```lisp
  __                                                  ____
 /\ \__      __                                      /\_ \       __
 \ \ ,_\    /\_\      ___      __  __                \//\ \     /\_\      ____    _____
  \ \ \/    \/\ \   /' _ `\   /\ \/\ \                 \ \ \    \/\ \    /',__\  /\ '__`\
   \ \ \_    \ \ \  /\ \/\ \  \ \ \_\ \      __         \_\ \_   \ \ \  /\__, `\ \ \ \L\ \
    \ \__\    \ \_\ \ \_\ \_\  \/`____ \    /\_\        /\____\   \ \_\ \/\____/  \ \ ,__/
     \/__/     \/_/  \/_/\/_/   `/___/> \   \/_/        \/____/    \/_/  \/___/    \ \ \/
                                   /\___/                                          \ \_\
                                   \/__/                                            \/_/
(defpackage :tiny (:use :cl))
(in-package :tiny)
(mapc #'load  '("lib/macros"     "lib/maths"   "lib/strings"  "lib/lists"
                "lib/settings"  "lib/structs"  "lib/egs"  ))

(defvar my (settings "
  TINY: semi-supervised multi-objective explanation facility.
  (c) 2022 Tim Menzies, BSD-2 clause license

  USAGE:  lisp eg.lisp [OPTIONS] [ARG]"
    '((far    "-F"   "how far is distant    " .95)
      (file   "-f"   "help file             " "../../data/auto93.lisp")
      (help   "-h"   "show help             " nil)
      (keep   "-K"   "items to keep         " 256)
      (k      "-k"   "nb low attributes classes" 1)
      (m      "-m"   "nb low frequency classes " 2)
      (p      "-p"   "distance coeffecient    " 2)
      (seed   "-s"   "random number seed      " 10019)
      (some   "-S"   "how many                " 512)
      (example "-e"  "example to run          " "ls"))))

(mapc #'load '("col/sample" "col/sym" "col/num" "col/cols" "row/row" "row/data"))
      .  /       .
 _. _ | /   _. _ | __
(_.(_)|/   (_.(_)|_)

(defstruct+ cols
  "Factory for making nums or syms."
  names  ; list of column names
  all    ; all the generated columns
  x      ; just the independet columns
  y      ; just the dependent columns
  klass) ; just the klass col (if it exists)

(defun make-cols (lst)
  "Upper/lowercase words ==> nums/syms. Kept in 'all' and maybe elsewhere."
  (let (all x y kl (at -1))
    (dolist (str lst (%make-cols
                        :names lst :x x :y y :klass kl :all (reverse all)))
      (let* ((what (if (upper-case-p (char str 0)) #'make-num #'make-sym))
             (col  (funcall what str (incf at))))
        (push col all)
        (unless (eq #\~ (charn str))
          (if (member (charn str) '(#\! #\- #\+)) (push col y) (push col x))
          (if (eq #\! (charn str)) (setf kl col)))))))
      .  /
 _. _ | /   ._ . ._ _
(_.(_)|/    [ )(_|[ | )

(defstruct+ num
   "summarize numeric columns"
   (txt "")   ; column name
   (at 0)     ; column position
   (n 0)      ; #items seen
   (w 1)      ; (1,-1) = (maximize, minimize)
   (lo most-positive-fixnum) ; least seen
   (hi most-negative-fixnum) ; most seen
   (_kept (make-sample)))     ; items seen

(defun make-num (&optional (s "") (n 0))
  "Create."
  (%make-num :txt s :at n :w (if (eq #\- (charn s)) -1 1)))

(defmethod add ((i num) (lst cons))
  "Add a list of items."
  (dolist (x lst i) (add i x)))

(defmethod add ((i num) x)
  "Add one thing, skipping 'dont know', updating 'lo,hi' and 'kept'."
  (unless (eq x #\?)
    (with-slots (lo hi) i
      (incf (? i n))
      (add (? i _kept) x)
      (setf lo (min x (? i lo))
            hi (max x (? i hi))))))

(defmethod norm ((i num) x)
  "Map 'x' 0 .. 1 (unless its unknown, unless gap too small."
  (with-slots (lo hi) i
    (cond ((eq x #\?)          x)
          ((< (- hi lo) 1E-9)  0)
          (t                   (/ (- x lo) (- hi lo))))))

(defmethod dist ((i num) x y)
  "Gap between things (0..1). For unknowns, assume max distance."
  (cond ((and (eq #\? x) (eq #\? y))
                     (return-from dist 1))
        ((eq #\? x)  (setf y (norm i y) x (if (< y .5) 1 0)))
        ((eq #\? y)  (setf x (norm i x) y (if (< x .5) 1 0)))
        (t           (setf x (norm i x) y (norm i y))))
  (abs (- x y)))

(defmethod mid ((i num))
  "Middle."
  (mid (? i _kept)))

(defmethod div ((i num))
  "Diversity"
  (div (? i _kept)))

(defmethod discretize ((i num) x &optional (bins (? my bins)))
  "Max 'x' to one of 'bins' integers."
  (with-slots (lo hi) i
    (let ((b (/ (- hi lo) bins)))
      (if (= hi lo) 1 (* b (floor (+ .5 (/ x b))))))))
      .  /                 .
 _. _ | /   __ _.._ _ ._ | _
(_.(_)|/   _) (_][ | )[_)|(/,
                      |
(defstruct+ sample
  "Keep up to 'max' numbers (after which, replace any old with new)."
  (_kept ; where to keep
          (make-array 2 :fill-pointer 0 :adjustable t))
  (n 0)
  max    ; how many to keep
  ok)    ; nil if items added and list not resorted yet

(defun make-sample (&optional (max (! my keep)))
  "Create."
  (%make-sample :max max))

(defmethod add ((i sample) (x number))
  "Update."
  (incf (? i n))
  (let ((size (length (? i _kept))))
    (cond ((< size  (? i max))
           (setf (? i ok) nil)
           (vector-push-extend x (? i _kept)))
          ((< (randf) (/ (? i n) (? i max)))
           (setf (? i ok) nil)
           (setf (elt (? i _kept) (randi size)) x)))))

(defmethod per ((i sample) p)
  "Return the pth item from 'kept'."
  (let* ((all (sorted i))
         (n  (1- (length all))))
    (elt all (max 0 (min n (floor (* p n)))))))

(defmethod mid ((i sample))
  "Middle"
  (per i .5))

(defmethod div ((i sample))
  "Diversity"
  (/ (- (per i .9) (per i .1)) 2.58))

(defmethod sorted ((i sample))
  "Return 'kept', sorted."
  (unless (? i ok)
    (sort (? i _kept) #'<)
    (setf (? i ok) t))
  (? i _kept))
      .  /
 _. _ | /   __ ._ _
(_.(_)|/   _) \_|[ | )
              ._|
(defstruct+ sym
  "Summarize symbolic columns"
  (txt "")  ; column name
  (at 0)    ; column position
  (n 0)     ; #items seen
  kept)     ; symbol counts of the items

(defun make-sym (&optional s n)
  "Create."
  (%make-sym :txt s :at n))

(defmethod add ((i sym) (lst cons))
  "Add a list of items."
  (dolist (x lst i) (add i x)))

(defmethod add ((i sym) x)
  "Add one items, skipping 'dont know', update frequency counts."
  (unless (eq x #\?)
    (incf (? i n))
    (incf (geta x (? i kept)))))

(defmethod adds ((i sym) x inc)
  "Bulk add of a symbol 'x', 'inc' times."
  (incf (? i n) inc)
  (incf (geta x (? i kept)) inc))

(defmethod mid ((i sym))
  "Middle"
  (loop for (key . n) in (? i kept) maximizing n return key))

(defmethod div ((i sym))
  "Diversity (entropy)."
  (labels ((fun (p) (* -1 (* p (log p 2)))))
    (loop for (_ . n) in (? i kept) sum (fun (/ n (? i n))))))

(defmethod dist ((i sym) x y)
  "Gap between 2 items; if unknown, assume max. distance."
  (cond ((and (eq #\? x) (eq #\? y)) 1)
        ((equal x y)                 0)
        (t                           1)))
              /
._. _ .    , /  ._. _ .    ,
[  (_) \/\/ /   [  (_) \/\/

(defstruct+ row
    "Hold one record"
    cells    ; cells
    _parent  ; pointer to someone who can say what are (e.g.) lo,hi
    evaled)  ; have we used the y values

(defun make-row (rows lst)
  "Create."
  (%make-row :_parent rows :cells lst))

(defmethod better ((row1 row) (row2 row))
  "Row1 better than row2 if jumping away is better jumping to."
  (let* ((s1 0) (s2 0)
                (cols (? row1 _parent cols y))
                (n (length cols)))
    (setf (? row1 evaled) t
          (? row2 evaled) t)
    (dolist (col cols (< (/ s1 n) (/ s2 n)))
      (with-slots (at w) col
        (let ((x (norm col (elt (? row1 cells) at)))
              (y (norm col (elt (? row2 cells) at))))
          (decf s1 (exp (* w (/ (- x y) n))))
          (decf s2 (exp (* w (/ (- y x) n)))))))))

(defmethod around ((row1 row) allrows)
  "Sort `allrows` by distance to `row1`."
  (labels ((two (row2) (cons (dist (? row1 _parent cols)  row1 row2) row2)))
    (sort (mapcar 'two allrows) 'car<)))

(defmethod far ((i row) allrows)
  "Return something far away from `i`. Avoid outliers by only going so `far`."
  (cdr (elt (around i allrows)
            (floor (* (length allrows) (? my far))))))
              /    .    ,
._. _ .    , /   _| _.-+- _.
[  (_) \/\/ /   (_](_] | (_]

(defstruct+ data
            "Stores multiple rows, and their summaries."
            rows   ; all the rows
            cols)  ; summaries of all the columns

(defun make-data (&optional src (i (%make-data)))
  "Eat first row for the column header,  add the rest"
  (labels ((top.row.is.special  (x) (if (? i cols)
                                        (push (add i x) (? i rows))
                                        (setf (? i cols) (make-cols x)))))
    (if (stringp src)
      (with-lines src (lambda (line) (top.row.is.special (cells line))))
      (mapcar #'top.row.is.special src))
    i))

(defmethod clone ((i data) &optional src)
  "Create a new table with same structure as `i`."
  (make-rows (cons (? i cols names) src)))

(defmethod add ((i data) (lst cons))
  "Row creation. Called in we try to add a simple list."
  (add i (make-row i lst)))

(defmethod add ((i data) (row1 row))
  "For all the unskipped columns, update from `row1`."
  (dolist (cols `(,(? i cols x) ,(? i cols y)) row1)
    (dolist (col cols)
      (add col (elt (? row1 cells) (? col at))))))

(defmethod dist ((i data) (row1 row) (row2 row))
  "Gap between `row1`, `row2`. At `p`=2, this is Euclidean distance."
  (let ((d 0) (n 0) (p (! my p)))
    (dolist (col (? i cols x))
      (incf n)
      (incf d (expt (dist col (elt (? row1 cells) (? col at))
                              (elt (? row2 cells) (? col at)))
                    p)))
    (expt (/ d n) (/ 1 p))))

(defmethod half ((i data) &optional all above)
  "Split rows in two by their distance to two remove points."
  (or all (? i rows))
  (print 1)
  (let (all some left right c tmp)
    (setf all  (or   all (? i rows)))
    (setf some  (many all (! my some)))
     (setf left  (or   above (far (any some) some)))
    (return-from half  (print (length some)))
     (setf right (far  left some))
     (setf c     (dist (? i _parent) left right))
     (setf tmp   (mapcar (lambda (row)
                         (print 2)
                         (let ((a (dist (? row _parent) row left))
                               (b (dist (? row _parent) row right)))
                           (cons (/ (+ (* a a) (* c c) (- (* b b))) (* 2 c)) row)))
                     all))
    (print 1)
    (let ((n 0) lefts rights)
      (dolist (one (sort tmp #'car<))
        (if (< (incf n) (/ (length tmp) 2))
          (push (cdr one) lefts)
          (push (cdr one) rights)))
      (values left right lefts rights c))))
. .    /
|*|_  /  ._ _  _. _.._. _  __
||[_)/   [ | )(_](_.[   (_)_)

; Simple alist access
(defmacro ! (l x)
  "Get into association lists."
  `(cdr (assoc ',x ,l)))

(defmacro ? (s x &rest xs)
 "(? obj x y z) == (slot–value (slot–value (slot–value obj 'x) 'y) 'z)"
 (if (null xs) `(slot-value ,s ',x) `(? (slot-value ,s ',x) ,@xs)))

(defmacro geta (x lst &optional (init 0))
  "Endure lst has a slot for ‘x‘. If missing, initialize it with ‘init‘."
  `(cdr (or (assoc ,x ,lst :test #'equal)
            (car (setf ,lst (cons (cons ,x ,init) ,lst))))))
. .    /           , .
|*|_  /  ._ _  _.-+-|_  __
||[_)/   [ | )(_] | [ )_)

(defun rnd (number &optional (digits 3))
  "Round to ‘digits‘ decimal places."
  (let* ((div (expt 10 digits))
         (tmp (/ (round (* number div)) div)))
    (if (zerop digits) (floor tmp) (float tmp))))

(defvar *seed* 10013)
(defun randf (&optional (n 1.0))
  "Random float 0.. n"
  (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d0))
  (* n (- 1.0d0 (/ *seed* 2147483647.0d0))))

(defun randi (&optional (n 1))
  "Random int 0..n"
  (floor (* n (/ (randf 1000000000.0) 1000000000))))
. .    /      ,
|*|_  /   __-+-._.*._  _  __
||[_)/   _)  | [   |[ )(_]_)
                         ._|
(defun charn (x)
  "Last thing from a string."
  (and (stringp x)
       (> (length x) 0)
       (char x (1- (length x)))))

(defun trim (x)
  "Kill leading tailing whitespace."
  (string-trim '(#\Space #\Tab #\Newline) x))

(defun thing (x &aux (y (trim x)))
  "Turn ‘x‘ into a number or string or ‘?‘."
  (cond ((string= y "?") #\?)
        ((string= y "t") t)
        ((string= y "nil") nil)
        (t (let ((z (read-from-string y nil nil)))
             (if (numberp z) z y)))))

(defun splits (str &key (char #\,) (filter #'identity))
  "Divide ‘str‘ on ‘char‘, filtering all items through ‘filter‘."
  (loop for start = 0 then (1+ finish)
    for        finish = (position char str :start start)
    collecting (funcall filter (trim (subseq str start finish)))
    until      (null finish)))

; String to lines or cells of things
(defun lines (string) (splits string :char   #\Newline))
(defun cells (string &key (char #\,)) (splits string :char char :filter #'thing))

(defun with-lines (file fun)
  "Call ‘fun‘ for each line in ‘file‘."
  (with-open-file (s file)
    (loop (funcall fun (or (read-line s nil) (return))))))
. .    /  .    ,
|*|_  /  |* __-+- __
||[_)/   ||_)  | _)

; sort predicates
(defun lt (x)     (lambda (a b) (< (slot-value a x) (slot-value b x))))
(defun gt (x)     (lambda (a b) (> (slot-value a x) (slot-value b x))))
(defun car< (a b) (< (car a) (car b)))
(defun car> (a b) (> (car a) (car b)))

; random sampling (with replacement).
(defmethod anv ((i cons))   (any (coerce 'vector i)))
(defmethod any ((i vector)) (elt i (random (length i))))

(defmethod manv ((i cons)   &optional (n 10)) (many (coerce i 'vector) n))
(defmethod many ((i vector) &optional (n 10)) (loop repeat n collect (any i)))
. .    /        ,   ,
|*|_  /   __ _ -+--+-*._  _  __
||[_)/   _) (/, |   | |[ )(_]_)
                           ._|
; Update `default` from command line.  Boolean flags just flip defaults.
(defun cli (key.flag.help.default)
  "If ‘flag‘ exists on command line, update ‘key‘."
  (destructuring-bind (key flag help default) key.flag.help.default
    (declare (ignore help))
    (let* ((args #+clisp ext:*args*
                 #+sbcl sb-ext:*posix-argv*)
           (it (member flag args :test 'equalp)))
      (cons key (cond ((not it)            default)
                      ((equal default t)   nil)
                      ((equal default nil) t)
                      (t                   (thing (second it))))))))

(defun settings (header options)
  "Update settings. If  ‘help‘ is set, print help."
  (let ((tmp (mapcar (lambda (x) (cli x)) options)))
    (when (! tmp help)
      (format t "~&~{~a~%~}~%OPTIONS:~%" (lines header))
      (dolist (one options)
        (destructuring-bind (flag help default) (cdr one)
          (format t " ~a  ~a = ~a~%" flag help default))))
    tmp))
```

```
. .    /      ,           ,
|*|_  /   __-+-._.. . _.-+- __
||[_)/   _)  | [  (_|(_. | _)

(defmacro defstruct+ (x doco &body body)
  "Creates %x for constructor, enables pretty print, hides slots with '_' prefix."
  (let* ((slots (mapcar    (lambda (x) (if (consp x) (car x) x))          body))
         (show  (remove-if (lambda (x) (eq #\_ (char (symbol-name x) 0))) slots)))
    `(progn
       (defstruct (,x (:constructor ,(intern (format nil "%MAKE–~a" x)))) ,@body)
       (defmethod print-object ((self ,x) str)
         (labels ((fun (y) (format nil ":~(~a~) ~a" y (slot-value self y))))
           (format str "~a" (cons ',x (mapcar #'fun ',show))))))))
. .    /
|*|_  /   _  _  __
||[_)/   (/,(_]_)
              ._|
(defvar *egs* nil)

(defmacro eg (what arg doc &rest src)
  "define a example"
  `(push (list ',what ',doc (lambda ,arg ,@src)) *egs*))

(defun demos (settings all &optional one)
  "Run ‘one‘ (or ‘all‘) the demos. Reset globals between each
 run.  Return to the operating systems the failure count (so
 fails=0 means ‘successs‘)."
  (let ((fails 0)
        (resets (copy-list settings)))
    (dolist (trio all)
      (destructuring-bind (what doc fun) trio
        (setf what (format nil "~(~a~)" what))
        (when (member what (list 'all one) :test 'equalp)
          (loop for (key . value) in resets do
            (setf (cdr (assoc key settings)) value))
          (setf *seed* (or (cdr (assoc 'seed settings)) 10019))
          (unless (eq t (funcall fun ))
            (incf fails)
            (format t "~&FAIL [~a] ~a ~%" what doc)))))
    #+clisp (ext:exit fails)
    #+sbcl  (sb-ext:exit :code fails)))
```

```

 _  _
(/,(_]
   ._|
; test suite
(load "tiny")
(in-package :tiny)

(eg my () "show options" (pprint my) t)

(eg any () "any, many"
    (print (sort (loop repeat 20 collect (any #(10 20 30 40))) #'<))
    (print (sort (many #(10 20 30 40 50 60 70 80 90
                         100 110 120 130 140 150) 5) #'<))
    t)

(eg sym () "sym"
    (let ((s (add (make-sym) '(a a a a b b c))))
      (and (= 1.379 (rnd (div s))) (eq 'c (mid s)))))

(eg sample () "sample"
    (setf (! my keep) 64)
    (let ((s (make-sample)))
      (dotimes (i 100) (add s (1- i)))
      (and (= 32.170544 (div s)) (= 56 (mid s)))))

(eg num () "num nums"
    (setf (! my keep) 64)
    (let ((n  (make-num)))
      (dotimes (i 100) (add n (1- i)))
      (and (= 98 (? n hi)) (= 32.170544 (div n)) (= 56 (mid n)))))

(eg cols () "cols"
    (print (make-cols '("aa" "bb" "Height" "Weight–" "Age–")))
    t)

(eg lines () "lines"
    (with-lines "../../data/auto93.csv"
                (lambda (x) (print (cells x))))
    t)

(eg rows () "rows"
    (let ((rows (make-rows "../../data/auto93.csv")))
      (print (? (? rows cols) y)))
    t)

(eg dist () "dist"
    (let (all
           (r (make-rows "../../data/auto93.csv")))
      (dolist (two (cdr (? r rows)))
        (push (dist r (car (? r rows)) two) all))
      (format t "~{ ~,3f~}" (sort all #'<))
     t))

(eg half () "half"
    (let ((r (make-rows "../../data/auto93.csv")))
      (half r)
     t))

(demos my *egs* (! my example))
```