RUNSTER User Notes
(R.S. Forsyth, September 2016)

Contents

RUNSTER

Regression Using Naturalistic Selection To Evolve Rules

is a **function-fitting** system inspired by the Darwinian concept of natural selection. In brief, it is a program suite which examines a database of examples and uses machine-learning techniques to create a rule or set of rules for estimating a numeric attribute associated with those examples, as well as other examples of the same type. RUNSTER's method relies on an analogue of "survival of the fittest" -- the same process that, according to Darwin's theory of evolution, gave rise to us all. This method involves the slicing and recombination of rules to produce better rules.

In statistical terms, it generates a **regression function** for assigning a numeric score to cases according to their attributes. It differs from conventional Multiple Regression firstly in employing a stochastic procedure to devise the regression rule/rules and secondly in using a highly non-linear, and hence much more expressive, description language for its regression rules. It is a companion to BEAGLE, which stands for Biological Evolutionary Algorithm Generating Logical Expressions, re-released earlier in 2016. RUNSTER applies essentially the same procedure to what statisticians call **regression** as BEAGLE does to **classification**.

1. Introduction

RUNSTER has its origin in BEAGLE, a Darwinian pattern-recognizer which I devised in 1980 (Forsyth, 1981), out of curiosity -- to find out whether an analogy with Darwinian evolution, which I termed "naturalistic selection", could be a viable machine-learning method. It worked surprisingly well, so I wrote a version in Turbo-Pascal under MS/DOS in 1985 (Forsyth & Rada, 1986) as a commercial product sold as PC/BEAGLE. That also worked well as a piece of software though less well as a money-spinner.

PC/BEAGLE did incorporate a regression mode, but it was something of an afterthought. It was clear that a more thorough implementation of its regression capability was needed. After many years, that is the lack that RUNSTER attempts to fill -- as part of a comprehensive rewrite of the whole system in a more flexible programming language, Python3, available under the GNU public licence.

RUNSTER is a suite of programs performing **symbolic regression**. A **regression function** is a function that takes one or more predictor-variable values as input and computes from them the expected value of a dependent or target variable. RUNSTER improves on conventional linear regression in two ways: (1) it selects a subset of useful variables from a larger set of input variables; (2) it doesn't just optimize coefficients in an additive formula, but constructs the functional form of the expression. This approach to nonlinear modelling offers the prospect of (semi-)automated discovery: you give the system data; it uncovers lurking patterns. Even if those patterns don't provide definitive answers, they often raise interesting questions.

To illustrate with a brief example, RUNSTER can recapitulate the discovery of Kepler's Third Law. Kepler himself needed Tycho Brahe's arduously assembled observations and many years of painstaking calculation to discover that a planet's period (P) equals that planet's mean orbital distance (D) to the power of 3/2:

P = D ^ 1.5

One of the datasets provided with the package (see Appendix D) contains details concerning the 33 largest moons of the four largest planets in our solar system (sats.dat). Another file (planets.dat) contains data on the 8 recognized planets, along with Ceres, Pluto and Eris. When RUNSTER's learning module, root.py, is trained on the satellite dataset with the objective of predicting a variable called reltime, the satellite's orbital period (relative to the largest moon of the planet concerned) from several other variables, including reldist, the satellite's relative distance from its planet, it produces the following output.

```
training data : c:\beagling\datasets\sats.dat
creation date : Sat Sep 17 15:48:46 2016
rule mode : standard
33 6
reltime
[0.038244514, 0.073129252, 0.171786834, 1.0, 61.25]
[33, 4.488086270393939, 12.949559136991365, 0.14721600199999998]
$
( ( ( $Root reldist * ( 0.9999092042625951 * reldist ) ) * 0.9999092042625951 ) *
0.9998645517007605 )
$
0  [1, 0.171786834, 12.949559136991365]
[-0.0012853953088402073, -10]
$
```

Details of what a RUNSTER rule contains will emerge in the following pages, but for the present it suffices to note that the line containing

((($Root reldist * (0.9999092042625951 * reldist)) * 0.9999092042625951) * 0.9998645517007605)

is, in effect, a re-expression of the right-hand side of Kepler's equation. Strictly speaking, it might be better

to call it 99.97% of Kepler's Third Law: if those three multiplicative constants beginning 0.9999 were 1, they would be redundant and what remains would be the exact function. This slight inaccuracy is doubtless due to slight inaccuracies in the training data (gathered from Moore (1999) with additions from Wikipedia), which serves to remind us that the software has no insight. Ideally, a near-bullseye like this should be close enough to provoke insight, or at least curiosity, in the human user.

When this rule is applied (by the program tree.py) to the data file planets.dat, it gives near-perfect predictions of planetary orbital periods (relative to the Earth's). Trained on moons, it gives excellent fits to data about planets. The program's main output listing is reproduced below.

```
dateline   Sat Sep 17 15:52:52 2016
progname   C:\beagling\p3\tree.py
id         C:\beagling\parapath\plansats.txt
testdat    c:\beagling\datasets\planets.dat
targval    reltime

====holdout trial :

rank  safeness case  name                      pred:true      cellsize      abdsiff     diffsqrd
   1      0.02   10  Eris                  565.5599 + 559            33         6.56        43.03
   2      0.05    9  Pluto                 249.0501 + 247.87         33         1.18         1.39
   3      0.07    8  Neptune               165.0556 + 164.8          33         0.26         0.07
   4      0.14    7  Uranus                 84.4723 + 84.01          33         0.46         0.21
   5      0.34    6  Saturn                 29.4631 + 29.4724        33         0.01         0.00
   6      0.64    5  Jupiter                11.8641 + 11.87          33         0.01         0.00
   7      0.99    4  Ceres                    4.594 + 4.6            33         0.01         0.00
   8      0.83    3  Mars                   1.8798 + 1.8805          33         0.00         0.00
   9      0.79    2  Terra Firma            0.9999 + 1               33         0.00         0.00
  10      0.77    1  Venus                  0.6148 + 0.6152          33         0.00         0.00
  11      0.75    0  Mercury                0.2405 + 0.2408          33         0.00         0.00
++++++++++

'success' percentage = 100.0
pearson correlation between predicted & true vals = 1.0
spearman rank-correlation between predicted & true vals = 1.0

mean abs.error = 0.771
mean error ^ 2 = 4.0641
correlation between safeness & abs.error (negative better) = -0.5363
```

Explanation of many elements of this printout will be found in subsequent sections. Here the point to emphasize is that predictions from this rule give an almost perfect correlation (+1 to four decimal places) with the actual target values. It is clear that for the eight planets, as well as asteroid Ceres, the divergences are negligible. Given that Pluto has a highly elliptical orbit, and Eris even more so, and that neither has in fact completed anything like a full orbit since it was discovered, it would not be surprising to find that the figures in my data file for Pluto and Eris will need revision in future -- quite possibly reducing these apparent errors.

This is a successful example of machine discovery, to whet your appetite for more. It is fair to note before passing on that expressing the periods and distances in relative units gives the system a big help. Given only raw distances and times in kilometres and days it fails to find a simple, accurate rule. Since the moon data comes from four planets with different masses, that really would be an impressive feat. Moreover, if you experiment with this same data by attempting to predict in the opposite direction, relative distances from relative times, you will find that RUNSTER only rarely discovers the converse equation. Nevertheless I hope this little astronomical vignette has encouraged you to do exactly that -- experiment with the system.

REMINDER: Results from RUNSTER should be treated as advisory. Inductively derived rules, whether created by people or machines, can never be proven correct. This venerable philosophical problem has broken the minds of some of the greatest thinkers since ancient times, and RUNSTER makes no pretence to solve it. In addition, it is always possible that there still remain errors in the programs. (If you encounter what seems to be a programming error, please inform me and I will endeavour to rectify it.) The real value of RUNSTER's kind of mechanistic creativity is to stimulate human creativity.

2. Getting Started

2.1 Setting Up

First you need Python3. If you don't have it already, the latest version can be downloaded and installed from the Python website: www.python.org. This is usually straightforward. The only snag is if you have Python2 and want to keep using it. (But isn't it about time to upgrade?) Then you'll probably have to set up a specific command to run whichever version you use less frequently.

Next step is to unpack the beagling.zip file, which contains both the BEAGLE and RUNSTER software. After unpacking it (into a top-level folder called "beagling", unless you want to do lots of editing), you should find the following subfolders.

datasets
op
p3
parapath

The programs are in p3. (You get the BEAGLE programs thrown is as well: see separate User Notes, beagling.pdf, for instructions.) Sample data sets for testing will be found in subfolder datasets. Subfolder op is the default location for output files and parapath is a convenient place for storing parameter files, which will be explained later.

In Windows, it is most convenient to install the system at the top level of the C:\ drive, at least to start with; otherwise you'll have to edit the sample parameter files to make sure their various file parameters point to the correct locations. On the Mac you'll probably have to unzip the distribution into a directory such as /Users/xxxx/beagling/ where "xxxx" is your user name. This will entail some editing of the parameter files provided. (Hint for Mac users: replacing "C:\" with "/Users/xxxx/" should do the trick.)

2.2 Data Format

The system expects to read its input values from data files such as can be exported from R (R Core Team, 2013) or Excel, with a header line giving column names, using the tab character as a delimiter. Data files can also be created in a text editor such as Notepad++ (http://notepad-plus-plus.org/), preferably in utf-8 encoding.

The first four and last four lines of the sample data file iris.dat are listed below to illustrate this format.

typename	sl	sw	pl	pw
setosa	5.1	3.5	1.4	0.2
setosa	4.9	3	1.4	0.2
setosa	4.7	3.2	1.3	0.2
[....]				
virgin	6.3	2.5	5	1.9
virgin	6.5	3	5.2	2
virgin	6.2	3.4	5.4	2.3
virgin	5.9	3	5.1	1.8

This dataset is a well-studied collection of 150 cases known as "Fisher's Iris Data". It was originally collected by Edgar Anderson who gathered the data to study the morphological variation of Iris flowers of three related species. Two of the three species were collected in the Gaspé peninsula in Quebec (Anderson, 1935). The dataset consists of 50 samples from each of three species of Iris (Iris setosa, Iris versicolor and Iris

virginica). Four features are measured from each sample: the length and the width of the sepals and petals, in centimetres. In the context of classification the point at issue is whether a rule or function can be devised to classify these example into 3 groups with high enough accuracy using the petal and sepal measurements.

This iris dataset is an example of a rectangular 'flat file' with instances as rows and attributes as columns, a format used by many machine-learning and statistical packages.

2.3 System Sketch

There are five programs in the RUNSTER suite and a typical run consists of running four or five of them in sequence.

Step	Program	Operation
0.	[None!]	Gathering & "cleansing" suitable example data. The system provides no software to support this explicitly even though it is the most crucial, and usually the most time-consuming, aspect of any machine-learning project! However, sample data sets are provided on subfolder datasets which allow you to become familiar with the data format, and how it is used, before collecting, checking and probably re-formatting, your own data.
1.	**seed.py** : Simple Exploratory Example Distributor	This program simply splits a data file into training and test sets. It takes a tab-delimited input file in rectangular format and randomly allocates items (rows) from that file to 2 output files. The proportion going into each file is approximately 0.618034 to 0.381966, respectively, but this proportion can be reset as a parameter option.
2.	**root.py** : Regression Oriented Optimization Tester	This is the main evolutionary learning program. It takes a training file of example cases and a target expression to be predicted and repeatedly uses an evolutionary algorithm to generate a ruleset for estimating the numeric target values. At the end it picks the best of these rulesets and writes it onto an output file to be read by the succeeding programs (and the user).
3.	**tree.py** : The Regression Estimation Evaluator	This program applies the ruleset written by root.py to predict target scores in a test file of example cases. Typically that will be the test data extracted by seed.py, to obtain a relatively unbiased error-rate estimate, but it may also be a holdout set of genuinely questionable examples for which estimates are required.
4.	**pear.py** : Program Exporting Applicable Rules	This program takes a rule file as written by root.py and translates it into Python3 or R so that it can be used in external software.
5.	**berries.py** : Bionically Evolved Regression Rule In Executable Software	This program essentially duplicates the function of tree.py: it uses the Python3 code written by pear.py to estimate dependent-variable values in a sample of test cases. Its usefulness is in illustrating how the generated functions can be incorporated into other Python programs, and as a check that the results are identical both when using RUNSTER's internal rule language (as in tree.py) and when using the derived Python code. (The generated R code contains a function named berries() that can be used for the same purpose, but within the R environment.)

2.4 Program Launch

Under Windows, there are three main ways to execute a Python program.

Perhaps the safest, and the mode closest to what is natural in Linux, is to open a command window and run the program from within that window. In Windows 10 that means right-clicking the bottom-left symbol and selecting "Command Prompt" from the menu that pops up. That should bring up an MS/DOS-style window,

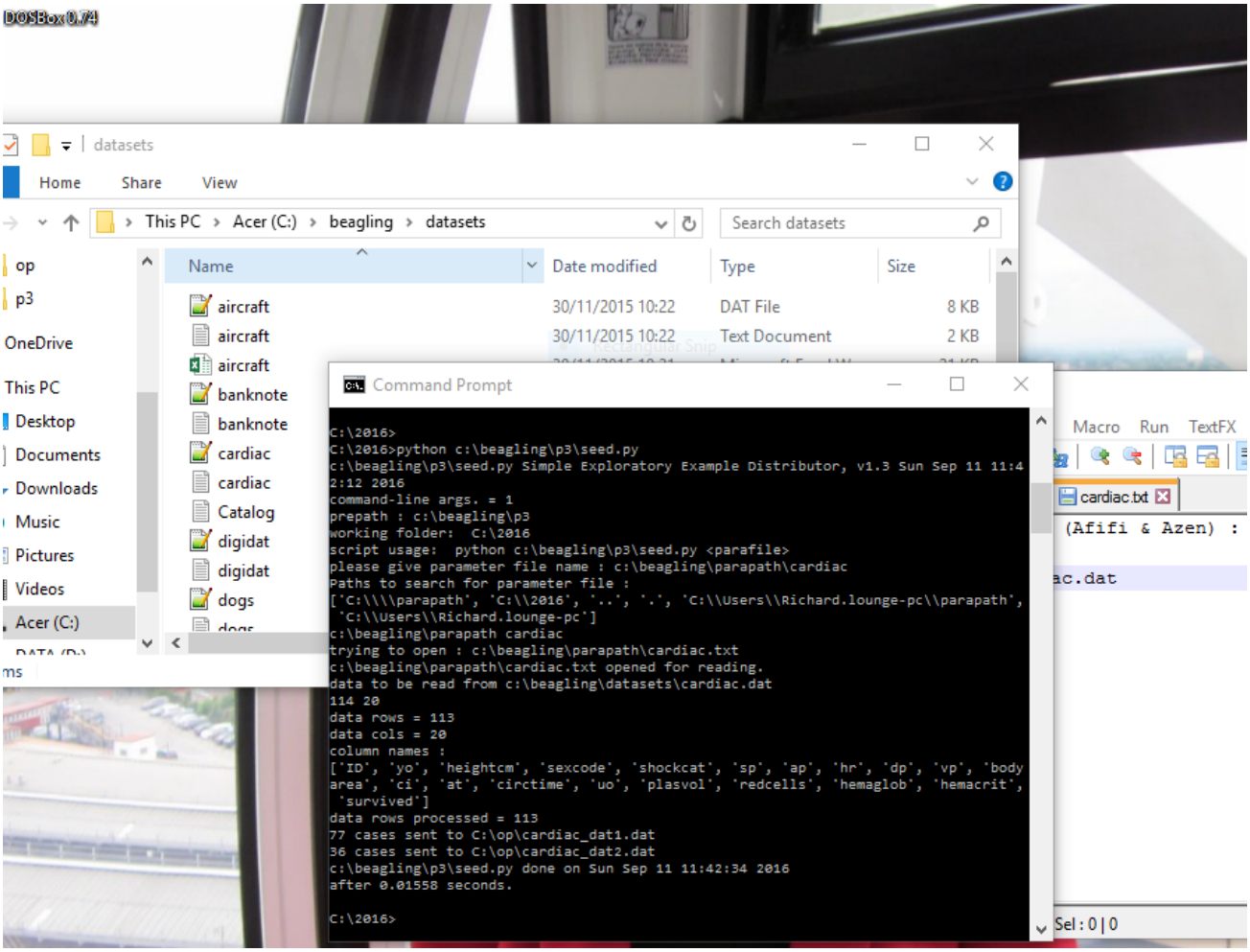
awaiting a command. At the prompt, you type (to run seed.py, for example) a command such as shown below, then press Enter.

C:\2016>**python c:\beagling\p3\seed.py**

Provided that you have Python3 installed normally, this will start the SEED program running. It will ask for a parameter file. In this case, since you're running from directory other than c:\beagling\p3\, namely C:\2016\, you'll need to give the full path of the parameter file, e.g.

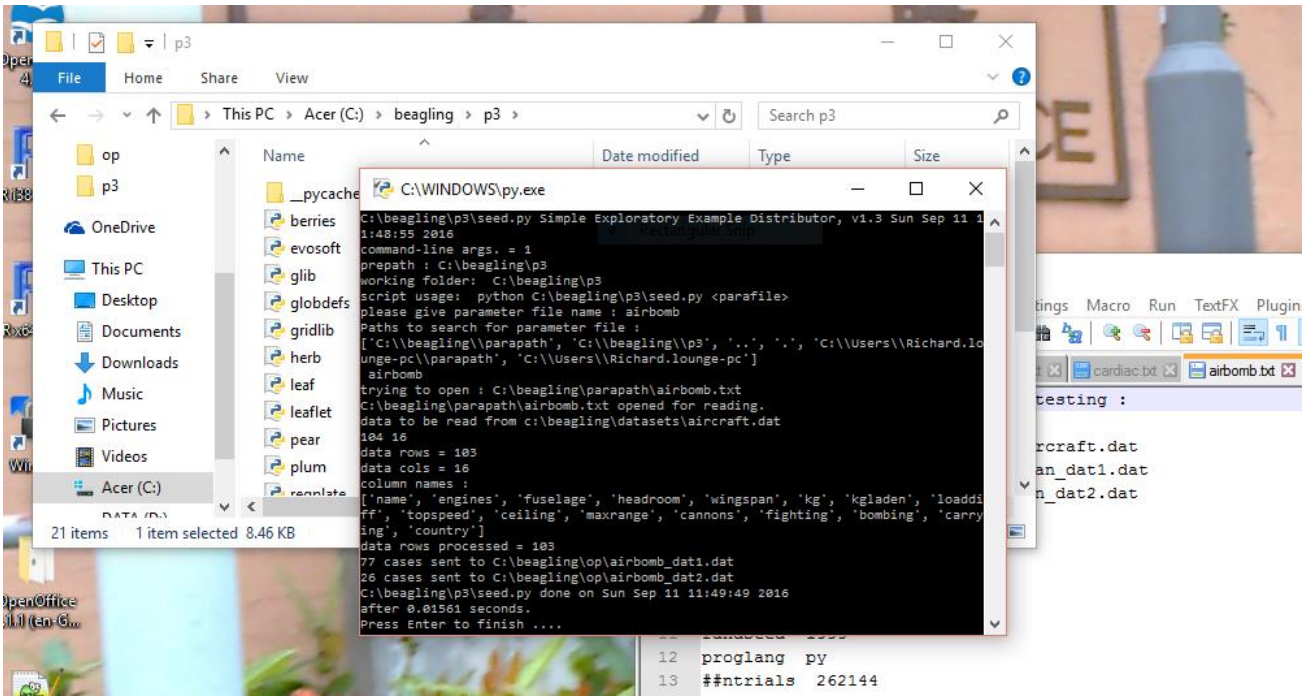
please give parameter file name : **c:\beagling\parapath\cardiac.txt**

where the user's input is in **bold**. (Actually, the .txt extension would be assumed if absent.) This refers to a parameter file, cardiac.txt, set up for analyzing the cardiac.dat data in the datasets folder (more details below). An example screen shot using this parameter file is shown below.



A second method is to navigate with File Explorer to the \beagling\p3\ directory, then select the program concerned (e.g. seed.py) and right-click on it. A menu should pop up with "Edit with IDLE" as an option near the top. Select that option and you'll be running IDLE with an active editing window containing the program. Along the top-line menu will be a "Run" option: click on that and pick "Run Module" to execute the program within a new window (which, on my desktop at any rate, always needs to be re-sized to fit my screen). Alternatively, just press Function Key F5. The snag with this method is that you might edit the program by accident, and, assuming you don't really want to alter it, the chances are that it won't work properly after that.

Thirdly, the lazy mode: having navigated to the \beagling\p3\ directory with File Explorer, you can just double-click on the program name. This will bring up a new command window in which the program runs. You'll then need to type in the parameter file name, as above, although if that file is located in the \beagling\parapath\ directory, you won't have to give its full path, just its name (with no need to type the extension either as long as it is ".txt"). A screen shot of running seed.py with the supplied airbomb.txt parameter file is shown below. (This parameter file refers to the aircraft.dat dataset, of which more later: see Appendix C.)



The only snag with this method is that the command window is temporary. The RUNSTER programs, if run in this manner, wait with the message

Press Enter to finish ....

when they are ready to finish, but as soon as you do press Enter (aka Return) the window vanishes. So if you want to examine the screen output, scroll back to look it over before pressing Enter to dismiss the window. Also, some errors can terminate the program abnormally, in which case the screen will disappear without showing the error message that you probably wanted to read.

2.5 Preparing a Parameter File

When you run a program in this suite it will ask for the name of a parameter file, as in the example above. Parameter files are used to select among RUNSTER's various option settings. Below is a listing of parameter file cardiac.txt which comes with the distribution in parapath.

```
comment  test on cardiac admissions data (Afifi & Azen) :
jobname  cardiac
maindat  c:\beagling\datasets\cardiac.dat
##targval  survived
targval  plasvol
idfield  ID
skipvars  ID
##ntrials  999999
rulemode  standard
brevity  1
```

```
randseed  3433
datfrac  .68
##proglang  r
proglang  py
```

This parameter file relates to a dataset (c:\beagling\datasets\cardiac.dat) which contains information published by Afifi & Azen (1979) on patients admitted to a Los Angeles hospital with heart failure. It describes 113 patients in terms of 20 attributes, all expressed as integers. It was collected with a view to finding ways of predicting survival status (the last column in the file) from measures that could be taken on arrival, but will be used here as an illustrative example of RUNSTER's symbolic regression capability to predict plasma volume index (plasvol) from the other variables. Plasma volume index is an estimate of the volume of liquid in the patient's blood, after deducting the volume of the cells in the blood. In humans it is usually between 4 and 5.5 litres. The attributes are briefly described below.

```
Cardiac variables:
ID,         Identification number
yo,         patient's age in years
heightcm,   height in centimters
sexcode,    1=male, 2=female
shockcat,   shock category
sp,         systolic pressure
ap,         mean arterial pressure
hr,         heart rate (pulse)
dp,         diastolic pressure
vp,         venous pressure
bodyarea,   body area
ci,         circulatory index
at,         appearance time
circtime,   circulation time
uo,         urinary output
plasvol,    plasma volume index, centilitres
redcells,   red cell index
hemaglob,   hemaglobin
hemacrit,   hemacrit
survived,   1=lived, 2=died
```

A parameter file such as \beagling\parapath\cardiac.txt is just a plain text file with one item per line. Each line should begin with the parameter name, then 1 or more blank spaces, then the parameter value. The following table interprets the above parameter file, line by line. Some parameters have default values that will be used if they are omitted from the parameter file or given an inapplicable value.

Parameter	Default value	Function
comment	[None]	This (or in fact any unrecognized parameter name, e.g. "##") can be used to insert reminders about what the file is meant to do.
jobname	runster	This gives the job a name. Any text string can be the value. It isn't necessary but it is useful as the jobname will be used as a prefix to the system's output files, so it can be seen that they form a related group.
maindat	[None]	This should be the full file specification of a file where the input data is stored (in tab-delimited form with a header line naming the columns, and each row representing a single instance).
targval	[None]	This parameter is used to define the target values, i.e. the numeric dependent variable. It is often simply a variable name (as here, "plasvol") but can be a more complex expression. Examples later.
idfield	[None]	This selects an identifying variable for used by root.py, tree.py & berries.py in their output listings.
skipvars	[None]	If used, this should be a comma-delimited list of column names that will be excluded from the rule-generation process. There is no need to

		exclude variables in the target expression.
rulemode	standard	In "standard" mode, a single RUNSTER expression will be generated to compute the value of the target expression. The alternative is "tabular" mode which will perform tabular regression, i.e. create a ruleset that indexes a table of possible output values.
brevity	1	This parameter applies to root.py. If it is 1, the system uses the size of each rule or ruleset in assessing its quality (in effect, as a tiebreaker, with bigger meaning worse); if it is 0, size is not taken into account when computing the quality of a rule or ruleset.
randseed	0	This number is used to initialize Python's pseudo-random number generator in seed.py and root.py -- except that if it is 0 or 1 or negative the generator will be seeded from the system clock, i.e. haphazardly, so each run will usually produce slightly different results. To have deterministic results, pick a number from 2 to 999999999, preferably a prime.
datafrac	0.61803398875	This specifies the fraction of the cases (rounded to the nearest whole number) in the full dataset (see maindat) that will be copied into the training datafile by seed.py. The remaining cases will go into the test datafile. Allocation of individual cases to each file is (pseudo-)random (see randseed).
proglang	r	This parameter applies to pear.py. It selects the programming language for the export of a ruleset. Valid options are py (Python3) & r.

Note that some parameters apply only to certain programs in the suite. If a parameter isn't relevant to a given program it will be ignored by that program; thus a single parameter file can be used for a complete run through the whole suite of programs. (More information about parameter files can be found in Appendix A.)

3. SEED.py : Simple Exploratory Example Distributor

SEED is used to split a dataset into training and test sets, which is a typical first step in using a system such as BEAGLE. The listing below reproduces the first and last four lines of the input data file cardiac.dat.

```
ID      yo      heightcm        sexcode shockcat        sp      ap      hr      dp      vp      bodyarea
        ci      at      circtime        uo      plasvol redcells        hemaglob        hemacrit
        survived
517     68      165     1       2       114     88      95      73      17      141     66      115     225
        110     562     206     113     340     1
537     37      171     1       2       149     115     76      97      36      182     355     82      156
        40      507     234     127     390     1
546     50      175     1       2       146     101     76      74      80      169     405     56      125
        0       644     239     134     410     1
[.... 96 lines omitted to save space ....]
543     52      152     2       3       82      52      106     38      189     155     589     28      97
        0       663     124     71      300     2
715     76      152     2       2       116     88      122     70      83      144     188     144     342
        23      498     171     96      290     2
634     52      185     1       2       112     67      73      49      150     200     380     82      151
        0       525     152     92      280     2
629     66      178     1       2       114     59      102     44      138     189     348     90      168
        0       495     206     93      280     2
```

By running seed.py and using the parameter file shown in the previous section, you should see on-screen output something like that below, which comes from running seed.py within the IDLE environment. The only user input is "cardiac" supplied in response to the program's request for a parameter file name. This parameter file is supplied with the RUNSTER distribution. (Extension .txt is assumed if no extension is given.)

```
>>>
```

```
C:\beagling\p3\seed.py Simple Exploratory Example Distributor, v1.3 Tue Sep 13 12:24:49
2016
command-line args. = 1
prepath : C:\beagling\p3
working folder:  C:\beagling\p3
script usage:  python C:\beagling\p3\seed.py <parafile>
please give parameter file name : cardiac
Paths to search for parameter file :
['C:\\beagling\\parapath', 'C:\\beagling\\p3', '..', '.', 'C:\\Users\\Richard.lounge-
pc\\parapath', 'C:\\Users\\Richard.lounge-pc']
 cardiac
trying to open : C:\beagling\parapath\cardiac.txt
C:\beagling\parapath\cardiac.txt opened for reading.
data to be read from c:\beagling\datasets\cardiac.dat
114 20
data rows = 113
data cols = 20
column names :
['ID', 'yo', 'heightcm', 'sexcode', 'shockcat', 'sp', 'ap', 'hr', 'dp', 'vp', 'bodyarea',
'ci', 'at', 'circtime', 'uo', 'plasvol', 'redcells', 'hemaglob', 'hemacrit', 'survived']
data rows processed = 113
77 cases sent to C:\beagling\op\cardiac_dat1.dat
36 cases sent to C:\beagling\op\cardiac_dat2.dat
C:\beagling\p3\seed.py done on Tue Sep 13 12:24:53 2016
after 0.04682 seconds.
>>>
```

From this it will be seen that 77 cases have been put into the training file and 36 into the test file. This is the whole-number approximation to the partition ratio given by the datfrac parameter (explained in the previous section and in Appendix A).

The program has read data in from cardiac.dat on the folder c:\beagling\datasets\ and created 2 new files in the op folder with the jobname as filename and "_dat1.dat" and "_dat2.dat" appended. This is because the parameter file specified an input source (maindat) but did not give values for parameters traindat and testdat. (For more details of parameters that can modify the behaviour of seed.py, see Appendix A.)

4. ROOT.py : Regression Oriented Optimization Tester

Continuing with the cardiac example, using the parameter file shown above, the natural next step is to create a regression rule with root.py. In this case we're seeking a rule to estimate plasvol from computations performed on a selection of other variables.

The only user input required at runtime is the parameter file name (here cardiac). The main data input will be read from whatever file is specified by the traindat parameter. If this is absent, as in the present example, the program will seek an input file composed of the jobname with "_dat1.dat" appended on the current output path (c:\beagling\op\ by default). Thus the traindat file which was output from seed.py becomes input to root.py.

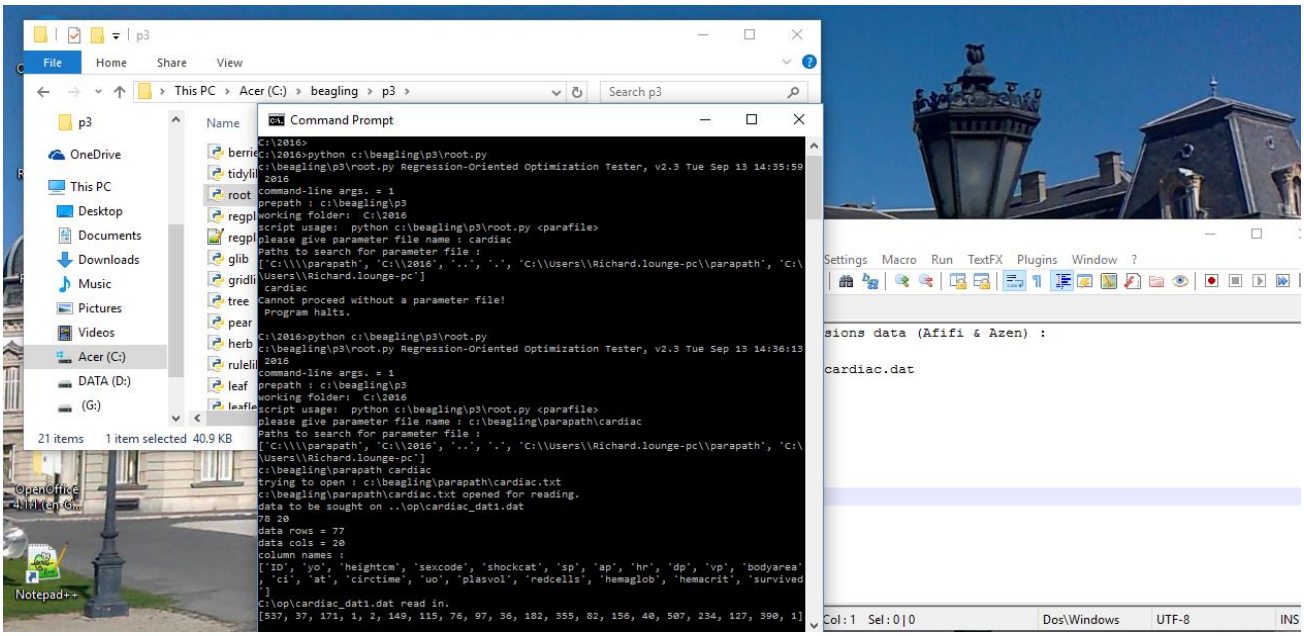
ROOT uses several cycles of repeated subsampling. That is, it applies its evolutionary algorithm several times (from five to nine times, depending on the dataset size). In each cycle, a small number of training cases (the square root of the total number of training cases, rounded) are set aside. The evolutionary loop is then applied to the remainder of the cases to generate a ruleset. This ruleset is then applied to the set-aside cases and its performance recorded. When all cycles are completed, statistics are accumulated for all the set-aside examples. This approach to internal testing, splitting the training set into sub-training and sub-test sets, is intended to provide error estimates that are not optimistically biased. The primary measure of fitness is the mean absolute deviation of each predicted value from the actual value of the target.

Finally, the best rulesets generated in each cycle are reapplied to the entire training set and the highest-

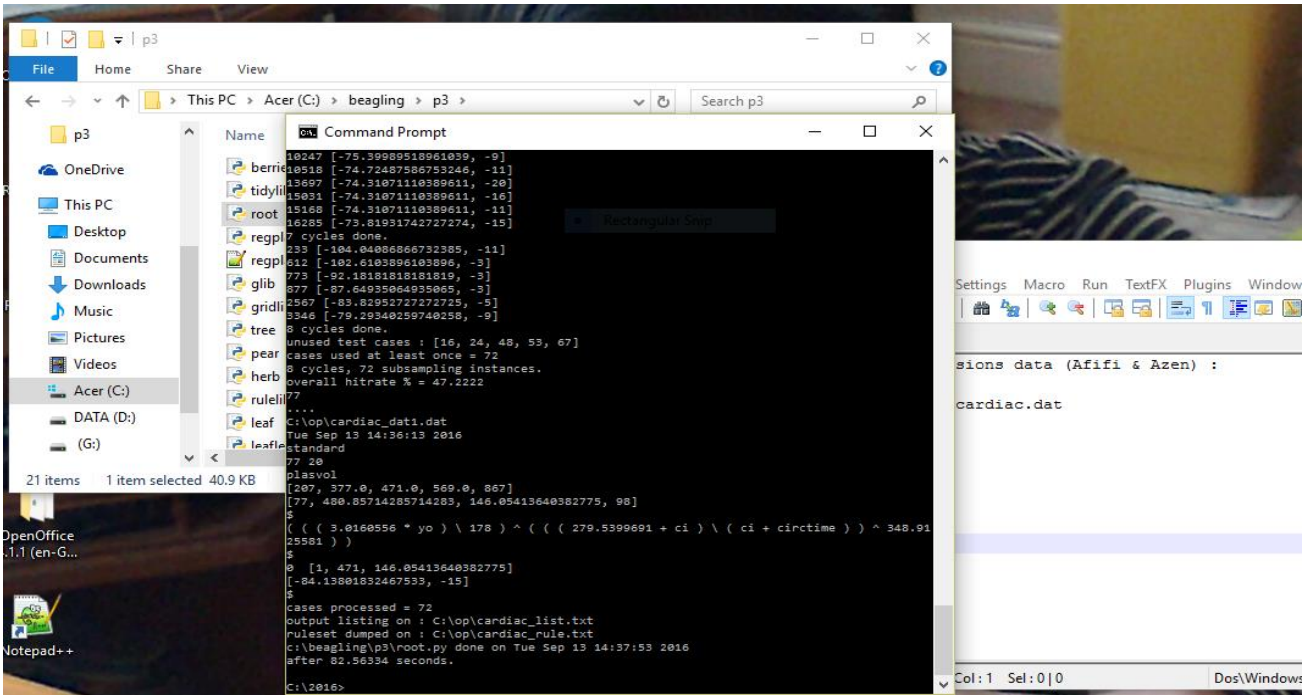
scoring ruleset among these is output to be used by subsequent programs.

During each rule-optimization cycle, the program displays on screen the new best score every time a new best-scoring rule is found, and at the end of each cycle it will also show the rule if its score is higher than the best from previous cycles. This ensures that even during a long run you can see that the program is working.

The screen shot below shows the output from the beginning of a run of root.py from a directory other than \beagling\p3\. In the first attempt, I only gave the parameter file name, cardiac, forgetting that I was running the program from a directory other than c:\beagling\p3\. This didn't work, so I ran it again, and this time gave the full parameter file specification c:\beagling\parapath\cardiac.



The screen shot below gives an idea of what to expect on screen at the finish of a ROOT run. It is the ending of the run using the cardiac dataset whose start is shown above.



4.1 ROOT's evolutionary algorithm

The core algorithm in ROOT that applies the evolutionary optimization method within each subsampling cycle is outlined in the following pseudocode. The overall number of subsampling cycles, K, will be between 5 and 10 depending on size of dataset.

[Parameters:
C counter to record number of fresh rulesets created;
P number of rulesets in the population, popsize;
R number of individual tree-structured rule expressions per ruleset (default 1);
T maximum number of fresh rulesets to be created in current cycle. T will be maximum number of fresh rulesets allowed overall (default 131072) divided by K.]

1. Create an initial random population of P rulesets, each containing R rule-expressions. Set C to P.
2. Examine M (4 by default) population members selected at random and call the highest-scoring item p1.
3. Examine M (4 by default) population members selected at random and call the lowest-scoring item p0.
4. Pick a member of the population at random and call it p2.
5. Mate p1 with p2 (i.e. apply 'crossover' operator) and replace p0 with the resulting 'offspring'.
6. Increment C.
7. With probability m1 (default 0.5) apply mutation to the newly made p0, i.e. make a small random change.
8. Evaluate the new ruleset p0 and show score if best so far.
9. With probability m2 (default 0.25) pick a population item randomly and apply mutation to it. Re-evaluate it and show score if best so far; also increment C.
10. If C exceeds T, exit cycle; otherwise continue from step 2.

The crossover routine, when applied to a tree-structured rule-expression, just consists of taking a random subtree from one parental expression (which could be the whole tree), doing the same to the other tree, and joining them with a connective randomly taken from either tree (or picked at random if that would violate the syntax rules). For example, mating

```
( ( ( ( at + yo ) + ci ) ^ redcells ) ^ (  ( ( at + vp ) + ci ) ^ heightcm ) )
```

with

```
( ( ci + circtime )  \  ( 341 ^ ( ci * 2.2503507 ) ) )
```

could produce

```
( ( (at + vp ) + ci ) \ ( 341 ^ (ci * 2.3503507) ) )
```

among many other possible 'offspring'. Then, if mutation were applied to the result, it might be changed to something like the following:

```
( ( (at + vp ) + $Root ci ) \ ( 344 ^ (ci * 2.3503507) ) )
```

again, among many other possibilities. The point is that such chopping, changing and recombining can be done mechanistically without knowing the meaning of the terms. (RUNSTER's rule language is described in Appendix B.)

4.2 Example ruleset derived from cardiac data

The two main output files of root.py are a rule file, which will be named as the jobname with "_rule.txt" appended unless otherwise specified, and a listing file (likewise with "_list.txt" appended).

A complete rule file (cardiac_rule.txt) derived from the cardiac_dat1.dat data follows.

```
training data : C:\op\cardiac_dat1.dat
creation date : Tue Sep 13 14:34:19 2016
rule mode : standard
77 20
plasvol
[207, 377.0, 471.0, 569.0, 867]
[77, 480.85714285714283, 146.05413640382775, 98]
$
( ( ( 3.0160556 * yo ) \ 178 ) ^ ( ( ( 279.5399691 + ci ) \ ( ci + circtime ) ) ^ 348.9125581 ) )
$
0  [1, 471, 146.05413640382775]
[-84.13801832467533, -15]
$
```

This can be subdivided into 3 sections, each ended by a dollar sign on a line of its own.

First come seven lines giving information about the training data. Next comes the rule itself, a one-line expression in this case. Thirdly come 2 more lines, one that will be explained in Appendix C and a second which indicates the quality or fitness score of this particular rule. This latter consist of 2 numbers, because brevity was set to 1 in the parameter file: the number -84.13801832467533 which is the mean absolute difference between the estimates computed by this rule and the actual target values (plasvol) in the training data, then -15 which is the length of the rule. Both these numbers are negated because the evolutionary algorithm maximizes but smaller deviations and shorter rules are preferred.

In the first section, lines 1 and 2 should be self-explanatory. The third line indicates that the rule mode used was "standard", i.e. that the rule is an expression which calculates an estimated target value directly. The alternative rule mode is "tabular", which uses several rules to index a cell in a 'signature table' (Samuel, 1967). Tabular mode is illustrated in Appendix C. Line four shows the number of rows and columns in the training data. The next 2 lines give the target expression, here just a variable name, followed by Tukey's "five-number summary" (Upton & Cook, 2006) of the target values in the training data. The final line before the dollar sign, gives the number of training instances, their mean target value, the standard deviation of the target values and the median absolute difference between each target value and its median (MADM). The latter is a nonparametric measure of variability.

The next section gives the regression rule itself. As a starting point, the subexpression on the left

```
( ( 3.0160556 * yo ) \ 178 )
```

multiplies yo, the age of the patient in years by 3.0160556 and then takes the lesser value of that multiplication or the constant 178 as its result, the backslash ('\') being RUNSTER's minimum operator. The full expression also employs RUNSTER's maximum operator ('^') as well as addition ('+'). RUNSTER's rule language is more fully explained in Appendix B. For the moment it suffices to note that the result has selected three variables, ci, circtime and yo. This feature-selection is a purely automatic by-product of the rule-optimization procedure: it does not require human intervention.

When applied to a fresh instance, this rule will be evaluated to produce a numeric estimate of the target.

4.3 Listing file derived from cardiac data

ROOT also produces a listing file (normally named from the jobname with "_list.txt" appended) that

summarizes how well the system performed during the resampling cycles. The subsampling process, which ensures in each cycle that rules are tested on (held-out) cases that were not used to measure the fitness scores of rulesets generated during the evolutionary optimization procedure, is designed to ensure that these summary statistics aren't optimistically biased.

An extract from the listing file produced by root.py from the run on the cardiac data that produced the rule described above follows.

```
dateline   Tue Sep 13 14:32:52 2016
progname   C:\beagling\p3\root.py
id         C:\beagling\parapath\cardiac.txt
traindat   C:\op\cardiac_dat1.dat
targval    plasvol

====subsampling trial :

rank  safeness case  ID                            pred:true      cellsize      abdsiff     diffsqrd
   1      0.28   21  732                           851 + 744            77       107.00     11449.00
   2      0.33   20  722                           777 - 494            77       283.00     80089.00
   3      0.37   15  689                           731 + 645            77        86.00      7396.00
   4      0.38   57  702                           715 - 561            77       154.00     23716.00
   5      0.44   36  699                           668 + 687            77        19.00       361.00
   6      0.45   70  593                           659 + 755            77        96.00      9216.00
   7      0.46   18  716                           653 + 867            77       214.00     45796.00
   8      0.46   73  541                        651.52 - 479            77       172.52     29763.15
   9      0.53   75  634                           612 - 525            77        87.00      7569.00
  10      0.54    1  546                      604.5051 + 644            77        39.49      1559.85

[.... 50 lines omitted to save space ....]

  61      0.52   52  620                           344 + 348            77         4.00        16.00
  62      0.51    9  649                      341.9136 - 386            77        44.09      1943.61
  63      0.51   31  696                      341.5051 - 553            77       211.49     44730.11
  64      0.51   45  527                           341 - 709            77       368.00    135424.00
  65      0.51   26  412                           341 + 362            77        21.00       441.00
  66      0.51   22  584                           341 + 210            77       131.00     17161.00
  67      0.51    4  583                           341 - 458            77       117.00     13689.00
  68      0.51    2  563                           341 + 294            77        47.00      2209.00
  69      0.51   65  444                           340 + 335            77         5.00        25.00
  70      0.50   27  518                      332.5051 + 321            77        11.51       132.37
  71      0.43   66  719                           286 + 300            77        14.00       196.00
  72      0.43   23  613                           286 + 349            77        63.00      3969.00
+-+-+++--+-+++--+-+++---+-----++------+---+--+---++--+-++++-+---++-+++++

'success' percentage = 47.22
pearson correlation between predicted & true vals = 0.5796
spearman rank-correlation between predicted & true vals = 0.6233

mean abs.error = 91.7103
mean error ^ 2 = 15686.9743
correlation between safeness & abs.error (negative better) = 0.0097

Resultant rule from all training cases :
C:\op\cardiac_dat1.dat
Tue Sep 13 14:32:52 2016
standard
77 20
plasvol
[207, 377.0, 471.0, 569.0, 867]
[77, 480.85714285714283, 146.05413640382775, 98]
$
( ( ( 3.0160556 * yo ) \ 178 ) ^ ( ( ( 279.5399691 + ci ) \ ( ci + circtime ) ) ^ 348.9125581 ) )
$
0  [1, 471, 146.05413640382775]
[-84.13801832467533, -15]
$

[Parameter settings omitted to save space ....]
```

The first five lines of this listing just display some of the more important parameter settings, for reference. Then, after "==== subsampling trial :", details of the 72 decisions made during the subsampling trial follow.

Fifty of these have been removed from this extract to save space, leaving only the first 10 and last 12 cases.

Taking the item at rank 70 as illustration,

```
  70      0.50   27  518                      332.5051 + 321           77        11.51       132.37
```

this line can be used to explicate the values in the various columns. The number 70 is the rank, out of 72, ranked by the predicted value in descending order. The second number, 0.50, in the column headed "safeness", is intended as an index of reliability. In "standard" mode, it is computed as (SD / (SD + d)) where SD is the target value standard deviation in the training set and d is the absolute difference of the predicted value from the mean of the target value in the training data. The idea is that predictions far from the expected mean value are more risky than those nearer that mean value. However, in practice, this hasn't proved to be a very reliable reliability measure. Finding a better one is work in progress. If and when I do find one, it will be incorporated in the next release.

The next two numbers give the relative position of the case within the training file and the value of the variable selected as the idfield in the parameter file, which in this case is just a serial number. Then the three items

```
        332.5051 + 321
```

show the predicted value, a 'success' marker, and the actual target value. Success in this sense is indicated as plus or minus ('+' or '-'). It is a fairly crude measure, included primarily by analogy with the BEAGLE classification software. It is defined in terms of Tukey's five-number summary: the lowest, lower quartile, median, upper quartile and highest of the training-data target values. These are carried within the rule. Here they are 207, 377.0, 471.0, 569.0 and 867. In effect, they define 6 ranges, from below the lowest to above the highest and all the gaps in between. In practice, values below the training-set minimum or above the training-set maximum are very rare, so the chance of 'success' -- which means that both predicted and actual target values fall in the same interval, as defined by these five numbers -- is roughly one in four. Getting 47.22% successes, as in this example, thus represents only a modest improvement on chance. The item at rank 70 is marked as a success ('+') because both the predicted (332.5051) and actual (321) values fall between the minimum and the lower quartile (377.0).

The number 77, under the heading "cellsize", doesn't vary in standard mode. It is just the number of training instances. It indicates the number of instances in which the prediction is based (which is useful information in tabular mode).

The last two numbers, 11.51 and 132.37, are the absolute and squared differences between the predicted and actual target values, respectively.

The line just after the last item

```
+-+++--+++--+++---+----+-----+---+--+---+--+++-+---+-+++++
```

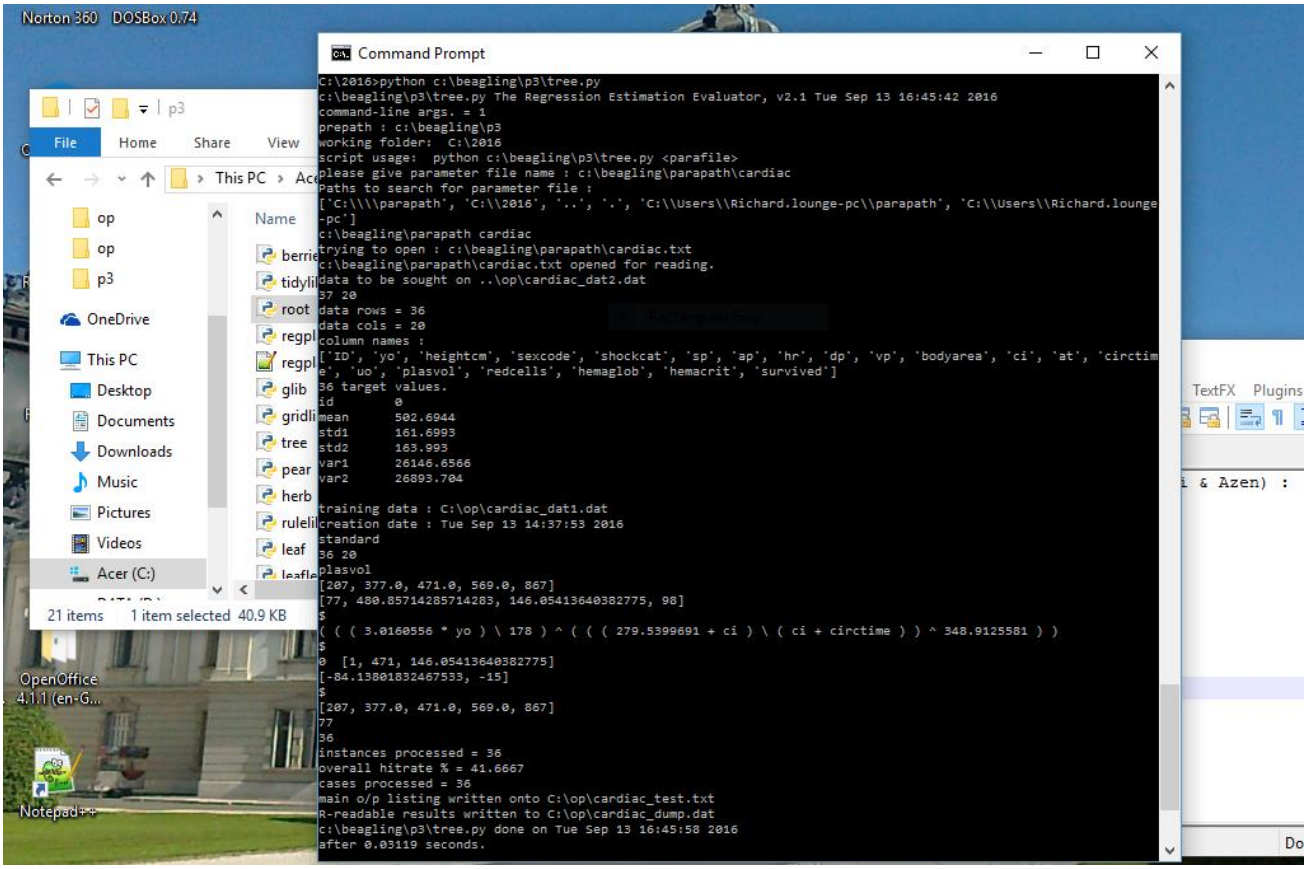
shows the error-status of each of the 72 decisions, ordered left to right as they are listed. However, a better measure of performance is the mean absolute error of 91.7103. This gives a relatively pessimistic estimate of the likely difference between predicted and actual values when the rule expression is used on fresh data of the same kind. It differs from the quality score of the rule itself (84.138) because of ROOT's subsampling behaviour. In this example root.py performed 8 subsampling cycles in each of which a rule was generated on 68 training instances and then tested on the remaining 9 held-out cases. Thus this value is internally cross-validated. The rule selected for final output is whichever of the eight achieved the lowest absolute deviation on all 77 training instances. This had a mean absolute error of 84.138, which can be regarded as an

optimistic estimate of future errors.

Whether this is satisfactory would be a clinical decision with medical data such as in the cardiac dataset. The fact that the standard deviation in this training set is 146.054 and the median absolute deviation from the median is 98, both greater than either the optimistic or pessimistic mean absolute deviation, suggests that this rule is not useless.

5. TREE.py : The Regression Estimation Evaluator

The normal next step after running seed.py and root.py is to run tree.py on the data held out as a test sample. A screen shot form running tree.py with the cardiac test data is shown below.



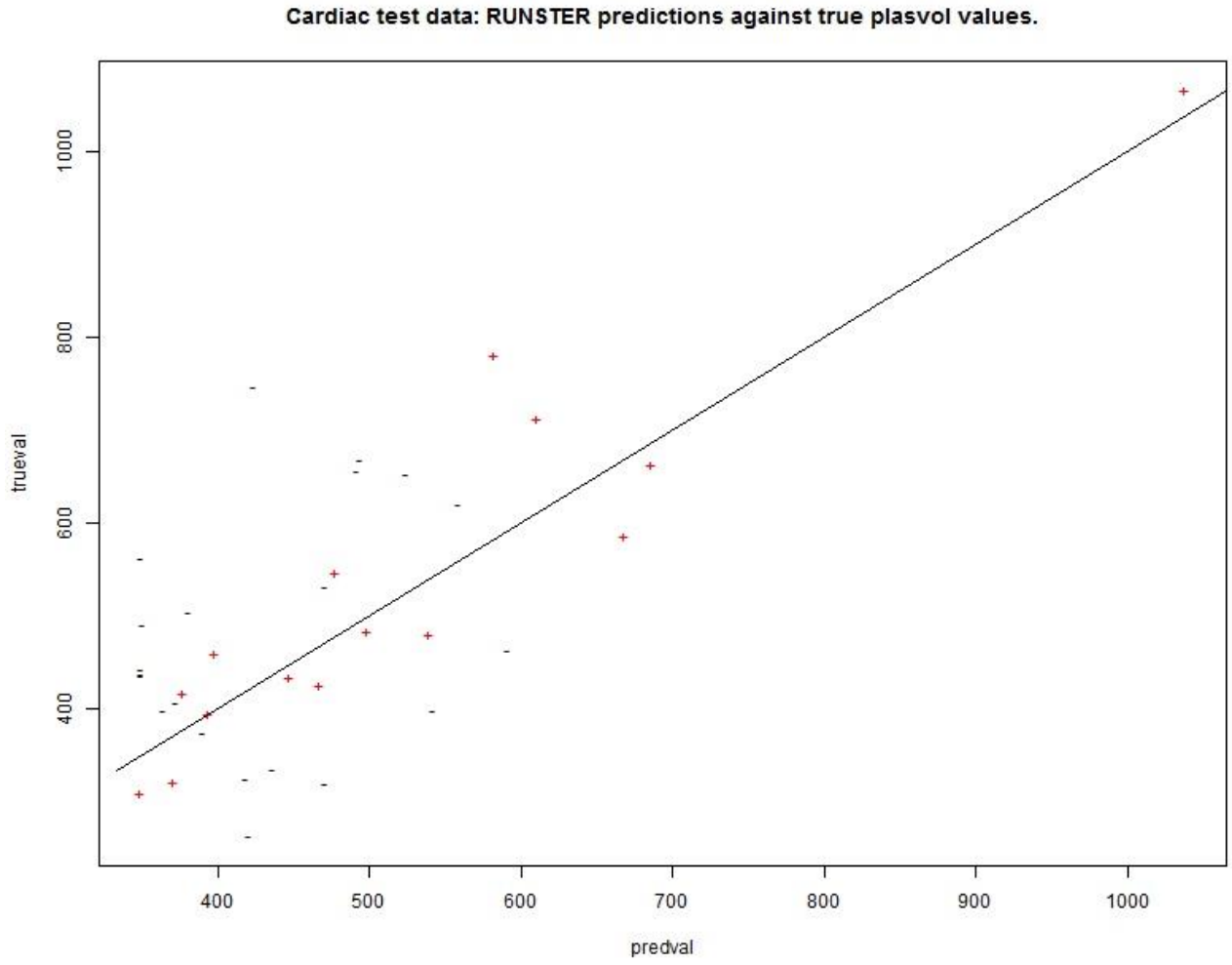
The following is the main listing produced by tree.py (suffixed "_test.txt") on the test data file cardiac_dat2.dat. The output is in the same format as the listing from root.py (suffixed "_list.txt") shown above and can be interpreted in essentially the same way.

```
dateline    Tue Sep 13 16:45:42 2016
progname    c:\beagling\p3\tree.py
id          c:\beagling\parapath\cardiac.txt
testdat     C:\op\cardiac_dat2.dat
targval     plasvol

====holdout trial :

rank  safeness case  ID                          pred:true      cellsize      abdsiff      diffsqrd
   1      0.21   19  723                       1038 + 1066            77        28.00        784.00
   2      0.42   35  543                        686 + 663             77        23.00        529.00
   3      0.44   20  731                        668 + 585             77        83.00       6889.00
   4      0.53   18  720                        611 + 712             77       101.00      10201.00
   5      0.57    7  713                        591 - 463             77       128.00      16384.00
   6      0.59   31  707                        583 + 780             77       197.00      38809.00
   7      0.65    2  639                        559 - 620             77        61.00       3721.00
```

```
   8      0.70    8  742                        542 - 398           77       144.00     20736.00
   9      0.71   34  588                        540 + 479           77        61.00      3721.00
  10      0.77   16  526                        524 - 653           77       129.00     16641.00
  11      0.89   11  734                        499 + 483           77        16.00       256.00
  12      0.92    4  684                        494 - 668           77       174.00     30276.00
  13      0.94    6  705                        491 - 656           77       165.00     27225.00
  14      0.98   30  725                        478 + 547           77        69.00      4761.00
  15      0.94   13  592                        471 - 531           77        60.00      3600.00
  16      0.93   27  691                        470 - 319           77       151.00     22801.00
  17      0.91    1  630                        467 + 425           77        42.00      1764.00
  18      0.82    3  664                        448 + 433           77        15.00       225.00
  19      0.76   33  529                      435.54 - 335          77       100.54     10108.29
  20      0.72   15  515                        423 - 747           77       324.00    104976.00
  21      0.71   32  660                      420.54 - 262          77       158.54     25134.92
  22      0.70    9  625                      418.54 - 324          77        94.54      8937.81
  23      0.64   22  554                        398 + 459           77        61.00      3721.00
  24      0.63   14  598                      394.54 + 395          77         0.46         0.21
  25      0.62   12  426                      389.54 - 373          77        16.54       273.57
  26      0.59   29  733                        380 - 504           77       124.00     15376.00
  27      0.59   26  704                      377.54 + 417          77        39.46      1557.09
  28      0.57   25  718                        372 - 407           77        35.00      1225.00
  29      0.57   10  672                      371.54 + 321          77        50.54      2554.29
  30      0.56    5  687                        364 - 397           77        33.00      1089.00
  31      0.53   17  657                        350 - 490           77       140.00     19600.00
  32      0.53   28  662                    348.9126 - 436          77        87.09      7584.22
  33      0.53   24  535                    348.9126 - 437          77        88.09      7759.40
  34      0.53   23  653                    348.9126 - 442          77        93.09      8665.27
  35      0.53   21  540                    348.9126 + 308          77        40.91      1673.84
  36      0.53    0  517                    348.9126 - 562          77       213.09     45406.26
++++-+--+-+--+--++----++--+-+-----+-

'success' percentage = 41.67
pearson correlation between predicted & true vals = 0.7342
spearman rank-correlation between predicted & true vals = 0.5503

mean abs.error = 92.969
mean error ^ 2 = 13193.4768
correlation between safeness & abs.error (negative better) = 0.1575

Resultant rule from all training cases :
training data : C:\op\cardiac_dat1.dat
creation date : Tue Sep 13 14:37:53 2016
standard
77 20
plasvol
[207, 377.0, 471.0, 569.0, 867]
[77, 480.85714285714283, 146.05413640382775, 98]
$
( ( ( 3.0160556 * yo ) \ 178 ) ^ ( ( ( 279.5399691 + ci ) \ ( ci + circtime ) ) ^ 348.9125581 ) )
$
0  [1, 471, 146.05413640382775]
[-84.13801832467533, -15]
$

[Parameter settings omitted to save space ....]
```

Here the Pearson correlation between predicted and actual target values (0.7342) is actually higher than estimated in the root.py listing (0.5796). The mean absolute error of 92.969 is very much in line with the pessimistic estimate of root.py.

To give a visual impression of the relationship between this rule's estimates and the actual plasma values, a scatter plot of predicted versus actual plasvol values is shown below. The diagonal line indicates where the points would fall if the correlation were perfect.



As a further comparison, I used R to form a linear model on the same training data, using the three variables identified by root.py. R's lm() function produced the following linear model.

```
lm(formula = plasvol ~ ci + circtime + yo, data = cardiac1)

Coefficients:
(Intercept)           ci     circtime           yo
   169.8403       0.8210       0.4940      -0.1721
```

When applied to the unseen training data (cardiac_dat2.dat) this linear model's estimates had a Pearson correlation with the true plasvol values of 0.6553, substantially lower than that of the root.py rule. The mean absolute error of the linear model was worse as well (102.26 versus 92.969). Some other combination of variables might give a better linear model, but the nice thing about RUNSTER is that the task of seeking good sets of predictive variables does not require extra work. (Traditional feature-subset selection in multiple regression is an arduous process.) In sum, this is not a trivial example. Although a 'success' rate of 41.67 percent may not seem impressive, the RUNSTER results are competitive with those of a more conventional statistical method.

6. PEAR.py : Program Exporting Applicable Rules

It is all very well to have a system learn a rule or ruleset, but it takes a lot of work: planning, data collection & collation, data checking, experimentation and so on. If you're lucky, and you have chosen your training data wisely, the reward for all that work is a ruleset that will make reliable target estimates from fresh examples

of the same sort of data.

It is unlikely that you'll be satisfied with expressions in RUNSTER's idiosyncratic rule-language as a final outcome, even if they appear to be highly accurate. For that reason, the pear.py module is provided, to translate from RUNSTER's expression language either into Python3 or R, the latter being the language of choice among people nowadays known as "data scientists".

PEAR works by taking in a rule file produced by root.py and combining it with one of 2 template files, regplate.py or regplate.r, provided with the distribution, which should reside in the same directory as pear.py (normally c:\beagling\p3\). These templates are program skeletons which pear.py fills in by translating the information in ROOT's rule file into a suitable format for the programming language concerned.

A complete listing of the Python source code file (cardiac_rule.py) produced by pear.py from the file cardiac_rule.txt follows.

```
##  Using RUNSTER Py template, version of 12/09/2016 :
##  rule written by pear.py ;
##  derived from training data : C:\op\cardiac_dat1.dat;
##  generated on creation date : Tue Sep 13 14:37:53 2016;
##  dumped on Tue Sep 13 19:33:22 2016.
##
beag_gold = 5.0 ** 0.5 * 0.5 + 0.5  ##  global

import  math  ##  math library called upon

##  helper functions :
def beag_bool (v):
       ##  ensures same bool/math treatment as in Beagle :
       return  (v > 0) + 0

def beag_exor (v1,v2):
       ##  exclusive or, as in Beagle :
       return  ((v1>0) != (v2>0))

def beag_root(v):
       ##  safe square root :
       if v >= 0.0:
              return  math.sqrt(v)
       else:
              return  -math.sqrt(abs(v))

def beag_slog (v):
       ##  safe natural logarithm :
       if v < 0:
              return -math.log1p(abs(v))
       else:
              return  math.log1p(v)

def runster_stabprep ():
       ##  sets up fallout table :

       stab = {}
       stab['0'] = [1, 471, 146.05413640382775]

       ##  unpacks stab lines.

       return (stab)
       ##  stabprep ends.


def runster_regrule (vals,stab):
       ##  input vals should be an object with appropriate attribute names.
```

```
       ##  target : plasvol.
       ##  rule mode is standard.

       bins = ['0','1']  ##  omit if demonic
       catlist = [207, 377.0, 471.0, 569.0, 867]
       priorvec = [77, 480.85714285714283, 146.05413640382775, 98]
       subrules = 1
       rule = [0] * 1
       ##  compute rule values :
       rule[0] = max(min((3.0160556 * vals.yo),178),max(min((279.5399691 +
vals.ci),(vals.ci + vals.circtime)),348.9125581))

       p = 0 ; b = []
       while (p < subrules):
              v = rule[p]  ##  omit if tabular
              b = ['0']  ##  omit if tabular
              p = p + 1  ##  early-r, late-py

       b = ''.join(b)  ##  should work in both modes
       ##  standard :
       predval = v  ##  omit if tabular
       cellvals = priorvec  ##  omit if tabular
       cellsize = priorvec[0] ; standev = priorvec[2]  ##  omit if tabular
       smalldif = priorvec[3]  ## should work in both modes
       ##  tabular :
       ##  retrieve cell frequencies :

       return ([b,predval,standev,smalldif,cellsize])
##  regression rule ends.

##  should be loadable into berries.py .

##  ending.
```

This module begins with comments and global definitions and then a definition of some helper functions for implementing the 'safe' natural logarithm and square root operators with the same semantics as in RUNSTER, and ensuring that RUNSTER's version of conversions between True/False and numeric values is preserved in Python.

Then comes beag_stabprep() which sets up the signature table with the data-derived sums to be used in tabular mode (irrelevant in standard mode but left in for compatibility).

Next follows the Python function runster_regrule(,) that makes decisions about individual instances. This function takes an input data object, vals, with appropriate attributes and produces an output list containing indicating its computed target value, the number of training instances on which the decision is based (more relevant in tabular mode).

Because you have the source code, you can examine the code at leisure to understand its workings, and of course you can import it and apply it to fresh data in a Python3 program. (R users: see example in Appendix C.)

With RUNSTER, you have data-driven automatic programming at your fingertips!

7. BERRIES.py : Bionically Evolved Regression Rule In Executable Software

This program is only applicable if you select Python as your output language (with py as the value of parameter proglang). If you choose R, the function berries() applies the learned ruleset to a dataframe of examples -- probably the most natural way of using the results of RUNSTER's machine learning for an R user. If you choose Python3, this program lets you perform essentially the same function, applying the rules to a

data file rather than an R dataframe.

It works by loading and compiling the "_rule.py" output. It should give exactly the same output as running tree.py on the same data, thus it functions as a check. More important, it allows a Python programmer to inspect the Python3 code of a valid method of incorporating a RUNSTER ruleset into Python3, and thus provides a pointer towards doing likewise with his or her programs.

The main listing of berries.py, suffixed with "_prop.txt" is the same as produced by tree.py except that the cases are listed in descending order of the "safeness" rather than by estimated target value. Comparing the two listings can be used as a check.

In addition, another output file, suffixed "_pout.dat", is produced in a format that can be read into R using R's read.delim() function with standard defaults. The first four lines of cardiac_pout.dat are reproduced below to illustrate the format.

```
safeness      cellsize      id     stabcell      predval       trueval       mark
0.9808131199240078  77     31     725    0      478    547    +
0.9367772305899463  77     14     592    0      471    531    -
0.9350636852057871  77     7      705    0      491    656    -
```

This is in the same format as the "_dump.dat" file produced by tree.py. Both can be read into R for checking consistency; and either can be used for graphical plots or further statistical analyses.

8. Concluding Remarks

Splitting a data file into training and test sets, perhaps several times, as done by seed.py, is standard practice in machine-learning projects during the exploratory phase. However, towards the end of such a project, once you have confidence that the system is able to produce reliable rules, it is advisable to use the entire dataset, not just a random sample, to generate rules for final export. A ruleset based on a larger training sample is likely to be more accurate than one based on a random subset. Therefore a final run using SEED with a datfrac of 1.0 before using ROOT and PEAR to export the learned ruleset for application "in the field" is also standard practice. (Following this advice may not be practicable with huge data sets because a ROOT run may take too long, but it is valid in principle.)

It is also fair to point out that, like most machine-learning systems, RUNSTER has an Achilles Heel. This is the "outlier" problem, which afflicts all model-based predictions, including that by humans, though models generated by computational learning are particularly vulnerable in this respect.

In fact, this concept of "outliers" still exercises the finest statistical thinkers. When a trained rule, ruleset or function is applied to instances from completely outside its training sample, i.e. outside what logicians refer to as the "universe of discourse", it will still give an answer. This sort of problem also rears its ugly head when, for instance, a regression rule trained to predict human plasma volumes is given instances from pigs or cattle. It may be possible to measure corresponding input attributes, but it is unlikely that the same relationship will hold between them and the dependent variable. Stated baldly, it may seem obvious that veterinary data shouldn't be presented to a rule derived from human training cases, but the computer has no knowledge of semantics, and frequently we apply rules blindly to data over whose collection we have no control.

The "safeness" index in TREE and BERRIES is an attempt to give a clue when this might be happening, but my initial experiments suggest that it is rather a weak heuristic. Like most practical learning systems RUNSTER tends to economize on the number of attributes employed, but to be able to raise some sort of "red flag" and say, in effect, "this example looks weird" would demand access to many variables so that their expected

interrelationships could be checked for anomalies.

A completely rigorous general solution to this problem is unattainable. There can be no context-free definition of what constitutes an outlier. I am working on alternative heuristics designed to do better with typical real-life data sets than RUNSTER's present "safeness" measure. It isn't a trivial task, but if I happen upon a better method, I intend to retrofit it to BEAGLE and RUNSTER.

Another limitation that should be mentioned is that RUNSTER in its present form is not suited to the kind of huge data sets that go by the name "big data", with tens of millions of instances measured on thousands of variables. Although Python3 is fast as interpreters go, analyzing such enormous data sets in RUNSTER on a desktop or laptop computer would take an unrealistic amount of time. RUNSTER is more at home with data sets of less than 100,000 data points (number of rows multiplied by number of columns). If you do have a very large dataset, and want to apply RUNSTER to it, using SEED with a small value of datafrac for the training subset is probably the best practical approach. At least that way you'll get some rules to try on the (presumably much larger) test sample.

Thus RUNSTER remains a work in progress. Feedback from users with error reports or suggestions for enhancements will be appreciated. I anticipate that this will not be the last version ever released, and hope to have time to improve the system in various ways over the coming months and years.

Meanwhile, may the Muses of Induction smile on your efforts!

Acknowledgements

Thanks to Phoenix Lam for teaching me about the Windows snipping tool, among other things; to James McDermott for encouraging me to re-visit the world of evolutionary computing; to Dean McKenzie for asking me from time to time when BEAGLE & RUNSTER would be runnable; to the UCI dataset repository for several example datasets.

Thanks also to you for reading this far. (:-)

References

Afifi, A.A. & Azen, S.P. (1979). *Statistical Analysis: a Computer Oriented Approach*, second edition. New York: Academic Press.

Anderson, E. (1935). "The irises of the Gaspé Peninsula". *Bulletin of the American Iris Society* 59, 2–5. http://en.wikipedia.org/wiki/Iris_flower_data_set

Andrews, D.F. & Herzberg, A.M. (1985). *Data: a Collection of Problems from many Fields for the Student and Research Worker*. New York: Springer.

Breiman, L., Friedman, J.H., Olshen, R.A. & Stone, C.J. (1984). *Classification and Regression Trees*. Monterey, California: Wadsworth.

Ethell, J.L. (1999). *World War II Aircraft.* Glasgow: HarperCollins.

Evett, I.W. & Spiehler, E.J. (1988). Rule induction in forensic science. In: Duffin, P.H. *Knowledge Based Systems in Administration*, 152-160. New York: Halsted Press. https://archive.ics.uci.edu/ml/datasets/Glass+Identification

Flury, B. & Riedwyl, H. (1988). *Multivariate Statistics: a Practical Approach*. London: Chapman & Hall.

Forsyth, R.S. (1981). BEAGLE -- a Darwinian approach to pattern recognition. *Kybernetes*, 10, 159-166. http://www.richardsandesforsyth.net/pubs/beagle81.pdf

Forsyth, R.S. & Rada, R. (1986). *Machine Learning: Applications in Expert Systems and Information Retrieval*. Chichester: Ellis Horwood.

Gorman, R. P., and Sejnowski, T. J. (1988). Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1, 75-89.

Kinnear, K.E. (1994) ed. Advances in Genetic Programming. *MIT Press*. (Volume 1.)

Manly, B.F.J. (1994). *Multivariate Statistical Methods: a Primer*. London: Chapman & Hall.

Moore, P. (1992). *Stars and Planets*. London: Chancellor Press.

R Core Team (2013). R: A language and environment for statistical computing. *R Foundation for statistical Computing*, Vienna, Austria. http://www.R-project.org/.

Samuel, A. (1967). Some studies of machine learning using the game of checkers. *IBM Journal of Research & Development,* 11(6), 601-617.

Selfridge, O.G. (1959). Pandemonium: a paradigm for learning. *Proceedings of Symposium held at The National Physical Laboratory*, November 1958, 513-526, London: HMSO.

Upton, G. & Cook, I. (2006). *Oxford Dictionary of Statistics*, second edition. Oxford: Oxford Univ. Press.

Appendix A : Parameter Files

The table below gives information about RUNSTER parameters with which users can choose various option settings. The letters under each parameter name indicate which programs in the suite take notice of the parameter (SRTP for seed.py, root.py, tree.py & pear.py). For example, SR-- would mean that SEED & ROOT take note of the parameter but the other programs ignore it.

Parameter files are plain text files, such as created by text-editors like Notepad or Notepad++, with each line setting a single parameter value. The parameter name comes first at the front of the line, followed by whitespace, followed by the parameter value. The order shown here is alphabetical, but ordering in a parameter file doesn't matter. Since programs only read values for the parameters that affect them, you should be able to make a single parameter file to control a complete run-through of the RUNSTER suite.

Parameter	Default value	Function
brevity -R--	1	If brevity is set to 1, the size of a ruleset will be used as the second element in its fitness score (negated, since ROOT maximizes) and thus function as a tie-breaker, with shorter rules favoured. If it is zero, the size of a ruleset will not affect its quality score.
comment ----	[None]	This (or in fact any unrecognized parameter name, e.g. "##") can be used to insert reminders about what the file is meant to do.
datfrac S---	0.61803398875	This specifies the fraction of the cases (rounded to the nearest whole number) in the full dataset (see maindat) that will be copied into the training datafile (see traindat) by SEED. The remaining cases will go into the test datafile (see testdat). Allocation of individual cases to each file is (pseudo-)random (see randseed).
dumpfile -RT-	[None]	ROOT creates a _"dump.txt" file and TREE creates a "_dump.dat" file. The former is intended mainly for program checking. The latter writes each decision in form that can be read into R.
idfield -RT-	[None]	ROOT & TREE use values from this named variable to identify each row in their output listings (see listfile). If none is given, the input row number is used as an identifier.
jobname SRTP	runster	This gives the job a name. Any string of alphanumeric characters can be the value. It isn't necessary but it is recommended, as the jobname will be used as a prefix to the program's output files, so it can be seen that they form a related group.
listfile -RTP	[Various]	This is a file specification for the main, human-readable, output listing of ROOT, TREE or PEAR. It is simplest not to specify a file, in which case the listing file will be named from the jobname (see above) with "_list.txt", "_test.txt" or "_pear.txt" appended, respectively, and placed in the outpath folder (see below).
m1 -R--	0.5	This number (from 0 to 1) specifies the probability that after a mating/crossover operation in ROOT a mutation will be performed on the resultant offspring.
m2 -R--	0.25	This number (from 0 to 1) specifies the probability, in ROOT, that a mutation will be performed on an existing item in the population of rulesets during each pass round the main evolutionary loop. In effect, half m2 specifies the proportion of new rulesets to be generated by 'asexual' reproduction.
maindat S---	[None]	This should be the full file specification of a file where the input data is stored (in tab-delimited form with a header line naming the columns, and each row describing a single instance).

ntrials -R--	131072	The total number of new structures to be generated during all ROOT's evolutionary trials. Minimum 256, maximum 1048576.
outpath SRTP	..\op\	This specifies the folder (directory) where the program will place its output. Default is the \op\ subfolder of the parent of the folder from which the program is executed.
popsize -R--	233	This specifies the number of quasi-organisms (rulesets) in the population being optimized by ROOT. Minimum 8, maximum 2048.
randseed SR--	0	This number is used to initialize Python's pseudo-random number generator in SEED and ROOT -- except that if it is 0 or 1 or negative the generator will be seeded from the system's clock, i.e. haphazardly, so each run will usually produce slightly different results. To have deterministic results, pick a number from 2 to 999999999, preferably a prime.
progfile ---P	[Various]	This specifies the file on which the export version of the input ruleset (see rulefile) will be written. If none is given and r is the programming language (see proglang) it will be jobname (see above) with "_prog.r" appended; if none is given and py is the programming language it will jobname with "_prog.py" appended.
proglang ---P	r	This chooses the programming language for export of a RUNSTER ruleset: r selects the R language; py is for Python3.
rulefile -RTP	[None] => jobname with "_rule.txt" appended	This specifies the file into which ROOT will write its highest-scoring ruleset at the end of its optimization process, and from which TREE and PEAR will read the ruleset to be used.
rulemode -R--	standard	When rulemode is tabular, the truth status of the rules in a ruleset (e.g. 110, meaning True,True,False) is used as index into a signature table to accumulate frequencies, sums and deviations in ROOT and to use those statistics in ROOT, TREE, PEAR and BERRIES. The only recognized alternative mode is standard, in which case there will be a single regression rule which will be evaluated directly to give a numeric estimate of the target value.
skipvars -R--	[None]	The value for skipvars should be a list of column/variable names separated by commas, e.g. skipvars bombing,country which will tell ROOT not to use these variables in any rules generated. There is no need to forbid variables used in the target expression (see targval) as they will be automatically excluded from the generation process.
targval -RTP	[None]	This parameter is used to define the target values. It can be a variable name (e.g. "wingspan") or a more complex expression. It must yield numeric values. An example can be seen in Appendix C.
testdat S-T-	[None] => jobname with "_dat2.dat" appended	This specifies a file into which SEED will write a test subset of the full data (see maindat) and which TREE & BERRIES will use by default for input.
traindat SR-P	[None] => jobname with "_dat1.dat"	This specifies a file into which SEED will write a training subset of the full data (see maindat) and which ROOT & PEAR will use by default for input.
trigfunx -R--	0	RUNSTER's rule language includes two trigonometric functions $Cosi and $Sine, but in the context of machine learning these can be rather dangerous. This parameter defaults to zero, which means

		that they won't be used in the evolutionary rule-generation process. Only set it to 1 if you're sure that you have (usually temporal) data where taking sines or cosines makes sense. Note that you can use trigonometric functions in target expressions (see targval) even when this parameter is zero: they are only excluded from the rule-generation loop.

BERRIES uses the same parameters as TREE except that it reads from progfile (written by PEAR) instead of rulefile (written by ROOT).

Appendix B: RUNSTER's rule language

RUNSTER's rule language is the same as BEAGLE's. It is modelled on that found in mainstream procedural programming languages, such as C, Fortran, Pascal and Python. It allows the user, or the computer, to frame logical and mathematical expressions. There are 20 recognized operators, as follows.

MONADIC (all of which are written with a dollar sign '$' as prefix)

- $! Logical negation (NOT)
- $~ Arithmetic negation (unary minus)
- $Fabs Floating-point absolute value (ignoring sign)
- $Root 'Safe' square root:
  $Root x is sqrt(abs(x)); result negated if x is less than zero
- $Slog 'Safe' natural logarithm:
  $Slog x is -ln(1+abs(x)) if x is negative, otherwise ln(1+x)
- $Tanh Hyperbolic tangent (a 'squashing' function mapping to the range -1 to +1)
- $Cosi Cosine
- $Sine Sine

[Note: if parameter trigfunx is 0 (the default value) $Cosi & $Sine will be disabled during ROOT's evolutionary cycle, though they can still be used in a target expression (see Appendix A). To enable these two trigonometric functions during rule generation, set parameter trigfunx to 1.]

ARITHMETIC

- + Addition
- - Subtraction
- * Multiplication
- ^ Maximum
- \ Minimum

BOOLEAN

- & Logical conjunction (AND)
- | Logical disjunction (inclusive OR)
- ; Logical exclusive OR

COMPARATIVE

- = Equality (EQ)
- < Inferiority (LT)
- > Superiority (GT)

STRINGY

- ? Only used as in (variable ? `text`) yielding 1 if variable contains substring 'text', else 0: quoted string constants enclosed by grave/backtick character (code point 96).

[Note that division is not provided. To get round this, alter X/2 to X*0.5, A/B < C to A < B*C and so on. Note also that the minus sign must be followed by one or more spaces when it means subtraction; if it isn't the system will presume that next character begins a negative number, such as -355.]

Expressions consist of variable names (such as wingspan) and numeric constants (such as 0.75) linked by operators. The only precedence recognized is that monadic (unary) operators have higher precedence than dyadic operators. Thus

($! x - 4)

performs the logical negation of x before subtracting 4. If you want the subtraction first,

```
$! (x - 4)
```

would be the correct form. This means that ordering among dyadic operations must be made explicit with parentheses. You will rarely have to write a complicated BEAGLE/RUNSTER expression, but you may have to interpret some.

This notation lets you intermix logical and numerical values. If an operation needs a logical value but is given a numeric one, it converts as follows.

```
x > 0     =>    True  (1)
x <= 0    =>    False  (0)
```

If it wants a numeric value but gets a logical one it uses the following conversions.

```
True      =>    1.0
False     =>    0.0
```

All computations are performed in floating-point double-precision arithmetic.

Appendix C: Case Study Using Aircraft Data :

C.1 Preliminaries

As an illustration, this Appendix describes a run-through of the RUNSTER suite on one of the example datasets provided with the distribution, a data file describing 103 World-War II military aeroplanes. The main source of this data was *Collins Jane's WWII Aircraft* (Ethell, 1999). The listing below gives the 16 column names with brief explanations of each.

```
name          name of aircraft
engines       number of engines
fuselage      length in metres
headroom      height in metres
wingspan      in metres
kg            empty weight in kg
kgladen       laden weight in kg
loaddiff      difference between loaded & empty (kg)
topspeed      maximum speed in km/hr
ceiling       maximum altitude in metres
maxrange      maximum range in km
cannons       number of cannon fitted
fighting      whether used as a fighter (0/1)
bombing       whether used as a bomber (0/1)
carrying      whether used as a transport plane (0/1)
country       nation of origin
```

N.B. Some aircraft were used in multiple roles, including roles such as reconnaissance which aren't noted here, and some changed role over time, e.g. fighter to (light) bomber. I have tried to indicate (with fighting, bombing & carrying) roles ascribed to each aircraft for the model whose specifications are recorded. Experts might disagree. (This still leaves a number of definite fighter-bombers.)

This data can be found in \beagling\datasets\aircraft.dat. The header line and the first and last data lines of this file are reproduced below to give an idea of its format. (These three lines appear as more than three owing to the restricted margins of this document.)

```
name   engines       fuselage      headroom      wingspan      kg     kgladen       loaddiff
       topspeed      ceiling       maxrange      cannons       fighting      bombing
       carrying      country
Boomerang_CA_12      1       7.78   3.51   11.06  2474   3450   976    476    8845   1496   2
       1      0      0       oz
[....]
Vought_F4U1D  1      10.17  4.6    12.47  3947   5465   1518   684    11285  1633   0      1
       0      0      us
```

The first thing to do in such an exercise is to choose a target. Often this is obvious, but with this data there are several possibilities. For example, we might train the system to predict the wingspan or loaddiff (carrying capacity) of these aircraft from other attribute values. To do that would require choosing wingspan or loaddiff as the value for the targval parameter. However, in this instance I have decided to exemplify the fact that the target should be an expression rather than a just a variable name. In fact, I have chosen an idiosyncratic target expression

(bombing - fighting)

which only gives three distinct values, since both variables are binary (0/1). In effect, this expression splits the aircraft into three groups: (-1) fighters that aren't bombers, (0) fighter-bombers or other types, such as transport aircraft, and (+1) bombers that aren't fighters. It could be used as a three-class classification in BEAGLE (and that would be an interesting comparison) but there is an implied ordering, so the numeric difference between predicted and target values is a legitimately treated as a number. In other words,

predicting -1 for a +1 case is a worse mistake than predicting 0. Hence a regression rule does make sense. And since this is on the borderline between classification and regression (also for illustrative purposes) I will employ tabular mode.

The parameter file listed below can be found as airbomb.txt in the parapath folder of the system as distributed. For ease of reference, line numbers have been inserted at the left of each line, though these are not part of the actual file, which can be found at c:\beagling\parapath\airbomb.txt.

```
1       ##  symbolic regression initial testing :
2       jobname  airbomb
3       maindat  c:\beagling\datasets\aircraft.dat
4       datfrac  0.75
5       outpath  c:\beagling\takeoff\
6       targval  bombing - fighting
7       rulemode  tabular
8       brevity  1
9       idfield  name
10      randseed  1999
11      proglang  r
12      ##ntrials  262144
```

Here the target appears on line 6. This target expression (bombing - fighting) will be +1 for what might be called "pure" bombers, aeroplanes only used as bombers. Likewise it will only yield -1 for what might be termed "pure" fighters. Since both variables in the target expression, bombing & fighting, will be excluded from HERB's generation process, it is not necessary to have a line such as

skipvars bombing,fighting

to prevent the system from generating circular rules.

Line 1 is merely a comment.

Line 2 gives a jobname, airbomb, to this task. This will be used as the core in the names of the various files that the system writes: it is RUNSTER's normal way of showing which files are related.

Line 3 specifies the data file containing the examples to be investigated.

Line 4 instructs the SEED program that 75% of the instances in the data should be placed in the training file, which will be called airbomb_dat1.dat since no traindat parameter has been specified, leaving 25% to go into the test file (called airbomb_dat2.dat, since no testdat parameter has been specified either).

Line 5 specifies an output folder (c:\beagling\takeoff\). This ensures that all the output files will be held in the same place, which is usually a good idea.

Line 6 is the target expression, discussed above.

Line 7 sets the rule mode to "tabular". This is a throwback to the Pascal PC/BEAGLE of 1985. It isn't normally the best method, but in a case like the present, where the target is quasi-categorical, it is worth trying. In this mode root.py will generate a small number of rules which will be interpreted as giving logical values (thus values greater than zero will be treated as 1 and zero or less as 0) and used to index a "signature table" (Samuel, 1967). Each combination of rule values, e.g. 01, will select a row of that table. During the generation process, the mean and standard deviation of the target values of cases with that combination will be accumulated in the row indicated. When used in predictive fashion, the appropriate mean will be selected as the estimated target value.

Line 8 requests that (negated) average rule length will be included as the second part of a rule's fitness score, i.e. as a tiebreaker. The first element of a rule's fitness score is the mean absolute deviation between predicted and true target values.

Line 9 provides a data field that will be used to label each output case in the listings of root.py and tree.py (and berries.py if proglang py is selected).

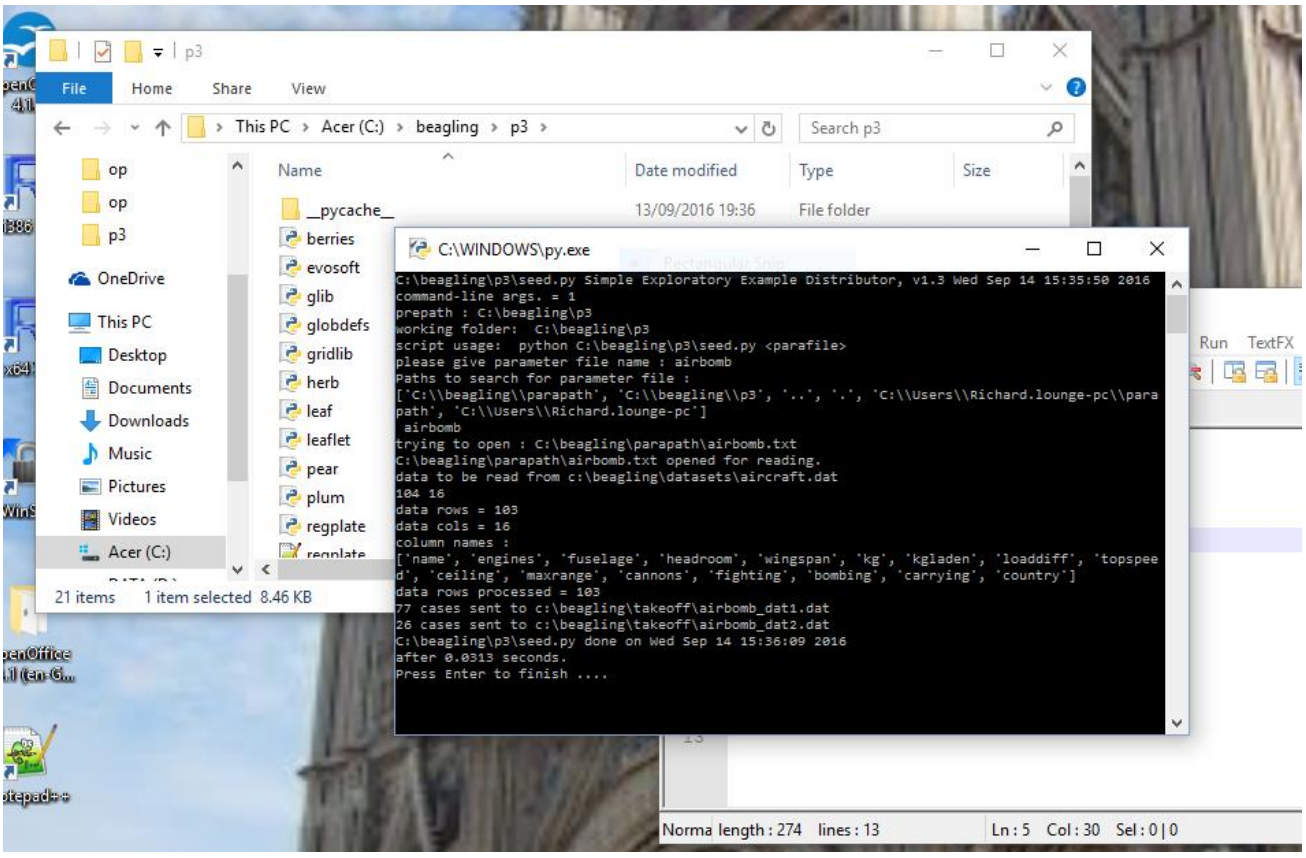
Line 10 will initialize Python's pseudo-random number generator to a fixed value, here 1999, to ensure that this run is repeatable.

Line 11 selects the export programming language of the ruleset to be R. This parameter is used by pear.py.

Line 12 is commented out. It requested an evolutionary run in root.py twice as long as the default value, but it didn't seem to lead to an improvement so I commented it out. I left it in as a reminder that comment lines in parameter files can be used to switch among option choices.

C.2 Running SEED

It is typical, as in this case, to start a RUNSTER run, by executing the seed.py program. If you double-click on seed.py in the \beagling\p3\ folder and type "airbomb" to specify the parameter file, you should see something rather like the screen shot below.



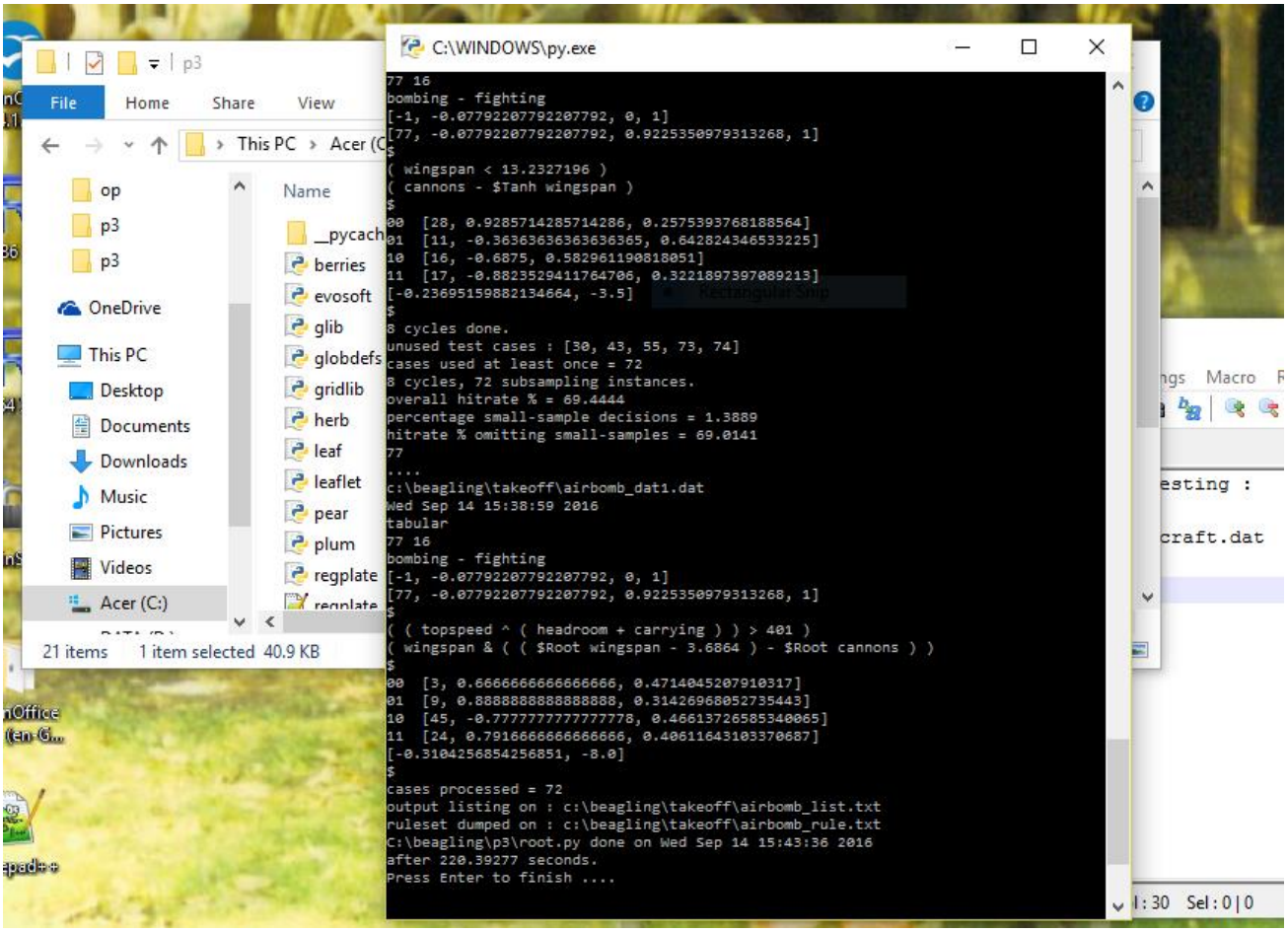
This reads the 103 cases in c:\beagling\datasets\aircraft.dat and puts 77 cases into airbomb_dat1.dat and 26 into airbomb_dat2.dat, both in the folder c:\beagling\takeoff\.

The header line and the first and last 2 data lines of airbomb_dat2.dat are listed below.

```
name    engines         fuselage        headroom        wingspan        kg      kgladen         loaddiff
        topspeed        ceiling         maxrange        cannons         fighting        bombing
        carrying        country
Fiat_CR42_Falco         1       8.25    3.35    9.7     1720    2300    580     430     10500   775     0
        1       0       0       italy
Fiat_G50_Freccia        1       7.79    2.9     10.97   1900    2706    806     471     10000   1000    0
        1       0       0       italy
[....]
NorthAmerican_B25J      2       16.13   4.8     20.6    9579    15876   6297    443     7320    2413    0
        0       1       0       us
Northrop_P61B 2         15.12   4.47    20.12   10896   17252   6356    589     10065   4505    4       1
        1       0       us
```

C.3 Running ROOT

The screen shot below shows the last several lines of output, under Windows 10, just after running root.py by clicking on it in the \beagling\p3\ directory and typing "airbomb" in reply to the parameter file request.



The best rule produced in this run is shown below.

```
training data : c:\beagling\takeoff\airbomb_dat1.dat
creation date : Wed Sep 14 15:43:36 2016
rule mode : tabular
77 16
bombing - fighting
[-1, -0.07792207792207792, 0, 1]
[77, -0.07792207792207792, 0.9225350979313268, 1]
$
```

```
( ( topspeed ^ ( headroom + carrying ) ) > 401 )
( wingspan & ( ( $Root wingspan - 3.6864 ) - $Root cannons ) )
$
00  [3, 0.6666666666666666, 0.4714045207910317]
01  [9, 0.8888888888888888, 0.31426968052735443]
10  [45, -0.7777777777777778, 0.46613726585340065]
11  [24, 0.7916666666666666, 0.40611643103370687]
[-0.3104256854256851, -8.0]
$
```

This contains a pair of independent rules, the first of which

```
( ( topspeed ^ ( headroom + carrying ) ) > 401 )
```

relates three variables, although, given knowledge of the data, two are superfluous. It will have the same effect on this data as

```
( topspeed > 401 )
```

because the inner subexpression (topspeed ^ (headroom + carrying)) is a maximum operation: the lowest topspeed in the aircraft dataset is 224 (km/hr) while the greatest headroom is 10.03, even adding 1 for carrying, topspeed will always exceed 11.03, so it will always be topspeed that is compared to the constant 401. This shows that the pressure towards brevity doesn't inevitably lead to a minimal rule.

The second expression

```
( wingspan & ( ( $Root wingspan - 3.6864 ) - $Root cannons ) )
```

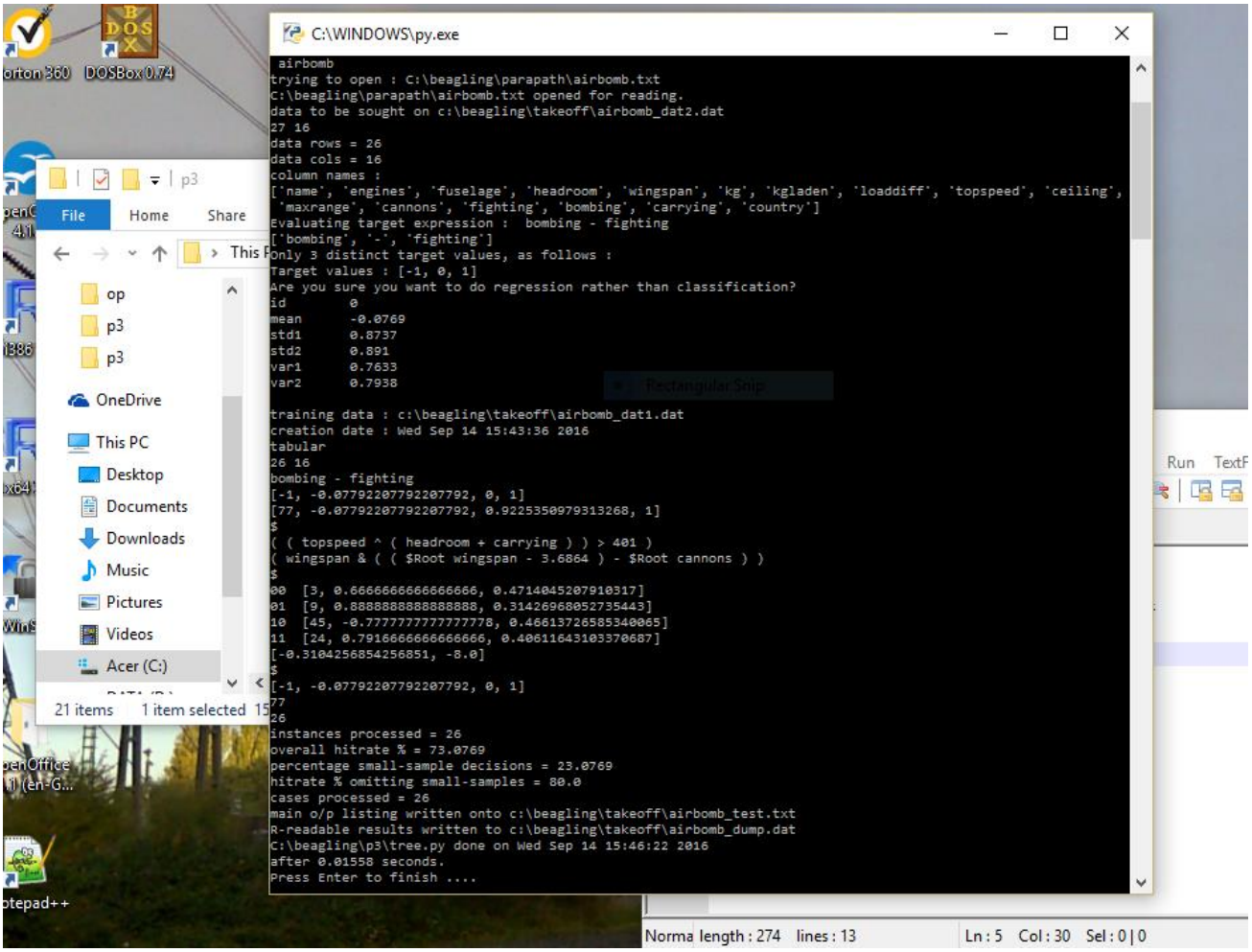
also contains a redundancy, since wingspan always exceeds zero, thus the logical '&' will always have True on its left, so will only depend on the right-hand subexpression. This latter part could be rewritten with the minus sign replaced by greater-than ('>') since it will be forced to 0 if the subtraction result is zero or less, 1 otherwise.

Between them, these rules will select one of four target estimates, ranging from -0.7777777777777778 if the rule-combination yields 10, to 0.7916666666666666 if both rules are true (11).

The last line before the final dollar sign ('$') of this rule listing indicates that the mean absolute deviation between values derived from using this ruleset and true target values were 0.3104 to four decimal places.

C.4 Running TREE

Running tree.py with the same parameter file should produce on-screen output resembling the screen shot below.



TREE applies the ruleset created by root.py (airbomb_rule.txt) to the 26 holdout cases written by SEED into the test file (airbomb_dat2.dat). Its main output file (aircraft_test.txt in this example) is listed below.

```
dateline   Wed Sep 14 15:46:12 2016
progname   C:\beagling\p3\tree.py
id         C:\beagling\parapath\airbomb.txt
testdat    c:\beagling\takeoff\airbomb_dat2.dat
targval    bombing - fighting

====holdout trial :
```

rank	safeness	case	name	pred:true	cellsize	abdsiff	diffsqrd
1	0.75	21	Douglas_Dakota_C47A	0.8889 - 0	9	0.89	0.79
2	0.75	16	Fairey_Swordfish_I	0.8889 + 1	9	0.11	0.01
3	0.75	15	Fairey_Battle_II	0.8889 + 1	9	0.11	0.01
4	0.75	9	FockeWulf_Fw202C_Con	0.8889 + 1	9	0.11	0.01
5	0.69	24	NorthAmerican_B25J	0.7917 + 1	24	0.21	0.04
6	0.69	23	Lockheed_PV1	0.7917 + 1	24	0.21	0.04
7	0.69	22	Grumman_TBF1	0.7917 + 1	24	0.21	0.04
8	0.69	20	Consolidated_B24J	0.7917 + 1	24	0.21	0.04
9	0.69	19	Boeing_B17G	0.7917 + 1	24	0.21	0.04
10	0.69	14	Petlyakov_Pe2	0.7917 - 0	24	0.79	0.63
11	0.69	6	Mitsubishi_Ki46	0.7917 - 0	24	0.79	0.63
12	0.69	5	Mitsubishi_Ki21	0.7917 + 1	24	0.21	0.04
13	0.66	12	Ilyushin_II2_Shturmo	0.6667 - 0	3	0.67	0.44
14	0.66	11	PZL_P11	0.6667 - -1	3	1.67	2.78
15	0.66	25	Northrop_P61B	-0.7778 - 0	45	0.78	0.60
16	0.66	18	Supermarine_Spitfire	-0.7778 + -1	45	0.22	0.05
17	0.66	17	Hawker_Tempest_V	-0.7778 + -1	45	0.22	0.05
18	0.66	13	Mikoyan_Gurevich_MiG	-0.7778 + -1	45	0.22	0.05
19	0.66	10	Heinkel_He219	-0.7778 + -1	45	0.22	0.05
20	0.66	8	Nakajima_Ki84	-0.7778 - 0	45	0.78	0.60
21	0.66	7	Nakajima_Ki44	-0.7778 + -1	45	0.22	0.05

```
  22      0.66    4  Mitsubishi_J2M           -0.7778 + -1            45         0.22         0.05
  23      0.66    3  Kawanishi_N1K1           -0.7778 + -1            45         0.22         0.05
  24      0.66    2  Reggiane_Re2000_Falc     -0.7778 + -1            45         0.22         0.05
  25      0.66    1  Fiat_G50_Freccia         -0.7778 + -1            45         0.22         0.05
  26      0.66    0  Fiat_CR42_Falco          -0.7778 + -1            45         0.22         0.05
-+++++++++-+---++++-++++++

'success' percentage = 73.08
pearson correlation between predicted & true vals = 0.8187
spearman rank-correlation between predicted & true vals = 0.8382

mean abs.error = 0.391
mean error ^ 2 = 0.2795
correlation between safeness & abs.error (negative better) = -0.152

Resultant rule from all training cases :
training data : c:\beagling\takeoff\airbomb_dat1.dat
creation date : Wed Sep 14 15:43:36 2016
tabular
77 16
bombing - fighting
[-1, -0.07792207792207792, 0, 1]
[77, -0.07792207792207792, 0.9225350979313268, 1]
$
( ( topspeed ^ ( headroom + carrying ) ) > 401 )
( wingspan & ( ( $Root wingspan - 3.6864 ) - $Root cannons ) )
$
00  [3, 0.6666666666666666, 0.4714045207910317]
01  [9, 0.8888888888888888, 0.31426968052735443]
10  [45, -0.7777777777777778, 0.46613726585340065]
11  [24, 0.7916666666666666, 0.40611643103370687]
[-0.3104256854256851, -8.0]
$

[Parameter settings omitted to save space ....]
```

This output is in a similar format to that of the airbomb_list.txt file produced by root.py, so they can easily be compared. In tabular mode the 'safeness' of a decision is calculated as (SD / (SD + sd)) where SD is the overall training-set standard deviation of the target values (0.922535 in this case) and sd is the standard deviation of the examples in whichever signature-table cell is being used.

On these genuinely unseen cases tree.py gets 19 out of 26 correct (73.08% 'success') which is slightly better than the success rate projected from ROOT's subsampling. One of its mistakes was assessing the Douglas Dakota, a workhorse transport aircraft, as a bomber. It certainly wasn't a fighter, but it wasn't used as a bomber. The worst decision in terms of absolute error was at rank 14, the Polish PLZ_P11. By the time of the German invasion of Poland in 1939, the P11, a monoplane fighter, was obsolete. Nevertheless, during the short campaign it downed more enemy aircraft than the number of P11's lost -- a tribute to the skill of the pilots, some of whom subsequently found their way into the RAF. The P11 could reasonably be regarded as an outlier; or, from another point of view, as a reminder that the training data consists of information from a short period (approximately 1938 to 1946) in a century of rapid developments in aviation. Trying this ruleset on 21st-century aircraft would undoubtedly increase the mean error.

A more practical way to assess the performance of RUNSTER in tabular mode on this data is to compare it with the most similar conventional technique, a regression tree (Breiman et al., 1984). To do this I created a regression tree from the airbomb_dat1.dat dataset using R's rpart() function with standard parameter settings.

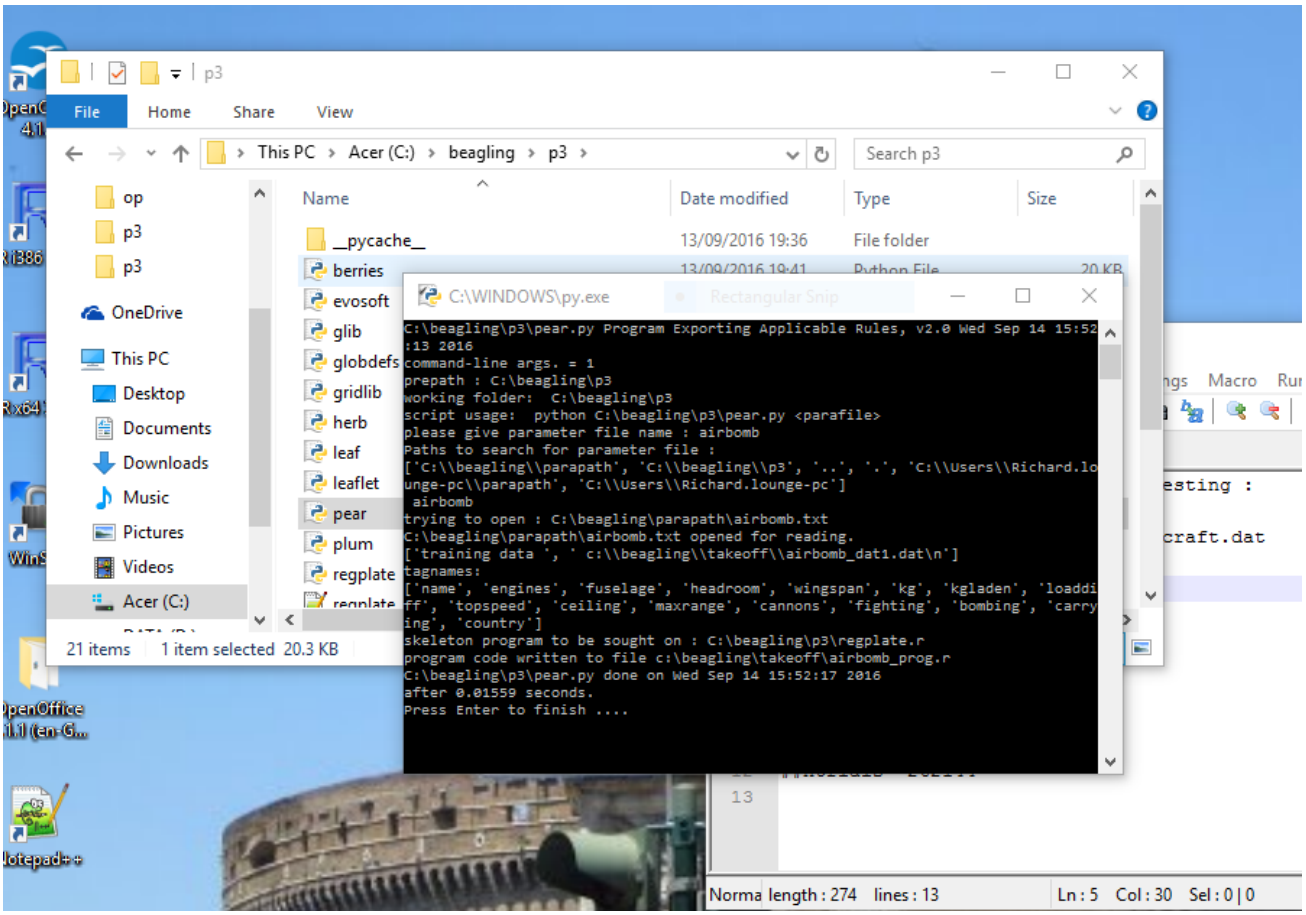
```
> airbtree=rpart((bombing-fighting)~.,data=airbomb1[,2:15])
> airbtree
n= 77

node), split, n, deviance, yval
      * denotes terminal node

 1) root 77 65.532470 -0.07792208
```

```
   2) wingspan< 13.335 36  9.638889 -0.80555560
     4) topspeed>=479.5 29  1.862069 -0.93103450 *
     5) topspeed< 479.5 7  5.428571 -0.28571430 *
   3) wingspan>=13.335 41 20.097560  0.56097560
     6) topspeed>=500.5 13  7.692308 -0.15384620 *
     7) topspeed< 500.5 28  2.678571  0.89285710
      14) ceiling>=8150 8  1.875000  0.62500000 *
      15) ceiling< 8150 20  0.000000  1.00000000 *
> airbtreg=predict(airbtree,newdata=airbomb2)
```

This tree contains five terminal 'leaf' nodes. When applied (using R's predict() function) to the held-out cases in the airbomb_dat2.dat dataset, the predicted values from this tree had a Pearson correlation of 0.8124 with the true values and a rank correlation of 0.8160. Both these correlations are marginally worse than those obtained by tree.py, although the mean absolute deviation achieved by the regression tree (0.3517) is slightly better than that of the RUNSTER ruleset (0.391). Overall, on this data, the two methods are pretty closely matched.

C5. Running PEAR

Running pear.py with this dataset will produce an output file containing an executable translation of ROOT's ruleset (airbomb_rule.txt), in R since the value of proglang in the parameter file is "r". A sample screen shot is shown below.



The program output file (airbomb_prog.r) is listed below.

```
##  Using RUNSTER R template, version of 12/09/2016 :
##  rule written by pear.py ;
##  derived from training data : c:\beagling\takeoff\airbomb_dat1.dat;
```

```
##  generated on creation date : Wed Sep 14 15:43:36 2016;
##  dumped on Wed Sep 14 15:52:17 2016.
##
beag_gold = 5.0 ** 0.5 * 0.5 + 0.5  ##  global

##  helper functions :
beag_bool = function (v) {
       ##  ensures same bool/math treatment as in Beagle :
       return  ((v > 0) + 0)
       }
beag_exor = function (v1,v2) {
       ##  exclusive or, as in Beagle :
       return ((v1>0) != (v2>0))
       }
beag_root = function (v) {
       ##  safe square root :
       if (v >= 0.0)  return (sqrt(v))
       else  return (-sqrt(abs(v)))
       }
beag_slog = function (v) {
       ##  safe natural logarithm :
       if (v < 0)  return (-log(1+abs(v)))
       else  return  (log(1+v))
       }

runster_stabprep = function () {
       ##  sets up fallout table :

       stab = list()
       stab[['00']] = c(3, 0.6666666666666666, 0.4714045207910317)
       stab[['01']] = c(9, 0.8888888888888888, 0.31426968052735443)
       stab[['10']] = c(45, -0.7777777777777778, 0.46613726585340065)
       stab[['11']] = c(24, 0.7916666666666666, 0.40611643103370687)

       ##  unpacks stab lines.

       return (stab)
       }  ##  stabprep ends.


runster_regrule = function (vals,stab) {
       ##  input vals should be a 1-row dataframe with appropriate colnames.
       ##  target : bombing - fighting.
       ##  rule mode is tabular.

       rule = c() ; bins = c('0','1')
       catlist = c(-1, -0.07792207792207792, 0, 1)
       priorvec = c(77, -0.07792207792207792, 0.9225350979313268, 1)
       subrules = 2
       ##  compute rule values :
       rule[1] = (max(vals$topspeed,(vals$headroom + vals$carrying)) > 401)
       rule[2] = (beag_bool(vals$wingspan) & beag_bool((beag_root(vals$wingspan) - 3.6864)
- beag_root(vals$cannons)))

       p = 0 ; b = c()
       while (p < subrules) {
              p = p + 1  ##  early-r, late-py
              v = (rule[p]>0)  ##  omit if standard
              b = c(b,bins[v+1])   ##  omit if standard
              }
       ##  standard mode :
       smalldif = priorvec[4]  ##  should work for both
       ##  tabular mode :
       b = paste(b,collapse='')  ##  omit if standard
       cellvals = stab[[b]]  ##  omit if standard
       cellsize = cellvals[1]  ##  omit if standard
       predval = cellvals[2]  ##  omit if standard
       standev = cellvals[3]  ##  omit if standard
```

```
       return
(list(cellcode=b,predval=predval,standev=standev,smalldif=smalldif,cellsize=cellsize))
       }
##  regression rule ends.


berries = function (datframe) {
       ##  Bionically Evolved Regression Rule In Executable Software :

       stab = runster_stabprep()
       rows = dim(datframe)[1]
       options(stringsAsFactors=FALSE)
       if (rows < 1)  return (NULL)
       for (r in 1:rows) {
              regvals = runster_regrule(datframe[r,],stab)
              if (r == 1) outframe = data.frame(regvals)
              else {
                     outframe = rbind(outframe,regvals)
                     }
              ##  outframe[j,] = c(datframe[r,],unlist(regvals))
              }
       temp = cbind(datframe,outframe)
       options(stringsAsFactors=default.stringsAsFactors())
       return (temp)
       }
##  returns dataframe with predicted values added to each record.
##  trueval may not be known, so no 'success' mark computed.

##  ending.
```

C6. Running BERRIES

To make use of this software within the R environment, it is necessary to load the R source code (top left-hand menu) and then apply the function berries() to a data frame containing at least the attributes included by name in the ruleset. The four output columns other than predval are provided as ingredients for various ways of assessing the likely reliability of the predicted value. For example,

airbtemp=berries(airbomb2)

which would create a new data frame with additional columns

cellcode	indicating which cell in the table was used to generate predval;
predval	the predicted target value;
standev	the standard deviation in the table cell used;
smalldiff	actually the MADM of the training-data target values;
cellsize	the number of cases in the table cell used.

Since the target value is an expression, not just an attribute, comparing predval with the true value requires generating another column, with a command such as

airbtemp$trueval = (airbtemp$bombing - airbtemp$fighting)

As far as predicted values are concerned, this should be the same as the output of tree.py (e.g. airbomb_dump.dat) which can be read into R for checking purposes. The berries() function does not attempt to compute a true value itself, since it is envisaged that it will be used on genuinely unknown cases, where that value can't be supplied.

Appendix D: Sample Datasets Provided

These are readable into R using read.delim() with default settings, i.e. tab-delimited with header line; meant to be suitable for classification &/or regression testing. The .dat files contain data; .txt files with same name give details.

Aircraft [103, 16]
Data about 103 World-War-II military aeroplanes, as from Collins/Jane's WWII Aircraft (Ethell, 1999).

Banknote [206, 8]
Data on 206 forged versus genuine Swiss banknotes (Flury & Riedwyl, 1988).

Cardiac [113, 20]
Sample data from Afifi & Azen (1979) on heart-attack patients in L.A.

Digidat [1024, 13]
Recreation of faulty light-emitting diode display data, as in example by Breiman et al. (1984).

Dogs [77, 12]
Mandible measurements of living & prehistoric Thai canines, as from Manly (1994).

Echo [201, 61]
Gorman & Sejnowski's (1988) Sonar dataset from UCI ML repository.
https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)

Elements [104, 14]
Information on chemical elements (Wikipedia periodic table).

Glasses [214, 10]
Evett & Spiehler's (1988) forensic glass identification example data.
https://archive.ics.uci.edu/ml/datasets/Glass+Identification

Iris [150, 5]
Fisher's (1936) Iris data. (Originator: Anderson, E. (1935). The Irises of the Gaspé peninsula.)
http://en.wikipedia.org/wiki/Iris_flower_data_set

Natflags [200, 30]
Information about nations & their flags.

Planets [11, 10]
Planetary data, with Ceres, Eris & Pluto promoted, from Moore (1992) combined with some information from Wikipedia. (See also sats, below.)

Rand [256, 16]
Pure random data (as null case) to test overfitting avoidance.

Roos [101, 20]
Kangaroo skull measurements (Andrews & Herzberg, 1985).

Sats [33, 6]
Data on 33 of the largest satellites of the four largest planets in the solar system, partly from Moore

(1992) combined with information from Wikipedia. (See also planets, above.)

Seed [210, 8]

Wheat seed data from Poland. Three varieties: Kama, Rosa & Canadian.
Seven measurements derived from soft x-rays.
https://archive.ics.uci.edu/ml/datasets/seeds

Vole [86, 8]

Measurements on 2 types of vole, from Flury & Riedwyl (1988).

Wine [178, 14]

Chemical measurements as predictors of type of Italian wine. Source: Forina et al.
https://archive.ics.uci.edu/ml/datasets/Wine

Zoobase [101, 18]

Zoological classification data as from Forsyth (1990).