Genetic Algorithms for Feature Selection

Michael Conklin – EVP Marketing & Data Science NA

GfK IMSM 2015
Leuven
Feature Selection in Predictive Models

Objective

Select a subset of a large number of possible predictors to maximize out of sample predictive accuracy

Approach

- Utilize Genetic Algorithms

Genetic Algorithms mimic biological genetic processes to create a “survival of the fittest” or evolutionary process for solving the problem.
The Genetic Algorithm Process

1. **Run model for each chromosome**
2. **Evaluate each chromosome’s fitness**
3. **Allow the remaining chromosomes to procreate**
4. **Send the worst fitness chromosomes to the convent**

Results in a new population of best parents and their children.
The procreation process

Random crossover point

Parent 1
1011000010001010101110111010111010

Parent 2
1001011010001010101000100010100010

Child 1
101100001000101010111011101100010100010

Child 2
10010110100010101010100111010111010
Mutation

Random crossover point

Parent 1
1011000010001010101101110110111010

Parent 2
100101101000101010110110100100010

Child 1
1011000010001010101101110110010100010

Child 2
1001011010001010101101100111000111010

Random mutation
## Steps in building the genetic algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Detail.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set global parameters</td>
<td>• Number of models in the population</td>
</tr>
<tr>
<td></td>
<td>• Number of generations to run the algorithm</td>
</tr>
<tr>
<td>Divide data into test and training sets</td>
<td>• Over-fitting is always a danger so splitting the data into a training set to fit the models and a test set to evaluate the performance of the models is essential</td>
</tr>
<tr>
<td>Create model fitting function</td>
<td>• This function needs to take a particular chromosome, select the variables that the chromosome indicates and then fit/tune the model on the training data.</td>
</tr>
<tr>
<td>Create model evaluation function</td>
<td>• Usually this can just be the predict method for the model being used, where we pass in a model that has been fit and evaluate the prediction on the test data. The type of evaluation is up to you.</td>
</tr>
<tr>
<td>Wrap it all together</td>
<td>• Create an initial population of chromosomes, and then loop through the number of generations. After each set of model fitting and evaluation, create the new population via the genetic algorithm steps shown previously.</td>
</tr>
</tbody>
</table>
Example: Retail Eyeware Predictive Model

- Panel model designed to forecast future sales for retail eyeware stores.
- Data includes observations for 940 stores for 34 months each.
- Variables include store characteristics, monthly store performance metrics (e.g. traffic, sales, number of doctor prescriptions), and monthly satisfaction data collected via GfK Echo.
- Training set had 600 stores while the test set had 340 stores.
Set Global Parameters

```r
# set population size for genetic algorithm - this is the number of different 
# models in each population
pop.size<-100

# set the maximum number of generations for the algorithm
num.gen<-100

mut.prob<-1/length(varlist)
```

We set the mutation probability to $1/\text{num. vars}$

This results in an average of 1 gene (variable) being mutated in each chromosome created by the genetic algorithm. This should prevent the algorithm getting stuck in a local maxima and provide an opportunity for all of the variables to have a fair chance of being included in the model.
Select candidate variables to include

`varlist <- select.list(names(Trainpdf), graphics=TRUE, multiple=TRUE)`

The actual set of variables selected for the feature selection step is:

```
# [1] "Nr.Total.Remakes"     "Nr.Service.Remakes"
  "Nr.Optical.Remakes"
  "Nr.Contact.Lens"
# [7] "Nr.Lens.es..Only"     "Nr.Completes"     "Nr.LC.EPP"
  "Nr.DONL.Eye.Exam.Rxs"
# [16] "Nr.A.R.Invisibles"   "lnSales"           "Q11_0C"
# [19] "Q11_0D"             "Q11_0E"           "Q11_0G"
# [22] "Q1_0"               "Q4_0"            "Q5_0"
# [25] "Q6_0A"             "Q6_0B"            "Q6_0C"
# [28] "Q6_0D"             "Q6_0E"            "Q6_0F"
# [31] "Q6_0G"             "Q6_0H"            "Q6_0I"
# [34] "Q7_0A"             "Q9_0K"            "Q9_0L"
# [37] "Q9_0M"             "Q9_0N"            "Q9_0S"
# [40] "Q9_0U"
```
Divide data into test and training sets

Setting the seed for the random number generator is critical, otherwise you cannot replicate your work executing this code.

```r
set.Seed(123)
TrainStores<- sample(unique(SQdata$Store.Number),600)
TestStores<- unique(SQdata$Store.Number)[!(unique(SQdata$Store.Number) %in% TrainStores)]
Trainpdf<- pdata.frame(SQdata[SQdata$Store.Number %in% TrainStores & !(SQdata$Fiscal.Month %in% TestPeriods)],index=c("Store.Number","Fiscal.Month"),row.names=TRUE)

## series GEO_OPENCLOSEINDICATOR, xClosingDate are constants and have been removed
Testpdf<- pdata.frame(SQdata[SQdata$Store.Number %in% TestStores],index=c("Store.Number","Fiscal.Month"),row.names=TRUE)
```
Create Model fitting function

In this particular case, we need to do a lot of manipulation of the variables to put them into this particular type of model. We first create a function that creates a formula for the model from the chromosome (called popelement in this code).

CreateFormula <- function(varlist, modelspec, popelement) {
  # varlist is the list of candidate variable names
  # modelspec has the number of lags and the dependent variable name
  # popelement is a binary vector selecting predictors for the model
  lhs <- paste(modelspec$depvar, "~")
  rhs <- paste(sapply(varlist[popelement == 1], function(x, numlags) {
    paste("lag(" , x , "," , 1: , numlags , ")", sep = "")
  }), modelspec$numlags), collapse = "+")
  fmla <- as.formula(paste(lhs, rhs, sep = ""))
  return(fmla)
}

fmla <- CreateFormula(varlist, modelspec, x)
mod <- plm(fmla, data = eval(parse(text = modelspec$pdataset)), na.action = na.omit, model = modelspec$method)
Wrap it all up

We start by setting up the initial population, the mutation probability and the loop through the generations.

```r
FeatureSelection<-function(modelspec,num.gen,pop.size,mut.prob=NULL,varlist,testdata=NULL,start pop=NULL){
  if(pop.size %% 2 != 0) stop("Population size must be even number")
  # this next bit allows us to start from a previously generated population so essentially
  # continue a run if we are not satisfied with the stability of the results
  if(is.null(startpop)){
    thispop<-initPop(varlist,pop.size) else {
      thispop<-startpop
    }
  }
  # this sets a mutation probability if one is not provided - in our case we used the default
  if (is.null(mut.prob)) mut.prob<-1/length(varlist)
  outlist<-vector("list",num.gen)
  for(i in 1:num.gen){
    # here we print to the screen to see the slow progress
    cat("Generation",i,"\n")
    # if we don't flush the console the previous lines won't show up until the function
    # finishes - defeating the purpose of printing the progress
    flush.console()
  }
}
```
Wrap it all up

Then we fit the model for each member of the population and evaluate the models on the holdout data

```r
# now we loop through the population of models and evaluate each one
evals <- sapply(thispop, function(x, varlist, modelspec) {
    # cat(sum(x))
    fmla <- CreateFormula(varlist, modelspec, x)
    mod <- plm(fmla, data = eval(parse(text = modelspec$pdataset)), na.action = na.omit, model = modelspec$method)
    vlist <- varlist[x == 1]
    qual <- EvalModel(mod, testdata = Testpdf, varlist = vlist, modelspec = modelspec)
    return(qual)
}, varlist, modelspec)
# cat(evals, "\n")
outlist[[i]] <- list(meanqual = mean(evals), pop = thispop)
# summarize how this population of models performs
cat("Mean Quality =", mean(evals), "\n")
flush.console()
```
Wrap it all up

Now comes the genetic algorithm part.

```r
sortpop<-thispop[rev(order(evals))]
  # create marriages from top half of population twice
parents<-matrix(rep(1:(pop.size/2),2),ncol=2,byrow=TRUE)
  # select random crossover point for each set of parents
crossovers<-sample(1:(length(varlist)-1),size=pop.size/2,replace=TRUE)
  # create children by combining chromosomes from two parents at the
crossover point
children<-lapply(1:nrow(parents),function(x){
  c(sortpop[[parents[x,1]]][1:crossovers[x]],sortpop[[parents[x,2]]][c(crossover[s[x]+1):length(varlist)]])
  # now randomly mutate chromosomes
mutchildren<-lapply(children,function(x){
  mutvec<-runif(length(x))<mut.prob
  out<-as.numeric(xor((x==1),mutvec))
  if(sum(out)==0){
    out<-x
  }else{
    return(out)
  }
  return(out)
})
  # return the original best models plus the mutated children
thispop<-c(sortpop[1:(pop.size/2)],mutchildren)
  } 
return(outlist)
```
How well does it work?

Well, it works very slowly ~24 hours to run 100 generations of 100 models each

But…convergence to a final set of variables seems relatively quick – so we probably did not have to run so many generations.
How well does it work?

```r
finalvarlist <- varlist[rowMeans(apply(sapply(test4[25:100], function(x) { sapply(x$pop, function(xx) { xx }) }), 2, function(xxx) { rowMeans(matrix(as.vector(xxx), nrow = 40)) })))].9]
```

The final list of variables selected makes sense and also includes some of the satisfaction variables.

Predictions on future data are also good.
How to make this easier – the genalg package
## Genalg package vs Build Your Own Genetic Algorithm

### Reasons to use genalg
- All the cross-over, mutation steps are automatically taken care of.
- Easy set up
- Much faster (unless you are a programming whiz)
- Can use continuous value genes (more later)

### Disadvantages of genalg
- You have less control over the functioning of the algorithm

### Overall - use genalg
- The speed and flexibility lets you try different parameters
Same problem – using genalg package

Now we need just two functions, one that fits the model and tests it, and one that generates some summary statistics

```r
lux.evaluate <- function(indices) {
  if(sum(indices)<1) {
    qual<-1
    return(qual)
  }
  fmla<-CreateFormula(varlist,modelspec,indices)
  mod<-plm(fmla,data=eval(parse(text=modelspec$pdataset)),na.action=na.omit,model=modelspec$method)
  vlist<-varlist[indices==1]
  qual<-EvalModel(mod,testdata=Testpdf,varlist=vlist,modelspec=modelspec)
  return(1-qual)
}

monitor <- function(obj) {
  minEval <-min(obj$evaluations);
  meanEval <-mean(obj$evaluations)
  cat(meanEval)
  plot(obj, type="hist");
}
```

Check to make sure we have at least 1 predictor

Reverse the quality metric as genalg minimizes instead of maximizes
Using genalg

```r
usingenalg <- rbga.bin(size=40, popSize=100, iters=100, mutationChance=1/40, elitism=50, zeroToOneRatio=4,
                        evalFunc=lux.evaluate, verbose=TRUE, monitorFunc=monitor)
```

Standard output for each generation shows the prevalence of each “gene” in the population.
As you can see, we pretty much have a clone army as a result.
How well does it work


More variables, but the previous set is a subset of this group with the same predictive ability

But, only 2 hours of runtime vs 24
Other Capabilities of genalg

**Use Continuous Genes**
Can use floating point genes to solve most any optimization problem.

**Speed**
Easy to set up and the speed of execution allows for tuning parameters.

**Control**
By adjusting your evaluation function you can implement any constraints.

Genalg is a useful tool for marketing scientists
Questions?