```python
#!/usr/bin/env python3.9
# vim: ts=2 sw=2 sts=2 et :
# autopep8 --exclude 'E20,E401,E226,E301,E302,E41'
"""
    ,-_|\   Contrast set learning
   /     \  (c) Tim Menzies, 2021, unlicense.org
   \_,-._*  Cluster, then report deltas between similiar
        v   clusters (which will be few).
"""
import re, sys, math, copy, argparse, random, itertools

def config(): return dict(
    BINS=(float, .5, 'bins are of size n**BINS'),
    COLS=(str, 'x', 'columns to use for inference'),
    DATA=(str, '../data/auto2.csv', 'where to read data'),
    FAR=(float, .9, 'where to look for far things'),
    GOAL=(str, 'best', 'learning goals: best|rest|other'),
    IOTA=(float, .3, 'small = sd**iota'),
    K=(int, 2, 'bayes low class frequency hack'),
    M=(int, 1, 'bayes low range frequency hack'),
    P=(int, 2, 'distance calculation exponent'),
    SAMPLES=(int, 20, '#samples to find far things?'),
    SEED=(int, 10013, 'seed for random numbers'),
    VERBOSE=(bool, False, 'set verbose'),
    TOP=(int, 20, 'focus on this many'),
    WILD=(int, False, 'run example, no protection'),
    XAMPLE=(str, "", "egs: '-x ls' lists all, '-x all' runs all"))

class o(object):
  def __init__(i, **k): i.__dict__.update(**k)
  def __setitem__(i, k, v): i.__dict__[k] = v
  def __getitem__(i, k): return i.__dict__[k]
  def __repr__(i): return i.__class__.__name__ + str(
      {k: v for k, v in i.__dict__.items() if k[0] != "_"})

# Columns ___________________________________________________
class Col(o):
  "Store columns in `Col`, `Skip`, `Sym`, `Num`."
  def __init__(i, at=0, txt="", inits=[]):
    i.n, i.at, i.txt = 0, at, txt
    i.w = -1 if "-" in txt else 1
    [i.add(x) for x in inits]

  def add(i, x, n=1):
    if x != "?":
      i.n += n
      x = i.add1(x, n)
    return x
# __________________
class Skip(Col):
  def add1(i, x, n=1): return x
# __________________
class Sym(Col):
  def __init__(i, **kw):
    i.has, i.mode, i.most = {}, None, 0
    super().__init__(**kw)

  def add1(i, x, n=1):
    new = inc(i.has, x, n)
    if new > i.most:
      i.most, i.mode = new, x
    return x

  def bins(i, j, _):
    for k in (i.has | j.has):
      yield True, i.has.get(k,0), i.at, (k, k)
      yield False, j.has.get(k,0), j.at, (k, k)

  def dist(i, x, y): return 0 if x == y else 1

  def ent(i):
    return sum(-v/i.n * math.log(v/i.n) for v in i.has.values())

  def merge(i, j):
    k = Sym(at=i.at, txt=i.txt)
    [k.add(x, n) for has in (i.has, j.has) for x, n in has.items()]
    return k

  def merged(i, j):
    k = i.merge(j)
    e1, n1, e2, n2, e, n = i.ent(), i.n, j.ent(), j.n, k.ent(), k.n
    tmp = n1/n*e1 + n2/n*e2
    if e1 + e2 < 0.01 or e * .95 < tmp:
      return k

  def mid(i): return i.mode
# __________________
class Num(Col):
  def __init__(i, **kw):
    i._all, i.ok = [], False
    super().__init__(**kw)

  def add1(i, x, n):
    x, i.ok = float(x), False
    for _ in range(n):
      i._all += [x]
    return x

  def all(i):
    if not i.ok:
      i.ok = True
      i._all = sorted(i._all)
    return i._all

  def bins(i, j, the):
    xy = [(z, True) for z in i._all]+[(z, False) for z in j._all]
    iota = the.IOTA * (i.n*i.sd() + j.n*j.sd()) / (i.n + j.n)
    for ((lo, hi), sym) in bins(xy, iota=iota, size=len(xy)**the.BINS):
      yield True, sym.has.get(True,0), i.at, (lo, hi)
      yield False, sym.has.get(False,0), j.at, (lo, hi)

  def dist(i, x, y):
    if x == "?":
      y = i.norm(y)
      x = 1 if y < 0.5 else 0
    elif y == "?":
      x = i.norm(x)
      y = 1 if x < 0.5 else 0
    else :
      x, y = i.norm(x), y.norm(y)
    return abs(x-y)

  def mid(i): return per(i.all(), p=.5)

  def norm(i, x):
    if x == "?":
      return x
    a = i.all()
    return max(0, min(1, (x-first(a))/(last(a)-first(a)+1E-32)))

  def sd(i)      : return (per(i.all(), .9) - per(i.all(), .1))/2.56
  def span(i)    : return (first(i.all()), last(i.all()))
  def wide(i, n=0): return last(i.all()) - first(i.all()) >= n

# Row and Rows __________________________________________
class Row(o):
  def __init__(i, lst, rows=None): i.rows, i.cells = rows, lst

  def __lt__(i, j):
    goals = i.rows.cols.y
    s1, s2, n = 0, 0, len(goals)
    for col in goals:
      a = col.norm(i.cells[col.at])
      b = col.norm(j.cells[col.at])
      s1 -= math.e**(col.w * (a - b) / n)
      s2 -= math.e**(col.w * (b - a) / n)
    return s1 / n < s2 / n

  def dist(i, j, the):
    d = n = 1E-32
    for col in i.rows.cols[the.COLS]:
      n += 1
      x, y = i.cells[at], j.cells[at]
      d += 1 if x == "?" and y == "?" else col.dist(x, y) ** the.P
    return (d/n) ** (1/the.P)

  def far(i, rows, the):
    tmp = [(dist(i, j), j) for _ in range(the.SAMPLE)]
    return per(sorted(tmp, key=first), the.FAR)

  def ys(i): return [i.cells[col.at] for col in i.rows.cols.y]
# __________________
class Rows(o):
  def __init__(i, inits=[]):
    i.rows = []
    i.cols = o(all=[], names=[], x=[], y=[], klass=None)
    [i.add(x) for x in inits]

  def add(i, a): i.data(a) if i.cols.names else i.header(a)

  def best(i, the):
    i.rows.sort()
    ds = [the.IOTA*y.sd() for y in i.cols.y]
    best, rest = i.clone(), i.clone()
    for n, row in enumerate(i.rows):
      bestp = False
      for n1, n2, d in zip(i.rows[0].ys(), row.ys(), ds):
        bestp |= abs(n1-n2) <= d
      (best if bestp else rest).add(row)
    return best, rest

  def clone(i, inits=[]): return Rows([i.cols.names] + inits)

  def data(i, a):
    a = a.cells if type(a) == Row else a
    i.rows += [Row([col.add(a[col.at]) for col in i.cols.all],
                   rows=i)]

  def header(i, a):
    i.cols.names = a
    for at, x in enumerate(a):
      new = Skip if i.skipp(x) else (Num if i.nump(x) else Sym)
      new = new(at=at, txt=x)
      i.cols.all += [new]
      if not i.skipp(x):
        i.cols["y" if i.yp(x) else "x"] += [new]
        if i.klassp(x):
          i.cols.klass = new

  def klassp(i, x): return "!" in x
  def nump(i, x): return x[0].isupper()
  def skipp(i, x): return "?" in x
  def yp(i, x): return "-" in x or "+" in x or i.klassp(x)
  def ys(i): return [col.mid() for col in i.cols.y]
  def ysd(i): return [col.sd() for col in i.cols.y]
# ____________________
def stratify(src):
  all, klass = None, {}
  for n, row in enumerate(src):
    if all:
      kl = row[all.cols.klass.at]
      here = klass[kl] = klass.get(kl, None) or all.clone()
      here.add(row)
      all.add(row)
    else:
      all = Rows([row])
  return o(all=all, klass=klass)

# Discretizations ____________________________________________
# Use ‘bins‘ to divide numeric data into ranges.
def bins(xy, iota=0, size=30):
  def merge(b4):
    j, tmp, n = 0, [], len(b4)
    while j < n:
      ((lo, _), ay) = a = b4[j]
      if j < n - 1:
        ((_, hi), by) = b4[j + 1]
        if cy := ay.merged(by):
          a = ((lo, hi), cy)
          j += 1
      tmp += [a]
      j += 1
    return merge(tmp) if len(tmp) < len(b4) else b4

  def divide(xy):
    bin = o(x=Num(), y=Sym())
    bins = [bin]
    for i, (x, y) in enumerate(xy):
      if bin.x.n >= size and x != b4:
        if i < len(xy)-size and bin.x.wide(iota):
          bin = o(x=Num(), y=Sym())
          bins += [bin]
      bin.x.add(x)
      bin.y.add(y)
      b4 = x
    return bins

  xy = sorted(xy, key=first)
  return merge([(bin.x.span(), bin.y) for bin in divide(xy)])

# Learn class deltas ________________________________________
def contrasts(here, there, the):
```

```
  goal = {’best’ : lambda b, r: b**2/b+r,
          ’rest’ : lambda b, r: r**2/(b+r),
          ’other’: lambda b, r: 1/(b+r)    }[the.GOAL]

  def like(d, kl):
    out = prior = (hs[kl] + the.K) / (n + the.K*2)
    for at, span in d.items():
      f = has.get((kl, (at, span)), 0)
      out *= (f + the.M*prior) / (hs[kl] + M)
    return out

  def val(d): return goal(like(d, True), like(d, False)), d
  def top(a): return sorted(a, reversed=True, key=first)[:the.TOP]

  has = {(kl, (at, (lo, hi))): f
         for col1, col2 in zip(here.cols.x, there.cols.x)
         for at, (lo, hi) in col1.bins(col2, the)}
  n = len(here.rows, there.rows)
  hs = {True: len(here.rows), False: len(there.rows)}
  solos = [val(dict(at=x)) for at, x in set([z for _, z in has])]
  ranges = {}
  for _, d in top(solos):
    for k in d:
      ranges[k] = ranges.get(k, set()).add(d[k])
  for rule in top([val(d) for d in dict_product(ranges)]):
    print(rule)

# Misc utils _______________________________________________________
# string stuff
def color(end="\n", **kw):
  s, a, z = "", "\u001b[", ";1m"
  c = dict(black=30, red=31, green=32, yellow=33,
           purple=34, pink=35, blue=36, white=37)
  for col, txt in kw.items():
    s = s+a + str(c[col]) + z+txt+"\033[0m"
  print(s, end=end)

def mline(m): m += [["-"*len(str(x)) for x in m[-1]]]

def printm(matrix):
  s = [[str(e) for e in row] for row in matrix]
  lens = [max(map(len, col)) for col in zip(*s)]
  fmt = ’ | ’.join(’{{:>{}}}’.format(x) for x in lens)
  for row in [fmt.format(*row) for row in s]:
    print(row)

# maths stuff
def r3(a): return [round(x, 3) for x in a]

# dictionary stuff
def has(d, k): return d.get(k, 0)
def inc(d, k, n=1): tmp = d[k] = n + d.get(k, 0); return tmp

def dict_product(d):
  keys = d.keys()
  for p in itertools.product(*d.values()):
    yield dict(zip(keys, p))

# list stuff
def first(a): return a[0]
def last(a): return a[-1]  # $\label{comment}$
def per(a, p=.5): return a[int(p*len(a))]
```

```
# file stuff
def csv(f=None, sep=","):
  def prep(s): return re.sub(r’([\n\t\r ]|#.*)’, ’’, s)
  if f:
    with open(f) as fp:
      for s in fp:
        if s := prep(s):
          yield s.split(sep)
  else:
    for s in sys.stdin:
      if s := prep(s):
        yield s.split(sep)

# command-line stuff
def cli(use, txt, config):
  fmt = argparse.RawTextHelpFormatter
  used, p = {}, argparse.ArgumentParser(prog=use, description=txt,
                                        formatter_class=fmt)
  for k, (_, b4, h) in config.items():
    k0 = k[0]
    used[k0] = c = k0 if k0 in used else k0.lower()
    if b4 == False:
      p.add_argument("-"+c, dest=k, default=False,
                     help=h,
                     action="store_true")
    else:
      p.add_argument("-"+c, dest=k, default=b4,
                     help=h + " [" + str(b4) + "]",
                     type=type(b4), metavar=k)
  return o( **p.parse_args().__dict__ )

# Unit tests _______________________________________________________
class Eg:
  def ls(the):
    "list  all examples."
    print("\nexamples:")
    for k, f in vars(Eg).items():
      if k[0] != "_":
        print(f"  {k:<13} {f.__doc__}")

  def _fail(the):
    "testing failure"
    assert False, "failing"

  def data(the, file="../data/vote.csv"):
    "simple load of data into  a table"
    r = Rows(csv(file))
    assert 435 == len(r.rows)
    assert 195 == r.cols.all[1].has[’y’]

  def nclasses(the, file="../data/diabetes.csv", kl="positive"):
    "read data with nclasses"
    rs = stratify(csv(file))
    assert 2 == len(rs.klass)
    assert 268 == len(rs.klass[kl].rows)
    assert 768 == len(rs.all.rows)
    assert 3.90625 == rs.klass[kl].cols.all[0].sd()

  def bins(the, file="../data/diabetes.csv",
           k1="positive", k2="negative"):
    "discretize some data"
    rs = stratify(csv(file))
    bins1(rs.klass[k1], rs.klass[k2], the)
```

```python

  def bestrest(the, file="../data/auto93.csv"):
    "discretize some multi-goal data"
    r = Rows(csv(file))
    goods, bads = r.best(the)
    bins1(goods, bads, the)

def bins1(goods, bads, the):
  for good, bad in zip(goods.cols.x, bads.cols.x):
    bins = sorted(good.bins(bad, the))
    if len(bins) > 1:
      print(f"\n{good.txt}")
      for bin in bins:
        print("\t", bin)

# Main program _____________________________________________
def main(the):
  def run(fun, fails, the):
    s = f" {fun.__name__:<12}"
    if the.WILD:
      print("wild")
      fun(copy.deepcopy(the))
      sys.exit()
    try:
      fun(copy.deepcopy(the))
      random.seed(the.SEED)
      color(green=(chr(10003) + s), white=fun.__doc__)
    except Exception as err:
      fails = fails + 1
      color(red=(chr(10007) + s), white=str(err))
    return fails
  # __________________
  fails = 0
  if the.XAMPLE == "all":
    for k, f in vars(Eg).items():
      if k[0] != "_" and k != "ls":
        fails = run(f, fails, the)
  else:
    if the.XAMPLE and the.XAMPLE in vars(Eg):
      f = vars(Eg)[the.XAMPLE]
      if the.XAMPLE == "ls":
        f(the)
      else :
        fails = run(f, fails, the)
  sys.exit(fails)


if __name__ == "__main__":
  main( cli("./keys", __doc__, config()) )
```