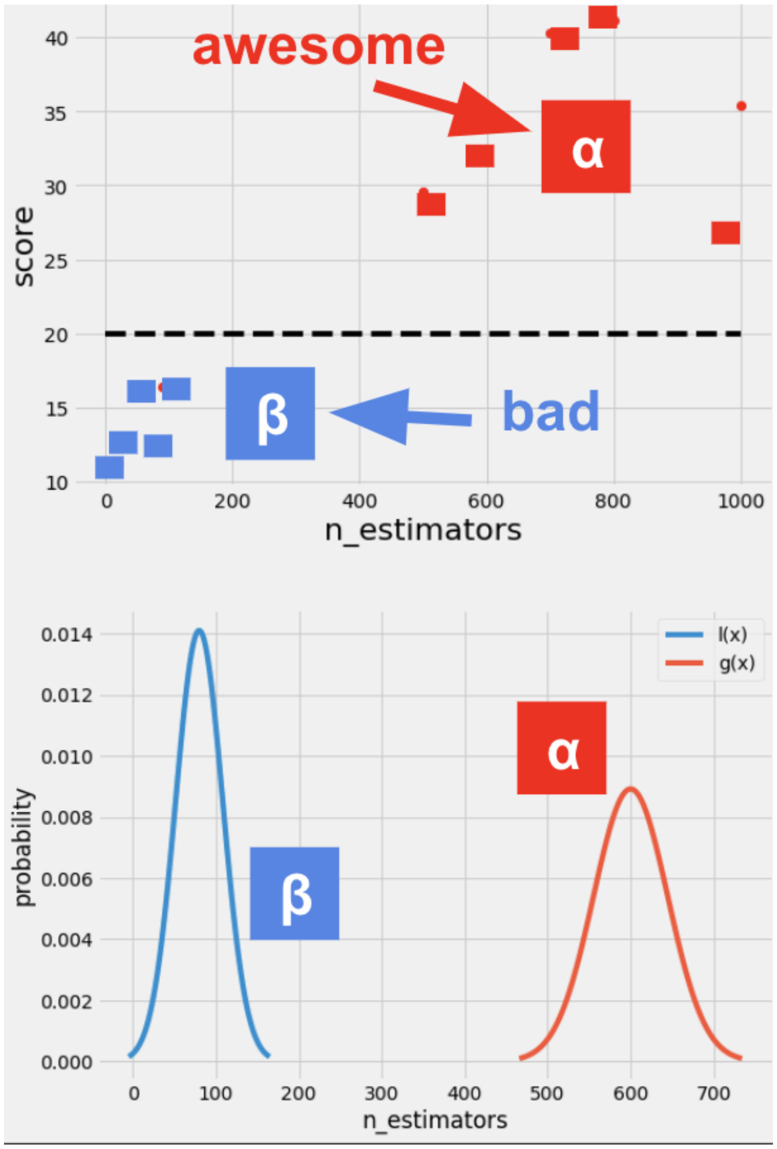


AI can be easy (using good SE)

Tim Menzies

January 8, 2024

Contents

1 Overview

This paper discusses `gate.lua`, which is a small script that implements sequential model optimization (defined below). GATE is a demonstrator that good AI (i.e. that is fast and understandable) can be build easily (i.e. that is simple to code) assuming that the software engineering is done right.

The great secret is that AI software is still software. Hence, better software engineering means better AI. Measured in terms of lines of code, any AI brain is very small compared to all the software that it needs to make it go [1] [2]. So rather than start with the inference procedure, we start with the data structures (the classes) that implement the under-the-hood tedium (e.g reading rows into a table of data, summarizing the rows into columns). Once that is working, we run the code and do lots of testing and ablation studies (where we throw away anything that barely changes the performance).

This approach generates two things:

- Some reusable classes (in this document, NUM, SYM, ROW, COL, COLS, DATA);
- And some very tiny functions implementing the actual AI.

For example, the DATA class defined in this document summarizes ROWs into its various NUMeric or SYMbolic columns. DATA turns out to be extraordinarily reusable for many things. E.g. if clustering, we an give each cluster its own DATA. If we want separate stats on each cluster, then we call the `DATA:stats()` method (shown below). For classification, we store information about different classes in its own separate data. Once that is done, then implementing a Naive Bayes Classifier takes just eight lines of code.

```
local function learn(data,row,  my,kl) --> nil, but updates the working memory "my"
  my.n = my.n + 1
  kl   = row.cells[data.cols.klass.at]
  if my.n > 10 then -- train on at least 10 items before testing
    my.tries = my.tries + 1
    my.acc   = my.acc + (kl == row:likes(my.datas) and 1 or 0) end
  my.datas[kl] = my.datas[kl] or DATA.new{data.cols.names}
  my.datas[kl]:add(row) end
```

In the above, ROWs from different classes are store in their own DATA instance. Also note that we test the classifier before updating the columns (so we are always testing on previously unseen examples). For full details on the above, see the rest of this paper.

Here's an example usage where the `learn()` is passed in as a call-back function, to be executed whenever we see new examples.

```
function eg.bayes()
  local wme = {acc=0,datas={},tries=0,n=0}
   DATA.new("../data/diabetes.csv", function(data,t) learn(data,t,wme) end)
   print(wme.acc/(wme.tries))
   return wme.acc/(wme.tries) > .72 end
```

Implementing Naive Bayes in eight lines is not so impressive (since the underlying algorithm is so simple). A more interesting example of how SE can simplify AI is the 30 lines needed to code up sequential model optimization (see the rest of this paper).

As for other examples where SE+AI leads to simpler SE, this code has not (yet) been applied to all things AI. But the track record so far is pretty interesting. The methods of this document have also been applied to clustering, data synthesis, anomaly detection, privacy, multi-objective optimization, streaming, and many other applications besides. But that's another story for another time.

[1] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison Hidden Technical Debt in Machine Learning Systems Advances in Neural Information Processing Systems 28 (NIPS 2015)

[2] T. Menzies, "The Five Laws of SE for AI," in IEEE Software, vol. 37, no. 1, pp. 81-85, Jan.-Feb. 2020, doi: 10.1109/MS.2019.2954841.

2 Use of Lua

GATE is implemented in Lua since that is a great `less is more` language [3]. Though mainly a procedural language, Lua lends itself to several other paradigms, including OO, functional programming, and data-driven programming. It also offers good support for data description, in the style of JSON.

The language is great for teaching since it is very simple (only 19 keywods).

```
and   break   do    else   elseif   end      false    for    function   if
in    local   nil   not    or       repeat   return   then   true       until while
```

For more details on Lua, see the cheat sheet at the end of this document for a quick tour of Lua.

Lua has the advantage of looking like a simplified Python (see the
Bayes classifier example on the last page). So as an introductory exercise, I get graduate students to port my Lua to Python. In exams, I then give them Lua code, with typos, and ask them to recognize and fix the bugs. This encourages them to study and understand the code (and not just delegate the porting task to ChatGPT).

My code tries to honor the following conventions (but there are lapses. . .):

Small functions: More than six lines per function makes me nervous, five lines makes me happy, less than three makes me gleeful. Also, if a line just contains `end`, the I add it to the line above.

Narrow lines: 90 chars/line or less (otherwise the pdf printing messes up).

Few globals: *N-1* globals is better than *N*. I just try to have one, the global config settings `the` (and I build this variable from the help text listed top of file). Optionally, `the` can be updated from the command-line options.

Function arguments: Two spaces denotes "start of optionals" and four spaces denotes "start of locals"

Tests: From command line we can run one, or all tests. - Before running a test, we reset the random number seed; - After running a test, we reset all the settings to their defaults; - Each test returns `true``,false, or `nil` (and `false` means fail). - If we run `all``, the code returns the number of failed tests; (so`$?==0' means "no errors").

Structs: Classes in my Lua are defined as tables that know know to look up methods in themselves, before looking elsewhere:

```
local function isa(x,y) return setmetatable(y,x) end
local function is(s,    t) t={a=s}; t.__index=t; return t end
```

`function ZZZ.new(...)` is a constructor that returns a new struct of type `ZZZ`. My structs support encapsulation, polymorphism, but no inheritance. Why?: - See [4] [5] about the errors introduced by objects; - If you really want inheritance, use a langua ge that truly supports it, like Smalltalk or Crystal).

Type hints: For function argumnets (but not for locals) I try to apply the following stadards:

- `zzz` (or `zzz1``) = instance of class`ZZZ'.
- `x` is anything
- `n` = number
- `s` = string
- `xs` = list of many `x`
- `t` = table.
- `a` = array (index 1,2,3..)
- `h` = hash (indexed by keys)
- `fun` = function

[3]Roberto Ierusalimschy, Luiz Henrique De Figueiredo, Waldemar Celes Communications of the ACM, November 2018, Vol. 61 No. 11, Pages 114-123 10.1145/3186277

[4]Jack Diederich. "Stop Writing Classes". Youtube video. Mar 15 , 2012. https://youtu.be/o9pEzgHorH0?si=KWLVXsHuD_hwGtLz

[5]Hatton, Les. "Does OO Sync with How We Think?" IEEE Softw. 15 (1 998): 46-54. https://www.cs.kent.edu/~jmaletic/Prog-Comp/Papers/Hatton98.pdf

3 Sequential Model Optimization

Using Lua, this code implements sequential model optimization. Suppose we have a database of examples of some function $X = F(Y)$:

$$\underbrace{y_1, y_2, ...}_{y,\ goals,\ dependents} = F(\underbrace{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10} ...}_{x,\ independents,\ controllables,\ observables})$$

In many cases it is expensive to compute Y. For example:

- If humans have to do the assessment then pretty quickly they get tired and make mistakes, or they have to run off and to all their other tasks.
- Consider hardware/software co-design of a car where "evaluation" means building the car and taking it out for a 10,000 mile drive.

In these cases, SMO build two models *best,rest* on a handul of evaluated examples seen so far. It then tries to better define the border between *best* and *rest* by applying that model to all the unevaluated examples. This procedure returns the *most interesting example* which is the one with:

- highest probability of being in *best*,
- but also with a very similar *rest* score.

This most interesting example is then evaluated; the two *best* and *rest* models are rebuilt; and the above procedure is called again.

In the experiments shown below, we see that for many SE problems, SMO can build good models after evaluating just a few dozen examples. Why does SMO work so well? Well:

- Firstly, it does not waste time checking things it already knows (i.e. it does not look at examples that fall safely into just *best* or just *rest*);
- Secondly, it is critical of itself. SMO finds the examples that could most confuse it (those right on the border), then asks about those.
- Thirdly, there is maths (see below) saying that, under some optimistic assumptions, then it is possible to find good examples within a large set, using just a few examples.

Note that in my code, *best* and *rest* are implemented via a Naive Bayes classifier. Other researchers prefer more complex frameworks.

3.1 Some Maths (about Sampling)

Under certain assumptions, it can be shown that it might be possible to sample large spaces with just a few assumptions. While these assumptions are somewhat questionable (see below), they suggest that it is somewhat possible that a little sampling can explore a large space.

According to Hamlet [6], the confidence of seeing an event at probability p after n random trials is

$$C = 1 - (1 - p)^n$$

This can be rearranged to show the number of samples needed find that event (at some level of confidence) is $n = \log(1 - C)/\log(1 - p)$.

An estimate for p can be derived using other maths that says two numbers from the same distribution are insignificantly different if they differ by less than $d \times \delta$ where

- δ is the difference between two items, divided by the standard deviation;

[6] Hamlet, R. G. (1987). Probable correctness theory. Information processing letters, 25(1), 17-22.

- and d is Cohen's constant (usually .35) [7] [8].

How big is this region d? Well, a z-curve is a Gaussian curve with $\sigma = 1$ and $\mu = 0$. That curve effectively runs $-3 \leq x \leq 3$ which means that the position of a solution that is indistinguishable from the best solution has size $d/6$. Hence, for z-curves, we can to rewrite our equation.

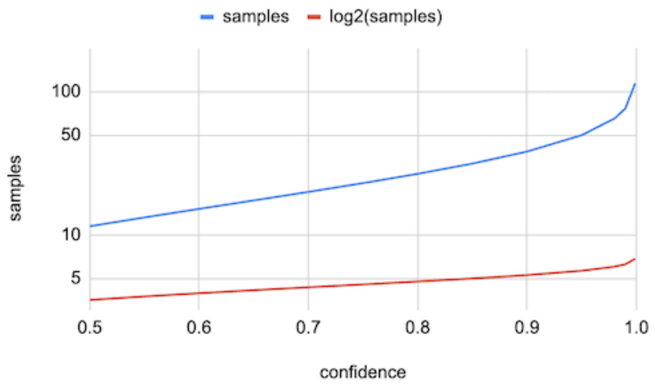
$$n = \log(1 - C)/\log(1 - d/6a)$$

An important aspect of this equation is that it does it mentions standard deviation, but not the size of the data. Hence, this maths works for 100 or 1,000,000 options, and tells us that as long as the variance is small, we do very much with just a little effort.

Further, it we have some way to sort the examples, then a binary chop will find that region after logarithmic samples. Putting all that together, then we should compare all the above to randomly sampling

$$n = \log_2 (\log(1 - C)/\log(1 - d/6))$$

The following graphs our two equations. Note that even without the sorting assumption, a few dozen examples can be quite informative especially if you are willing to accept low levels of confidence (e.g. 75%). And if it is possible to sort examples, then the number of samples required for confidence is far lower (less than dozen will get you near 100% confidence). These graphs go so way to explaining the strange success of human beings who, despite all their cognitive impairments, have managed to get to the moon, split the atom, and feed billions of people [9]. eg



It should be stressed that the assumptions behind these equations are *extremely optimistic*. For example, they assume that

- all solutions are spaced equally across decision space and objective space;
- all distributions conform to a Gaussian (single peak, symmetrical, continuous).

More importantly, they assume that

- we are *optimizing* which means we can quickly prune much of the space in our quest for a small corner with more desired properties.
- This is not true for classification and regressions tasks (where we seek labels across the Whole space) or generative tasks (where we need all the association weights between all the labels).

Nevertheless, it is good to know that for at least some tasks, humans have a chance of reasoning effectively about the world.

[7] Rosenthal, R., Cooper, H., Hedges, L.: Parametric measures of effect size. The handbook of research synthesis 621(2), 231–244 (1994)
[8] Sawilowsky, S.S.: New effect size rules of thumb. Journal of Modern Applied Statistical Methods 8(2), 26 (2009)
[9] For a depressing long list of ways that humans routinely get it wrong, see Wikipedia's list of cognitive biases.

4 Installation

4.1 Install LUA

```
brew install lua5.3 # mac os/x
sudo apt update; sudp apt install lua5.3 # debian/unix
```

4.2 Make directories

```
mkdir gate
mkdir gate/data
mkdir gate/src
```

4.3 Load the script.

```
cd gate/src
curl https://raw.githubusercontent.com/timm/lo/6jan24/src/gate.lua
```

4.4 Load the sample data.

```
cd ../data
Files="auto93  china   coc1000 coc10000 diabtes
       healthCloseIsses12mths0001-hard healthCloseIsses12mths0011-easy
       nasa93dem pom soybean"
for f in $Files do
  curl https://github.com/timm/lo/blob/6jan24/data/$f.csv
done
```

(Note that the diabetes and soybean and classification data sets (only one symbolic goal) while the rest are optimization data sets (To test the installation install

4.5 Test your installation.

```
cd ../src
lua gate.lua -t all
```

If this works, the last line of the test output should be say "PASS 0 fail(s)".

5 GATE's Input Data Format

GATE reads comma-seperated files (e.g. auto93.csv) whose first row names the columns.

- Names starting with uppercase are numeries; e.g. `Volume`. All other names are symbolic columns (e.g. `origin`)
- Names ending with X are ignored by the reasoning; e.g. `HpX`.
- Numeric names ending with `-` or `+` are goals to be minimized or maximized; e.g. `Lbs-` and `Acc+`.
- Symolic names ending with `!` are classes to be recognized; e.g. `happy!`.

```
          {Clndrs, Volume,HpX,   Model,  origin, Lbs-,  Acc+,  Mpg+}
1         {4,      97,   52,     82,      2,     2130,  24.6,  40}
2         {4,      90,   48,     80,      2,     2335,  23.7,  40}
3         {4,      90,   48,     78,      2,     1985,  21.5,  40}
4         {4,      90,   48,     80,      2,     2085,  21.7,  40}
...
394       {8,      429,  198,    73,      1,     4952,  11.5,  10}
395       {8,      383,  180,    71,      1,     4955,  11.5,  10}
396       {8,      440,  215,    70,      1,     4312,  8.5,   10}
397       {8,      455,  225,    73,      1,     4951,  11,    10}
398       {8,      400,  175,    71,      1,     5140,  12,    10}
```

Most of GATE's reasoning occurs over the X columns with only an occasional peek at the Y values.

But sometimes, just to generate baselines, we look at everything. For example, after looking at all the Y values, the above rows are sorted by `distance to heaven`:

$$H = \sqrt{\left(\sum_i^n (\overline{y_i} - h_i)^2\right) / n}$$

That is, for n goals normalized to $\overline{x_i}$ as 0..1 min..max, find the distance to the best value ($h_i = 0$ for goals we are minimizing and $h_i = 1$ for goals we are maximizing).

The net result is that the rows closest to the goals are shown first (lightest, fastest, most economical cars) and the worst cars are shown last (heaviest, slowest, worst MPG).

Note that sometimes, we do not know the contents of a cell. This is indicated with "?". Any code processing this kind of data has to check for missing values before doing any other calculation.

6 Help and Settings

The top of gate.lua is a help string from which this code extracts the system's config.

```
help =[[
gate: guess, assess, try, expand
(c) 2023, Tim Menzies, BSD-2
Learn a little, guess a lot, try the strangest guess, learn a little more, repeat

USAGE:
  lua gate.lua [OPTIONS]

OPTIONS:
  -c --cohen    small effect size               = .35
  -f --file     csv data file name              = ../data/diabetes.csv
  -h --help     show help                       = false
  -k --k        low class frequency kludge      = 1
  -m --m        low attribute frequency kludge  = 2
  -s --seed     random number seed              = 31210
  -t --todo     start up action                 = help]]
```

The help text parser a regular expression to find a word after two dashes:

```
[-][-]([%S]+)[^=]+= ([%S]+)
```

That pattern is used in `l.settings` as follows:

```
function l.settings(s,    t,pat) --> a dictionary with the config options
  t,pat = {}, "[-][-]([%S]+)[^=]+= ([%S]+)" -- @\ding{202}@
  for k, s1 in s:gmatch(pat) do t[k] = l.coerce(s1) end
  t._help = s
  return t end
```

That parser needs a function to coerce strings to (e.g.) numbers or booleans.

```
function l.coerce(s1,    fun) --> nil or bool or int or float or string
  function fun(s2)
    if s2=="nil" then return nil else return s2=="true" or (s2~="false" and s2) end end
  return math.tointeger(s1) or tonumber(s1) or fun(s1:match'^%s*(.*%S)') end
```

This code is used to generate a variable `the` storing the config.

```
the = l.settings(help)
```

Optionally, we can update the built in defaults via command-line flags using the `cli` function (which, incidently, uses coerce to turn command line strings into values):

```
function l.cli(t) --> the table `t` updated from command line
  for k, v in pairs(t) do
    v = tostring(v)
    for argv,s in pairs(arg) do
      if s=="-"..(k:sub(1,1)) or s=="--"..k then
        v = v=="true" and "false" or v=="false" and "true" or arg[argv + 1]
        t[k] = l.coerce(v) end end end
  if t.help then os.exit(print("\n"..t._help)) end
  return t end

the = l.cli(the)
```

7 Examples and Demos Library

The code ends with a few dozen tests that test/demo different parts of the system. For example, here are two examples:

- `eg.oo` tests the code for printing nested structures.
- `eg.sym` tests class that accepts atoms and can report the mode and entropy of that collection (code for `SYM` is presented below).

```
local eg={}

function eg.oo() --> bool
  return l.o{a=1,b=2,c=3,d={e=3,f=4}}  == "{a: 1, b: 2, c: 3, d: {e: 3, f: 4}}" end

function eg.sym(      s,mode,e) --> bool
  s = SYM.new()
  for _, x in pairs{1,1,1,1,2,2,3} do s:add(x) end
  mode, e = s:mid(), s:div()
  print(mode, e)
  return 1.37 < e and e < 1.38 and mode == 1 end
```

All these tests can be run at the command line:

```
lua gate.lua -t oo
lua gate.lua -t sym
```

The example library needs some support code. The `run` function checks what is returned by each example (and if it is `false`, then this test returns `true` indicating a failure). Note also that `run` has does some *setup* and *tearDown*:

- In the *tearDown* step, we restore the global config.
- As *setup*, the global config is cached (so it can be restored in `tearDown`) and the randomseed is restored to some default value.

```
local function run(k,    oops,b4) --> bool
  b4 = l.copy(the)          -- set up
  math.randomseed(the.seed) -- set up
  oops = eg[k]()==false
  io.stderr:write(l.fmt("# %s %s\n",oops and " FAIL" or " PASS",k))
  for k,v in pairs(b4) do the[k]=v end -- tear down
  return oops end
```

The `eg.all` command runs all the examples (via `run`) and returns to the operating system the number of failures. Note the use of `l.keys()`: this returns the example names, sorted alphabetically (so the tests are run in that order).

```
function eg.all(     bad) --> failure count to operating system
  bad=0
  for _,k in pairs(l.keys(eg)) do
    if k ~= "all" then
      if run(k) then bad=bad+1 end end end
  io.stderr:write(l.fmt("# %s %s fail(s)\n",bad>0 and " FAIL" or " PASS",bad))
  os.exit(bad) end
```

Note that, like anything else in eg, `eg.all` can be called from the command line

```
lua gate.lua -t all
```

8 Column Classes

GATE reads data and stores them in rows. Row columns are summarized in either NUMeric or SYMbolic column classes. These classes respond to the same polymorphic methods:

- `mid()`: central tendency (mean for NUMs and mode for SYMs);
- `div()`: the tendency to move away from `mid()` (standard deviation for NUMs and entropy for SYMs);
- `small()`: indistinguishable differences. For SYMs, NUMs, that is zero or .35*standard deviation [10];
- `like(x)`: how likely is `x` to belong to the distribution stored in the NUM or SYM

8.1 SYMbolic Classes

This class incrementally maintains a count of symbols seen so far, plus the most seen symbol (the `mode`).

```
local SYM=is"SYM"
function SYM.new(s,n) --> SYM
  return isa(SYM,{txt=s or " ", at=n or 0, n=0, has={}, mode=nil, most=0}) end

function SYM:add(x) --> nil
  if x ~= "?" then
    self.n = self.n + 1
    self.has[x] = 1 + (self.has[x] or 0)
    if self.has[x] > self.most then
      self.most,self.mode = self.has[x], x end end end

function SYM:mid()  --> any
   return self.mode end

function SYM:div(    e) --> num
  e=0; for _,v in pairs(self.has) do e=e-v/self.n*math.log(v/self.n,2) end; return e end

function SYM:small() --> 0
   return 0 end

function SYM:like(x, prior) --> num
  return ((self.has[x] or 0) + the.m*prior)/(self.n +the.m) end
```

Note the last method (`like`) has some low frequency tricks to handle lightly sampled regions of the data space (see `prior` and `the.m`).

[10] Sawilowsky, S.S.: New effect size rules of thumb. Journal of Modern Applied Statistical

8.2 NUMeric class

NUMerics incrementally update the control parameters of a Gaussian function [11].

This class supports the same methods as SYMbolics; i.e. `mid()`, `div()`, `small(x)`, `like(x,prior)`.

```
local NUM=is"NUM"
function NUM.new(s, n) --> NUM
  return isa(NUM,
          {txt=s or " ",                                 -- column name
           at=n or 0,                                    -- column position
           n=0, mu=0, m2=0                               -- used to calculate mean an sd
           hi=-1E30, lo=1E30,                            -- used when normalizing
           heaven = (s or ""):find"-$" and 0 or 1}) end  -- 0,1 for min,maximization

function NUM:add(x,     d) --> nil
  if x ~="?" then
    self.n  = self.n+1
    d       = x - self.mu
    self.mu = self.mu + d/self.n
    self.m2 = self.m2 + d*(x - self.mu)
    self.lo = math.min(x, self.lo)
    self.hi = math.max(x, self.hi) end end

function NUM:mid() --> num
  return self.mu end

function NUM:div() --> num
  return self.n < 2 and 0 or (self.m2/(self.n - 1))^.5 end
```

See Finch [12] for a proof that the above correctly and incrementally calculates mean `mu` and standard deviation (in `div()`).

```
function NUM:small() --> num
  return the.cohen*self:div() end

function NUM:like(x,_,      nom,denom) --> num
  local mu, sd =  self:mid(), (self:div() + 1E-30)
  nom   = 2.718^(-.5*(x - mu)^2/(sd^2))
  denom = (sd*2.5 + 1E-30)
  return  nom/denom end
```

`NUM:like(x)` is just a standard Gaussian probability distribution function[13].

NUMs have one ohter method called `norm(x)` that returns 0..1, min..max.

```
function NUM:norm(x) --> num
  return x=="?" and x or (x - self.lo) / (self.hi - self.lo + 1E-30) end
```

[11]https://en.wikipedia.org/wiki/Gaussian_function
[12]Tony Finch, "Incremental calculation of weighted mean and variance" University of Cambridge Computing Service, February 2009, https://fanf2.user.srcf.net/hermes/doc/antiforgery/stats.pdf
[13]https://en.wikipedia.org/wiki/Normal_distribution

9 DATA, ROW, and COLS Class

In order to read rows and summarize their contents in NUMeric or SYMbolic columns, we need three things:

1. Something to read the first row of our CSV files to create the right NUMs and SYMs columns. In the following, this will be the COLS object.
2. Some struct to hold the rows. In the following, this will be the ROW object.
3. Some place to store the rows and columns generated via points 1,2. In the following, this will be the DATA object.

So COLS store columns, ROWs stores a single record, and DATA stores the `rows` and `cols`. Due to their interconnections, explaining these classes have a bit of a "chicken and egg" problem. But lets see how we go:

DATA turns out to be extraordinarily useful for many things. E.g. if clustering, we an give each cluster its own DATA. For classification, we can store information about different classes in its own seperate data. For all these cases, if we want seperate stats on just one part of the data, we ensure that part is a DATA, then we call the `DATA:stats()` method (shown below).

9.1 COLS

Converts a list of stings into sets of columns, according to the rules described in *GATE's Input Data Format*.

COLS categorizes the columns as `all,x,y,klass`:

```
local COLS=is"COLS"
function COLS.new(row) --> COLS
  local x,y,all = {},{},{}
  local klass,col
  for at,txt in pairs(row.cells) do
    col = (txt:find"^[A-Z]" and NUM or SYM).new(txt,at)
    all[1+#all] = col
    if not txt:find"X$" then
      if txt:find"!$" then klass=col end
      (txt:find"[!+-]$" and y or x)[at] = col end end
  return isa(COLS,
          {x     = x,         -- all the independent columns
           y     = y,         -- all the dependent columns
           all   = all,       -- all the columns
           klass = klass,     -- just the klass column (if it exists)
           names = row.cells -- names of all the columns
          }) end
```

The `COLS:add()` method takes a ROW and updates the `x,y` columns.

```
function COLS:add(row) --> row
  for _,cols in pairs{self.x, self.y} do
    for _,col in pairs(cols) do
      col:add(row.cells[col.at]) end end
  return row end
```

9.2 ROWs

The following ROWs code uses two features of DATA object (defined below):

- `data.cols` : what comes out of the above COLS code;
- `data.rows` : stores all the rows.

```
local ROW=is"ROW"
function ROW.new(t) return isa(ROW, { cells = t }) end
```

Here's distance to heaven (as defined above in *GATE's Input Data Format*).

```
function ROW:d2h(data, d, n) --> num in the range 0..1
  d, n = 0, 0
  for _, col in pairs(data.cols.y) do
      n = n + 1
      d = d + math.abs(col.heaven - col:norm(self.cells[col.at])) ^ 2 end
  return d ^ .5 / n ^ .5 end
```

ROWs implements the likelihood calculation of Baye's rule

- Suppose we have rows describing `nHypotheses` number different things (dogs, horse, cats) etc. We can work out how much a ROW `likes` being a dog, horse, or cat.
- The `prior` is the ratio of (e.g.) `dogs` amongst all the `dogs,horses,cats`. e.g. Suppose we have 100 dogs horses and cats. Then our prior belief in dogs is $\frac{100}{300} = 0.33$.
- Then we multiply the prior by how frequently we see ROW values amongst (e.g.) the dogs; i.e.:

$$like(H|E) = prior \times \sum_x like(E_x|H)$$

Here H is one of the hypotheses (e.g. dogs, horses, or cats); and E is the *evidence* available for each hypothesis (this is just the distributions seen in ROWs of a DATA). This likelihood calculation is coded as follows:

```
function ROW:like(data,n,nHypotheses,       prior,out,v,inc) --> num
  prior = (#data.rows + the.k) / (n + the.k * nHypotheses)
  out   = math.log(prior) -- use logs to handle very small numbers
  for _,col in pairs(data.cols.x) do
    v= self.cells[col.at]
    if v ~= "?" then
      inc = col:like(v,prior)
      out = out + math.log(inc) end end
  return math.exp(1)^out end
```

(Aside: we use some low frequency tricks when calculating *prior*, see `the.k`.)

Once we can find `like` for one hypothesis, we can implement a classifier that searches many hypotheses. Note that in this example `datas` contains key-value pairs where the key is a class name and the value is a DATA devoted just to rows from that class.

```
function ROW:likes(datas,       n,nHypotheses,most,tmp,out) --> sym,num
  n,nHypotheses = 0,0
  for k,data in pairs(datas) do
    n = n + #data.rows
    nHypotheses = 1 + nHypotheses end
  for k,data in pairs(datas) do
    tmp = self:like(data,n,nHypotheses)
    if most==nil or tmp > most then most,out = tmp,k end end
  return out,most end
```

9.3 DATA

DATA is the ringmaster than reads information (from disk, from another source), creates ROWs (if reading from disk), creates the COLS, updates the COLS, and stores the ROWs.

DATE's creation methods reads from disk if `src` is a filename string. Note the optional `fun` (a function). This is some call-back function that will be executed each time we see a new ROW.

```lua
local DATA=is"DATA"
function DATA.new(src,  fun,     self) --> DATA
  self = isa(DATA,{rows={}, cols=nil})
  if   type(src) == "string"
  then for _,x in l.csv(src)       do self:add(x, fun) end
  else for _,x in pairs(src or {}) do self:add(x, fun) end end
  return self end
```

DATE's update method ensures that the thing being added is a row [14]. After that, there are two cases. Firstly, if this is the first ROW we have seen, then we need to create a COLS object. Otherwise, we need to update our COLS with information from this ROW, then store the ROW.

```lua
function DATA:add(t,  fun,row) --> nil
  row = t.cells and t or ROW.new(t)
  if   self.cols
  then if fun then fun(self,row) end
       self.rows[1 + #self.rows] = self.cols:add(row)
  else self.cols = COLS.new(row) end end
```

Note in the above that the `fun` call-back is executed just before we update each row.

One use of DATA is to report stats from different parts of the data. Those stats are computed recursively using the `mid(), div()` functions of NUM, SYM (see above), `DATA:stats()` is the most general form on these queries. It can be called different ways to return different stats, rounded to however many decimals you like, for what ever cols you like. If called with no arguments, it reports the `mid()` of the y columns, rounded to 2 decimal places.

```lua
---           stats(cols:str,fun:str, ndivs:int)
function DATA:stats(  cols,fun,ndivs,    u) --> table
  u = {[".N"] = #self.rows}
  for _,col in pairs(self.cols[cols or "y"]) do
    u[col.txt] = l.rnd(getmetatable(col)[fun or "mid"](col), ndivs or 2) end
  return u end
```

The other queries are more specific:

```lua
function DATA:mid(cols,   u) --> ROW
  u = {}; for _, col in pairs(cols or self.cols.all) do u[1 + #u] = col:mid() end
  return ROW.new(u) end

function DATA:div(cols,    u) --> ROW
  u = {}; for _, col in pairs(cols or self.cols.all) do u[1 + #u] = col:div() end;
  return ROW.new(u) end

function DATA:small(    u)a --> ROW
  u = {}; for _, col in pairs(self.cols.all) do u[1 + #u] = col:small(); end
  return ROW.new(u) end
```

[14]When reading from disk, the information arrives in a raw Lua table and *not* a ROW. In that case, we create a new ROW fom that table– see first line of `DATA:add()`.

10 Sequential Model Optimization with GATE

As promised in the introduction, once the right data structures are in place, then implementing an AI is very simple. For example, here is the GATE Sequential Model Optimizer in under 30 lines.

This code uses the following terminoly:

- Examples are `dark` if we do not know that Y values. Otherwise they are `lite`. Initially, all the examples are `dark`.
- The `lite` examples divide into two sets: `best` and `rest` where `best` are closer to heaven than `rest` (where `heaven` was defined above in `ROW:d2d()`).
- As GATE executes, the most informative examples are moved one at a time from `dark` to `lite` where they are then classified as `best` or `rest`.

This means our examples divide as follows:

$$examples = \left(\underbrace{\overbrace{lite}^{y=known}}_{best,rest}, \quad \overbrace{dark}^{y=unknown} \right)$$

```
function DATA:bestRest(rows, want, best, rest, top) --> DATA, DATA
    table.sort(rows, function(a, b) return a:d2h(self) < b:d2h(self) end)
    best, rest = { self.cols.names }, { self.cols.names }
    for i, row in pairs(rows) do
        if i <= want then best[1 + #best] = row else rest[1 + #rest] = row end end
    return DATA.new(best), DATA.new(rest) end
```

Now what is fun here is that GATE runs on so few examples. Intially, we start with just four examples in the `lite` so `DATA:bestRest()` only wants to move $\sqrt{4} = 2$ examples into the `best` and `rest`.

GATE then tries to better define the border between *best* and *rest* by applying that model to all the unevaluated examples. This procedure returns the *most interesting example* which is the one with:

- Highest likelihood b of being in *best*,
- But also with a very similar r *rest* likelihood.
- Specifically, we use $abs(b+r)/abs(b-r)$

Just for the evaluation purposes, GATE returns the example that maximizes the above measure as well as all the *dark* examples that would have been `selected` by the model learned to date (i.e. all the examples where b is greater than r).

```
function DATA:split(best,rest,lite,dark)
  local selected,max,out
  selected = DATA.new{self.cols.names}
  max = 1E30
  out = 1
  for i,row in pairs(dark) do
    local b,r,tmp
    b = row:like(best, #lite, 2)
    r = row:like(rest, #lite, 2)
    if b>r then selected:add(row) end
    tmp = math.abs(b+r) / math.abs(b-r+1E-300)
    --print(b,r,tmp)
    if tmp > max then out,max = i,tmp end end
  return out,selected end
```

10.1 Main Loop

The main loop of GATE (shown below) squirrels away the `selected` rows ito the `stats` list. This list can be used to report how well we would have found `best` and `rest` things if we'd stopped after a budget of B evaluations.

The main loop also show the sequential optimization process. At each iteration of that loop, one more example is removed from `dark` and moved into the `lite`. Them, in the next iteration, the `lite` is our `best,rest` model is rebuilt and the process repeats.

```
function DATA:gate(budget0,budget,some)
  local rows,lite,dark
  local stats,bests = {},{}
  rows = l.shuffle(self.rows)
  lite = l.slice(rows,1,budget0)
  dark = l.slice(rows, budget0+1)
  for i=1,budget do
    local best, rest     = self:bestRest(lite, (#lite)^some)  -- assess
    local todo, selected = self:split(best,rest,lite,dark)
    stats[i] = selected:mid()
    bests[i] = best.rows[1]
    table.insert(lite, table.remove(dark,todo)) end
  return stats,bests end
```

Rebuilding the model for each loop seems like a slow thing to do, but there are two things to remember here. First, we are using Naive Bayes classifiers here so rebuilding just means adding some numbers to some frequency counts. Secondly, we are rebuilding using very small data sets. In the runs shown below, `lite` ever grows to more than 20 examples so the whole process is very fast.

Before showing this running, one last comment:

- In `DATA:gate()` the line `stats[i] = selected:mid()` collects the the median of the values selected at this iteration;
- While the line `bests[i] = best.rows[1]` collects the best row seen at each iteration

`bests` reports the best performance seen on the current data while `stats` reports a prediction for how well this model might perfrom, on avewrage, on new data (from the same source as the `lite,dark` used for this training). The different between "best performance" and "on average" is the different between optimism and realism:

- Optimistically we expect the best;
- But realistically, we think we will see the average.

10.2 Running the code

Here's the function that runs GATE. Its a bit of mess (all report drivers usually are) but it can be simplified into four parts:

First we set the control parameters:

```
function eg.gate(stats, bests, d, say,sayd)
  local budget0,budget,some = 4,10,.5
```

That is, our initial `lite` sample will be 4 and after that we will iterate 10 times (total evaluations = 14). At each step, the `lite` examples will be divided into $|lite|^{0.5}$ (i.e. square root) *best* and the *rest*.

Next we do some initial set up. We report the random seed we are using (useful for reproduction) and print statistics that baseline our data. Reporting such baselines is an important principle of experimental SE: always compare your supposedly better system against some baseline:

```
-- eg.gate(), continued
print(the.seed)
d = DATA.new("../data/auto93.csv")
function sayd(row, txt) print(l.o(row.cells), txt, l.rnd(row:d2h(d))) end
function say( row,txt)  print(l.o(row.cells), txt) end
print(l.o(d.cols.names),"about","d2h")
print"#overall" -------------------------------------
sayd(d:mid(), "mid")
say(d:div() , "div")
say(d:small(),"small=div*"..the.cohen)
```

Note the called to `small()`. This illustrates another important principles. Two numbers from the same distribution are the insignificantly different if they differ by less than a small amount. This knowledge lets us rule out trivial results.

Next we run GATE and report the average and optimal results (using `stats` and `bests`).

```
-- eg.gate(), continued
print"#average" ----------------------------------
stats,bests = d:gate(budget0, budget, some)
for i,stat in pairs(stats) do sayd(stat,i+budget0) end
print"#iterative" ----------------------------------------------------------
for i,best in pairs(bests) do sayd(best,i+budget0) end
print"#optimum" ------------------------------------------------------
table.sort(d.rows, function(a,b) return a:d2h(d) < b:d2h(d) end)
sayd(d.rows[1], #d.rows)
```

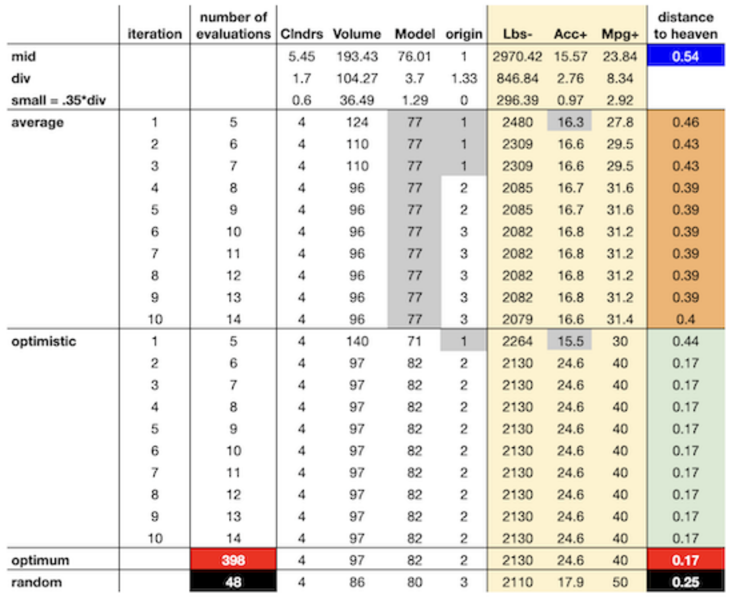
Finally, we compare our results against the simple sampling policy we know (random, no ordering). This implies selecting n examples at random an evaluation them all:

$$n = \log(1-(C=.95))/\log(1-(d=.35)/6) = 49$$

This is also good empirical practice: compare your results against some reasonable (but simple) search.

```
-- eg.gate(), continued
print"#random" ------------------------------------------------------
local rows=l.shuffle(d.rows)
rows = l.slice(rows,1,math.log(.05)/math.log(1-the.cohen/6))
table.sort(rows, function(a,b) return a:d2h(d) < b:d2h(d) end)
sayd(rows[1]) end
```

10.3 Output



	iteration	number of evaluations	Clndrs	Volume	Model	origin	Lbs-	Acc+	Mpg+	distance to heaven
mid			5.45	193.43	76.01	1	2970.42	15.57	23.84	0.54
div			1.7	104.27	3.7	1.33	846.84	2.76	8.34	
small = .35*div			0.6	36.49	1.29	0	296.39	0.97	2.92	
average	1	5	4	124	77	1	2480	16.3	27.8	0.46
	2	6	4	110	77	1	2309	16.6	29.5	0.43
	3	7	4	110	77	1	2309	16.6	29.5	0.43
	4	8	4	96	77	2	2085	16.7	31.6	0.39
	5	9	4	96	77	2	2085	16.7	31.6	0.39
	6	10	4	96	77	3	2082	16.8	31.2	0.39
	7	11	4	96	77	3	2082	16.8	31.2	0.39
	8	12	4	96	77	3	2082	16.8	31.2	0.39
	9	13	4	96	77	3	2082	16.8	31.2	0.39
	10	14	4	96	77	3	2079	16.6	31.4	0.4
optimistic	1	5	4	140	71	1	2264	15.5	30	0.44
	2	6	4	97	82	2	2130	24.6	40	0.17
	3	7	4	97	82	2	2130	24.6	40	0.17
	4	8	4	97	82	2	2130	24.6	40	0.17
	5	9	4	97	82	2	2130	24.6	40	0.17
	6	10	4	97	82	2	2130	24.6	40	0.17
	7	11	4	97	82	2	2130	24.6	40	0.17
	8	12	4	97	82	2	2130	24.6	40	0.17
	9	13	4	97	82	2	2130	24.6	40	0.17
	10	14	4	97	82	2	2130	24.6	40	0.17
optimum		398	4	97	82	2	2130	24.6	40	0.17
random		48	4	86	80	3	2110	17.9	50	0.25

Here's the out from the above code, annotated with some colors.

- Yellow denotes the goal columns;
- Blue shows results from the centroid of our data. Here we see that, on average our unoptimized data isa round 0.54 away from heaven. In that unoptimized data, we see weights, acceleartions and miles per hour in our cars of around (3000,16,24).
- Gray cells denote optimization results where the new values are only a small difference from the original. When intepreting these results, we cannot make conclusions from these gray cells.
- Red cells shows the optimum results; i.e. how well we can do if we evaluate all the data. Here we see that after 398 evaluations we can find weights, acceleartions and miles per hour in our cars of around (2300, 25,40).
- In the optimistic case (shown in the green rows), afer 14 evaluations, we can reach weights, acceleartions and miles per hour of (2100, 25,40).
- More realistically (in the orange rows) we after 14 evaluations, we can reach weights, acceleartions and miles per hour of (2000,17,31). While this is clearly not as good as the optimal case, its not bad after jsut 14 evaluations.
- Black results shows what happens if we just evalauted 49 examples. It do not perform as well as the red optimal but does better than the green.

The above results suggest we might want to think more about our stopping rules:

- The black results suggest we might want to reflect on our stopping rules (since our results after 14 evaluations) are close, but not as good as after 49 evalautions.
- On the other hand, the results after 7 evalautions are not necessirily better than than what we see after 14. So we might be able to stop even earlier than 14.

Finally, the results seen from one run in one data set really need to be checked using (say) 10 to 20 data sets and say 20 runs (with different random number seeds). But that is another story for another time.

11 Appendix: A Quick Tour of Lua

Source: from https://github.com/rstacruz/cheatsheets/.

11.1 Basic examples

11.1.1 References

- https://www.lua.org/pil/13.html

11.1.2 Comments

```
-- comment
--[[ Multiline
     comment ]]
```

11.1.3 Invoking functions

```
print()
print("Hi")

-- You can omit parentheses if the argument is one string or table literal
print "Hello World"     <-->     print("Hello World")
dofile 'a.lua'          <-->     dofile ('a.lua')
print [[a multi-line    <-->     print([[a multi-line
 message]]                        message]])
f{x=10, y=20}           <-->     f({x=10, y=20})
type{}                  <-->     type({})
```

11.1.4 Tables / arrays

```
t = {}
t = { a = 1, b = 2 }
t.a = function() ... end

t = { ["hello"] = 200 }
t.hello

-- Remember, arrays are also tables
array = { "a", "b", "c", "d" }
print(array[2])       -- "b" (one-indexed)
print(#array)         -- 4 (length)
```

11.1.5 Loops

```
while condition do
end

for i = 1,5 do
end

for i = start,finish,delta do
end

for k,v in pairs(tab) do
```

```
end

repeat
until condition

-- Breaking out:
while x do
  if condition then break end
end
```

11.1.6 Conditionals

```
if condition1 then
  print("yes")
elseif condition2 then
  print("maybe")
else
  print("no")
end
```

print(condition1 and "yes" or (condition2 and "maybe" or "no")

11.1.7 Variables

Variables are global by default. They can be mde local using the `local` keyword, or if they are included as extra arguments in functions.

```
local x = 2
two, four = 2, 4
```

11.1.8 Functions

```
function myFunction()
  return 1
end
```

If `myFunctionWithArgs` is called with one arguments, then `b` is a local variables.

```
function myFunctionWithArgs(a, b)
  -- ...
end

myFunction()

anonymousFunctions(function()
  -- ...
end)

-- Not exported in the module
local function myPrivateFunction()
end

-- Splats
function doAction(action, ...)
  print("Doing '"..action.."' to", ...)
  --> print("Doing 'write' to", "Shirley", "Abed")
end
```

```
doAction('write', "Shirley", "Abed")
```

11.1.9 Lookups

```
mytable = { x = 2, y = function() .. end }

-- The same:
mytable.x
mytable['x']

-- Syntactic sugar, these are equivalent:
mytable.y(mytable)
mytable:y()

mytable.y(mytable, a, b)
mytable:y(a, b)

function X:y(z) .. end
function X.y(self, z) .. end
```

11.2 More concepts

11.2.1 Constants

```
nil
false
true
```

11.3 Operators (and their metatable names)

11.3.1 Logic

```
-- Logic (and/or)
nil and false  --> nil
false and nil  --> false
0 and 20       --> 20
10 and 20      --> 20
```

11.3.2 Tables

```
-- Length
-- __len(array)
#array


-- Indexing
-- __index(table, key)
t[key]
t.key
```

11.4 API

11.4.1 API: Some Global Functions

```
assert(x)    -- x or (raise an error)
assert(x, "failed")

type(var)  -- "nil" | "number" | "string" | "boolean" | "table" | "function" | "thread" | "userdata"

_ENV  -- Global context
setfenv(1, {})  -- 1: current function, 2: caller, and so on -- {}: the new _G

pairs(t)     -- iterable list of {key, value}

tonumber("34")
tonumber("8f", 16)
```

11.4.2 API: Strings

```
'string'..'concatenation'

s = "Hello"
s:upper()
s:lower()
s:len()    -- Just like #s

s:find()
s:gfind()

s:match()
s:gmatch()

s:sub()
s:gsub()

s:rep()
s:char()
s:dump()
s:reverse()
s:byte()
s:format()
```

11.4.3 API: Tables

```
table.foreach(t, function(row) ... end)
table.setn
table.insert(t, 21)          -- append (--> t[#t+1] = 21)
table.insert(t, 4, 99)
table.getn
table.concat
table.sort
table.remove(t, 4)
```

11.4.4 API: Math

```
math.abs     math.acos    math.asin       math.atan    math.atan2
math.ceil    math.cos     math.cosh       math.deg     math.exp
math.floor   math.fmod    math.frexp      math.ldexp   math.log
math.log10   math.max     math.min        math.modf    math.pow
math.rad     math.random  math.randomseed math.sin     math.sinh
math.sqrt    math.tan     math.tanh
```

11.4.5 API: Misc

```
io.output(io.open("file.txt", "w"))
io.write(x)
io.close()

for line in io.lines("file.txt")

file = assert(io.open("file.txt", "r"))
file:read()
file:lines()
file:close()
```