```python
  5:  #!/usr/bin/env python3.9
  6:  # vim: ts=2 sw=2 sts=2 et :
  7:  """
  8:    ,-_|\   Contrast set learner( c) Tim Menzies, 2021, unlicense.org
  9:   /   \    The delta between things is
 10:   \_,-._*   simpler than the things.
 11:      v
 12:  """
 13:  import re, sys, math, copy, argparse, random, itertools
 14:
 15:  def config(): return dict(
 16:    BINS   = (float, .5,   'bins are of size n**BINS'),
 17:    COLS   = (str,  'x',   'columns to use for inference'),
 18:    DATA   = (str,  '../data/auto2.csv', 'where to read data'),
 19:    FAR    = (float, .9,   'where to look for far things'),
 20:    GOAL   = (str,  'best', 'learning goals: best|rest|other'),
 21:    IOTA   = (float, .3,    'small = sd**iota'),
 22:    K      = (int,   2,     'bayes low class frequency hack'),
 23:    M      = (int,   1,     'bayes low range frequency hack'),
 24:    P      = (int,   2,     'distance calculation exponent'),
 25:    SAMPLES = (int,   20,    '#samples to find far things?'),
 26:    SEED   = (int,   10013, 'seed for random numbers'),
 27:    VERBOSE = (bool,  False, 'set verbose'),
 28:    TOP    = (int,   20,    'focus on this many'),
 29:    WILD   = (int,   False, 'run example, no protection'),
 30:    XAMPLE = (str, "", "egs: '-x ls' lists all, '-x all' runs all"))
 31:
 32:  class o(object):
 33:    def __init__(i, **k):   i.__dict__.update(**k)
 34:    def __setitem__(i, k, v): i.__dict__[k] = v
 35:    def __getitem__(i, k):  return i.__dict__[k]
 36:    def __repr__(i):        return i.__class__.__name__ + str(
 37:   {k:v for k, v in i.__dict__.items() if k[0] != "_"})
 38:
 39:  # Columns _____
 40:  class Col(o):
 41:    "Store columns in 'Col', 'Skip', 'Sym', 'Num'."
 42:    def __init__(i, at=0, txt="", inits=[]):
 43:      i.n, i.at, i.txt = 0, at, txt
 44:      i.w = -1 if "-" in txt else 1
 45:      [i.add(x) for x in inits]
 46:
 47:    def add(i, x, n=1):
 48:      if x != "?": i.n += n; x = i.add1(x, n)
 49:      return x
 50:  # _____
 51:  class Skip(Col):
 52:    def add1(i, x, n=1): return x
 53:  # _____
 54:  class Sym(Col):
 55:    def __init__(i, **kw):
 56:      i.has, i.mode, i.most = {}, None, 0
 57:      super().__init__(**kw)
 58:
 59:    def add1(i, x, n=1):
 60:      new = inc(i.has, x, n)
 61:      if new > i.most: i.most, i.mode = new, x
 62:      return x
 63:
 64:    def bins(i, j, _):
 65:      for k in (i.has | j.has):
 66:        yield i.at, (k, k)
 67:
 68:    def dist(i, x, y): return 0 if x == y else 1
 69:
 70:    def ent(i):
 71:      return sum(-v/i.n * math.log(v/i.n) for v in i.has.values())
 72:
 73:    def merge(i, j):
 74:      k = Sym(at=i.at, txt=i.txt)
 75:      [k.add(x, n) for has in (i.has, j.has) for x, n in has.items()]
 76:      return k
 77:
 78:    def merged(i, j):
 79:      k = i.merge(j)
 80:      e1,n1, e2,n2, e,n = i.ent(), i.n, j.ent(),j.n, k.ent(),k.n
 81:      tmp = n1/n*e1 + n2/n*e2
 82:      #print(e1,e2,e,tmp)
 83:      if e1 + e2 < 0.01 or e * .95 < tmp:
 84:        return k
 85:
 86:    def mid(i): return i.mode
 87:  # _____
 88:  class Num(Col):
 89:    def __init__(i, **kw):
 90:      i._all, i.ok = [], False
 91:      super().__init__(**kw)
 92:
 93:    def add1(i, x, n):
 94:      x, i.ok = float(x), False
 95:      for _ in range(n): i._all += [x]
 96:      return x
 97:
 98:    def all(i):
 99:      if not i.ok: i.ok = True; i._all = sorted(i._all)
100:      return i._all
101:
102:    def bins(i, j, the):
103:      xy = [(z, True) for z in i._all]+[(z, False) for z in j._all]
104:      iota = the.IOTA * (i.n*i.sd() + j.n*j.sd()) / (i.n + j.n)
105:      for ((lo, hi),_) in bins(xy,iota=iota, size=len(xy)**the.BINS):
106:        yield i.at, (lo, hi)
107:
108:    def dist(i, x, y):
109:      if   x == "?": y = i.norm(y); x = 1 if y < 0.5 else 0
110:      elif y == "?": x = i.norm(x); y = 1 if x < 0.5 else 0
111:      else         : x, y = i.norm(x), y.norm(y)
112:      return abs(x-y)
113:
114:    def mid(i): return per(i.all(), p=.5)
115:
116:    def norm(i, x):
117:      if x == "?": return x
118:      a = i.all()
119:      return max(0, min(1, (x-first(a))/(last(a)-first(a)+1E-32)))
120:
121:    def sd(i)    : return (per(i.all(), .9) - per(i.all(), .1))/2.56
122:    def span(i)  : return (first(i.all()), last(i.all()))
123:    def wide(i,n=0): return last(i.all()) - first(i.all()) >= n
124:
125:  # Row and Rows _____
126:  class Row(o):
127:    def __init__(i, lst, rows=None): i.rows, i.cells = rows, lst
128:
129:    def __lt__(i, j):
130:      goals = i.rows.cols.y
131:      s1, s2, n = 0, 0, len(goals)
132:      for col in goals:
133:        a  = col.norm(i.cells[col.at])
134:        b  = col.norm(j.cells[col.at])
135:        s1 -= math.e**(col.w * (a - b) / n)
136:        s2 -= math.e**(col.w * (b - a) / n)
137:      return s1 / n < s2 / n
138:
139:    def dist(i, j, the):
140:      d = n = 1E-32
141:      for col in i.rows.cols[the.COLS]:
142:        n += 1
143:        x, y = i.cells[at], j.cells[at]
144:        d += 1 if x=="?" and y=="?" else col.dist(x, y) ** the.P
145:      return (d/n) ** (1/the.P)
146:
147:    def far(i, rows, the):
148:      tmp = [(dist(i, j), j) for _ in range(the.SAMPLE)]
149:      return per(sorted(tmp, key=first), the.FAR)
150:
151:    def ys(i): return [i.cells[col.at] for col in i.rows.cols.y]
152:  # _____
153:  class Rows(o):
154:    def __init__(i, inits=[]):
155:      i.rows = []
156:      i.cols = o(all=[], names=[], x=[], y=[], klass=None)
157:      [i.add(x) for x in inits]
158:
159:    def add(i, a): i.data(a) if i.cols.names else i.header(a)
160:
161:    def best(i,the):
162:      i.rows.sort()
163:      ds = [the.IOTA*y.sd() for y in i.cols.y]
164:      best, rest = i.clone(), i.clone()
165:      for n,row in enumerate(i.rows):
166:        bestp = False
167:        for n1,n2,d in zip(i.rows[0].ys(), row.ys(), ds):
168:          bestp |= abs(n1-n2) <= d
169:        (best if bestp else rest).add(row)
170:      return best, rest
171:
172:    def clone(i, inits=[]): return Rows([i.cols.names] + inits)
173:
174:    def data(i, a):
175:      a = a.cells if type(a) == Row else a
176:      i.rows += [Row([col.add(a[col.at]) for col in i.cols.all],
177:          rows=i)]
178:
179:    def header(i, a):
180:      i.cols.names = a
181:      for at, x in enumerate(a):
182:        new = Skip if i.skipp(x) else (Num if i.nump(x) else Sym)
183:        new = new(at=at, txt=x)
184:        i.cols.all += [new]
185:        if not i.skipp(x):
186:          i.cols["y" if i.yp(x) else "x"] += [new]
187:          if i.klassp(x):
188:            i.cols.klass = new
189:
190:    def klassp(i, x): return "!" in x
191:    def nump(i, x): return x[0].isupper()
192:    def skipp(i, x): return "?" in x
193:    def ys(i): return [col.mid() for col in i.cols.y]
194:    def ysd(i): return [col.sd() for col in i.cols.y]
195:    def yp(i, x): return "-" in x or "+" in x or i.klassp(x)
196:  # _____
197:  def stratify(src):
198:    all, klass = None,{}
199:    for n,row in enumerate(src):
200:      if all:
201:        kl  = row[all.cols.klass.at]
202:        here = klass[kl] = klass.get(kl,None) or all.clone()
```

```
203:       here.add(row)
204:       all.add(row)
205:     else:
206:       all = Rows([row])
207:     return o(all=all, klass=klass)
208:
209:  # Discretizations _____
210:  # Use 'bins' to divide numeric data into ranges.
211:  def bins(xy, iota=0, size=30):
212:    def merge(b4):
213:      j, tmp, n = 0, [], len(b4)
214:      while j < n:
215:        ((lo,_),ay) = a = b4[j]
216:        if j < n - 1:
217:          ((_,hi),by) = b4[j + 1]
218:          if cy := ay.merged(by):
219:            a = ((lo, hi), cy)
220:            j += 1
221:        tmp += [a]
222:        j += 1
223:      return merge(tmp) if len(tmp) < len(b4) else b4
224:
225:    def divide(xy):
226:      bin  = o(x=Num(), y=Sym())
227:      bins = [bin]
228:      for i, (x, y) in enumerate(xy):
229:        if bin.x.n >= size and x != b4:
230:          if i < len(xy)-size and bin.x.wide(iota):
231:            bin = o(x=Num(), y=Sym())
232:            bins += [bin]
233:        bin.x.add(x)
234:        bin.y.add(y)
235:        b4 = x
236:      return bins
237:
238:    xy = sorted(xy, key=first)
239:    return merge([(bin.x.span(), bin.y) for bin in divide(xy)])
240:
241:  # Learn class deltas _____
242:  def contrasts(here, there, the):
243:    goal= {'best' : lambda b, r: b**2/b+r,
244:           'rest' : lambda b, r: r**2/(b+r),
245:           'other': lambda b, r: 1/(b+r)   }[the.GOAL]
246:
247:    def like(d, kl):
248:      out = prior = (hs[kl] + the.K) / (n + the.K*2)
249:      for at, span in d.items():
250:        f = has.get((kl, (at, span)), 0)
251:        out *= (f + the.M*prior) / (hs[kl] + M)
252:      return out
253:
254:    def val(d): return goal(like(d, True), like(d, False)), d
255:    def top(a): return sorted(a,reversed=True,key=first)[:the.TOP]
256:
257:    has = {(kl, (at, (lo, hi))): f
258:            for col1, col2 in zip(here.cols.x, there.cols.x)
259:            for at, (lo, hi) in col1.bins(col2, the)}
260:    n = len(here.rows, there.rows)
261:    hs = {True: len(here.rows), False: len(there.rows)}
262:    solos = [val(dict(at=x)) for at, x in set([z for _, z in has])]
263:    ranges = {}
264:    for _, d in top(solos):
265:      for k in d:
266:        ranges[k] = ranges.get(k, set()).add(d[k])
267:    for rule in top([val(d) for d in dict_product(ranges)]):
268:      print(rule)
```

```
269:
270:  # Misc utils _____
271:  # string stuff
272:  def color(end="\n", **kw):
273:    s, a, z = "", "\u001b[", ";1m"
274:    c = dict(black=30, red=31,  green=32, yellow=33,
275:          purple=34, pink=35,  blue=36, white=37)
276:    for col,txt in kw.items(): s = s+a + str(c[col]) +z+txt+"\033[0m"
277:    print(s, end=end)
278:
279:  def mline(m): m+= [["-"*len(str(x)) for x in m[-1]]]
280:
281:  def printm(matrix):
282:    s = [[str(e) for e in row] for row in matrix]
283:    lens = [max(map(len, col)) for col in zip(*s)]
284:    fmt = ' | '.join('{{:>{}}}'.format(x) for x in lens)
285:    for row in [fmt.format(*row) for row in s]: print(row)
286:
287:  # maths stuff
288:  def r3(a): return [round(x,3) for x in a]
289:
290:  # dictionary stuff
291:  def has(d, k): return d.get(k, 0)
292:  def inc(d, k, n=1): tmp = d[k] = n + d.get(k, 0); return tmp
293:
294:  def dict_product(d):
295:    keys = d.keys()
296:    for p in itertools.product(*d.values()):
297:      yield dict(zip(keys, p))
298:
299:  # list stuff
300:  def first(a): return a[0]
301:  def last(a): return a[-1]   #$\label{comment}$
302:  def per(a, p=.5): return a[int(p*len(a))]
303:
304:  # file stuff
305:  def csv(f=None, sep=","):
306:    def prep(s): return re.sub(r'([\n\t\r ]|#.*)', '', s)
307:    if f:
308:      with open(f) as fp:
309:        for s in fp:
310:          if s := prep(s): yield s.split(sep)
311:    else:
312:      for s in sys.stdin:
313:        if s := prep(s): yield s.split(sep)
314:
315:  # command-line stuff
316:  def cli(use, txt, config):
317:    fmt= argparse.RawTextHelpFormatter
318:    used, p = {}, argparse.ArgumentParser(prog=use, description=txt,
319:                    formatter_class=fmt)
320:    for k, (_, b4, h) in config.items():
321:      k0 = k[0]
322:      used[k0] = c = k0 if k0 in used else k0.lower()
323:      if b4==False:
324:        p.add_argument("-"+c, dest=k, default=False,
325:              help=h,
326:              action="store_true")
327:      else: p.add_argument("-"+c, dest=k, default=b4,
328:              help=h + " [" + str(b4) + "]",
329:              type=type(b4), metavar=k)
330:    return o( **p.parse_args().__dict__ )
331:
332:  # Unit tests _____
333:  class Eg:
334:    def ls(the):
```

```
335:      "list all examples."
336:      print("\nexamples:")
337:      for k, f in vars(Eg).items():
338:        if k[0] != "_": print(f"  {k:<13} {f.__doc__}")
339:
340:    def _fail(the):
341:      "testing failure"
342:      assert False, "failing"
343:
344:    def data(the,file="../data/vote.csv"):
345:      "simple load of data into a table"
346:      r = Rows(csv(file))
347:      assert 435 == len(r.rows)
348:      assert 195 == r.cols.all[1].has['y']
349:
350:    def nclasses(the,file="../data/diabetes.csv", kl="positive"):
351:      "read data with nclasses"
352:      rs = stratify(csv(file))
353:      assert 2     == len(rs.klass)
354:      assert 268   == len(rs.klass[kl].rows)
355:      assert 768   == len(rs.all.rows)
356:      assert 3.90625 == rs.klass[kl].cols.all[0].sd()
357:
358:    def bins(the, file="../data/diabetes.csv",
359:          k1= "positive", k2= "negative"):
360:      "discretize some data"
361:      rs = stratify(csv(file))
362:      bins1(rs.klass[k1], rs.klass[k2], the)
363:
364:    def bestrest(the, file="../data/auto93.csv"):
365:      "discretize some multi-goal data"
366:      r = Rows(csv(file))
367:      goods, bads = r.best(the)
368:      bins1(goods, bads, the)
369:
370:  def bins1(goods, bads, the):
371:    for good,bad in zip(goods.cols.x, bads.cols.x):
372:      bins = list(good.bins(bad, the))
373:      if len(bins) > 1:
374:        print(f"\n{good.txt}")
375:        for bin in bins: print("\t",bin)
376:
377:  # Main program _____
378:  def main(the):
379:    def run(fun, fails, the):
380:      s= f" {fun.__name__:<12}"
381:      if the.WILD:
382:        print("raw")
383:        fun(copy.deepcopy(the)); sys.exit()
384:      try:
385:        fun(copy.deepcopy(the))
386:        random.seed(the.SEED)
387:        color(green= (chr(10003)+ s), white=fun.__doc__)
388:      except Exception as err:
389:        fails = fails + 1
390:        color(red=   (chr(10007)+ s), white= str(err))
391:      return fails
392:    # _____
393:    fails=0
394:    if the.XAMPLE == "all":
395:      for k, f in vars(Eg).items():
396:        if k[0] != "_" and k!="ls": fails = run(f,fails, the)
397:    else:
398:      if the.XAMPLE and the.XAMPLE in vars(Eg):
399:        f = vars(Eg)[the.XAMPLE]
400:        if the.XAMPLE=="ls": f(the)
```

```
401:      else          : fails=run(f,fails,the)
402:    sys.exit(fails)
403:
404:  if __name__ == "__main__":
405:    main( cli("./keys", __doc__, config()) )
```