

keys

```

5: #!/usr/bin/env python3.9
6: # vim: ts=2 sw=2 sts=2 et :
7: # autpep8: ignore E20,E401,E226,E302,E41
8: import re, sys, argparse, itertools
9: from argparse import ArgumentParser as parse
10: from argparse import RawTextHelpFormatter as textual
11: Float = Str = Int = Bool = lambda "l": [0]
12: #####
13:
14: def keys(BINS : Float("bins are of size n**BINS") = .5,
15:         COLS : Str( "columns to use for inference") = "x",
16:         DATA : Str( "where to read data") = "../data/auto2.csv",
17:         EPSILON : Float("small = sd**EPSILON") = .3,
18:         FAR : Float("where to look for far things") = .9,
19:         GOAL : Str( "learning goals: best|rest|other") = "best",
20:         K : Int( "bayes low class frequency hack") = 2,
21:         M : Int( "bayes low range frequency hack") = 1,
22:         P : Int( "distance calculation exponent") = 2,
23:         SAMPLE : Int( "#samples to find far things?" ) = 20,
24:         VERBOSE : Bool( "set verbose") = False,
25:         TOP : Int( "focus on this many") = 20,
26:         XAMPLE : Str( "Egs: '-x ls' lists all, '-x all' runs all") = ""
27:         ):
28:     """
29:     ,_/_\ (c) Tim Menzies, 2021, unlicense.org.
30:     /_/_\ The delta between things is
31:     \_/_/_ simpler than the things.
32:     v """"
33:
34:     GOAL = {'best': lambda b, r: b**2/b+r,
35:            'rest': lambda b, r: r**2/(b+r),
36:            'other': lambda b, r: 1/(b+r)
37:            }[GOAL]
38:
39:     # -----
40:     class Col(o):
41:         def __init__(i, at=0, txt="", inits=[]):
42:             i.n, i.at, i.txt = 0, at, txt
43:             i.w = -1 if "-" in txt else 1
44:             [i.add(x) for x in inits]
45:
46:         def add(i, x, n=1):
47:             if x != "?":
48:                 i.n += 1
49:                 x = i.add1(x, n)
50:             return x
51:
52:     # -----
53:     class Skip(Col):
54:         def add1(i, x, n=1): return x
55:
56:     # -----
57:     class Sym(Col):
58:         def __init__(i, **kw): i.has = {}; super().__init__(**kw)
59:
60:         def add1(i, x, n=1): inc(i.has, x, n); return x
61:
62:         def bins(i, j):
63:             for k in (i.has | j.has):
64:                 yield i.has.get(k, 0), True, (i.at, (k, k))
65:                 yield j.has.get(k, 0), False, (j.at, (k, k))
66:
67:         def dist(i, x, y): return 0 if x == y else 1

```

```

68:
69: def ent(i):
70:     return sum(-v/i.n * math.log(v/i.n) for v in i.has.values())
71:
72: def merge(i, j):
73:     k = Sym(at=i.at, txt=i.txt)
74:     [k.add(x, n) for has in (i.has, j.has) for x, n in has.items()]
75:     return k
76:
77: def merged(i, j):
78:     k = i.merge(j)
79:     e1, n1, e2, n2, e, n = i.ent(), i.n, j.ent(), j.n, k.ent(), k.n
80:     if e1 + e2 < 0.01 or e * .95 < n1 / n * e1 + n2 / n * e2:
81:         return k
82:
83:     # -----
84:     class Num(Col):
85:         def __init__(i, **kw):
86:             i._all, i.ok = [], False
87:             super().__init__(**kw)
88:
89:         def add1(i, x, n):
90:             x, i.ok = float(x), False
91:             for _ in range(n):
92:                 i._all += [x]
93:             return x
94:
95:         def all(i):
96:             if not i.ok:
97:                 i.ok = True
98:                 i._all = sorted(i._all)
99:             return i._all
100:
101:         def bins(i, j):
102:             xy = [(z, True) for z in i._all]+[(z, False) for z in j._all]
103:             eps = EPSILON * (i.n*i.sd() + j.n*j.sd()) / (i.n + j.n)
104:             for ((lo, hi), sym) in bins(xy, epsilon=eps, enough=len(xy)**BINS):
105:                 for klass, n in sym.has.items():
106:                     yield n, klass, (i.at, (lo, hi))
107:
108:         def dist(i, x, y):
109:             if x == "?":
110:                 y = i.norm(y)
111:                 x = 1 if y < 0.5 else 0
112:             elif y == "?":
113:                 x = i.norm(x)
114:                 y = 1 if x < 0.5 else 0
115:             else:
116:                 x, y = i.norm(x), y.norm(y)
117:             return abs(x-y)
118:
119:         def norm(i, x):
120:             if x == "?":
121:                 return x
122:             a = i.all()
123:             return max(0, min(1, (x-first(a))/(last(a)-first(a)+1E-32)))
124:
125:         def sd(i): return (per(i.all(), .9) - per(i.all(), .1))/2.56
126:         def span(i): return (first(i.all()), last(i.all()))
127:         def wide(i, n=0): return last(i.all()) - first(i.all()) >= n
128:
129:     # -----
130:     class Row(o):

```

```

131:         def __init__(i, lst, tab=None): i.tab, i.cells = tab, lst
132:
133:         def dist(i, j):
134:             d = n = 1E-32
135:             for col in i.tab.cols[COLS]:
136:                 n += 1
137:                 x, y = i.cells[at], j.cells[at]
138:                 d += 1 if x == "?" and y == "?" else col.dist(x, y) ^ P
139:             return (d/n) ^ (1/P)
140:
141:         def far(i, rows):
142:             tmp = [(dist(i, j), j) for _ in range(SAMPLE)]
143:             return per(sorted(tmp, key=first), FAR)
144:
145:     # -----
146:     class Table(o):
147:         def __init__(i, inits=[]):
148:             i.rows = []
149:             i.cols = o(all=[], names=[], x=[], y=[], klass=None)
150:             [i.add(x) for x in inits]
151:
152:         def add(i, a): i.data(a) if i.cols.names else i.header(a)
153:         def clone(i, inits=[]): return Table([i.cols.names] + inits)
154:
155:         def data(i, a):
156:             a = a.cells if type(a) == Row else a
157:             a = [col.add(a[col.at]) for col in i.cols.all]
158:             i.rows += [Row(a, tab=i)]
159:
160:         def header(i, a):
161:             i.cols.names = a
162:             for at, x in enumerate(a):
163:                 new = Skip if i.skipp(x) else (Num if i.nump(x) else Sym)
164:                 new = new(at=at, txt=x)
165:                 i.cols.all += [new]
166:                 if not i.skipp(x):
167:                     i.cols["y" if i.y(x) else "x"] += [new]
168:                 if i.klassp(x):
169:                     i.cols.klass = new
170:
171:         def klassp(i, x): return "!" in x
172:         def nump(i, x): return x[0].isupper()
173:         def skipp(i, x): return "?" in x
174:         def yp(i, x): return "-" in x or "+" in x or i.klassp(x)
175:
176:     # -----
177:     def bins(xy, epsilon=0, enough=30):
178:         def merge(b4):
179:             j, tmp, n = 0, [], len(b4)
180:             while j < n:
181:                 a = b4[j]
182:                 if j < n - 1:
183:                     b = b4[j + 1]
184:                     if cy := a.y.merged(b.y):
185:                         a = o(x=(a.x[0], b.x[1]), y=cy)
186:                         j += 1
187:                     tmp += [a]
188:                     j += 1
189:             return merge(tmp) if len(tmp) < len(b4) else b4
190:
191:         def divide(xy):
192:             xy = sorted(xy, key=first)
193:             bin = o(x=Num(), y=Sym())

```

keys

```

194: bins = [bin]
195: for i, (x, y) in enumerate(xy):
196:     if bin.x.n >= enough:
197:         if x != b4:
198:             if i < len(xy) - enough:
199:                 if bin.x.wide(epsilon):
200:                     bin = o(x=Num()), y=Sym()
201:                     bins += [bin]
202:             bin.x.add(x)
203:             bin.y.add(y)
204:             b4 = x
205:         return bins
206:
207:     return merge(
208:         divide([o(bin.x.span(), y=bin.y) for bin in bins]))
209: # -----
210:
211: def contrasts(here, there, t):
212:     def like(d, kl):
213:         out = prior = (hs[kl] + K) / (n + K*2)
214:         for at, span in d.items():
215:             f = has.get((kl, (at, span)), 0)
216:             out *= (f + M*prior) / (hs[kl] + M)
217:         return out
218:
219:     def val(d): return GOAL(like(d, True), like(d, False)), d
220:     def top(a): return sorted(a, reversed=True, key=first)[:TOP]
221:
222:     has = {(kl, (at, (lo, hi))): f
223:             for col1, col2 in zip(here.cols.x, there.cols.x)
224:             for f, kl, (at, (lo, hi)) in col1.bins(col2)}
225:     n = len(here.rows, there.rows)
226:     hs = {True: len(here.rows), False: len(there.rows)}
227:     solos = [val(dict(at=x)) for at, x in set([z for _, z in has])]
228:     ranges = {}
229:     for _, d in top(solos):
230:         for k in d:
231:             ranges[k] = ranges.get(k, set()).add(d[k])
232:     for rule in top([val(d) for d in dict_product(ranges)]):
233:         print(rule)
234: # -----
235:
236: class Eg:
237:     def ls():
238:         "list all examples."
239:         print("\nexamples:")
240:         for k, f in vars(Eg).items():
241:             if k[0] != "_":
242:                 print(f" {k:<13} {f.__doc__}")
243:
244:     def data(file="./data/vote.csv", goal="democrat"):
245:         "simple load of data into a table"
246:         t = Table(csv(file))
247:         assert(435 == len(t.rows))
248:         assert(195 == t.cols.all[1].has['y'])
249:
250:     def clone(file="./data/diabetes.csv", k="positive"):
251:         "discretize test"
252:         t = Table(csv(file))
253:         kl = t.cols.klass.at
254:         u, v = t.clone(), t.clone()
255:         [(u if k == row.cells[kl] else v).add(row) for row in t.rows]
256:         good, bad = u.cols.x[1], v.cols.x[1]

```

```

257:     #print(good.all()[::25], len(good.all()), good.sd())
258:     print(bad.all()[::25], len(bad.all()), bad.sd())
259:
260:     def bins(file="./data/diabetes.csv", k="positive"):
261:         "discretize test"
262:         t = Table(csv(file))
263:         kl = t.cols.klass.at
264:         u, v = t.clone(), t.clone()
265:         print("=====")
266:         for row in t.rows:
267:             (u if k == row.cells[kl] else v).add(row)
268:         good, bad = u.cols.x[1], v.cols.x[1]
269:         for x in good.bins(bad):
270:             print(good.at, x)
271: # -----
272: # main program for keys
273: if XAMPLE == "all":
274:     for k, f in vars(Eg).items():
275:         if k[0] != "_":
276:             print("\n" + k)
277:             f()
278: else:
279:     if XAMPLE and XAMPLE in vars(Eg):
280:         vars(Eg)[XAMPLE]()
281:
282: #####
283: # things that don't use the config vars
284: # dictionaries
285: def has(d, k): return d.get(k, 0)
286: def inc(d, k, n=1): tmp = d[k] = n + d.get(k, 0); return tmp
287: def public(d): return {k: v for k, v in d.items() if k[0] != "_"}
288:
289: def dict_product(d):
290:     keys = d.keys()
291:     for p in itertools.product(*d.values()):
292:         yield dict(zip(keys, p))
293:
294: # -----
295: # lists
296: def first(a): return a[0]
297: def last(a): return a[-1]
298: def per(a, p=.5): return a[int(p*len(a))]
299:
300: # -----
301: # objects
302: class o(object):
303:     def __init__(i, **k): i.__dict__.update(**k)
304:     def __getitem__(i, k): return i.__dict__[k]
305:     def __repr__(i): return i.__class__.__name__ + str(public(i.__dict__))
306:     def __setitem__(i, k, v): i.__dict__[k] = v
307:
308: # -----
309: # files
310: def csv(f=None, sep=","):
311:     def prep(s): return re.sub(r'([\n\r ]|#.*)', '', s)
312:     if f:
313:         with open(f) as fp:
314:             for s in fp:
315:                 if s := prep(s):
316:                     yield s.split(sep)
317:     else:
318:         for s in sys.stdin:
319:             if s := prep(s):

```

```

320:         yield s.split(sep)
321:
322: # -----
323: # command line options
324: def cli(f):
325:     p = parse(prog="./" + f.__name__,
326:               description=f.__doc__,
327:               formatter_class=Textual)
328:     for (k, h), b4 in zip(list(f.__annotations__.items()),
329:                           f.__defaults__):
330:         if b4 == False:
331:             p.add_argument("-" + k[0].lower(), dest=k, help=h,
332:                            default=False, action="store_true")
333:         else:
334:             p.add_argument("-" + k[0].lower(), dest=k, default=b4,
335:                            help=h + " [" + str(b4) + "]", type=type(b4),
336:                            metavar=k)
337:     f(**p.parse_args().__dict__)
338:
339: #####
340: if __name__ == "__main__":
341:     cli(keys)

```