BEAGLE User Notes
(R.S. Forsyth, August 2016)

Contents

BEAGLE

Biological Evolutionary Algorithm Generating Logical Expressions

is a **rule-finder** system inspired by the Darwinian concept of natural selection. A rule-finder is a program which examines a database of examples and uses machine-learning techniques to create a rule or set of rules for classifying those examples, as well as other examples of the same type. BEAGLE'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 **discriminant function** for classifying cases according to their attributes. It differs from conventional Linear Discriminant Analysis firstly in employing a stochastic procedure to devise the discriminant functions and secondly in using a highly non-linear, and hence much more expressive, description language for its classification rules. (A companion system, RUNSTER, which stands for Regression Using Naturalistic Selection To Evolve Rules, will be released later in 2016, if work goes to plan. This applies essentially the same logic to what statisticians call **regression** as BEAGLE does to **classification**.)

1. Why I Wrote this Software

Broadly speaking, there have been three 'incarnations' of BEAGLE. I devised the first, in 1980, at the Polytechnic of North London (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 the second 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 but less well as a money-spinner.

Despite its modest commercial success, PC/BEAGLE represented a milestone in the field. Arguably it was the first working example of a genetic-programming system, since it incorporated the four distinctive features of Genetic Programming identified by Kinnear (1994), namely:

1. tree-structured heritable material;
2. variable-length heritable material;
3. syntax-preserving crossover;
4. executable heritable material.

When it was first released as a commercial software package in 1985, it was 15 years ahead of its time. Hardly anyone then knew what Genetic Programming was, even its author. Now, however, PC/BEAGLE is 16 years behind the times. Since Windows 7, it doesn't even run in the Windows command-line window any more. (It will run under DOSBox v0.74, but with a couple of niggly quirks that require work-arounds.)

Thus it is high time for a BEAGLE upgrade, taking advantage of modern hardware and some of the lessons learnt since 1985. Hence the third incarnation -- a comprehensive rewrite of the whole system in a more flexible programming language, Python3. This is freely available to any interested parties at
http://www.richardsandesforsyth.net/software.html
under the GNU public licence. My goal in this reimplementation has been to keep the good points of 1980s BEAGLE while correcting some deficiencies and adding some desirable new features.

Good points of 1980s BEAGLE

- It worked! (Fast enough on MS/DOS personal computers to solve practical problems.)
- It exported what it had learned as executable subprograms in C, Fortran or Pascal generated from example data. (Practical genetic programming.)
- It dealt with numeric targets (tabular regression) as well as logical target expressions (classification).
- It avoided 'bloat' with a prolixity penalty.
- It handled string fields as well as numeric variables.
- It broke beyond the bitstring barrier. (Fortunately I didn't then know of the Schema Theorem or its apparent support for low-cardinality alphabets.)
- It didn't fall into the fitness-proportional selection trap. (Floating approximate median gave quasi-rank-based selection without the expense of sorting.)

Enhancements in 21st-century BEAGLE

- A move from generational to incremental search. (Mammals versus mayflies.)
- Ability to handle multi-class classification problems.
- A more intelligible string-handling technique.
- A slightly richer expression language (though not too abstruse).
- A more principled Implementation of the brevity-bias. (Simplify rules after evolution, not during!)
- Correct Bayesian reasoning!
- Rule export in Python or R.
- Zero cost. (People these days have come to expect high-quality software to be free!)

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 quite 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. 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. 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 BEAGLE 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.	**herb.py** : Heuristic Evolutionary Rule Breeder	This is the main evolutionary learning program. It takes a training file of example cases and a target expression that divides them into categories and repeatedly uses an evolutionary algorithm to generate a ruleset for classifying those examples into the appropriate categories. 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.	**leaf.py** : Likelihood Estimator And Forecaster	This program applies the ruleset written by herb.py to classify 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 a decision is required.
4.	**plum.py** : Procedural Language Utility Module	This program takes a rule file as written by herb.py and translates it into Python3 or R so that it can be used in external software.
5.	**leaflet.py** : Language Export Application For Likelihood Estimation Trials	This program essentially duplicates the function of leaf.py: it uses the Python3 code written by plum.py to classify 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 BEAGLE's internal rule language (as in leaf.py) and when using the derived Python code. (The generated R code contains a function named leaflet() 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.

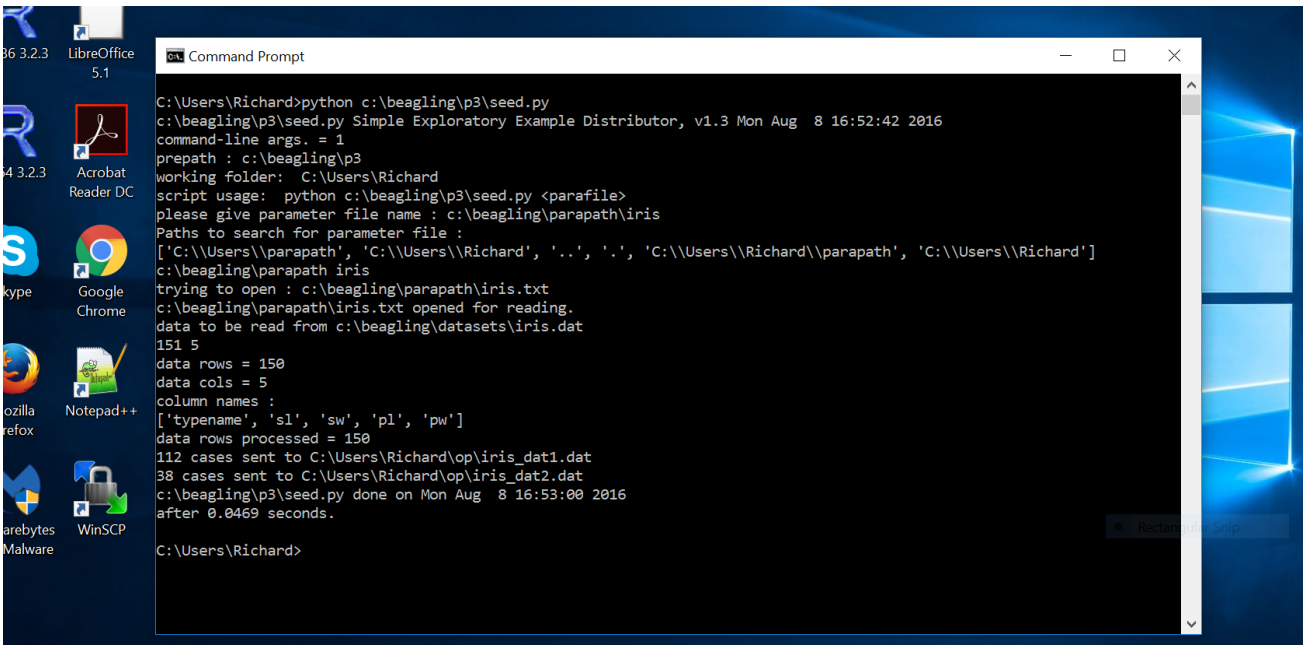
Perhaps the safest, and the mode closest to what is natural in Linux, is to open a command window and run the program from within that window. In Windows 10 that means right-clicking the bottom-left symbol and selecting "Command Prompt" from the menu that pops up. That should bring up an MS/DOS-style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

This will start the SEED program running. It will ask for a parameter file. In this case, since you're running from directory C:\2016\ you'll need to give the full path of the parameter file, e.g.

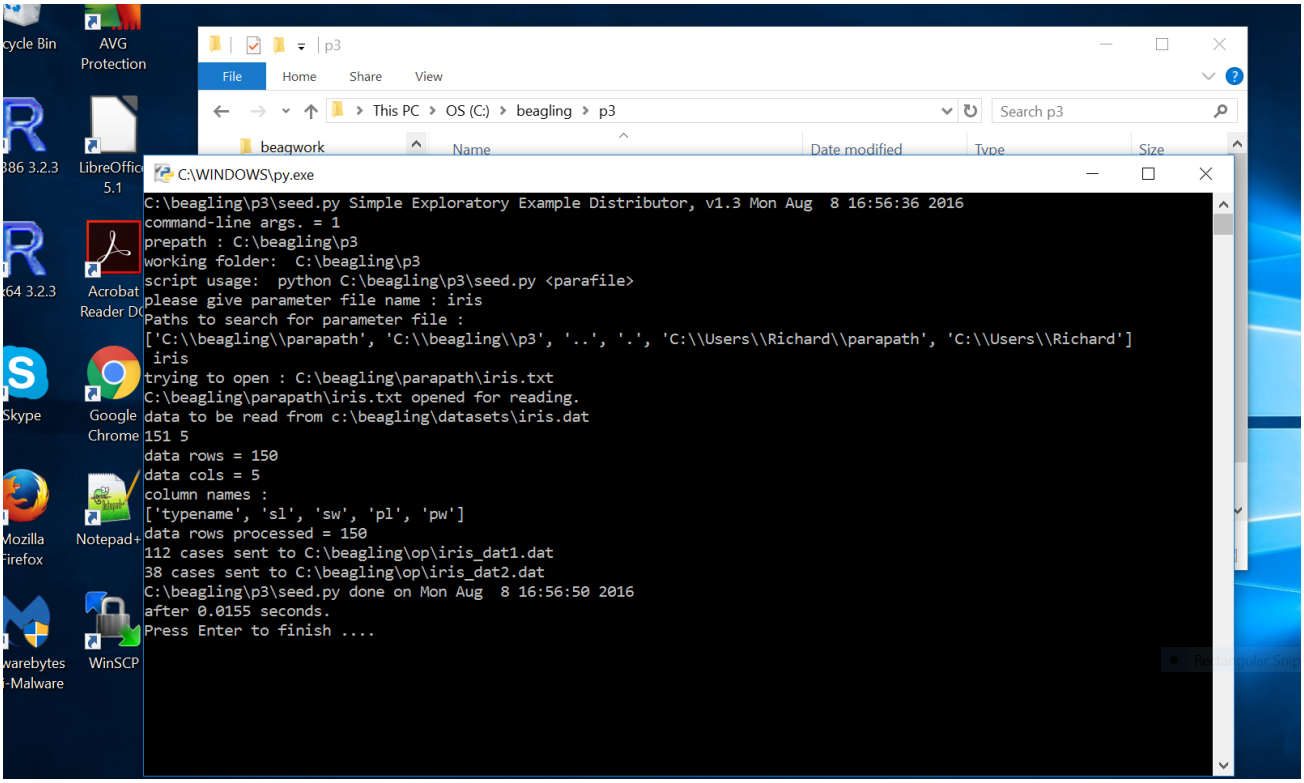
please give parameter file name : **c:\beagling\parapath\iris**

where the user's input is in **bold**. An example screen shot using this parameter file, iris.txt, 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 right 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 iris parameter file is shown below.



The only snag with this method is that the command window is temporary. The BEAGLE programs, if run in this manner, do 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.

2.5 Preparing a Parameter File

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

```
comment   simple zoological example :
jobname   zoobase
maindat   c:\beagling\datasets\zoobase.dat
targval   type
idfield   name
brevity   1
proglang   r
```

This dataset (c:\beagling\datasets\zoobase.dat) refers to a simple example describing 101 animal species in terms of 18 attributes, mostly binary (0/1 indicating absence/presence). It is zoologically naive, but quite easy to understand, so it will be used as an initial example to illustrate each of the BEAGLE programs. The attributes are briefly described below.

```
Zoobase variables:
name          (unscientific) species name
hair          whether it has hair
feathers      whether it has feathers
eggs          whether it lays eggs
milk          whether it gives milk to its young
```

```
airborne      whether it can fly
aquatic       whether it lives (at some stage) in water
predator      whether it eats meat (i.e. isn't a herbivore)
toothed       whether it has teeth
backbone      whether it has a spinal chord
breathes      whether it breathes air
venomous      whether it produces venom
fins          whether it has fins
legs          number of legs
tail          whether it has a tail
domestic      whether it has been domesticated
catsize       whether it is at least as big as typical domestic cat
type          category code:
              1  mammal
              2  bird
              3  reptile
              4  fish
              5  amphibian
              6  insect
              7  other kinds of animal
```

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.

A parameter file 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	beagle16	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 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 categories. It can be a variable name (as here, "type") or a more complex expression. Examples later.
idfield	[None]	This selects an identifying variable for used by herb.py, leaf.py & leaflet.py in their output listings.
brevity	1	This parameter applies to herb.py. If it is 1, the system uses the size of each 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 a ruleset's quality.
proglang	r	This parameter applies to plum.py. It selects the programming language for the export of a ruleset. Valid options are py (Python3) & r.

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 zoobase.dat.

```
name    hair    feathers        eggs    milk    airborne        aquatic predator        toothed backbone
        breathes        venomous        fins    legs    tail    domestic        catsize type
aardvark        1       0       0       1       0       0       1       1       1       1       0       0
        4       0       0       1       1
antelope        1       0       0       1       0       0       0       1       1       1       0       0
        4       1       0       1       1
bass    0       0       1       0       0       1       1       1       1       0       0       1       0
        1       0       0       4
[.... 94 lines omitted ....]
wasp    1       0       1       0       1       0       0       0       0       1       1       0       6
        0       0       0       6
wolf    1       0       0       1       0       0       1       1       1       1       0       0       4
        1       0       1       1
worm    0       0       1       0       0       0       0       0       0       1       0       0       0
        0       0       0       7
wren    0       1       1       0       1       0       0       0       1       1       0       0       2
        1       0       0       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. The only user input is "zoobase" supplied in response to the program's request for a parameter file name. This parameter file is supplied with the BEAGLE distribution. (Extension .txt is assumed if no extension is given.)

```
C:\beagling\p3\seed.py Simple Exploratory Example Distributor, v1.2 Thu Aug  4 15:51:55
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 : zoobase
Paths to search for parameter file :
['C:\\beagling\\parapath', 'C:\\beagling\\p3', '..', '.', 'C:\\Users\\Richard.lounge-
pc\\parapath', 'C:\\Users\\Richard.lounge-pc']
 zoobase
trying to open : C:\beagling\parapath\zoobase.txt
C:\beagling\parapath\zoobase.txt opened for reading.
data to be read from c:\beagling\datasets\zoobase.dat
102 18
data rows = 101
data cols = 18
column names :
['name', 'hair', 'feathers', 'eggs', 'milk', 'airborne', 'aquatic', 'predator', 'toothed',
'backbone', 'breathes', 'venomous', 'fins', 'legs', 'tail', 'domestic', 'catsize', 'type']
data rows processed = 101
62 cases sent to C:\beagling\op\zoobase_dat1.dat
39 cases sent to C:\beagling\op\zoobase_dat2.dat
C:\beagling\p3\seed.py done on Thu Aug  4 15:51:58 2016
after 0.06251 seconds.
```

From this it will be seen that 62 cases have been put into the training file and 39 into the test file. This is the default partition ratio: to alter this ratio use the datfrac parameter (explained in Appendix A).

The program has read data in from zoobase.dat on the folder c:\beagling\datasets\zoobase.dat 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. HERB.py : Heuristic Evolutionary Rule Breeder

Continuing with the zoobase example, using the parameter file shown above, the natural next step is to

create a ruleset with herb.py. In this case we're seeking a rule to classify each animal into one of the seven classes given by the type variable.

The only user input required at runtime is the parameter file name (here zoobase). 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 herb.py.

HERB uses several cycles of repeated subsampling. That is, it applies its evolutionary algorithm several times (from 5 to 10 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, success/failure 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-rate estimates that are not optimistically biased.

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 highest score every time a new highest-scoring ruleset 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 ensure that even during a long run you can see that the program is working.

4.1 HERB's evolutionary algorithm

The core algorithm within HERB that applies the evolutionary optimization method within each subsampling cycle is outlined in the following pseudocode. The overall number of 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;
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 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. 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

(wingspan > (fuselage * 2.47891))

with

((engines = cannons) & (maxrange > (ceiling * 0.8098014))

could produce

((fuselage * 2.47891) > (ceiling * 0.8098014))

among many other possible 'offspring'. (BEAGLE's rule language is described in Appendix B.)

The two main output files of herb.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).

4.2 Example ruleset derived from zoobase data

A complete rule file (zoobase_rule.txt) derived from the zoobase_dat1.dat data follows.

```
training data : C:\beagling\op\zoobase_dat1.dat
creation date : Thu Aug  4 15:54:20 2016
rule mode : tabular
62 18
type
['1', '2', '3', '4', '5', '6', '7']
[0.27092212241224867, 0.1879897902429265, 0.104277973536091, 0.13794679259913573,
0.0903073741374162, 0.104277973536091, 0.104277973536091]
$
( tail > milk )
( ( 2 ; toothed ) - ( feathers < backbone ) )
( breathes ; milk )
$
000  [27, 0, 0, 0, 0, 0, 0]
001  [0, 0, 0, 0, 2, 0, 0]
010  [0, 0, 0, 0, 0, 0, 3]
011  [0, 0, 0, 0, 0, 4, 1]
100  [0, 0, 0, 7, 0, 0, 0]
101  [0, 0, 4, 0, 1, 0, 0]
110  []
111  [0, 13, 0, 0, 0, 0, 0]
[0.9428571428571428, -4.333333333333333]
$
```

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

First come seven lines (one split over 2 lines by the formatting of this document) giving information about the training data. Next come the rules themselves, three in this case. Thirdly comes a "signature table" (Samuel, 1967) which indicates how many training example of each type are found for each of the 8 possible combinations of True(1)/False(0) values of the three rules. The extra line at the end of the signature table is the quality or fitness score of this rule. This consist of 2 numbers, because brevity was set to 1 in the parameter file: the number starting with 0.9428 is Goodman & Kruskal's lambda, a measure between 0 and 1 of how well two categorical variables associate in a contingency table (Upton & Cook, 2006); the number

starting -4.3333 is the average length of the rules, negated because the algorithm maximizes but 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 tabular, i.e. that rules in the ruleset are combined by the signature table (see below). The alternative rule mode is "demonic", named in honour of Selfridge's "Pandemonium" (Selfridge, 1959). In demonic mode there is a rule for each category in the training data and the class to be assigned is decided by evaluating each rule and picking the one with the highest value. The forth line shows that the training data consisted of 62 rows each with 18 columns. The next 2 lines give the target expression, here just a variable name, and the category labels. The final line before the dollar sign, gives the 'prior probabilities' to be assumed for each category: these are used in the Bayesian reasoning when the rule is applied to a particular data instance.

The next section gives the three rules of this ruleset. BEAGLE's rule language is explained in Appendix B. As illustration here, we consider the third rule

(breathes ; milk)

which uses one of BEAGLE's less obvious operators, the semi-colon ";", which stands for Exclusive-Or. The variable breathes will be 1 or 0, depending on whether the animal breathes air or not; milk will be 1 or 0 depending on whether the animal species is one that gives milk to its offspring or not. Thus this rule will be true for air-breathing animals that don't give milk as well as milk-giving animals that don't breathe air (if any exist, perhaps waiting to be discovered by science near some geothermal vent in the deep ocean), false for all others.

The third section is the signature table itself, followed by the quality score on the last line before the final dollar. Here the first line

```
000  [27, 0, 0, 0, 0, 0, 0]
```

shows that when all rules were false (000) there were 27 examples of class '1' (mammal) and no examples of any other class. Similarly, the line

```
101  [0, 0, 4, 0, 1, 0, 0]
```

shows the class distribution when the first and third rules were true and the second rule false (101). There were five such cases in the training data, of which four belonged to class '3' (reptile) and one to class '5' (amphibian). Finally,

```
111  [0, 13, 0, 0, 0, 0, 0]
```

shows that all 13 cases when all rules were true were of class '2' (birds).

When applied to classifying a fresh instance, the rules will be evaluated to determine which row of the signature table is selected, and the frequencies in that row will modulate the prior probabilities to determine the posterior probabilities to be assigned to each category.

4.3 Listing file derived from zoobase data

HERB also produces a listing file (normally named with the jobname with "_list.txt" appended) that summarizes how well the system performed during the resampling cycles. The resampling 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 herb.py from the run on the zoobase data that produced the rule described above follows.

```
dateline   Thu Aug  4 15:52:09 2016
progname   C:\beagling\p3\herb.py
id         C:\beagling\parapath\zoobase.txt
traindat   C:\beagling\op\zoobase_dat1.dat
targval    type

====subsampling trial :

rank  strength case  name      pred:true  cellsize  predvals
   1      0.97   33  opossum     1 + 1    23    0.97    0.01    0.00    0.00    0.00    0.00    0.00
   2      0.97   30  mole        1 + 1    23    0.97    0.01    0.00    0.00    0.00    0.00    0.00
   3      0.97   18  gorilla     1 + 1    23    0.97    0.01    0.00    0.00    0.00    0.00    0.00
   4      0.97   58  wallaby     1 + 1    22    0.97    0.01    0.00    0.01    0.00    0.00    0.00
   5      0.97   43  reindeer    1 + 1    22    0.97    0.01    0.00    0.01    0.00    0.00    0.00
   6      0.97   29  mink        1 + 1    22    0.97    0.01    0.00    0.01    0.00    0.00    0.00
   7      0.97   22  hare        1 + 1    22    0.97    0.01    0.00    0.01    0.00    0.00    0.00

[.... items 8 to 57 deleted to save space ....]

  58      0.44   46  seawasp     7 + 7     2    0.14    0.10    0.05    0.07    0.05    0.05    0.53
  59      0.42   59  wasp        6 + 6     4    0.10    0.07    0.20    0.05    0.03    0.52    0.04
  60      0.42   27  lobster     6 - 7     4    0.10    0.07    0.20    0.05    0.03    0.52    0.04
  61      0.42   16  gnat        6 + 6     4    0.10    0.07    0.04    0.05    0.03    0.52    0.20
  62      0.42    7  crab        6 - 7     4    0.10    0.07    0.20    0.05    0.03    0.52    0.04
  63      0.22   52  toad        5 + 5     1    0.19    0.14    0.07    0.10    0.35    0.07    0.07
  64      0.22   14  treefrog    5 + 5     1    0.19    0.14    0.07    0.10    0.35    0.07    0.07
++++++++++++++++++++++++++++++++++-++++++++++--+++---++-++-+-++

Confusion matrix :

Truecat =         1    2    3    4    5    6    7
Predcat : 1      27    0    0    0    0    0    0
Predcat : 2       1   14    0    0    0    0    0
Predcat : 3       0    0    0    0    1    0    0
Predcat : 4       0    0    2    7    0    0    0
Predcat : 5       0    0    1    0    2    0    0
Predcat : 6       0    0    1    0    0    4    3
Predcat : 7       0    0    0    0    0    0    1

Kappa value =  0.8085
Precision (%) by category :
1     100.0
2     93.3333
3     0.0
4     77.7778
5     66.6667
6     50.0
7     100.0
Recall (%) by category :
1     96.4286
2     100.0
3     0.0
4     100.0
5     66.6667
6     100.0
7     25.0

cases = 64
cases with unseen category labels = 0
hits =  55
percent hits = 85.94
Hedges's g (z-gap) between strengths of right & wrong answers = 1.3048

Resultant rule from all training cases :

[rule listing removed to save space (rule shown in previous section) ....]
```

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 64 decisions in the subsampling trial follow. Fifty of

these have been removed from this extract to save space, leaving only the first and last 7 cases.

All the first seven assignments are correct (mammals). Among the last seven there are two mistakes, a lobster and a crab classified as type '6' (insect) when they should have been type '7' (miscellaneous).

Taking the item at rank 60 as illustration,

```
  60      0.42   27  lobster     6 - 7     4    0.10    0.07    0.20    0.05    0.03    0.52    0.04
```

this line can be used to explicate the values in the various columns. The number 60 is the rank, out of 64, ordered by the number in the second column, headed "strength". This is 0.42 and gives a measure of certainty about the decision. It is computed by comparing the highest of the class posterior probabilities with the remaining posterior probabilities. The next two items ("27 lobster") identify the instance itself: 27 is its position within the training file and "lobster" is the value of the chosen idfield (name). The next three symbols ("6 - 7") show the predicted category, 6, the success status, '-', and the true category, 7. For correct decisions, the success status is '+'. Then comes the number 4, in a column labelled "cellsize". This gives the number of examples in the signature-table row on which the decision was based. In general, smaller numbers are associated with less statistical reliability, hence more doubtful decisions. Then come the 7 posterior probabilities, rounded to 2 places of decimals. In this row the largest probability is 0.52 (applying to insects) and the next largest 0.20 (applying to reptiles): the true category (miscellaneous, at position 7) only gets a posterior probability of 0.04. So this was a clear mistake.

However, there were only nine mistakes in these 64 decisions, so the overall performance wasn't bad. The line just after the last item

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

shows the error-status of each of these 64 decisions, ordered left to right as they are ranked, i.e. from most to least confident (according to the "strength" column). It is relatively reassuring that the first 35 decisions are all correct, and all but one of the mistakes are found in the last 18. To summarize: the system is telling us that it knows pretty well how to identify mammals and birds, but isn't so sure about other types.

This conclusion is reinforced by looking at the "confusion matrix" which shows how the predicted and true categories were associated. The Kappa value derived from this is a measure of statistical association, Cohen's Kappa (Upton & Cook, 2006). The Precision and Recall of each category is also given. From this it can be inferred that the ruleset is particularly poor on reptiles, of which only four were present in the training data: 41 of the 55 correct decisions were birds or mammals. The overall success rate was nearly 86%.

The values given as Hedges's g, represents how many standard deviations apart the means of the correct and incorrect "strength" values were. It is a z-score intended to give an indication of the reliability of the strength ranking. Interpretation in terms of signal detection theory is possible, though tricky. As a rule of thumb, anything above 1 is satisfactory.

5. LEAF.py : Likelihood Estimator And Forecaster

The normal next step after running SEED and HERB is to run LEAF on the data held out as a test sample. The following is an extract from LEAF's main listing (suffixed "_test.txt") on the test data file zoobase_dat2.dat. The output is in the same format as the listing from herb.py (suffixed "_list.txt") and can be interpreted in essentially the same way. Only the first 7 and last 10 individual decisions (out of 39) are shown below, to save space.

```
dateline   Thu Aug  4 15:55:01 2016
progname   C:\beagling\p3\leaf.py
```

```
id         C:\beagling\parapath\zoobase.txt
testdat    C:\beagling\op\zoobase_dat2.dat
targval    type

====holdout trial :

rank  strength case  name      pred:true  cellsize  predvals
   1      0.97   37  vole        1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00
   2      0.97   33  squirrel    1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00
   3      0.97   26  raccoon     1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00
   4      0.97   25  puma        1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00
   5      0.97   24  porpoise    1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00
   6      0.97   23  pony        1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00
   7      0.97   22  polecat     1 + 1    27    0.98    0.01    0.00    0.00    0.00    0.00    0.00

[.... items 8 to 29 omitted, to save space ....]

  30      0.55   34  starfish    7 + 7     3    0.12    0.08    0.04    0.06    0.04    0.04    0.62
  31      0.55   20  octopus     7 + 7     3    0.12    0.08    0.04    0.06    0.04    0.04    0.62
  32      0.55    7  crayfish    7 + 7     3    0.12    0.08    0.04    0.06    0.04    0.04    0.62
  33      0.55    6  clam        7 + 7     3    0.12    0.08    0.04    0.06    0.04    0.04    0.62
  34      0.50   38  worm        6 - 7     5    0.08    0.06    0.03    0.04    0.03    0.59    0.17
  35      0.50   36  termite     6 + 6     5    0.08    0.06    0.03    0.04    0.03    0.59    0.17
  36      0.50   19  moth        6 + 6     5    0.08    0.06    0.03    0.04    0.03    0.59    0.17
  37      0.50   16  housefly    6 + 6     5    0.08    0.06    0.03    0.04    0.03    0.59    0.17
  38      0.50   10  flea        6 + 6     5    0.08    0.06    0.03    0.04    0.03    0.59    0.17
  39      0.39   11  frog        5 + 5     2    0.15    0.11    0.06    0.08    0.49    0.06    0.06
++++++++++++++++-++++++-+++++++++-+++++

Confusion matrix :

Truecat =          1    2    3    4    5    6    7
Predcat : 1       14    0    0    0    0    0    0
Predcat : 2        0    7    0    0    0    0    1
Predcat : 4        0    0    1    6    0    0    0
Predcat : 5        0    0    0    0    1    0    0
Predcat : 6        0    0    0    0    0    4    1
Predcat : 7        0    0    0    0    0    0    4

Kappa value =  0.9008
Precision (%) by category :
1     100.0
2     87.5
4     85.7143
5     100.0
6     80.0
7     100.0
Recall (%) by category :
1     100.0
2     100.0
3     0.0
4     100.0
5     100.0
6     100.0
7     66.6667

cases = 39
cases with unseen category labels = 0
hits =  36
percent hits = 92.31
Hedges's g (z-gap) between strengths of right & wrong answers = 0.4049
```

Here percentage success (92.31) is actually higher than estimated in the herb.py listing (85.94) though the reliability of the strength ranking is less.

6. PLUM.py : Procedural Language Utility Module

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 reliably classify fresh examples of the same sort of data.

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

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

A complete listing of the R source code file (zoobase_rule.r) produced by PLUM from the file zoobase_rule.txt follows.

```
##  Using BEAGLE R template, version of 28/07/2016 :
##  rule written by plum.py ;
##  derived from training data : C:\beagling\op\zoobase_dat1.dat;
##  generated on creation date : Thu Aug  4 15:54:20 2016;
##  dumped on Thu Aug  4 15:59:04 2016.
##
beag_gold = 5.0 ** 0.5 * 0.5 + 0.5  ##  global

##  helper functions :
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))
       }

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

       stab = list()
       stab[['000']] = c(27, 0, 0, 0, 0, 0, 0)
       stab[['001']] = c(0, 0, 0, 0, 2, 0, 0)
       stab[['010']] = c(0, 0, 0, 0, 0, 0, 3)
       stab[['011']] = c(0, 0, 0, 0, 0, 4, 1)
       stab[['100']] = c(0, 0, 0, 7, 0, 0, 0)
       stab[['101']] = c(0, 0, 4, 0, 1, 0, 0)
       stab[['110']] = c(0, 0, 0, 0, 0, 0, 0)
       stab[['111']] = c(0, 13, 0, 0, 0, 0, 0)

       ##  unpacks stab lines.

       return (stab)
       }  ##  stabprep ends.


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

```
       rule = c() ; bins = c('0','1')
       catlist = c('1', '2', '3', '4', '5', '6', '7')
       priorvec = c(0.27092212241224867, 0.1879897902429265, 0.104277973536091,
0.13794679259913573, 0.0903073741374162, 0.104277973536091, 0.104277973536091)
       subrules = 3
       ##  compute rule values :
       rule[1] = (vals$tail > vals$milk)
       rule[2] = (beag_exor(2,vals$toothed) - (vals$feathers < vals$backbone))
       rule[3] = beag_exor(vals$breathes,vals$milk)

       p = 0 ; b = c()
       while (p < subrules) {
              p = p + 1  ##  early-r, late-py
              v = (rule[p]>0)  ##  omit if demonic
              b = c(b,bins[v+1])  ##  boolean string, omit if demonic
              }
       b = paste(b,collapse='')  ##  omit if demonic
       ##  retrieve cell frequencies :
       frex = stab[[b]]
       cellsize = sum(frex)
       ##  attenuate frex :
       cats = length(priorvec)
       slug = beag_gold / cats
       modfrex = slug + frex
       ##  defer to Reverend Bayes :
       postvec = priorvec * modfrex
       postvec = postvec / sum(postvec)
       pc = which.max(postvec)
       predcat = catlist[pc]

       return (list(cellcode=b,predcat=predcat,frex=frex,postvec=postvec))
       }
##  decision rule ends.


leaflet = function (datframe) {
       ##  Language-Export Application For Likelihood Estimation Trials.

       stab = beag_stabprep()
       rows = dim(datframe)[1]
       if (rows < 1)  return (NULL)
       options(stringsAsFactors=FALSE)
       ##  prepare extra cols :
       predcat = character(rows) ; stabcode = character(rows)
       cellsize = numeric(rows)
       strength = numeric(rows)
       topprob = numeric(rows) ; nextprob = numeric(rows)
       for (r in 1:rows) {
              beagvals = beag_decrule(datframe[r,],stab)
              if (r == 1) cats = length(beagvals$frex)
                     ##  what if cats < 2 (?!)
                     ##  also test if predvals[[2]] is character ?
              stabcode[r] = beagvals$cellcode
              predcat[r] = as.character(beagvals$predcat)  ##  might be numeric
              cellsize[r] = sum(beagvals$frex)
              post = sort(beagvals$postvec,decreasing=TRUE)
              topprob[r] = post[1] ; nextprob[r] = post[2]
              ##  nextprob's category tricky to obtain; worth doing ?
              pd = post[1] - post[2]  ##  winning margin
              if (cats <= 2)  strength[r] = pd
              else {
                     ps = c(nextprob[r],post[2:cats])  ##  duplicate second-highest prob
                     strength[r] = topprob[r] - sum(ps) / cats  ##  compare top with
augmented remainder
                     }
              }
       outframe = data.frame(strength,predcat,topprob,nextprob,stabcode,cellsize)
```

```
       temp = cbind(datframe,outframe)
       options(stringsAsFactors=default.stringsAsFactors())
       return (temp)
       }  ##  returns data frame with additional columns.

##  ending.
```

This module begins with a definition of some helper functions for implementing the 'safe' natural logarithm and square root operators with the same semantics as in BEAGLE, and ensuring that BEAGLE's version of Exclusive Or is implemented as expected in respect of conversions between True/False and numeric values.

Then comes beag_stabprep() which sets up the signature table with the data-derived frequency counts to be used in the Bayesian reasoning.

Next follows the R function beag_decrule(,) that makes decisions about individual instances; while the function leaflet() (Language-Export Application For Likelihood Estimation Trials) uses that function to take an input data frame and produce an output data frame in which each row has additional columns indicating its computed category membership and the probability assigned to that classification, as well as stabcode, indicating the row in the signature table used and cellsize, the number of training instances that fell into that row.

Because you have the source code, you can examine the code at leisure to understand its workings, and of course you can export it and apply it to fresh data within the R environment. (For Pythoneers, an example of Python3 output from plum.py is shown in Appendix C.)

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

7. LEAFLET.py : Language Export Application For Likelihood Estimation Testing

LEAFLET.py 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 leaflet() applies the learned ruleset to a dataframe of examples -- probably the most natural way of using the results of BEAGLE'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 leaf.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 BEAGLE ruleset into Python, and thus provides a pointer towards doing likewise with his or her programs.

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 HERB and PLUM 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 HERB run may take too long, but it is valid in principle.)

It is also fair to point out that, like most machine-learning systems, BEAGLE has an Achilles Heel. This is the "none-of-the-above" problem, which afflicts all classifiers, including humans, though computational

classifiers are particularly vulnerable in this respect.

In fact, this is a variation on the theme of "outliers" -- a concept that 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. For instance, if you throw information about daffodils or orchids at a ruleset trained to distinguish the three types of iris flower in the iris.dat file, it will classify them as setosa or versicolor or virginica. It will pick the most likely of those three classes, but in this case the most likely is still impossible.

The "strength" index in LEAF and LEAFLET 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 software using Bayesian reasoning, BEAGLE, in effect, assigns zero prior probability to never-seen categories. There isn't a general rule for assigning prior probability to a dustbin category such as "none of the above".

A completely rigorous general solution to this problem is unattainable. There can be no context-free definition of what constitutes an outlier. However, I am working on alternative heuristics designed to do better with typical real-life data sets than BEAGLE's present "strength" measure. With average-to-good luck, I hope to incorporate such a technique into RUNSTER and release it before the end of 2016; and then retrofit the same technique into BEAGLE.

Another limitation that should be mentioned is that BEAGLE 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 BEAGLE on a desktop computer would take an unrealistic amount of time. BEAGLE 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 BEAGLE 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.

Even after 36 years, BEAGLE is 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 happy BEAGLING! 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 would be runnable again; 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.

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 ed. Oxford: Oxford Univ. Press.

Appendix A : Parameter Files

The table below gives information about BEAGLE 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 (SHLP for seed.py, herb.py, leaf.py & plum.py). For example, SH-- would mean that SEED & HERB 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 BEAGLE suite.

Parameter	Default value	Function
brevity -H--	1	If brevity is set to 1, the average size of a ruleset will be used as the second element in its fitness score (negated, since HERB 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 -HL-	[None] => jobname with "_dump.txt" appended	HERB & LEAF dump versions of their decisions on each case in form that can be read back into Python or, less easily, R in this file. It is mainly intended to assist the programmer in checking the software, but you're welcome to look at it.
idfield -HL-	[None]	HERB & LEAF use this variable name to identify each row in their output listings (see listfile). If none is given, the input row number is used as an identifier.
jobname SHLP	beagle16	This gives the job a name. Any text string 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 group.
listfile -HLP	[None]	This is a file specification for the main, human-readable, output listing of HERB, LEAF or PLUM. 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 "_plum.txt" appended, respectively, and placed in the outpath folder (see below).
m1 -H--	0.5	This number (from 0 to 1) specifies the probability that after a mating/crossover operation in HERB a mutation will be performed on the resultant offspring.
m2 -H--	0.25	This number (from 0 to 1) specifies the probability, in HERB, 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, it specifies the proportion of new rulesets to be generated by 'asexual' reproduction.
maindat	[None]	This should be the full file specification of a file where the input

S---		data is stored (in tab-delimited form with a header line naming the columns, and each row describing a single instance).
ntrials -H--	131072	The total number of new structures to be generated during all HERB's evolutionary trials. Minimum 256, maximum 1048576.
outpath SHLP	..\op\	This specifies the folder (directory) where the program will place its output. Default is the \op\ subfolder of the parent of the folder in which the program resides.
popsize -H--	233	This specifies the number of quasi-organisms (rulesets) in the population being optimized by HERB. Minimum 8, maximum 2048.
randseed SH--	0	This number is used to initialize Python's pseudo-random number generator in SEED and HERB -- 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.
progfile ---P	[None]	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 BEAGLE ruleset: r selects the R language; py is for Python3.
rulefile -HLP	[None] => jobname with "_rule.txt" appended	This specifies the file into which HERB will write its highest-scoring ruleset at the end of its optimization process, and from which LEAF and PLUM will read the ruleset to be used.
rulemode -H--	tabular	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 in HERB and to use frequencies in HERB, LEAF, PLUM and LEAFLET. The only recognized alternative mode is demonic, in which case there will be as many rules as categories with a frequency line for each and the decision for each instance will be based on the rule that gives the highest numerical value.
skipvars -H--	[None]	The value for skipvars should be a list of column/variable names separated by commas, e.g. skipvars bombing,country which will tell HERB 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 -HLP	[None]	This parameter is used to define the target categories. It can be a variable name (e.g. "type") or a more complex expression. An example can be seen in Appendix C.
testdat S-L-	[None] => jobname with "_dat1.dat" appended	This specifies a file into which SEED will write a test subset of the full data (see maindat) and which LEAF & LEAFLET will use by default for input.
traindat SH-P	[None] => jobname with "_dat2.dat"	This specifies a file into which SEED will write a training subset of the full data (see maindat) and which HERB & PLUM will use by default for input.

trigfunx -H--	0	BEAGLE'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.

LEAFLET uses the same parameters as LEAF except that it reads from progfile (written by PLUM) instead of rulefile (written by HERB).

Appendix B: BEAGLE's rule language

BEAGLE's rule language 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 HERB'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 expression, but you may have to interpret some.

BEAGLE 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 complete run-through of the BEAGLE 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 k/h
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, i.e. decide on a classification scheme. Often this is obvious, but with this data there are several possibilities. For example, we might train the system to classify these aircraft by country of origin. To do that would require choosing country as the value for the targval parameter. However, in this instance I have decided that the system should attempt to learn a ruleset for picking out fighters. To do so, I prepared the parameter file listed below. 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\aircraft.txt.

```
1      ##  beagle testing on ww2 military aircraft :
2      jobname  aircraft
3      maindat  c:\beagling\datasets\aircraft.dat
4      datfrac  0.75
5      outpath  c:\beagling\takeoff\
6      randseed  3433
7      ##targval  bombing
8      ##skipvars  fighting
```

```
9      targval  (fighting > bombing)
10     skipvars  carrying
11     idfield      name
12     brevity  1
13     proglang  py
```

Here the target appears on line 9. I have left in lines 7 and 8, which are commented out, so won't have any effect on the programs, to show that this file has been modified from one which used bombing as the target variable -- and also specified that the variable fighting shouldn't be allowed in generated rules, to avoid what might be viewed as cheating, hence "##skipvars fighting" on line 8. (There are some fighter-bombers in the data, so fighting and bombing are not perfect inverses, but it seemed better not to allow the system to use a variable so clearly related to the outcome being predicted.)

In fact the target expression on line 9 (fighting > bombing) will only be true for what might be called "pure" fighters, aeroplanes only used as fighters, not bombers. Since both variables in the target expression, fighting & bombing, will be excluded from HERB's generation process, it is not necessary to have a line such as

skipvars bombing

but I have inserted

skipvars carrying

at line 10 to prevent the system from using knowledge about whether the aircraft is used as a transport plane in its classification of fighters.

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

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

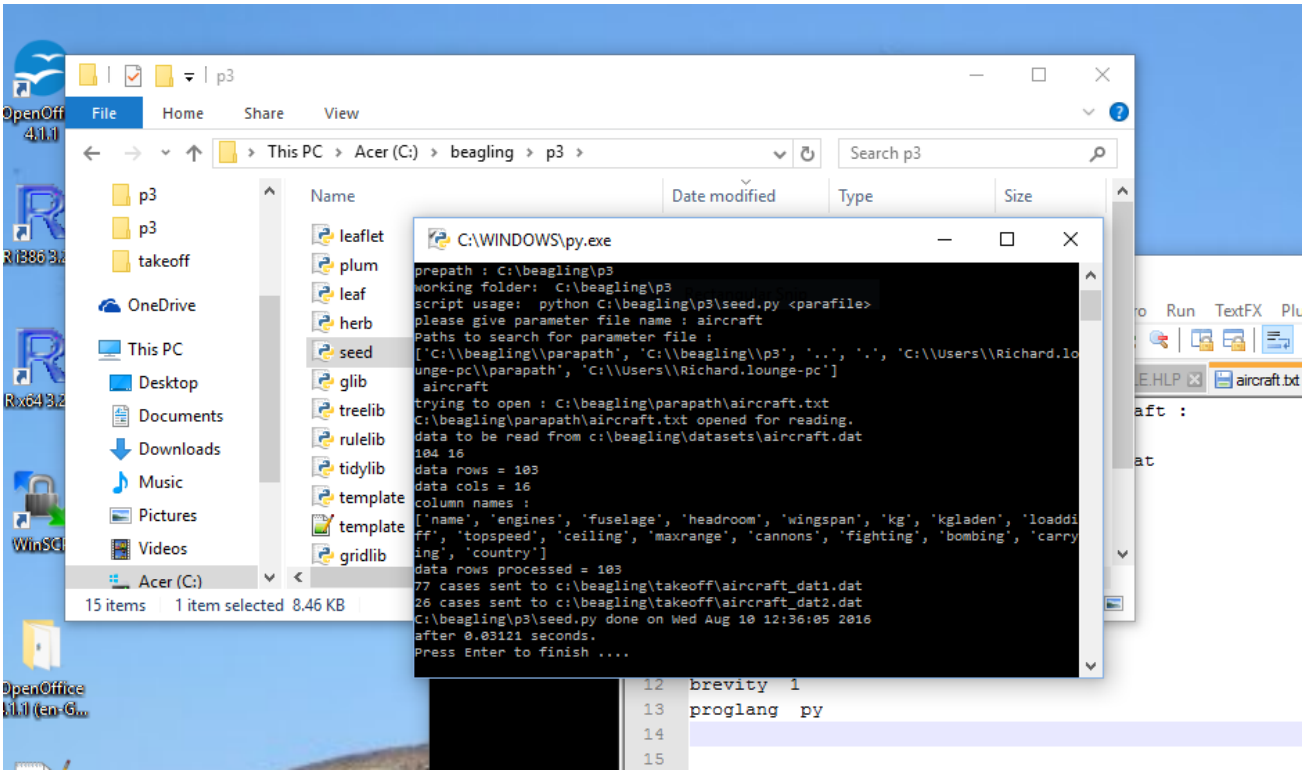
Line 4 instructs the SEED program that 75% of the instances in the data should be placed in the training file, which will be called aircraft_dat1.dat since no traindat parameter has been specified, leaving 25% to go into the test file (called aircraft_dat2.dat, since no testdat parameter has been specified 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 (randseed 3433) ensures that SEED and HERB will set Python's pseudorandom number generator to a particular starting point. This ensures that the whole exercise is repeatable. You should get exactly the same results if you use this as a trial on your computer, and I won't have to start again from scratch if I happen to delete some output that I later decide should have been included in this guide!

C.2 Running SEED

It is typical, as in this case, to start a BEAGLE run, by executing the seed.py program. If you double-click on seed.py in the \beagling\p3\ folder and type "aircraft" to specify the parameter file, you should see something rather like the screen shot below. Lines 1-6 of the parameter file, shown in section C.1, are in fact all that are needed for seed.py.



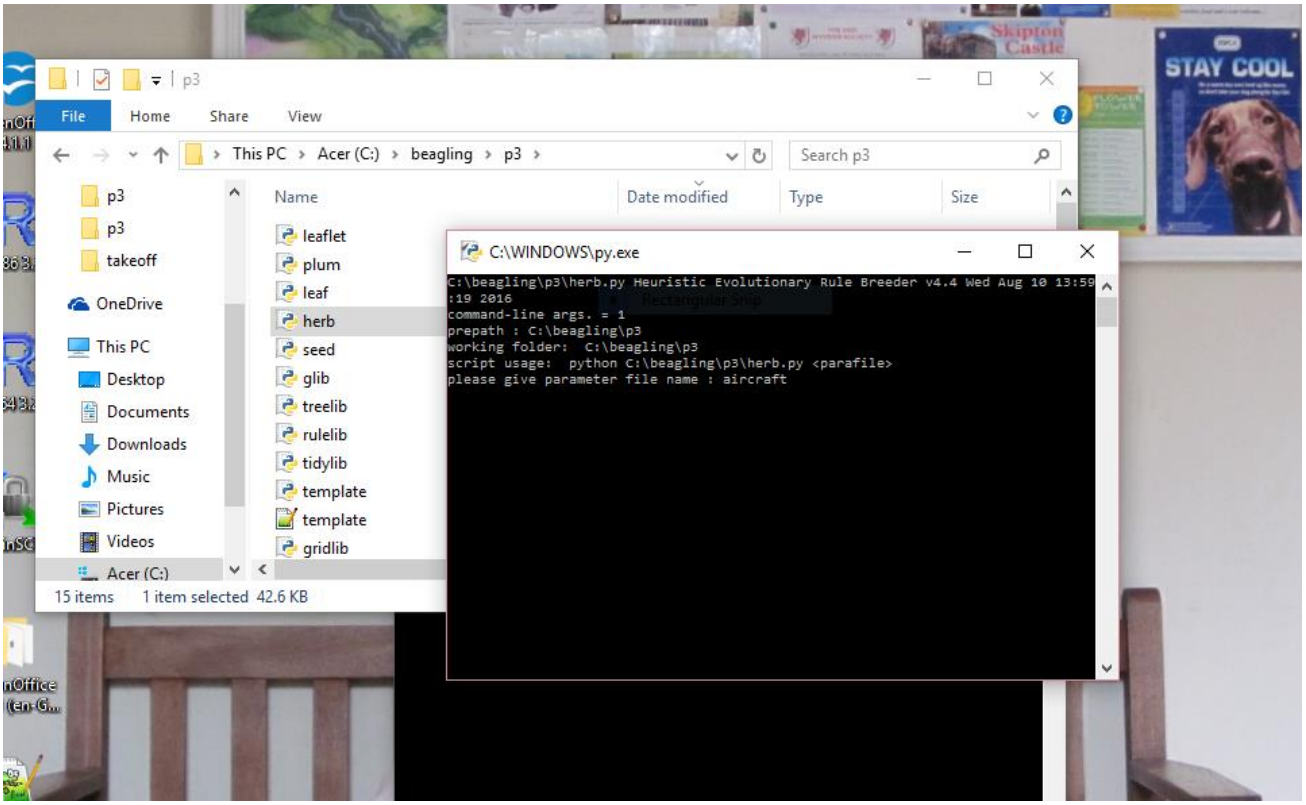
This reads the 103 cases in c:\beagling\datasets\aircraft.dat and puts 77 cases into aircraft_dat1.dat and 23 into aircraft_dat2.dat, both in the folder c:\beagling\takeoff\.

The header line and the first and last 2 lines of aircraft_dat2.dat are listed below.

```
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
Fiat_BR20_Cicogna       2       16.78   4.75    21.56   6700    10450   3750    430     6750    2000    0
        0       1       0       italy
[....]
Martin_Baltimore_III            2       14.79   5.41    18.69   6900    10442   3542    486     7320
        1705    0       0       1       0       us
NorthAmerican_Mustang_P51D      1       9.82    4.17    11.3    3232    5266    2034    703     12780
        2092    0       1       0       0       us
```

C.3 Running HERB

The screen shot below shows initial HERB output, under Windows 10, just after running herb.py and typing "aircraft" in reply to the parameter file request, but before pressing the Enter key.



After pressing Enter, 8 cycles of on-screen output rather like the extract below will appear.

```
233 [0.35714285714285715, -5.5]
889 [0.39285714285714285, -4.0]
1106 [0.42857142857142855, -7.5]
1107 [0.42857142857142855, -6.5]
1850 [0.42857142857142855, -6.0]
1868 [0.4642857142857143, -9.0]
1874 [0.4642857142857143, -8.0]
1909 [0.5, -4.5]
2074 [0.5357142857142857, -12.5]
3070 [0.5357142857142857, -8.5]
3455 [0.5357142857142857, -8.0]
4229 [0.5357142857142857, -7.0]
4470 [0.5357142857142857, -6.5]
4718 [0.5714285714285714, -8.0]
4753 [0.5714285714285714, -7.5]
5642 [0.5714285714285714, -7.0]
6769 [0.6071428571428571, -10.5]
7112 [0.6428571428571429, -8.5]
8516 [0.6428571428571429, -8.0]
12450 [0.6785714285714286, -11.0]
13416 [0.6785714285714286, -10.0]
14642 [0.6785714285714286, -5.0]
15204 [0.7142857142857143, -6.0]
15267 [0.75, -8.0]
15943 [0.7857142857142857, -10.0]
c:\beagling\takeoff\aircraft_dat1.dat
Wed Aug 10 14:06:10 2016
tabular
77 16
(fighting > bombing)
['0', '1']
[0.5358983848622454, 0.4641016151377546]
$
( 19.95 & ( wingspan > 13.0871497 ) )
( cannons < ( cannons < ( ( ( ( country ? `ussr`) * kgladen ) - ( ( country ? `japan`) ;
```

```
engines ) ) + engines ) ) )
$
00  [1, 23]
01  [2, 0]
10  [12, 5]
11  [25, 0]
[0.7857142857142857, -10.0]
$
2 cycles done.
233 [0.25925925925925924, -4.0]
491 [0.25925925925925924, -3.5]
569 [0.25925925925925924, -2.0]
603 [0.3333333333333333, -5.0]
797 [0.5185185185185185, -6.0]
1180 [0.5555555555555556, -8.0]
1285 [0.5555555555555556, -7.5]
3064 [0.5925925925925926, -10.0]
3589 [0.6296296296296297, -6.0]
3950 [0.6296296296296297, -4.0]
7197 [0.6666666666666666, -8.5]
7988 [0.6666666666666666, -7.5]
9213 [0.6666666666666666, -5.0]
10220 [0.7037037037037037, -7.0]
10904 [0.7037037037037037, -4.0]
3 cycles done.
```

This illustrates the second and third evolutionary cycle. Lines such as

```
15943 [0.7857142857142857, -10.0]
```

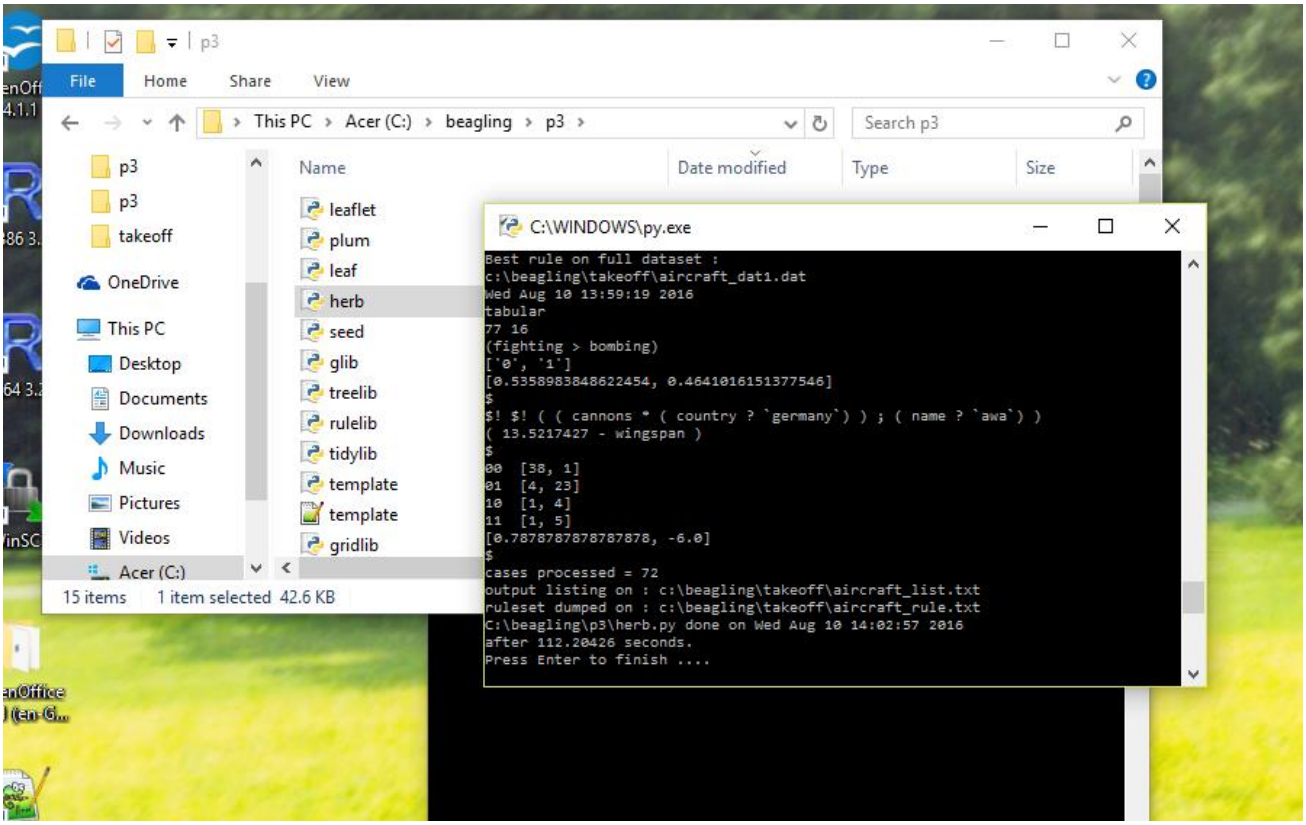
appear each time a new highest-scoring ruleset is found during each cycle. This line can be interpreted as follows: after 15943 trials (i.e. 15943 new rulesets created in the current cycle) a new best ruleset was found with a score of 0.7857142857142857 (Goodman & Kruskal's lambda) and an average rule length of 10 elements. This rule-size measure is included as second element of the fitness score because parameter brevity was set to 1.

Because this was a better score than the best rule from the previous cycle (cycle 1) the ruleset itself is printed as well. (A better and shorter ruleset was discovered in a later cycle, so I won't attempt to interpret this one.) Cycle 3 ended with the line

```
10904 [0.7037037037037037, -4.0]
```

indicating that the best rule in that cycle was created on trial 10904, with a score of 0.7037037037037037 and a mean length of 4 elements. This didn't improve on the score reached in cycle 2, so the rule isn't displayed.

A screen shot showing the ending of this particular HERB run, which comprised 8 subsampling cycles, is shown below. This shows the overall highest-scoring ruleset, written into file aircraft.rule.txt. You should expect to view something similar if using the same dataset and parameter file.



This output ruleset is reproduced below.

```
training data : c:\beagling\takeoff\aircraft_dat1.dat
creation date : Wed Aug 10 14:08:11 2016
rule mode : tabular
77 16
(fighting > bombing)
['0', '1']
[0.5358983848622454, 0.4641016151377546]
$
$! $! ( ( cannons * ( country ? `germany`) ) ; ( name ? `awa`) )
( 13.5217427 - wingspan )
$
00  [38, 1]
01  [4, 23]
10  [1, 4]
11  [1, 5]
[0.7878787878787878, -6.0]
$
```

It contains a pair of independent rules, the first of which begins with 2 logical negation operators. In logic, a double negation is redundant, so it might be expected that HERB's rule-tidying procedure would have eliminated this redundancy. In this example, the double negation is indeed redundant, because the subexpression it operates upon yields a logical value. However, given the way that BEAGLE mixes numeric and logical values, double negatives cannot in general be eliminated without exploring the rule tree rather extensively, so this particular simplification has not (so far) been incorporated into BEAGLE's rather elementary rule-tidying functions.

Ignoring the double negation, which here won't affect the result, this first rule performs an Exclusive Or (';') between the two subexpressions below.

((cannons * (country ? `germany`))

(name ? `awa`)

The second will be true if the aircraft name contains the substring "awa", which in practice means it was manufactured by Kawanishi or Kawasaki. The first rule multiplies the number of cannons (always a number between 0 and 6 in this data) by the truth value (0 or 1) of the test whether the country of origin is "germany". This component will be true only for German aircraft that have cannons (not just machine-guns). Thus the whole rule will be true of aircraft that are either German with cannons or have "awa" in their names but not both. (Both, we know from history, didn't ever happen.)

The second rule

(13.5217427 - wingspan)

is simpler; and illustrates how HERB deals with numeric expressions when in "tabular" mode. In tabular mode a subrule is expected to be true or false: if an expression yields a numeric value, it will be treated as 0 (false) if it is zero or negative and treated as 1 (true) if it is greater than zero. So in this context this rule will have the same effect as

(13.5217427 > wingspan)

being considered true for those (relatively narrow) aeroplanes with wingspan less than 13.5217427 metres, false otherwise.

In the training file there were 39 cases when both rules were false, only 1 being a "pure" fighter and 38 being aircraft used in other roles such as bombing, transport or possibly as fighter-bombers. There were only 6 cases when both rules were true, five of which were "pure" fighters.

An extract from HERB's other output file (aircraft_list.txt), which summarizes how the rules performed during the subsampling cycles, is reproduced below.

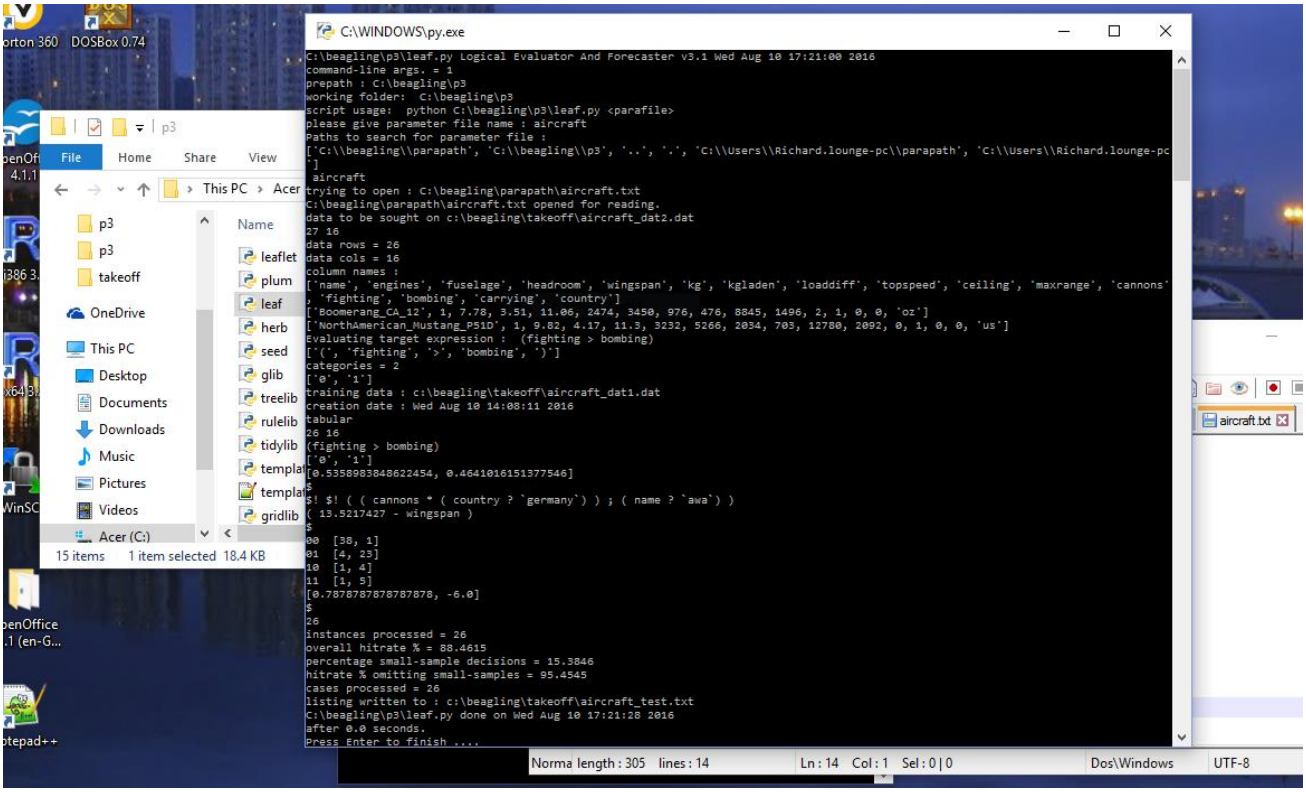
```
dateline   Wed Aug 10 14:06:10 2016
progname   C:\beagling\p3\herb.py
id         C:\beagling\parapath\aircraft.txt
traindat   c:\beagling\takeoff\aircraft_dat1.dat
targval    (fighting > bombing)

====subsampling trial :

rank  strength case  name                          pred:true      cellsize  predvals
   1      0.95   53  Short_Stirling_III              0 + 0             25    0.97    0.03
   2      0.95   36  Ilyushin_II4                    0 + 0             25    0.97    0.03
   3      0.91   75  Republic_P47D                   0 - 1             35    0.96    0.04
   4      0.91   64  Douglas_Dakota_C47A             0 + 0             35    0.96    0.04
   5      0.91   18  Mitsubishi_Ki46                 0 + 0             35    0.96    0.04
   6      0.91   68  Grumman_TBF1                    0 + 0             33    0.95    0.05
   7      0.91   63  Douglas_A20G                    0 + 0             33    0.95    0.05
   8      0.91   46  Fairey_Swordfish_I              0 + 0             33    0.95    0.05
   9      0.91   44  Fairey_Battle_II                0 + 0             33    0.95    0.05
  10      0.91   42  Bristol_Blenheim_IV             0 + 0             33    0.95    0.05
  11      0.91   29  Junkers_Ju88                    0 + 0             33    0.95    0.05
  12      0.87   72  Martin_B26B                     0 + 0             36    0.93    0.07
  13      0.87   59  Consolidated_B24J               0 + 0             36    0.93    0.07
  14      0.87   38  Petlyakov_Pe2                   0 + 0             36    0.93    0.07
  15      0.87   27  Heinkel_He219                   0 - 1             36    0.93    0.07

[ .... items 16 to 60 omitted to save space .... ]

  61      0.54   58  Brewster_Buffalo_F2A            1 + 1             22    0.23    0.77
```

```
  62      0.54   13  Mitsubishi_A6M_Zero            1 + 1              22    0.23    0.77
  63      0.42   40  Tupolev_SB2                    0 + 0               4    0.71    0.29
  64      0.39   24  FockeWulf_Fw190D               1 + 1               5    0.30    0.70
  65      0.35   26  Heinkel_He162                  1 + 1               8    0.32    0.68
  66      0.32   43  DeHavilland_Mosquito           1 - 0               1    0.34    0.66
  67      0.19   74  Northrop_P61B                  1 - 0               6    0.40    0.60
  68      0.19   50  Hawker_Typhoon_IB              1 + 1               6    0.40    0.60
  69      0.19   32  Messerschmitt_Me410            1 + 1               6    0.40    0.60
  70      0.10   35  Ilyushin_II2_Shturmo           1 - 0              10    0.45    0.55
  71      0.10   31  Messerschmitt_Bf110            1 + 1              10    0.45    0.55
  72      0.07   11  Kawanishi_N1K1                 0 - 1               0    0.54    0.46
++-+++++++++++-+++++++++-++++++++++++++++++-++++-++++++++++++++++++-+++-+-

Confusion matrix :

Truecat =         0    1
Predcat : 0      33    3
Predcat : 1       8   28

Kappa value =  0.693
Precision (%) by category :
0     91.6667
1     77.7778
Recall (%) by category :
0     80.4878
1     90.3226

cases = 72
cases with unseen category labels = 0
hits =  61
percent hits = 84.72
Hedges's g (z-gap) between strengths of right & wrong answers = 0.744

Resultant rule from all training cases :
c:\beagling\takeoff\aircraft_dat1.dat
Wed Aug 10 14:06:10 2016
tabular
77 16
(fighting > bombing)
['0', '1']
[0.5358983848622454, 0.4641016151377546]
$
$! $! ( ( cannons * ( country ? `germany`) ) ; ( name ? `awa`) )
( 13.5217427 - wingspan )
$
00  [38, 1]
01  [4, 23]
10  [1, 4]
11  [1, 5]
[0.7878787878787878, -6.0]
$


Parameter settings :

[ .... parameter dump omitted to save space .... ]
```

A point to note about this listing is that it does not record the application of the final best rule itself (even though that rule is listed, mainly as a convenience), but of the highest-scoring rule in each subsampling cycle when applied to the subsample of held-out cases. Thus the performance figures, such as the success rate of 84.72%, are effectively predictions of how the best overall rule is expected to perform on data from the same source. The idea of this output file is to provide evidence for judging the likely practical value of the generated ruleset.

C.4 Running LEAF

Running leaf.py with the same parameter file should produce output resembling the screen shot below.



LEAF applies the ruleset created by HERB (aircraft_rule.txt) to the 26 holdout cases written by SEED into the test file (aircraft_dat2.dat). Its main output file (aircraft_test.txt in this example) is listed below.

```
dateline    Wed Aug 10 17:21:00 2016
progname    C:\beagling\p3\leaf.py
id          C:\beagling\parapath\aircraft.txt
testdat     c:\beagling\takeoff\aircraft_dat2.dat
targval     (fighting > bombing)

====holdout trial :

rank  strength case  name                      pred:true      cellsize  predvals
   1      0.92   24  Martin_Baltimore_III        0 + 0            39     0.96     0.04
   2      0.92   23  Douglas_A26B                0 + 0            39     0.96     0.04
   3      0.92   22  Boeing_B29                  0 + 0            39     0.96     0.04
   4      0.92   20  HandleyPage_Halifax_        0 + 0            39     0.96     0.04
   5      0.92   18  Bristol_Beaufort            0 + 0            39     0.96     0.04
   6      0.92   17  Bristol_Beaufighter_        0 - 1            39     0.96     0.04
   7      0.92   16  ArmstrongWhitworth_W        0 + 0            39     0.96     0.04
   8      0.92   14  Tupolev_Tu2                 0 + 0            39     0.96     0.04
   9      0.92    9  Junkers_Ju87_Stuka          0 + 0            39     0.96     0.04
  10      0.92    8  Junkers_Ju53                0 + 0            39     0.96     0.04
  11      0.92    1  Fiat_BR20_Cicogna           0 + 0            39     0.96     0.04
  12      0.62   25  NorthAmerican_Mustan        1 + 1            27     0.19     0.81
  13      0.62   21  Bell_P39Q                   1 + 1            27     0.19     0.81
  14      0.62   19  Gloster_Meteor_I            1 + 1            27     0.19     0.81
  15      0.62   15  Yakolev_Yak3                1 + 1            27     0.19     0.81
  16      0.62   13  Polikarpov_I16              1 + 1            27     0.19     0.81
  17      0.62   12  Mikoyan_Gurevich_MiG        1 + 1            27     0.19     0.81
  18      0.62   11  Lavochkin_La5               1 + 1            27     0.19     0.81
  19      0.62    5  Nakajima_Ki44               1 + 1            27     0.19     0.81
  20      0.62    4  Nakajima_Ki43               1 + 1            27     0.19     0.81
```

```
  21      0.62    2  Macchi_C202_Folgore            1 + 1             27    0.19    0.81
  22      0.62    0  Boomerang_CA_12                1 + 1             27    0.19    0.81
  23      0.47   10  Messerschmitt_163_Ko           1 + 1              6    0.26    0.74
  24      0.47    6  FockeWulf_Fw190                1 - 0              6    0.26    0.74
  25      0.47    3  Kawasaki_Ki61                  1 + 1              6    0.26    0.74
  26      0.39    7  FockeWulf_Fw202C_Con           1 - 0              5    0.30    0.70
+++++-++++++++++++++-+-

Confusion matrix :

Truecat =         0    1
Predcat : 0      10    1
Predcat : 1       2   13

Kappa value =  0.7661
Precision (%) by category :
0     90.9091
1     86.6667
Recall (%) by category :
0     83.3333
1     92.8571

cases = 26
cases with unseen category labels = 0
hits =  23
percent hits = 88.46
Hedges's g (z-gap) between strengths of right & wrong answers = 0.7886

Resultant rule from all training cases :
(Rule size = 12 )
training data : c:\beagling\takeoff\aircraft_dat1.dat
creation date : Wed Aug 10 14:08:11 2016
tabular
26 16
(fighting > bombing)
['0', '1']
[0.5358983848622454, 0.4641016151377546]
$
$! $! ( ( cannons * ( country ? `germany`) ) ; ( name ? `awa`) )
( 13.5217427 - wingspan )
$
00  [38, 1]
01  [4, 23]
10  [1, 4]
11  [1, 5]
[0.7878787878787878, -6.0]
$

[.... parameter dump omitted to save space ....]
```

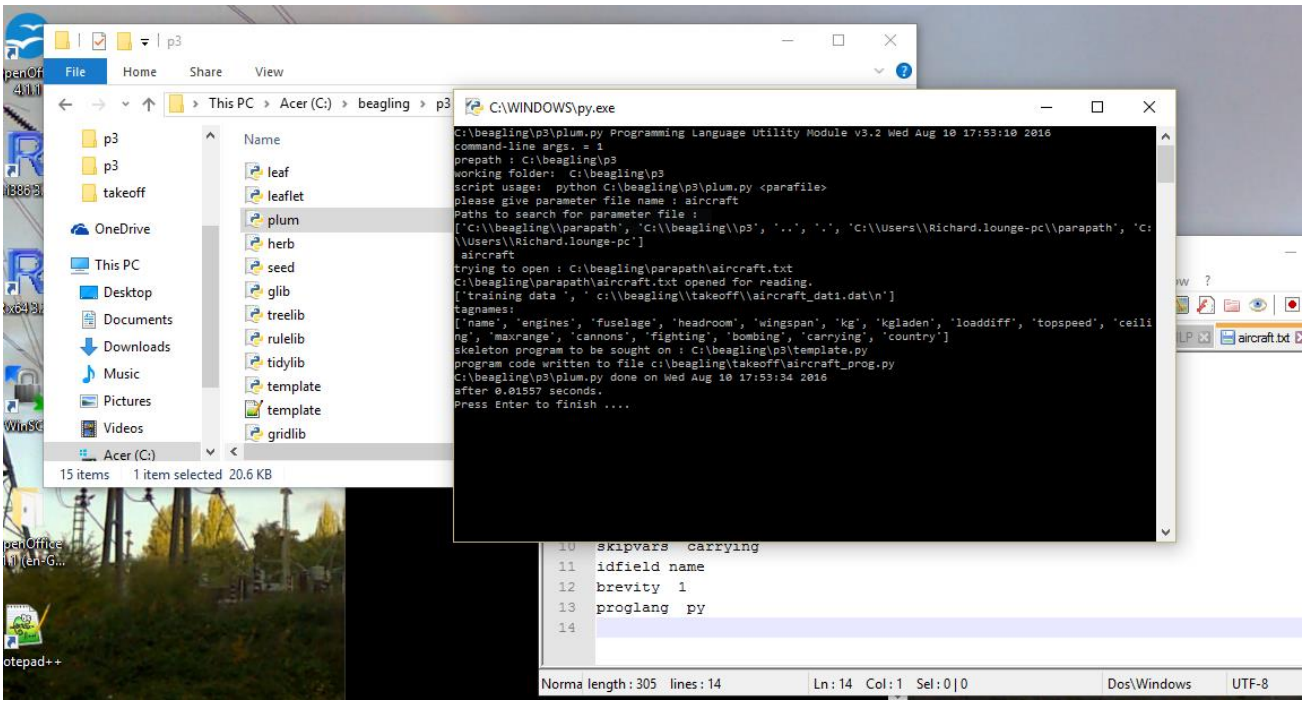
This output is in a similar format to that of the aircraft_list.txt file produced by HERB, so they can easily be compared. On these genuinely unseen cases LEAF gets 23 out of 26 correct (88.46% success) which is slightly better than the success rate projected from HERB's subsampling.

One of its mistakes was assigning the Bristol Beaufighter, a 2-engined fighter developed from a bomber (the Bristol Beaufort), with a broad wingspan of 17.63 metres, to the class of non-fighters. Another was classifying the Focke-Wulf Fw190, a fighter-bomber, as a pure fighter. The third mistake was classifying the Focke-Wulf Condor as a pure fighter: it was a long-range German bomber with cannons, so it fell into the sparsely populated '10' row of the signature table.

C5. Running PLUM

Running plum.py with this dataset will produce an output file containing an executable translation of HERB's ruleset (aircraft_rule.txt), in Python3 since the value of proglang in the parameter file is "py". A sample

screen shot is shown below.



The program output file (aircraft_prog.py) is listed below.

```
##  Using BEAGLE Py template, version of 28/07/2016 :
##  rule written by plum.py ;
##  derived from training data : c:\beagling\takeoff\aircraft_dat1.dat;
##  generated on creation date : Wed Aug 10 14:08:11 2016;
##  dumped on Wed Aug 10 17:53:34 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 beag_stabprep ():
       ##  sets up fallout table :
```

```
        stab = {}
        stab['00'] = [38, 1]
        stab['01'] = [4, 23]
        stab['10'] = [1, 4]
        stab['11'] = [1, 5]

        ##  unpacks stab lines.

        return (stab)
        ##  stabprep ends.


def beag_decrule (vals,stab):
        ##  input vals should be an object with appropriate attribute names.
        ##  target : (fighting > bombing)
        ##  rule mode is tabular.

        bins = ['0','1']  ##  omit if demonic
        catlist = ['0', '1']
        priorvec = [0.5358983848622454, 0.4641016151377546]
        subrules = 2
        rule = [0] * 2
        ##  compute rule values :
        rule[0] = ((beag_exor((vals.cannons * ("germany" in vals.country)),("awa" in
vals.name))<= 0)<= 0)
        rule[1] = (13.5217427 - vals.wingspan)

        p = 0 ; b = []
        while (p < subrules):
                v = (rule[p] > 0) + 0  ##  ensure numeric, omit if demonic
                b.append(bins[v])  ##  boolean string, omit if demonic
                p = p + 1  ##  early-r, late-py

        b = ''.join(b)  ##  omit if demonic
        ##  retrieve cell frequencies :
        frex = stab[b]
        cellsize = sum(frex)
        ##  attenuate frex :
        cats = len(priorvec)
        slug = beag_gold / cats
        modfrex = [slug + f for f in frex]
        ##  defer to Reverend Bayes :
        postvec = [priorvec[j] * modfrex[j] for j in range(cats)]
        psum = sum(postvec)
        postvec = [p / psum for p in postvec]  ##  rescale to total 1.
        m = max(postvec) ; pc = postvec.index(m)
        predcat = catlist[pc]

        return ([b,predcat,frex,postvec])
##  decision rule ends.

##  ending.
```

To make use of this software in a Python3 program, you would need to call the function beag_stabprep() once, to set up the signature table, as in

```
        sigtab = beag_stabprep()
```

and then call beag_decrule(,) for each instance to be classified, as in

```
        reslist = beag_decrule(vals,sigtab)
```

where vals should be a Python object with attributes cannons, country, name and wingspan. In this case, the attributes cannons and wingspan ought to be numeric and the values of attributes country and name should

be strings, otherwise Python will raise an error. In other words this software assumes that the attributes used in the rules have the same types as in the training data.

C6. Running LEAFLET

Running leaflet.py is like running leaf.py except that it reads the translated ruleset written by plum.py (aircraft_prog.py in this case) instead of the BEAGLE ruleset produced by herb.py (aircraft_rule.txt in this case). It doesn't apply when R is the selected programming language. The main output file (aircraft_prop.txt in this case) should be identical to that produced by LEAF, except that the ruleset will be listed in its Python form. If the results differ between LEAF and LEAFLET something has gone wrong! With this example, reassuringly, they didn't differ, so the output isn't reproduced here. Comparing the two listing files is left as an exercise for the reader.

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, 11+5, at least 4 potential classification variables]
World-War-II military aeroplanes, as from Collins/Jane's WWII Aircraft (1999).

Banknote [206, 6+2, 2 cats]
Forged versus genuine Swiss banknotes (Flury & Riedwyl, 1988).

Cardiac [113, 19+1, 2 cats]
Sample data from Afifi & Azen (1979) on heart-attack patients in L.A.

Digidat [1024, 11+2, 10 cats]
Recreation of faulty light-emitting diode display data, as in example by Breiman et al. (1984).

Dogs [77, 10+2, 5 cats]
Mandible measurements of living & prehistoric Thai canines, as from Manly (1994).

Echo [201, 60+1, 2 cats]
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, various]
Information on chemical elements (Wikipedia periodic table).

Glasses [214, 9+1, 7 cats]
Evett & Spiehler's (1988) forensic glass identification example data.
https://archive.ics.uci.edu/ml/datasets/Glass+Identification

Iris [150, 4+2, 3 cats]
Fisher's (1936) Iris data. (Originator: Anderson, E. (1935). The Irises of the Gaspe peninsula.)
http://en.wikipedia.org/wiki/Iris_flower_data_set

Natflags [200, 29+1, many possible choices of x & y vars]
Information about nations & their flags.

Rand [256, 15+1, no genuine cats (2 with "coin" as a test?)]
Pure random data (as null case) to test overfitting avoidance.

Roos [101, 19+1, 3 cats]
Kangaroo skull measurements (Andrews & Herzberg, 1985).

Seed [210, 7+1, 3 cats]
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, 7+1, 2 cats]
Measurements on 2 types of vole, from Flury & Riedwyl (1988).

Wine [178, 13+1, 3 cats]
Chemical measurements as predictors of type of Italian wine. Source: Forina et al.
https://archive.ics.uci.edu/ml/datasets/Wine

Zoobase [101, 16+2, 7 cats]
Zoological classification data as from Forsyth (1990).