```lisp
; vim: ts=2 sw=2 et :
;.                                   __
;.                                  /\ \__
;.     __  __     ___     ___    \ \ ,_\
;.    /\ \/\ \   /' _ `\   / __`\  \ \ \/
;.    \ \ \_\ \  /\ \/\ \ /\ \L\ \  \ \ \_
;.     \/`____ \ \ \_\ \_\\ \____/   \ \__\
;.      `/___/> \ \/_/\/_/ \/___/     \/__/
;.         /\___/
;.         \/__/
;
; __preable '(__settings __macros __globals)
;;;; Ynot
(defpackage :ynot (:use :cl))
(in-package :ynot)

(defun help (lst)
  (terpri)
  (format t "ynot (v1.0) : not-so-supervised multi-objective optimization~%")
  (format t "(c) 2022 Tim Menzies, MIT (2 clause) license~%")
  (format t "~%OPTIONS:~%")
  (loop for (x(s y)) on lst by #'cddr do
    (format t " -~(~10a~) ~a = ~a~%" x s y)))

; Define settings.
(defvar *settings*
  '(enough ("how many numbers to keep"   512)
    file   ("load data from file   "   "../data/auto93.csv")
    help   ("show help             "   nil)
    p      ("distance coeffecient  "   2)
    seed   ("random number seed    "   10019)
    todo   ("start up action       "   "nothing")))

; List for test cases
(defvar *demos* nil)

; Counter for test failures (this number will be the exit status of this code).
(defvar *fails* 0)

; To reset random number generator, reset this variable.
(defvar *seed* 10019)

;.    _ _  ____  ____  ____  ____  ____
;.   |\/| |__|  |     |__/  |  |  [__
;.   |  | |  |  |___  |  \  |__|  ___]

;;; Macros.

; Shorthand for accessing settings.
(defmacro ? (x) `(second(getf *settings* ',x)))

; Shorthand for nested struct access.
(defmacro o (s x &rest xs)
  (if xs `(o (slot-value ,s ',x) ,@xs) `(slot-value ,s ',x)))

; Anaphoic if.
(defmacro aif (expr then &optional else)
  `(let (it) (if (setf it ,expr) ,then ,else)))

; Loop over file
(defmacro with-csv ((lst file &optional out) &body body)
  (let ((str (gensym)))
    `(let (,lst)
       (with-open-file (,str ,file)
         (loop while (setf ,lst (read-line ,str nil)) do ,@body)
         ,out))))

; Ensure `a` has a cells `(x . number)` (where number defaults to 0).
(defmacro has (key dictionary)
  `(cdr (or (assoc ,key ,dictionary :test #'equal)
            (car (setf ,dictionary (cons (cons ,key 0) ,dictionary))))))

; Define a demo function (see examples at end of file).
(defmacro defdemo (name params &body body)
  `(progn (pushnew ',name *demos*) (defun ,name ,params  ,@body)))

;.    ___  ___  ___  _    ____
;.     |  |  | |  | |    [__
;.     |  |__| |__| |___ ___]

;;; Library

;.      _   _   _   ._  _   _
;.     (_  (_) (/_  |  (_  (/_
;; coerce
; Cull silly white space.
(defun trim (s) (string-trim '(#\Space #\Tab) s))

; String to number (if we can).
(defun asAtom (s &aux (s1 (trim s)))
  (if (equal s1 "?") #\? (let ((x (ignore-errors (read-from-string s1))))
                           (if (numberp x) x s1))))

; String to list of strings
(defun asList (s &optional (sep #\,) (x 0) (y (position sep s :start (1+ x))))
  (cons (subseq s x y) (and y (asList s sep (1+ y)))))

; String to list of atoms
(defun asAtoms(s) (mapcar #'asAtom (asList s)))

;.      ._   _.  ._    _|   _   ._ _
;.      |   (_|  | |  (_|  (_)  | | |
;; Random
; Unlike LISP, it is easy to set the seed of this random number genertor.
(labels ((park-miller () (setf *seed* (mod (* 16807.0d0 *seed*) 2147483647.0d00)
)
                           (/ *seed* 2147483647.0d0)))
  (defun randf (&optional (n 1)) (*   n (- 1.0d0 (park-miller)))) ;XX check this
  (defun randi (&optional (n 1)) (floor (* n (park-miller)))))

; Return sample from normal distribution.
(defun normal (&optional (mu 0) (sd 1))
  (+ mu (* sd (sqrt (* -2 (log (randf)))) (cos (* 2 pi (randf))))))

;.      _  -|-  _.  -|-  _
;.     _>   |_ (_|   |_ _>
;; round
(defun round2 (number digits)
  (let* ((div (expt 10 digits))
         (tmp (/ (round (* number div)) div)))
   (if (zerop digits) (floor tmp) (float tmp))))

(print (round2 10.1234 3))

;; Stats
(defun norm (lo hi x)
  (if (< (abs (- hi lo)) 1E-9) 0 (/ (- x lo) (- hi lo))))

; Any item
(defun any (seq)    (elt seq (randi (length seq))))
(defun many (seq n) (let (a) (dotimes (i n a) (push (any seq) a))))

; Return `p`-th item from seq.
(defun per (seq &optional (p .5) &aux (v (coerce seq 'vector)))
  (elt v (floor (* p (length v)))))

; Find sd from a sorted list.
(defun sd (seq &optional (key #'identity))
  (if (<= (length seq) 5) 0
    (/ (- (funcall key (per seq .9)) (funcall key (per seq .1))) 2.56)))

; Return entropy of symbols in an assoc list.
(defun ent (alist &aux (n 0) (e 0))
  (dolist (two alist) (incf n (cdr two)))
  (dolist (two alist e) (let ((p (/ (cdr two) n))) (decf e (* p (log p 2))))))

;.      ._ _   o   _   _
;.      | | |  |  _>  (_
;; misc
; For each setting `x`, look for `-x` on the command line.
(defun update-settings-from-command-line (lst)
  (let ((args #+clisp ext:*args*
              #+sbcl  sb-ext:*posix-argv*))
    (loop for (slot (help b4)) on lst by #'cddr do
      (setf (second (getf lst slot))
            (aif (member (format nil "-~a" slot) args :test #'equalp)
              (cond ((eq b4 t)   nil) ; boolean flags flip the default
                    ((eq b4 nil) t)   ; boolean flags flip the default
                    (t (asAtom (elt it 1))))
              b4)))))

;.      ._ _   _.  o  ._
;.      | | | (_|  |  | |
; Handle tests within a test function"
(defun ok (test msg)
  (cond (test (format t "~aPASS ~a~%" #\Tab  msg))
        (t    (incf *fails* )
              (if (? dump)
                (assert test nil msg)
                (format t "~aFAIL ~a~%" #\Tab msg)))))

; Update *options* from command-line. Show help or run demo suite.
; Before demo, reset random number seed (and the settings).
; Return the number of fails to the operating system.
(defun main (&aux defaults)
  (labels ((stop () #+clisp (exit *fails*)
                    #+sbcl  (sb-ext:exit :code *fails*))
           (fun (x) (find-symbol (string-upcase x)))
           (demo (todo) (when (fboundp todo)
                          (format t "~a~%"  todo)
                          (setf *settings* (copy-tree defaults)
                                *seed*     (? seed))
                          (funcall todo))))
    (update-settings-from-command-line *settings*)
    (setf defaults (copy-tree *settings*))
    (cond ((? help)                   (help *settings*))
          ((equalp "all" (? todo)) (dolist (one *demos*) (demo (fun one))))
          (t                          (demo (fun (? todo)))))
    (stop)))
```

```lisp
;.    ____ _     ____ ____ ____ ____ ____
;.   |     |     |__| [__  [__  |___ [__
;.   |___ |___  |  | ___] ___] |___ ___]

;;; Classes

;.     o  _
;.     |  _>

; The first/last char of a column name defines meta-knowledge for that column.
(defun is (s kind)
  (let
    ((post '((ignore #\X) (klass #\!) (less #\-) (more #\+) (goal #\+ #\- #\!)))
     (pre  '((num #\$))))
    (or (member (char s (1- (length s))) (cdr (assoc kind post)))
        (member (char s 0)                (cdr (assoc kind pre))))))

;.     _      ._ _
;.    _>  \/  | | |
;.        /
;; Sym
(defstruct (sym  (:constructor %make-sym )) (n 0) at name all mode (most 0))

(defun make-sym (&optional (at 0) (name ""))
  (%make-sym :at at :name name))

(defmethod add ((self sym) x)
  (with-slots (n all mode most) self
    (unless (eq x #\?)
      (incf n)
      (let ((now (incf (has x all))))
        (if (> now most)
          (setf most now
                mode x)))))
  x)

(defmethod div ((self sym)) (ent (sym-all self)))
(defmethod mid ((self sym)) (sym-mode self))

;.    ._        ._ _
;.    | |  |_|  | | |
;; Num
(defstruct (num  (:constructor %make-num ))
  (n 0) at name
  (all (make-array 5 :fill-pointer 0 :adjustable t ))
  (max (? enough))
  ok w (hi -1E32) (lo 1E32))

(defun make-num (&optional (at 0) (name ""))
  (%make-num :at at :name name :w (if (is name 'less) -1 1)))

(defmethod add ((self num) x)
  (with-slots (n lo hi ok all max) self
    (unless (eq x #\?)
      (incf n)
      (setf lo (min x lo)
            hi (max x hi))
      (cond ((< (length all) max)  (setf ok nil) (vector-push-extend x all))
            ((< (randf) (/ max n)) (setf ok nil)
                                   (setf (elt all (randi (length all))) x)))))
  x)

(defmethod holds ((self num))
  (with-slots (ok all) self
    (unless ok (setf all (sort all #'<)))
    (setf ok t)
    all))

(defmethod div ((self num)) (sd  (holds self)))
(defmethod mid ((self num)) (per (holds self)))

;.     _   _   |   _
;.    (_  (_)  |  _>
;; cols
(defstruct (cols (:constructor %make-cols)) all x y klass)

(defun make-cols (names &aux (at -1) x y klass all)
  (dolist (s names (%make-cols :all (reverse all) :x x :y y :klass klass))
    (let ((now (funcall (if (is s 'num)  #'make-num #'make-sym) (incf at) s)))
      (push now all)
      (when (not (is s 'ignore))
        (if (is s 'goal)  (push  now y) (push now x))
        (if (is s 'klass) (setf klass now))))))

;.     _    _    _
;.    (/_  (_|  _>
;.          _|
;; egs
(defstruct (egs (:constructor %make-egs))
  cols (rows (make-array 5 :fill-pointer 0 :adjustable t)))

(defun make-egs (data &aux (self (%make-egs)))
   (if data
     (if (stringp data)
       (with-csv (row data) (add self (asAtoms row)))   ; for string = file name

       (map nil #'(lambda (row) (add self row)) data))) ; for array or list
   self)

(defmethod add ((self egs) row)
  (with-slots (rows cols) self
    (if cols
      (progn (vector-push-extend (mapcar #'add (o cols all) row) rows))
      (setf cols (make-cols row)))))
```

```lisp

(defmethod size ((self egs)) (length (o self rows)))
;.    ____ _    _  _ ____ ___ ____ ____
;.   |     |    |  | [__   |  |___ |__/
;.   |___ |___ |__| ___]  |  |___ |  \

;;; Cluster

(defmethod dist ((self egs) row1 row2)
  (let ((n 0) (d 0) (p (? p)))
    (dolist (col (o self cols x) (expt (/ d n) (/ 1  p)))
      (let ((inc (dist col (elt row1 (o col at))
                           (elt row2 (o col at)))))
        (incf d (expt inc p))
        (incf n)))))

(defmethod dist ((self num) x y)
  (with-slots (lo hi) self
    (cond ((and (eq x #\?) (eq y #\?)) (return-from dist 1))
          ((eq x #\?) (setf y (norm lo  hi y)
                            x (if (< y .5) 1  0)))
          ((eq y #\?) (setf x (norm lo  hi x)
                            y (if (< x .5) 1 0)))
          (t          (setf x (norm lo hi x)
                            y (norm lo hi y))))
    (abs (- x y))))

(defmethod dist ((self sym) x y)
  (if (and (eq x #\?) (eq y #\?))
    0
    (if (equal x y) 0 1)))

; (defun half ((self egs) rows)
;   (labels ((far (row,t
;
; (defstruct (cluster (:constructor %make-cluster)) egs top (rank 0) lefts right
s)
;
; (defmethod leaf ((self egs)) (not (o self lefts) (o self rights)))
;
; (defun make-cluster (top &optional (egs top))
;   (multiple-value-bind (half top (o egs rows))
;     (lefts rights left right border c)
;     (let ((self (%make-cluster :egs egs :top top :left left :right right
;                                :c c :border border)))
;       (when (>= (size egs) (* 2 (expt (size top) (? minItems))))
;         (when (<  (size lefts) (size egs))
;           (setf (o self lefts)  (cluster top lefts)
;                 (o self rights) (cluster top rights))))
;       self)))
;
; (defmethod show ((self cluster) &optional (pre ""))
;   (let ((front (format t "~a~a" pre (length (o egs rows)))))
;     (if  (leaf (o self egs))
;       (format t "~20a~a" front (mid (o self egs) (o self egs cols y)))
;       (progn
;         (print front)
;         (if (o self lefts)  (show (o lefts)  (format nil "|.. ~a" pre)))
;         (if (o self rights) (show (o rights) (format nil "|.. ~a" pre)))))))
;.    ___  ____ _  _ ____ ____
;.    |  \ |___ |\/| |  | [__
;.    |__/ |___ |  | |__| ___]

;;; Demos

(defdemo .rand() (print (randf)))

(defdemo .egs()
  (print 1)
  (let ((eg (make-egs (? file))))
    (holds (second (o eg cols y)))
    (print (o eg cols y))))

(defdemo .dist(&aux (eg (make-egs (? file))))
  (print (sort (loop repeat 32 collect
                 (round2 (dist eg (any (o eg rows)) (any (o eg rows))) 2)) #'<))
)

(main)
```