```
local help=[[

XPLAN.lua: semi-supervised multi-objective explanation
(c)2022, Tim Menzies <timm@ieee.org>, BSD-2 license

SYNOPSIS:
  Data, with multiple dependent goals, is recursive
  bi-clustered on independent variables by partitioning
  on the distance to two distant points (found after a
  few dozen random projections).  A decision tree reports
  the difference between the "best" and "worst" clusters
  (defined using a multi-objective domination predicate).
  This process only makes log2(N) queries to y-values
  (while clustering, just on the pairs of distance objects).

USAGE:
  lua xplan.lua [OPTIONS]

OPTIONS:
 -b bins          max number of bins    = 16
 -c cohen         difference in nums    = .35
 -f file          source                = ../../data/auto93.csv
 -F Far           how far is far        = .95
 -g go            action                = nothing
 -h help          show help             = false
 -m min           size of small         = .5
 -p p             distance coefficient  = 2
 -r rests         number of rests to use= 4
 -P Projections   number of random projections   = 64
 -s seed          random number seed    = 10019 ]]

---- ---- ---- ---- Names
---- ---- Locals
-- Store old names (so, on last line, we can check for rogue locals)
local b4={}; for k,_ in pairs(_ENV) do b4[k]=k end
-- Define new names from library code (so we can mention them before defining them).
local any,big,cat,chat,cli,coerce,csv,data,fmt,kap,lt
local map,max,min,on,per,push,rand,rnd,rnds,shuffle,sort,sum
-- Place for test suites (to disable a test, move it `go` to `no`.
local go,no = {},{}
-- Place for settings (parsed from help text; e.g. `the.bins=16`).
local the = {}

---- ----  Objects
----  obj(str,fun): class -- `Fun` is a constructor for instances of class `str`.
-- Polymorphism, encapsulation, classes, instance, constructors: all in 3 lines. :-)
local function obj(txt,fun,  t,i)
  local function new(k,...) i=setmetatable({},k); fun(i,...); return i end
  t={__tostring = cat}; t.__index = t;return setmetatable(t,{__call=new}) end

---- ROW(tab) -- Stores one record. ROWs are stored in a contained called ROWS.
-- Implementation note: ROWs are created when data is read from CSV files.
-- After that, if a ROW is added to more than one ROWS object then the same
-- ROW will be held in different ROWSs. This makes certain labelling and
-- record keep tasks easier (e.g. tracking how many rows we have evaluated).
local ROW=obj("ROW", function(self,cells)
  self.cells = {}             -- place to hold one record
  self.label  = false         -- true if we have decided  this ROW is "best"?
  self.evaled = false end)    -- have we accessed this row's y-values?

----  SYM(?num=0, ?str="") -- Summarizes streams of symbols in ROWs
local SYM=obj("SYM", function(self,at,txt)
  self.n   = 0          -- number of items seen
  self.at  = at or 0    -- column number
  self.txt = txt or ""  -- column name
  self.kept= {}   end)  -- counters for symbols

---- NUM(?num=0, ?str="") -- Summarize streams of numbers in ROWs
local NUM=obj("NUM", function(self,at,txt)
  self.n   = 0                      -- number of items seen
  self.at  = at    or 0             -- column number
  txt=txt or ""
  self.txt = txt                    -- column name
  self.w   = txt:find"-$" and -1 or 1 -- If minimizing, then -1. Else 1
  self.kept= {}                     -- some sample of the seen items
  self.ok  = false end)             -- true if sorted, set to false by each add

----  COLS([str]+) -- Factory for making NUMs or SYMs from list of col names.
-- Column names starting with upper case are NUMs (others are SYMs).
-- Anything adding with "!+-" is a dependent goal column.
-- Column names ending with ":`" are "skipped"; i.e.
-- not added to the list of independent or dependent columns.
local COLS=obj("COLS", function(self,  names)
  self.names= names -- list of column names
  self.all  = {}    -- [NUM|SYM] all names, converted to NUMs or SYMs
  self.x    = {}    -- [NUM|SYM] just the independent columns
  self.y    = {}    -- [NUM|SYM] just the dependent columns
  self.klass= nil   -- SYM       the klass column (if it exists)
  for k,v in pairs(names) do
    col= push(self.all, (v:find"^[A-Z]" and NUM or SYM)(at,txt))
    if not v:find":$" then
      if v:find"!$" then self.klass = col end
      push(v:find"[!+-]$" and self.y or self.x, col) end end end)

---- ROWS() -- Stores `rows` and their summaries in `cols`.
local ROWS=obj("ROWS", function(self)
  self.rows = {}        -- [ROW] records, stored as ROW
  self.cols = nil end)  -- a COLS instance (if nil, no data read yet)

---- BIN( NUM|SYM, num, ?num=lo, ?SYM) -- Values from same rows in 2 columns
local BIN=obj("BIN",function(self,col, lo, hi, has)
  self.col = col              -- What column does this bin handle?
  self.lo  = lo               -- Lowest value of columnl.
  self.hi  = hi or lo         -- Highest value of columnl
  self.has = has or SYM() end) -- Symbol counts of column2 values.


---- ---- ---- ----  Columns
---- ---- ---- Sym
---- ---- Create
---- SYM:merge(SYM): SYM --  Create a new SYM by merging two others.
function SYM:merge(other,    k)
  k= SYM(self.at, self.txt)
  for x,n in pairs(self.kept)  do k:add(x,n) end
  for x,n in pairs(other.kept) do k:add(x,n) end
  return k end

---- ---- Update
---- SYM:add(any,?num=1) -- Add a symbols `x`. Do it `n` times.
function SYM:add(x,n)
  n = n or 1
  if x~="?" then self.n=self.n+n; self.kept[x]=n + (self.kept[x]+0) end end

---- ---- Query
----  SYM:div():num -- Diversity. Return entropy.
function SYM:div()
  return sum(self.kept, function(n) return -n/i.n*math.log(n/i.n,2) end) end

---- SYM:mid():num -- Return `mid`dle (mode) symbol.
function SYM:mid()
  local most,mode = -1,nil
  for x,n in pairs(self.kept) do if n>most then most,mode=n,x end end
  return mode end

---- ----  Distance
---- SYM:dist(atom,atom):num -- Identical symbols have distance 0. Otherwise, 1.
-- If any unknowns, assume max distance.
function SYM:dist(x,y) return (x=="?" or  y=="?") and 1 or x==y and 0 or 1 end

---- ---- Discretization
---- SYM:bin(any):any -- Discretize a symbol (do nothing)
function SYM:bin(x) return x end
---- SYM:merges(t,...):SYM --  Merge adjacent bins (do nothing: SYMs don't merge)
function SYM:merges(t,...) return t end

---- ---- ---- Num
---- ---- Update
---- NUM:add(num) -- Add `x`. If no space, at prob `some/n`, replace any old number.
function NUM:add(x)
  if x~="?" then
    self.n = self.n + 1
    local pos
    if #self.kept < the.some        then pos= #i.kept+1
    elseif rand() < the.some/self.n then pos= rand(#i.kept) end
    if pos then
      self.ok=false  -- the `kept` list is no longer in sorted order
      self.kept[pos]=x end end end

---- ---- Query
----  NUM:has() -- Return `kept`, ensuring it is sorted.
function NUM:has()
  self.kept = self.ok and self.kept or sort(self.kept)
  self.ok = true
  return self.kept end

---- NUM:mid():num -- Return `mid`dle (median) number.
function NUM:mid() return per(self.has(),.5) end

---- NUM:norm(x):num -- Normalize x,y to 0..1.
function NUM:norm(x)
  local a =  self:has()
  local lo,hi = a[1], a[#a]
  return x=="?" and x or math.abs(hi-lo)<1E-9 and 0 or (x-lo)/(hi-lo+1/big) end

---- ---- Distance
---- NUM:dist(x,y):num -- Normalize x,y to 0..1, report their difference.
-- If any unknowns, assume max distance.
function NUM:dist(x,y)
  if x=="?" and y=="?" then return 1
  elseif x=="?" then y=self:norm(y); x=y<.5 and 1 or 0
  elseif y=="?" then x=self:norm(x); y=x<.5 and 1 or 0
  else   x,y = self:norm(x), self:norm(y) end
  return math.abs(x-y) end

---- ---- Discretization
---- NUM:bin(any):any -- Discretize a num to one of `the.bins`.
function NUM:bin(x)
  local a = self:has()
  local lo,hi = a[1], a[#a]
  local b = (hi - lo)/the.bins
  return hi==lo and 1 or math.floor(x/b+.5)*b end

---- NUM:merges({BIN],...) :[BIN] -- Prune superflous bins.
function NUM:merges(b4, min)
  local n,now = 1,{}
  while n <= #b4 do
    local merged = n<#b4 and b4[n]:merged(b4[n+1],min) -- defined in BIN
    now[#now+1]  = merged or b4[n]
    n            = n + (merged and 2 or 1)  -- if merged, skip over merged bin
  end -- end while
  if #now < #b4 then return self:merges(now,min) end     -- seek others to merge
  bins[1].lo,bins[#bins].hi = -big,big            -- grow to plus/minus infinity
  return bins end


---- ---- ---- ---- Data
---- ---- ---- COLS
---- ---- Update
---- COLS:add(ROW) --  update the non-skipped columns with values from ROW
function COLS:add(row)
  for _,cols in pairs{self.x, self.y} do
    for _,col in pairs(cols) do col:add(row.cells[col.at]) end end end

---- ---- Distance
---- COLS:dist(ROW,ROW) :num --  Using `x` columns, compute distance.
function COLS:dist(r1,r2)
  local d,x1,x2 = 0
  for _,col in pairs(self.x) do
    x1 = r1.cells[col.at]
    x2 = r2.cells[col.at]
    d  = d+(col:dist(x1,x2))^the.p end
  return (d/#self.x)^(1/the.p) end

---- COLS:half([ROW]) -- Divide `rows` by their distance to two distant points A,B
-- Find two distant points A,B using a few dozen random projections.
function COLS:half(rows,    b4)
  local function ABc (A,B) return {A=A, B=B, As={}, Bs={}, c=self:dist(A,B)} end
  local ABcs={}
  for n=1,the.Projections do
    push(ABcs, ABc(b4 or any(rows), -- if b4, use it for one pole
                   any(rows))) end
  local i = per(sort(ABcs,lt"c"), the.Far) -- avoid outliers: only go so Far
  local function xCs(C)
          return {x = (self:dist(C,i.A)^2+i.c^2-self:dist(C,i.B)^2)/(2*i.c),
                  C = C} end
  for j,xC in pairs(sort(map(rows,xCs),lt"x")) do
    push(j<#rows/2 and i.As or i.Bs, xC.C) end
  return i end

---- ---- Optimize
----  COLS:best(ROW,ROW) :bool -- True if better on multi-objectives
function COLS:best(r1,r2)
  r1.evaled,r2.evaled = true,true
  local s1, s2, ys, e = 0, 0, self.y, math.exp(1)
  for _,col in pairs(ys) do
    local x = col:norm(r1.cells[col.at])
    local y = col:norm(r2.cells[col.at])
    s1      = s1 - e^(col.w * (x-y)/#ys)
    s2      = s2 - e^(col.w * (y-x)/#ys) end
  return s1/#ys < s2/#ys end -- i.e. we lose less going to r2->r1 than r2->r1

---- COLS:bests([ROW]):bests=[ROW],rests=[ROW] -- Recursively apply `best`.
-- Returns `bests` and everything else as `rest`.
function COLS:bests(rows,    b4,stop,rests)
  rests = rests or {}
  stop  = stop or (#rows)^the.min
  if #rows < stop then return rows,rests end -- return best=[ROW],rests=[ROW]
  local two = self:half(rows,b4) -- if b4 supplied, then half will use it as one pole.
  local best, bests, rests1
  if   self:best(two.A, two.B)
  then best, bests, rests1 = two.A, two.As, two.Bs
       for i=#rests1,1,-1 do push(rests,rests1[i]) end --sort L to R, worst to better
  else best, bests, rests1 = two.B, two.Bs, two.As
       for i=1,#rests1, 1 do push(rests,rests1[i]) end --sort L to R, worst to better
  end
  return self:best(bests, best, stop,rests) end

---- ---- ---- ---- COLS
---- ---- ---- BINS
---- ---- Create
---- BIN:merged(BIN, num) -- Combine two bins if we should or can do so.
-- "Should" is true if either is too small.
-- "Can" is true of the whole is simpler than the parts.
function BIN:merged(j, min)
  local a, b, c = self.has, j.has, self.has:merge(j.has)
  local should = a.n < min or b.n < min
  local can    = c:div() <= (a.n*a:div() + b.n*b:div())/c.n
  if should or can then return BIN(a.col,self.lo, j.hi, c) end end

---- ---- Update
---- BIN:add(num,sym) -- extend `lo,hi` to cover `x`; remember we saw `y.
function BIN:add(x,y)
  self.lo = min(x,self.lo)
  self.hi = max(x,self.hi)
  self:has(y) end

---- ---- Query
---- BIN:show() -- pretty print the range
function BIN:show(i)
  local x,lo,hi = self.ys.txt, self.lo, self.hi
  if     lo ==  hi  then return fmt("%s == %s", x, lo)
  elseif hi ==  big then return fmt("%s > %s", x, lo)
  elseif lo == -big then return fmt("%s <= %s", x, hi)
  else                   return fmt("%s < %s <= %s", lo,x,hi) end end

---- BIN:selects([ROW]):[ROW] -- Returns rows that fall within this BIN.
-- Returns nil if the subset is same size as original sets.
function BIN:selects(rows,    select,tmp)
  function select(row,   v)
    v= row.cells[self.col.at]
    if v=="?" or self.lo==self.hi or self.lo<v and v <=self.hu then return row end end
  tmp= map(rows,select)
  if #tmp < #rows then return rows end end
```

```lua
---- ---- ROWS
function ROWS:clone(t) return ROWS():add(self.cols.names):adds(t or {}) end

function ROWS:file(x) for t in csv(x) do self:add(t) end; return self end

function ROWS:adds(t) for _,t1 in pairs(t) do self:add(t1) end; return self end

function ROWS:add(t)
  if   self.cols
  then self.cols:add(push(i.rows, t.cells and t or ROW(t)))
  else self.cols=COLS(t) end end

-- ROWS:mids(?int=2,?[COL]=self.cols.y):[key=num] -- Return `mid` of columns
-- rounded to `p` places.
function ROWS:mids(p,cols)
  local t={n=#self.rows}
  for _,col in pairs(cols or self.cols.y) do t[col.txt]=col:mid(p) end
  return rnds(t,p or 2) end

function ROWS:splitter(rows)
  function split(col)
    for _,row in pairs(rows) do
      local v = row.cells[col.at]
      if v ~= "?" then
        n=n+1
        local pos = col:bin(v)
        dict[pos] = dict[pos] or push(list, BIN(col,v))
        dict[pos]:add(v,row.label) end end end
    list = col:merges(sort(list,lt"lo"), n^the.min)
    return {bins = list,
            div  = sum(list,function(z) return z.has:div()*z.ys.n/n end)}
  end ------------------------------------------
  return sort(map(self.cols.x, split),lt"div")[1].bin end

function ROWS:tree()
  local bests,rests= self.cols:bests(self.rows)
  local rows={}
  for _,best in pairs(bests) do push(rows,    best).label=1 end
  for i = 1,the.rests*#bests do push(rows,rests[i]).label=0 end
  self:grow(rows, (#rows)^the.min)
  return self end

function ROWS:grow(rows,stop,when)
  local function kid(bin)
    local t = bin:selects(rows)
    if t then return self:clone(t):grow(t,stop,bin) end end end
  self.when=when
  if #rows < stop then return self end
  self.kids = map(self:splitter(rows), kid) end

---- ---- ---- ---- Lib
big=math.huge
min=math.min
max=math.max
fmt=string.format
rand=math.random

function any(a)  return a[rand(#a)] end
function per(t,p) p=p*#t//1; return t[math.max(1,math.min(#t,p))] end

function push(t,x) t[1+#t]=x; return x end
function map(t,f,     u) u={};for _,x in pairs(t) do u[1+#u]=f(x)end;return u end
function kap(t,f,  u) u={};for k,x in pairs(t)do u[1+#u]=f(k,x)end;return u end
function sum(t,f,     u) u=0; for _,x in pairs(t) do u=u+f(x)     end;return u end

---- rnd(num, places:int):num  -- Return `x` rounded to some number of `place`.
function rnd(x, places)  --   &#9312;
  local mult = 10^(places or 2)
  return math.floor(x * mult + 0.5) / mult end
---- rnds(t:num, places:?int=2):num -- Return items in `t` rounds to `places`.
function rnds(t, places)
  local u={};for k,x in pairs(t) do u[k]=rnd(x,places or 2)end;return u end

function sort(t,f) table.sort(t,f); return t end
function lt(x)     return function(a,b) return a[x] < b[x] end end

function shuffle(t,    j)
  for i=#t,2,-1 do j=rand(i); t[i],t[j]=t[j],t[i] end; return t end

function coerce(x)
  x = x:match"^%s*(.-)%s*$"
  if x=="true" then return true elseif x=="false" then return false
  else return math.tointeger(x) or tonumber(x) or x end  end

function cli(t)
  for k,v in pairs(t) do
    v = tostring(v)
    for n,x in ipairs(arg) do if x=="-"..(k:sub(1,1)) then
      v = v=="false" and "true" or v=="true" and "false" or arg[n+1] end end
    t[k] =  coerce(v) end
  if t.help then os.exit(go.help()) end
  return t end

function chat(t) print(cat(t)) return t end
function cat(t,   show,u)
  function show(k,v) return #t==0 and (":%s %s"):format(k,v) or tostring(v) end
  u={}; for k,v in pairs(t) do u[1+#u]=show(k,v) end
  return (t._is or "").."{"..table.concat(#t==0 and sort(u) or u," ")..."}" end

function csv(file,fun)
  function lines(file, fun)
    local file = io.input(file)
    while true do
      local line = io.read()
      if not line then return io.close(file) else fun(line) end end
  end ----------------------------
  function words(s,sep,fun,      t)
    fun = fun or same
    t={};for x in s:gmatch(fmt("([^%s]+)",sep)) do t[1+#t]=fun(x) end; return t
  end ----------------------------------------------------------------
  lines(file, function(line) fun(words(line, ",", coerce)) end) end

----  on(tab,tab) -- Runs some (or all) of the demos. Return number of failures.
-- Resets `the` and the random number seed before each demo.
function on(the,go)
  local the, fails, defaults=cli(the), 0, {}
  for k,v in pairs(the) do defaults[k]=v end
  local todos = sort(kap(go,function(k,_) return k end))
  for _,todo in pairs(the.go=="all" and todos or {the.go}) do
    if type(go[todo])=="function" then
      for k,v in pairs(defaults) do the[k]=v end
      math.randomseed(the.seed)
      if true ~= go[todo]() then
        print("FAIL:",todo)
        fails=fails+1 end end end
  for k,v in pairs(_ENV) do if not b4[k] then print("?",k,type(v)) end end
  os.exit(fails) end

---- ---- ---- ---- Start-up
function go.the() chat(the) ; return true end
function go.help()
  print(help:gsub("[%u][%u%d]+","\27[1;32m%1\27[0m"),""); return true end

help:gsub("\n[-]%S[%s]+([%S]+)[^\n]+= ([%S]+)",function(k,x) the[k]=coerce(x)end)
if   pcall(debug.getlocal, 4, 1)
then return {ROWS=ROWS, the=the}
else on(the,go) end
```