**Source path:** svn/ trunk/ scripts/ fractionation/ fractionation_ga.py

‹r181 **r184**

Show details

```
"""
This module does operations on data that looks like:
    >>> runs_data = '1__1111_11_1__111__111_11111'

where '1' indicates a gene retention and '_', a gene loss.
that can be converted by count_runs to a runs list like:
    >>> count_runs(runs_data)
    [(1, 2), (2, 1), (3, 2), (4, 1), (5, 1)]

indicating 2 runs of 1, 1 run of 2, 2 runs of 3, etc.

the run_sim() function takes a string as in runs_data above
and runs a genetic algorithm to find the deletion length
frequencies that best reproduce it's pattern.

it does so by calling the `gen_deletions` function, which
takes a list of deletion lengths and a string of '1's and
randomly chooses a deletion length from the list of deletion
lengths and a point to start deleting. this continues until
the simulated string contains as meny deletions as the observed
string. then the runs are counted with count_runs()
and the numbers are compared to the runs data of the observed
string. the GA tries to minimize the difference between
the run counts.

"""
import sys

from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve.Consts import minimaxType

from itertools import groupby
from random import randint, choice
try:
    import psyco
    psyco.full()
    print >>sys.stderr, "using psyco"
except:
    print >>sys.stderr, "no psyco"

def count_deletion_runs(astr, deletion="_"):
    """
    like count runs except it counts the run-lengths
    of deletions, not retentions
    `astr` is a string where anything that is not `deletion`
    is considered to be retained.
    """
    delstr = list(set(astr).difference(deletion))
    assert len(delstr) == 1, delstr
    delstr = delstr[0]
    return count_runs(astr, splitter=delstr)

def count_runs(astr, splitter="_"):
    """find the length of runs in the str
    ::

```

```
58          >>> count_runs("11_111_1_1_1_1_111111")
59          [(1, 4), (2, 1), (3, 1), (4, 0), (5, 0), (6, 1)]
60
61      """
62      lens = sorted([len(s) for s in astr.split(splitter) if s != ''])
63      nmax = lens[-1]
64      runs = dict((x, len(list(y))) for (x, y) in groupby(lens, lambda a: a))
65      for i in range(1, nmax + 1):
66          # if it's not there, set it.
67          runs.setdefault(i, 0)
68      return sorted(runs.items())
69
70  def gen_deletions(region_length, deletion_lengths, num_deletions=0.5,
71                    count_retentions=False):
72      """
73      :param region_length: the length in genes of the region to simulate
74      :param deletion_lengths`: an iterable of numbers to choose randomly for the
75                            number of genes to delete
76      :param num_deletions: if > 1, then the number of times to do a deletion event
77                            if < 1, then the proportion of the region to delete before
78                            stopping
79      :param count_retentions: by default (False), this will count the runs of deletions
80                               if True, it will count runs of retention.
81      :rtype: the generated string and the run counts and the number of deletion
82              events. (see count_runs in misc.py)
83      """
84      if num_deletions < 1:
85          num_deletions = int(round(num_deletions * region_length ))
86
87      region = range(region_length)
88      deletion_count = 1
89      current_region_length = region_length
90
91      for event in range(num_deletions):
92
93          # can be: randint(0, region_length - 1 - deletion_count)
94          #deletion_start = randint(0, len(region) - 1)
95          #assert current_region_length == len(region), (current_region_length, len(region), event)
96          deletion_start = randint(0, current_region_length - 1)
97
98          deletion_length = choice(deletion_lengths)
99
100         deletion_stop = deletion_start + deletion_length
101         if deletion_stop > current_region_length:
102             deletion_length -= (deletion_stop - current_region_length)
103             deletion_stop = current_region_length
104
105
106
107         del region[deletion_start:deletion_stop]
108         current_region_length -= deletion_length
109
110
111         deletion_count += deletion_length
112         if deletion_count > num_deletions:
113             break
114
115     region = frozenset(region)
116     deletion_string = "".join(["1" if i in region else "_" \
117                                for i in range(region_length)])
118
119     if deletion_lengths == (1,):
120         assert len(deletion_string) == region_length, (len(deletion_string), )
121
122     if count_retentions:
123         runs = count_runs(deletion_string)
124     else:
125         runs = count_runs(deletion_string, "1")
126
127     return deletion_string, runs
128
129 def initializator(genome, **args):
```

```
    """ used for the genetic algorithm initialize the deletion lengths
    to randomly chosen values between 1 and 6"""
    genome.clearList()
    rmin = genome.getParam("rangemin")
    rmax = genome.getParam("rangemax")
    for i in range(GA_LEN):
        genome.append(randint(rmin, rmax))

def mutator(genome, **args):
    """
    used for the genetic algorithm. mutate a "chromosome" by
    picking between 0 and 4 mutations in random spots and incrementing
    or decrementing by 1.
    """
    rmax = genome.getParam("rangemax")
    rmin = genome.getParam("rangemin")
    #l = len(genome)
    l = GA_LEN
    n_mutations = randint(0, 4)
    mutations = 0
    while mutations < n_mutations:
        idx = randint(0, l - 1)
        change = choice([-1, 1])
        v = genome[idx]
        newval = v + change

        if newval > rmax: newval -= 2
        if newval < rmin: newval += 2

        genome[idx] = newval
        mutations += 1
    return mutations


def run_sim(astr):
    """
    given an example deletion/retention string, where
    "_" is a deletion, run a ga simulation to determine
    the deletion lengths likely to have created that pattern
    of deletion-lengths
    """

    num_deletions = astr.count('_')
    region_length = len(astr)


    real_runs = count_deletion_runs(astr)
    max_real_run_len = real_runs[-1][0]

    def evaluator(chromosome):
        deletion_lengths = list(chromosome)

        # since gen_deletions is random, do multiple tries to
        # make sure an outlier doesnt screw it up.
        ntries = 10
        asum = 0.0
        for tries in range(ntries):
            sim_str, sim_runs = gen_deletions(region_length,
                                              deletion_lengths=deletion_lengths,
                                              num_deletions=num_deletions,
                                              count_retentions=False)
            sim_runs = dict(sim_runs)

            for run_length, real_count in real_runs:
                asum += run_length * abs(real_count - sim_runs.get(run_length, 0))
            # maybe the simulation had some really long runs...
            for run_length in range(max_real_run_len + 1, max_real_run_len + 10):
                asum += run_length * sim_runs.get(run_length, 0)
        return asum / ntries


    genome = G1DList.G1DList(len(astr))
```

```
202     # deletion lenghts vary between 1 and 5
203     genome.setParams(rangemin=1, rangemax=5, roundDecimal=5)
204     genome.initializator.set(initializator)
205     genome.mutator.set(mutator)
206     genome.evaluator.set(evaluator)
207 
208     ga = GSimpleGA.GSimpleGA(genome)
209     ga.setMinimax(minimaxType['minimize'])
210     ga.setGenerations(GA_GENERATIONS)
211     ga.evolve(freq_stats=0)
212     best = ga.bestIndividual()
213     return {'deletion_lengths': sorted(list(best)), 'fitness': best.fitness,
214             'score': best.score}
215 
216 GA_LEN = 25
217 GA_GENERATIONS = 10000
218 MAX_DELETION_SIZE = 10 # > 1000 is same as no removal.
219 
220 if __name__ == "__main__":
221     # expects a string of deletions 1_1111_11 where "_" is the deletion
222     # and a number which is the max deletions to allow (more are removed)
223     import sys
224     if len(sys.argv) > 1:
225         print "testing..."
226         import doctest
227         doctest.testmod()
228         sys.exit()
229 
230 
231     import re
232     delstrs = dict(
233         #overs = open('over.txt').read().strip(),
234         #unders = open('under.txt').read().strip(),
235         both = open('both.txt').read().strip()
236     )
237     # all deletions longer than this are collapsed to nothing
238     # assumed not to be real.
239     print "removing deletions longer than % i" % MAX_DELETION_SIZE
240     for delname, deletion_str in delstrs.items():
241         deletion_str = re.sub("_{%i,}" % MAX_DELETION_SIZE, "", deletion_str)
242         print
243         print delname
244         print run_sim(deletion_str)
```