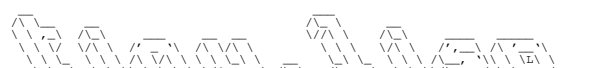
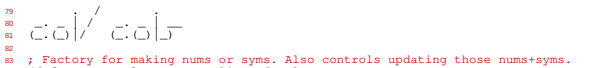
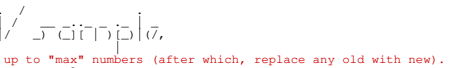


```
(defpackage :tiny (:use :cl))
(in-package :tiny)
(mapc #'load  '("lib/macros"     "lib/maths"   "lib/strings"  "lib/lists"
                "lib/settings"  "lib/structs"  "lib/demos" ))

(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 '("col/sample" "row/row" "col/sym" "col/num" "col/cols" "row/rows"))

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

(defun make-row (cols l) (%make-row :cells l :_cols cols))

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

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

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

(defmethod clone ((i rows) &optional src)
  (make-rows (? i cols names) src))

(defmethod add ((i rows) x)
  (if (? i cols)
    (push (add (? i cols) x) (? i rows))
    (setf (? i cols) (make-cols x))))
```



```
; Factory for making nums or syms. Also controls updating those nums+syms.
(defstruct+ cols 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)
  (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)))))))

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

(defmethod dist ((self cols) (row1 row) (row2 row))
  (let ((d 0) (n 0))
    (dolist (col (? self x) (float (expt (/ d n) (! my p))))
      (incf n)
      (incf d (dist col (elt (? row1 cells) (? col at))
                        (elt (? row2 cells) (? col at)))))))

; 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)
  (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)
  (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)
  (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 div ((i num)) (div (? i _kept)))
(defmethod mid ((i num)) (mid (? i _kept)))

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



```
; 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))
  (n 0)
  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) (lst cons)) (dolist (x lst i) (add i x)))
(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 per ((i sample) p)
  (let* ((all (sorted i))
         (n  (1- (length all))))
    (elt all (max 0 (min n (floor (* p n)))))))

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

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

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

; 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)))
```

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

; 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") 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 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< (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))
         (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))))))))
. .     /     .
|*|_  /    _| _ ._ _  _  __
||[_)/    (_](/,[ | )(_)_)

; 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)

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

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

(defdemo 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)))))

(defdemo 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)))))

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

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

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

(defdemo dist () "dist"
  (let ((r (make-rows "../../data/auto93.csv")))
    (dotimes (i 20 t)
      (let ((one (nth (randi (length (? r rows))) (? r rows)))
            (two (nth (randi (length (? r rows))) (? r rows))))
        (print (dist (? r cols) one two))))))

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