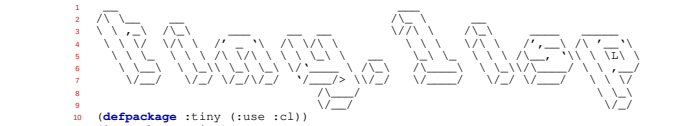

```
(defpackage :tiny (:use :cl))
(in-package :tiny)
(mapc #'load  '("lib/macros"     "lib/maths"   "lib/strings" "lib/lists"
                "lib/settings" "lib/structs" "lib/demos" ))

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

  USAGE:  lisp eg.lisp [OPTIONS] [ARG]"
  '((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)
    (go    "-g"  "start up action        " "ls"))))

(mapc #'load '("sample" "row" "sym" "num" "about" "data"))

; Keep up to "max" numbers (after which, replace any old with new).
(defstruct+ sample
  (_kept ; where to keep
         (make-array 2 :fill-pointer 0 :adjustable t))
  max    ; how many to keep
  ok)    ; nil if items added and list not resorted yet

(defun make-sample (&optional (max (? my keep_))) (%make-sample :max max))

(defmethod add ((i sample) (x number))
  (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 sorted ((i sample))
  (unless (? i ok)
    (sort (? i _kept) #'<)
    (setf (? i ok) t))
  (? i _kept))

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

(defun make-row (about l) (%make-row :cells l :_about about))

(defmethod better ((row1 row) (row2 row))
  (let* ((s1 0) (s2 0) (d 0) (n 0)
         (cols (? row1 _about 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) rows)
  (labels ((two (row2) (cons row2 (gap row1 row2))))
    (sort (mapcar #'two rows) #'car<)))

(defmethod gap ((row1 row) (row2 row))
  (let ((d 0) (n 0))
    (dolist (col (? row1 _about x) (expt (/ d n) (? my p)))
      (incf n)
      (incf d (dist col (elt (? row1 cells) (? col at))
                        (elt (? row2 cells) (? col at)))))))

; Summarize symbolic columns
(defstruct+ sym  (txt "")  ; column name
                 (at 0)    ; column position
                 (n 0)     ; #items seen
                 kept)     ; symbol counts of the items

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

(defmethod add ((i sym) (lst cons)) (dolist (x lst i) (add i x)))
(defmethod add ((i sym) x)
  (unless (eq x #\?)
    (incf (? i n))
    (incf (geta x (? i kept)))))

(defmethod adds ((i sym) x inc)
  (incf (? i n) inc)
  (incf (geta x (? i kept)) inc))

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

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

(defmethod dist ((i sym) x y)
  (cond ((and (eq #\? x) (eq #\? y)) 1)
        ((equal x y)                 0)
        (t                           1)))

; Summarize numeric columns.
(defstruct+ num (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))
  (%make-num :txt s :at n :w (if (eq #\- (charn s)) -1 1)))

(defmethod add ((i num) (lst cons)) (dolist (x lst i) (add i x)))
(defmethod add ((i num) x)
  (with-slots (lo hi) i
    (unless (eq x #\?)
      (incf (? i n))
      (setf lo (min lo (? i lo))
            hi (max hi (? i hi)))
      (add (? i kept) x))))

(defmethod norm ((i num) x)
  (with-slots (lo hi) i
    (cond ((eq x #\?)           x)
          ((< (- hi lo) 1E-12) 0)
          (t                   (/ (- x lo) (- hi lo))))))

(defmethod dist ((i num) x y)
  (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 discretize ((i num) x)
  (with-slots (lo hi) i
    (let ((b (/ (- hi lo) (? my bins))))
      (if (- hi lo) 1 (* b (floor (+ .5 (/ x b))))))))

; Factory for making nums or syms. Also controls updating those nums+syms.
(defstruct+ about 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-about (lst)
  (let (all x y kl (at -1))
    (dolist (str lst (%make-about :names lst :x x :y y :klass kl
                                  :all (reverse all)))
      (incf at)
      (let ((col (if (eq #\$ (char str 0)) (make-num str at) (make-sym str 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)))))))

(defmethod add ((i about) (lst cons)) (add i (make-row i lst)))
(defmethod add ((i about) (row1 row))
  (dolist (cols `(,(? i x) ,(? i y)) row1)
    (dolist (col cols)
      (add col (elt (? row1 cells) (? col at))))))

; Place to hold rows, and their sumamries.
(defstruct+ data rows    ; all the rows
                 about)  ; summaries of all the columns

(defun make-data (names &optional src (i (%make-data :about (make-about names))))
  (if (stringp src)
    (with-lines src (lambda (line) (add i (cells line))))
    (dolist (row src) (add i row)))
  i)

(defmethod clone ((d data) &optional src) (make-data (? d about names) src))

(defmethod add ((i data) x) (push (add (? i about) x) (? i rows)))
```

```
. .    /
|*|_  /  ._ _  _. _ _. _  __
||(_)/   [ | )(_](_.[  (_)_)
                        ._|
```

```
; Simple alist access
(defmacro ! (l x) `(cdr (assoc ',x ,l)))

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

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

; round
(defun rnd (number &optional (digits 3))
  (let* ((div (expt 10 digits))
         (tmp (/ (round (* number div)) div)))
    (if (zerop digits) (floor tmp) (float tmp))))

; Random number control (since reseeding in LISP is... strange).
(defvar *seed* 10013)

(defun randf (&optional (n 1.0))
  (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d0))
  (* n (- 1.0d0 (/ *seed* 2147483647.0d0))))

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

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

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

; Divide `str` on `char`, filtering all items through `filter`.
(defun splits (str &key (char #\,) (filter #'identity))
  (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))

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

; sort predicates
(defun !< (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< (x) (lambda (a b) (< (car a) (car b))))
(defun car> (x) (lambda (a b) (> (car a) (car b))))

. .    /        ,  ,
|*|_  /   __ _ -+--+-*._  _  __
||(_)/   _) (/, |  | |[ )(_|_)
                        ._|
; Update `default` from command line.  Boolean flags just flip defaults.
(defun setting (key.flag.help.default)
  (destructuring-bind (key flag help default) key.flag.help.default
    (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))))))))

; Update settings.  If  `help` is set, print help.
(defun settings (header options)
  (let ((tmp (mapcar #'setting options)))
    (when (! tmp help)
      (format t "~&~{~a~%~}~%OPTIONS:~%" (lines header))
      (dolist (one options)
        (format t " -a  ~a = ~a~%" (second one) (third one) (fourth one))))
    tmp))
. .    /     ,           ,
|*|_  /   __-+-.__.. , _.-+- __
||(_)/   _)  | [  (_|(_. | _)

; Creates %x for constructor, enables pretty print, hides slots with "_" prefix.
(defmacro defstruct+ (x &body body)
  (let* ((slots  (mapcar    #'(lambda (x) (if (consp x) (car x) x))          body))
         (public (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 ',public)))))))
. .    /    .
|*|_  /   _| _ ._ _  _  __
||(_)/   (_|(_. | _)  _)

; Define one demos.
(defvar *demos* nil)
(defmacro defdemo (what arg doc &rest src)
  `(push (list ',what ',doc (lambda ,arg ,@src)) *demos*))

; Run `one` (or `all`) the demos. Reset globals between each run.
; Return to the operating systems the failure count (so fails=0 means "success").
(defun demos (settings all &optional one)
  (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)

;(print (make-row 12 '(1 2 3 4)))
; (print (make-about '("$aa" "bb!~" "cc+")))
; (print (! my 'seed))
; (dotimes (i 20) (print (randi  200)))
; ; (defmethod clone ((d data) &optional src) (make-data (? d about names) src))
; ;(reads "../../data/auto93.lisp" 'print)

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

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

(defdemo num () "num nums"
  (let ((n  (make-num)))
   ; (add n 10)
    ;(print n)
    ))

(defdemo nums () "num nums"
  (let ((n (add (make-num) (loop for j from 0 to 10 collect j))))
    (format t "~a ~a~%" (div n) (mid n))
    t))

(demos my *demos* (! my go))
```