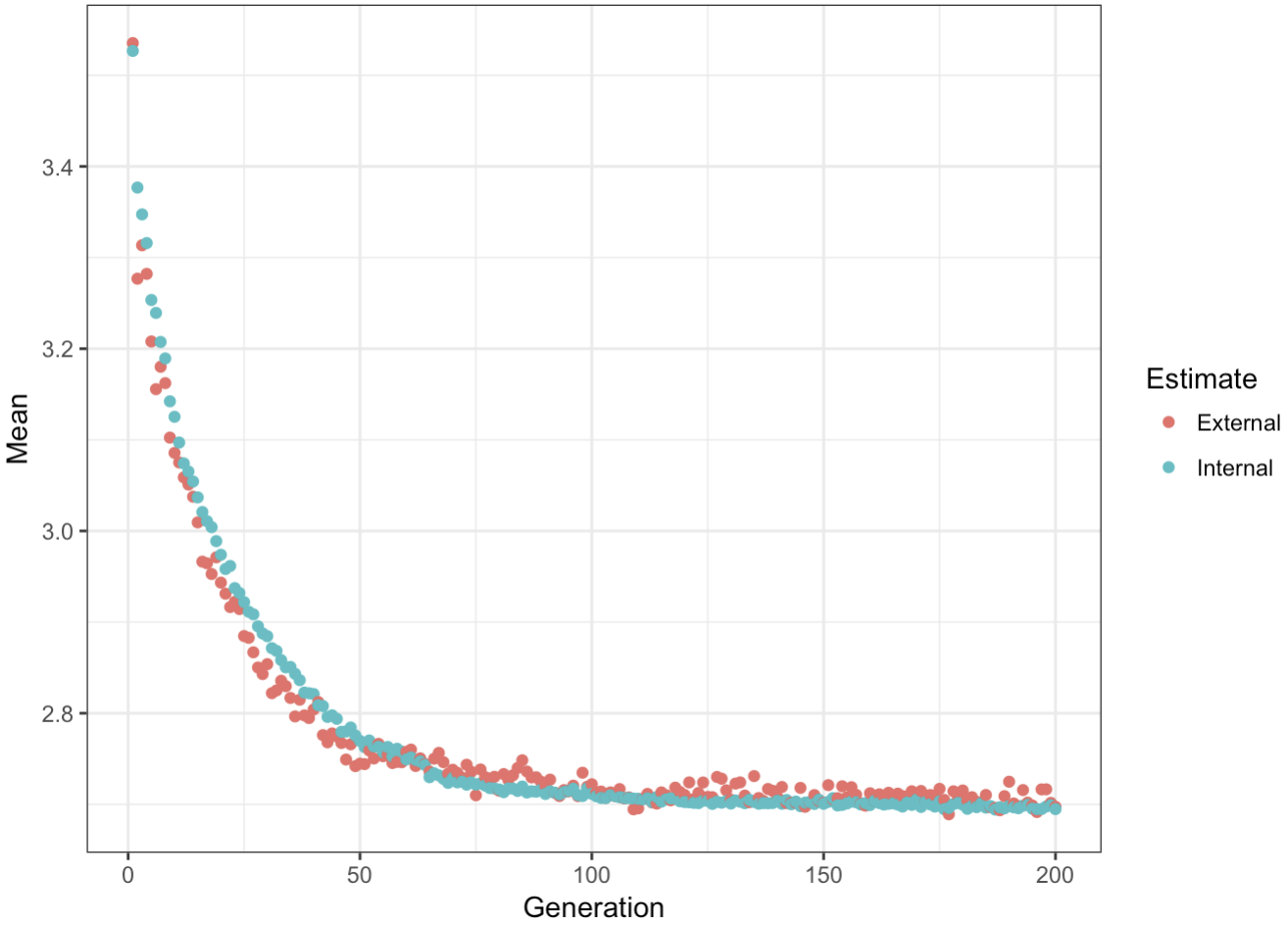
To use the overall desirability to drive the feature selection, the `internal` function requires replacement. We make a copy of `rfGA` and add code using the `desirability` package and the function returns the estimated RMSE and the overall desirability. The `gafsControl` function also need changes. The `metric` argument needs to reflect that the overall desirability score should be maximized internally but the RMSE estimate should be minimized externally.

```r
library(desirability)
rfGA2 <- rfGA
rfGA2$fitness_intern <- function (object, x, y, maximize, p) {
  RMSE <- rfStats(object)[1]
  d_RMSE <- dMin(0, 4)
  d_Size <- dMin(1, p, 2)
  overall <- dOverall(d_RMSE, d_Size)
  D <- predict(overall, data.frame(RMSE, ncol(x)))
  c(D = D, RMSE = as.vector(RMSE))
  }
ga_ctrl_d <- gafsControl(functions = rfGA2,
                         method = "repeatedcv",
                         repeats = 5,
                         metric = c(internal = "D", external = "RMSE"),
                         maximize = c(internal = TRUE, external = FALSE))

set.seed(10)
rf_ga_d <- gafs(x = x, y = y,
                iters = 200,
                gafsControl = ga_ctrl_d)

rf_ga_d
```

```
##
## Genetic Algorithm Feature Selection
##
## 100 samples
## 50 predictors
##
## Maximum generations: 200
## Population per generation: 50
## Crossover probability: 0.8
## Mutation probability: 0.1
## Elitism: 0
##
## Internal performance values: D, RMSE
## Subset selection driven to maximize internal D
##
## External performance values: RMSE, Rsquared
## Best iteration chose by minimizing external RMSE
## External resampling method: Cross-Validated (10 fold, repeated 5 times)
##
## During resampling:
##   * the top 5 selected variables (out of a possible 50):
##     real1 (100%), real2 (100%), real4 (100%), real5 (100%), real3 (36%)
##   * on average, 5.2 variables were selected (min = 4, max = 6)
##
## In the final search using the entire training set:
##    * 6 features selected at iteration 177 including:
##      real1, real2, real3, real4, real5 ...
##    * external performance at this iteration is
##
##        RMSE    Rsquared
##      2.6891      0.7682
```

Here are the RMSE values for this search:

```
plot(rf_ga_d) + theme_bw()
```



The final GA found 6 that were selected: real1, real2, real3, real4, real5, bogus26. During resampling, the average number of predictors selected was 5.2, indicating that the penalty on the number of predictors was effective.