

The SNOB program (Vanilla version)

Copyright C.S.Wallace
23 Aug 2002, 3 Dec 2002, 21 Feb 2002

March 26, 2004

1 Overview

Snob is a program for clustering, that is, for discovering class structure in a multivariate population given data on a sample of things which are assumed to be a random sample from the population. Given such a sample, Snob tries to find a population model in which the statistical distribution of things is modeled as the union of some number of class distributions. The model of a single class is simple, modeling the distribution of each variate with a simple unimodal statistical distribution. The model for the whole population is a weighted sum of the class distributions, the weight given to each class being the estimated relative abundance of members of that class in the population.

Snob estimates the number of classes, the relative abundance of each class, the distribution parameters for each variate within each class, and the class to which each thing in the sample most probably belongs. In so doing, it uses an inference principle known as Minimum Message Length (MML). In this application, the principal implies that the population model is used to provide a concise encoding of the data in a message which first specifies the model (number of classes, distribution parameters etc.), then specifies the class to which each thing is assumed to belong, and finally encodes the data for each thing using a code based on the statistical distribution of the class to which the thing has been assigned. The “best” model for the population is that which minimizes the total length of this message.

The MML principle has an inherent trade-off between complexity of model and fit to the data. Hence, no further principle or “significance test” is needed to choose the appropriate number of classes.

2 License, Copyright, Reference, Availability

Snob-vanilla is released under the Gnu Public License (GPL) and copyright 2002 Chris Wallace. See the file `gpl.txt` for details.

Published or otherwise recorded work using Snob should please cite Wallace and Dowe 1986 [2] and Wallace and Boulton 1968 [1].

This and other versions of Snob are maintained at and available from the Monash Data Mining Centre in the School of Computer Science and Software Engineering.

www.datamining.monash.edu.au/~software/snob

3 History

The original version of Snob was written by David Boulton and Chris Wallace in 1968 in Algol [1]. The original proved successful in applications to do with the species of fur seals, the diagnosis of clinical depression, the ecology of Scottish heathers, and diverse other data. However, it gave an inconsistent method for estimation, since it totally (rather than probabilistically) assigned things to classes.

A hierarchical version was later developed by Boulton, again in Algol, but has been less widely used.

A Fortran version including some algorithmic improvements was developed by Wallace, and extended by David Dowe, over the late 70s and 80s. In particular, Wallace 1986 [2], 1990 [3], and Wallace and

Dowe 1994 [4] correct this, with Wallace and Dowe (1994) also introducing Poisson and von Mises circular distributions.

The present “Vanilla” version is written in C, and has minor algorithmic and interface improvements over the Fortran version. It was written by Wallace in July and August 2002. It includes only Gaussian and multistate distributions. It has slightly different file formats as well.

4 Data (general)

The input data required by Snob is basically a set of records, one for each thing or case in the data sample. Each record contains values for each of several variables as measured or observed for that thing. The same set of variables appears for each thing. Thus, the sample data is essentially just a rectangular table of values with a row (i.e. record) for each thing and a column for each variable.

Not every value need be known for each thing: Snob provides for “missing” values in the table.

4.1 Variable types

The Vanilla version currently provides for just two types of variable, “Continuous” and “Multistate”. I expect to add a third, for angular data, shortly.

Each variable type has a type code.

4.1.1 Continuous data. Type code 1


A “continuous” variable is one that takes real values in a continuum. A value of this type appears as a decimal number with or without a fractional part, e.g. “327”, “-108.2”, “.777”, “.0099”, “0”.

In the Snob model, the distribution of a continuous variable within one class is modeled by a Normal or Gaussian distribution with parameters Mean and Standard Deviation (SD).

As this distribution form is symmetrical, and can cover both negative and positive values, it may be inappropriate for a data variable which is by its nature strictly positive, e.g. mass, height, age since birth etc. It may be preferable to transform values of such a variable by replacing them by their logarithms before submitting to Snob. However, previous experience suggests that this transformation makes little difference to the class structure found, unless the values cover a wide ratio range.

Multiplicative and/or additive scaling of continuous values, if done uniformly to the same variable for all things, does not affect the structure found by Snob. However, as the mean and SD estimates for a class distribution are printed by Snob to 6 decimal places, prescaling to bring data values to a typical magnitude not less than 0.1 and not more than 1000 is advisable.

Precision: Snob takes note of the precision with which continuous data has been recorded. This precision has a small effect on the results, but may become important if some discovered class has, for some variable, a SD not much bigger than the precision quantum. The precision quantum for a real-valued variable is called ‘eps’, and must be stated in the data file as described later. Some examples may clarify the matter. Suppose that a data variable is “bank account balance”, that the data values involved range between \$1000000 credit and \$300000 debt, and balances have been recorded only to the nearest \$100. Then the limitations of the Snob output formats suggests that the data values be scaled to units of \$10,000, so the data values will then range from +100 to -30. The ‘eps’ for the balance variable is then 0.01, because 0.01 times \$10,000 is \$100. Suppose another variable is (a person’s) age, which is to be used untransformed, and has been recorded as an integer number of years. Then ‘eps’ for this variable is 1 (or 1.0), representing a quantum of one year.

Note that the eps quantum shows the precision with which the data values are stated, not their accuracy. For instance, systolic blood pressures are normally recorded as an integer number (of mm of Hg) but are rarely reproducible to better than ± 3 mm. The precision of the data is then 1, even though the likely error in each value is of order 3. ‘eps’ should be entered as 1 or 1.0. 

4.1.2 Multistate variables. Type code 2

A “multistate” variable can take only a finite number of discrete values. The possible values are termed “states”, and the values must appear in the data as integers ranging from 1 up to the number of states. Different multistate variables may have different numbers of states. For example, the variable “sex”

could have 2 states, 1 = female, 2 = male. The integers representing the values have no arithmetic significance. In particular, the sequence of integers does **not** imply an ordering of the states.

The vanilla version allows up to 50 states for a discrete variable.

Within a class, the statistical distribution of a discrete variable is modeled by a multinomial distribution, i.e. by a probability for each state, the probabilities summing to 1.

5 Input Files

Snob uses four sorts of input file. These are Variable-Set files, Sample files, Member files and Population files. Hereafter, these are referred to as **vset**, **samp**, **mrep** and **prep** files. Files of each type must have names ending with the suffixes “.vset”, “.samp”, “.mrep”, or “.prep”.

Snob requires at least a vset file and a samp file to do anything useful. The use of mrep and prep files as input will be described later.

5.1 Variable Set files (vset)

A vset file specifies the name, type, and other essential information of every variable in a sample data set. The format is as follows:

Line 1: A name for the vset. This need not be the same as the name of the vset file but preferably should be the same as the file name less its “.vset” suffix.

Line 2: The number of variables. That is, the number of columns in the data table describing a sample.

Following lines, one for each variable:

<Variable name> <variable type code> <other info depending on variable type>

- For Continuous (type code 1) there is no ‘other info’.
- For Multistate (type code 2) the ‘other info’ is the number of states.

An example vset file is shown below:

```
sd1
  3
A_REAL_VAR 1
A_3-state_multi 2 3
A_binary 2 2
```

Preferably, the file name should be sd1.vset.

5.2 Sample files (samp)

A samp file contains the actual sample data, with some heading information. The format follows:

Line 1: A name for the samp. This need not be the same as the name of the samp file but preferably should be the same as the file name less its “.samp” suffix.

Line 2: The name of the vset describing the variables appearing in the sample data. Note, it is the name of the vset, not the name of the .vset file containing it, which should appear here.

Following lines, one for each variable in the vset which requires some further sample-specific info. These must appear in the same order as the variables are listed in the vset. Some variable types require no sample-specific info, and need not have a line, but it seems a good idea to include a blank line for each such variable.

- Continuous variables each require a real value for ‘eps’, the precision with which values of this variable are known.

- Multistate variables require no sample-specific info.

A line: containing the sample size, i.e. the number of things in the sample. This concludes the *heading* information.

Following lines, one for each record: Each thing in the sample gets a record.

- Each record must begin on a new line, but may occupy several lines.
- Blank lines are ignored.
- Each record begins with an integer thing identifier, which must be unique, but has no arithmetic significance. Identifiers need not be consecutive or in order.
- The record for a thing continues with its data values listed in the same order as in the vset list of corresponding variables. Each record must contain an item for every variable. If the value of a variable is not known for the thing, it must still appear in the record as a “**missing value**” flag.
- Items in a record are separated by one or more space, tab or newline characters.

5.2.1 Missing Values

A missing data value appears in a samp file as a “missing value” flag. This is a string of one or more consecutive '=' characters. Whatever the number of '=' characters, the string represents a single missing value. ⚠

5.2.2 Example Sample file (using the example vset “sd1” above)

```
sd1dat sd1 0.1

300

2000 28.12 2 2
2001 43.71 1 2
2002 ===== 2 1
3003 61.50 3 2
3004 51.09 3 =
2005 20.98 2 2
2006 16.14 3 2
3007 56.70 3 1
1008 -8.83 3 2
1009 -2.02 1 1
3010 53.73 2 1
3011 38.10 1 2
etc etc.
```

6 Internal Storage of Vsets, Samples and Models

The present version has internal storage for just one variable set, which will be used throughout a run.

Snob can store up to 10 samples, which must all use the same vset.

It can hold up to 15 population models (which are called poplms or models interchangeably, I fear). These may have been found using different samples, but several different models can be held for and applied to the same sample. A model trained on one sample may be applied to another sample.

At any time during the run, one sample will be the “current” sample, and until another is selected from among the stored samples, the current one will be used in all operations.

Stored models have names. At any time, one model will be current, and will be used in all operations until another is picked. **The current model is named “work”**, and may be thought of as an evolving model of the current sample. If a different named model is picked from among the stored models, the

current “work” model is lost and replaced by a copy of the picked model, which is unaltered. The current “work” model may be copied, with a new name, into the store of models, overwriting any previous stored model of the same name. This leaves the “work” model unchanged and still current. ⚠

Besides “work”, there are three special model names.

- “TrialPop” is a name used for a temporary model employed in some operations. This name should not be used for any other model.
- “BST_<samplename>” is a reserved name used automatically for the best model found so far for the named sample. There may be several “BST_...” models automatically made and stored, one for each sample which has been used.
- “BSTP<samplename>” is a name applied to a model which was (in this or a previous run) generated as a “BST_...” model, recorded on a file, and in this run read back into model storage.

7 Running Snob

The Snob executable program is designed to be run in one of two different ways.

7.1 Direct execution

If the program is started as a normal terminal-started program, it will, after a delay of a second or two, output a message like:

```
Enter variable-set file name:  
There being no comms file, input will be taken from StdInput
```

At this point, the user should input the name of the vset file to be employed in the run. (the “.vset” suffix may be omitted.) Thereafter, Snob will prompt for a sample file name and then (if all goes well) will prompt for commands directing its actions.

If this direct execution mode is to be used, before starting Snob **ensure** that there is no file around called “comms”. Otherwise Snob will try to read from that file, possibly waiting forever. ⚠

7.2 Indirect execution

This version of Snob has a crude but platform-independent way to execute in the background, allowing you to interrupt its search without losing its state.¹ Snob can run in the background and get its input from a file, while a separate program (“pro”) captures keyboard input and puts it in the file.

In this mode of use, first, **remove** any file named “comms”. Then start the **snob-vanilla** executable as a background program, and immediately start another program called “pro” (compiled from source file “prompt.c”).

These three steps are best done via a command script. For Unix-like systems, the script file “go” will do the job. It is shown below:

```
rm comms  
snob-vanilla &  
pro  
rm comms
```

Indirect execution is thus initiated just by entering “go”. When so started, Snob will output the message:

```
Enter variable-set file name:
```

¹If there is a better way that is still platform-independent, please let us know.

and then carry on as for direct execution.

(For Windows-like systems, just start two command-line windows in the same directory, run `snob-vanilla` in one, and `pro` in the other.)

The advantage of indirect execution is that it allows Snob to be interrupted in the middle of a time-consuming operation without losing state. It will stop what it is doing if a new command or a blank line is entered from the terminal, and ask for a new command.

The disadvantage of indirect execution is that Snob may take a second or two to respond to each command. Also, some catastrophic failures of Snob may leave the 'pro' program still running, in which case `pro` has to be killed off before another run. The prompt formatting may also be messier.

8 Classes, Leaves, Subclasses, POP and TC

Some confusing nomenclature has crept into Snob from earlier versions. In the present vanilla version, a model comprises some number of class models, only one if Snob has found no evidence for distinct classes, or possibly quite a number. These classes together represent the model for the sample. They are called 'leaves' and are the principal result of a Snob run.

However, Snob maintains in association with the "work" model several other class models. These are not really part of the model, but are used in the search for a better model.

"POP" is a class model for the whole sample. That is, the distribution of each variable over the entire sample is modeled as a single, simple statistical distribution. This class contains all things, and does not change unless a different sample is selected. POP is shown in a printout of 'classes' because the overall distribution of variables may be of interest, but it is not a leaf of the model.

Subclasses are class models which are not part of the model. Each leaf of the model may have a pair of subclasses, which represent a *potential* split of the leaf into two. If Snob determines that such a split is worthwhile, it will replace the leaf by its two subclasses, which then become true leaves of the work model.

TC is a class model which, like subclasses, represents a potential leaf, not a true leaf of the sample model. It represents the potential union of two true leaves into a single leaf. If Snob determines that such a union is worthwhile, it will replace the two leaves by TC, which becomes a new leaf. There is at most one TC class in the "work" model.

The properties of subclasses and TC may be printed out just as for true leaves, but usually are of little interest.

The word "class" is used generally to refer to all class models, not just the true leaves. Leaves are referred to as leaves.

9 Class Serial Numbers

Every class has a serial number.

POP is serial 0 always.

Leaf serials are allocated from 1 upwards as leaves are created in the search for a good model. As leaves may be destroyed in this search, e.g. by being replaced by subclasses, the set of leaf serials in the current work model is not usually consecutive.

The potential combination class TC has a serial allocated as if it were a leaf. As most TC classes are discarded as not worthwhile, and replaced by the potential union of a different pair of leaves, successive TC classes will often have different serials.

The serials of the two subclasses of a leaf whose serial is S are shown as Sa and Sb. Thus if leaf 22 has subclasses, these have serials 22a and 22b.

10 Commands (General)

When a run is started, Snob first asks for the file name of a vset file, then for the file name of a sample file. God willing, it will accept these files, and construct an initial "work" model comprising a single leaf. For example:

```

> snob-vanilla
Enter variable-set file name:
There being no comms file, input will be taken from StdInput
sd1.vset
Readvset returns 0
Enter sample file name:
sd1dat.samp

```

It will show this model in a little summary, showing the existence of the POP class and a leaf with serial 1. For example:

```

Number of active cases = 300
Begin sort of 300 cases
Finished sort
Readsample returns 0
Allocated space      7704 chars
AaaA
Popln  1 on sample  1,   1 leaves,   300 things  Cost      15.89

  Assign mode Partial    --- Adjust: Params Tree
POP    0  RelAb 0.998  Size   300.0
      1  RelAb 1.000  Size   300.0
aa
Firstpop returns 0
Allocated space      9252 chars

S#     0   POP Age#    3  SampSz#   300.0  RelAb 1.000  Sz    300.0
Pcost   17.53  Tcost  2585.09  Total  2602.62

```

For obscure reasons, it will then output a line “??? line 0” and then expect to receive a command.

Snob commands all begin with a **lower-case** key word, and may then need some parameters. If just the key word is entered and parameter(s) are needed, Snob will prompt with a brief description of the command and ask for the parameters.

Unintelligible commands or parameters will usually result in Snob’s writing “???” and expecting another command.

Command keywords may be abbreviated to a prefix, and Snob will warn of ambiguity.

10.1 Command Summary

Whenever Snob is run, it writes a file called Snob.Menu containing a summary of all commands in alphabetic order. As this is produced by Snob itself, it may be more accurate and up-to-date than these notes.

As Snob.Menu is only 2 pages long, I strongly suggest that a user print it out on paper.

10.2 Help

The command “help” needs a command keyword as parameter, and will output a brief description of the command. “help help” will list all keys in alphabetic order.

At present, “help” may be abbreviated to “h”, and “h h” will list all commands.

10.3 Automatic Search Commands

When Snob is first started by reading in a vset and samp, the “work” model is set up with a single leaf. Work is also left with a single leaf after a new sample is selected.

Snob can be asked to search for a better model by modifying work whether or not work is in this initial one-leaf state.

The command to perform an automatic search is “doall”, but the action of doall is subject to some settings which can be changed using the “assign”, “adjust” and “nosubs” commands before issuing doall.

10.3.1 doall <N>

Performs up to N cycles of attempted improvement. A cycle basically consists of assigning things to classes, collecting statistics for each class from the things assigned to it, and then re-estimating the classes’ abundance and variable distribution parameters.

As well as these basic operations, a cycle may:

- Add subclasses to a leaf.
- Destroy subclasses of a leaf if they seem to offer no hope of improvement.
- Split a leaf, i.e. replace it by its subclasses which then become leaves.
- Destroy a leaf which has become too small.
- Choose a pair of leaves to combine to form a TC potential leaf.
- Destroy the TC class if it seems to offer no hope of improvement.
- Combine the two leaves of the TC class, i.e., replace the two leaves by TC, which then becomes a leaf.
- If “work” is better than the stored model “BST_<samplename>” (i.e. has a shorter message length), copies work into BST...

Some of these additional operations are only attempted every 10 cycles or so.

While doall is cycling, it outputs a character on every cycle. If the cycle improves the model, it outputs ‘A’, otherwise ‘a’. Splits and combines are reported.

If doall finds no improvement for about 60 consecutive cycles, it gives up even although it may not have completed N cycles.

If Snob was started in indirect mode, entering a blank line or new command while doall is in progress will stop doall at the end of the current cycle.

10.3.2 assign <c>

Normally, doall allows for the fact that the class of a thing may be uncertain, given the present work model. It then partially assigns the thing several classes, in proportion to its probability of being a member of the class. This is the default assignment mode, and for a given model, leads to the shortest message length.

The assign <c> command allows the assignment mode to be altered. The character <c> selects the new mode.

- “assign p” sets the normal, “Partial” mode.
- “assign m” sets the “most_likely” mode, in which a thing is assigned to the class most likely to contain it. Most_likely mode can lead to inconsistent estimates of class parameters if many things have uncertain membership, so it is not recommended as a permanent setting. However, doing a few cycles in this mode and then reverting to Partial mode may sometimes help Snob to escape from a local sub-optimal minimum.
- “assign r” sets Random mode. In this mode, a thing is assigned to a class selected at random in proportion to the probabilities of its belonging to the different classes. As the assignment is freshly chosen on each cycle, the model will jitter about a bit. The use of Random mode for a number of cycles may help Snob to escape from a local sub-optimal minimum.

Assign settings remain until another assign command is given but may revert to Partial if work is overwritten.

10.3.3 `adjust` <paramstring>

This command changes setting which enable cycles to modify work.

- “`adjust p-`” inhibits changes to class distribution parameters,
- “`adjust p+`” enables them.
- “`adjust t-`” inhibits changes to class structure, in particular, splitting and combining.
- “`adjust t+`” enables them.

Both settings may be set in a single command, e.g. “`adjust t-p+`”. “`adjust a+`” and “`adjust a-`” control both parameter and structure changes.

10.3.4 `nosubs` < N >

“`nosubs 1`” destroys all subclasses and stops the making of new ones. This applies even if the `adjust` setting allows structure changes.

“`nosubs 0`” reverts to the default situation where subclasses may be made provided the `adjust` setting allows.

10.4 Manual Structure Change Commands

These allow alteration to “work” without use of `doall`. In the descriptions following, ‘ S ’ refers to a class serial number, or a serial number followed by ‘ a ’ or ‘ b ’ if a subclass is meant.

`kclass` < S > Destroys the class.

`ksons` < S > Destroys the subclasses of leaf S .

`splitleaf` < S > If leaf S has subclasses, replaces S by its subclasses which become new leaves.

`combine` < $S1$ > < $S2$ > Combines leaves $S1$ and $S2$ into a single leaf.

`flatten` Splits all splittable leaves, then sets NOSUBS.

`ranclass` < N > Replaces all leaves of work by N leaves with randomly selected members.

The classes should have approximately equal sizes, but some may fail to attract enough members to survive.

If `ranclass` is later done again, a different set of classes will be generated, even if N is the same as previously.

`Rmrep` < F > Reads in a file named F .mrep, which must have a format resembling that produced by the “`mrep`” command, which see under the general heading of reporting commands. The file must refer to the current sample.

The effect is to replace work by a new work model with leaves having the serial numbers given in the file, and distribution parameters which are estimates based on the things named in the class beginning with that serial. The new work model is then subjected to one `doall` cycle.

Not all members of the sample need be mentioned in the file. The aim is to set up a model based on the user’s guess as to what might be typical things in suspected classes.

The required format is shown below, “junk” meaning any strings not extending onto a new line.

Line 1: VanillaSnob-Member-Report-File

Line 2: Sample <name of current sample>

Line 3: <number of classes> junk

Lines : Then some number of lines (maybe none) containing any junk not including ‘+’.

Next line: +

Then for each class: a block of lines with the format:

Line 1: Class <class serial number>

Following lines: A list of thing identifiers, each being the identifier of some member of the current sample. The list is terminated by “-1”. This ends the class block.

An example file appears below.

```
VanillaSnob-Member-Report-File
Sample  sd1dat
  3 junk if you like..
+
Class   3:
  1008   1009   1016   1023   1026   1030   1034   1040   1041   1045
  1046   1050   1051   1055   1058   1059   1061   1062   1063   1065
-1
Class   4:
  3087   3093   3095   3096   3101   3102   3103   3104   3107   3109
  3113   3116   3117   3118   3121   3124   3133   3138   3140   3141
-1

Class   1
  2120   2125   2126   2132   2134   2137   2144   2146   2148   2149
  2150   2152   2166   2169   2173   2175   2179   2185   -1
```

Files produced by the `mrep` command are acceptable as input files for `rmrep`, and will more or less reconstruct the model from which the file was produced. However, as the file does not provide for partial assignment of things to classes, the reconstructed model will in general be slightly different from the original.

10.5 Commands about Models

Snob may store several models internally. Each model has a name, and is stored at some “index” or address in the model store. Indexes are chosen by Snob and may be reused.

pops Lists the indices and names of each model in the model store. When a model is to be specified as a parameter in a command, either its name (a character string) or its index may be used.

In command descriptions, $\langle P \rangle$ refers to a model name or index.

kpop $\langle P \rangle$ Destroys model P . Its index becomes available for reuse. Model “work” may not be destroyed. If model “BST_(current sample name)” is destroyed, it is immediately replaced by a copy of work.

copypop $\langle P \rangle$ Copies work to the model store as a new model called P . The stored model will have no subclasses or TC class.

$\langle P \rangle$ may not be “work” or “Trial_Pop”.

pickpop $\langle P \rangle$ Replaces work with a copy of model P , which remains in store. The copy is, of course, named “work”.

The current sample is modeled by the new work, and a message length computed.

save $\langle P \rangle$ Writes a copy of model P to a file named “ P .save”. This is a binary file, and so is not in general transferable among different types of computer or operating system. The advantage of ‘save’ as compared to ‘prep’ (which reports a model description in a readable text file) is that the binary .save file holds an exact representation of the model, whereas a .prep file has parameters etc. rounded to a modest number of decimal digits.

If P is “work”, save requires a second parameter which will be used as the name of the filed model instead of “work”. The form then is “save work $\langle newname \rangle$ ” and the file will be called *newname.save*.

If P begins with “BST_” (or is the index of a model with such a name) the filed model will be called “BSTP...” and its file name will use this modified name. The purpose is that the model can then be restored in this or a later run without overwriting a possibly better model then in store.

rsave $< F >$ Tries to read a file named “ F .save”. If successful, installs the saved model in the model store at the lowest vacant index. Any previous stored model named F is destroyed. work is not affected.

rprep $< F >$ Tries to read a file from a “.prep” file (see command “**prep**” below). If successful, installs a model whose name is taken from the file, and is normally F .

The file defines the model in terms of its number of leaves, the sizes of each leaf, and the distribution parameters of each variable within each leaf. “**rprep**” does not use all the information in a .prep file as produced by the command “**prep**”. The fields it actually uses are the first two lines (a heading file and the model name), and then such fields as are preceded by a ‘#’ character in the .prep file. Other fields are ignored.

It is fairly easy to prepare manually an acceptable .prep file. This allows a user to suggest a classification in terms of its number of classes, their relative abundances, and the parameters of their variable distributions.

A manually-prepared .prep file should begin with the same text lines as would be found in a file made by the “**prep**” command.

Line 1: “VanillaSnob-Model”

Line 2: $< model\ name >$, preferably the same as the file name without .prep. After these 2 lines, **rprep** only reads fields which are preceded by a ‘#’ character. Other fields which are produced by “**prep**” are ignored. A manual .prep file can thus continue simply as under:

Line 3: # $< variable\ set\ name >$

Line 4: # $< sample\ name >$, which will be ignored. Use “???”.

Line 5: # $< sample\ size >$ # $< cost >$ (These numbers must be present, but will have no meaning in a manual .prep file.)

Line 6: # $< number\ of\ leaf\ classes >$

Then follow sections, one for each leaf class: Command “**prep**” begins with a section for the POP ‘class’, but omit this, and include only sections for leaves. The section for a leaf begins with a line:

7a: # $< serial\ number >$ # $< age >$ # $< size >$

Then the section for a leaf continues with a record for each variable. All variables in the variable-set must be included in the same order. Each record begins:

7b # $< 's' \text{ or } 'I' >$

(‘s’ indicates the variable is significant, ‘I’ that it isn’t.)

7c # $< sample\ size >$

(If this value is less than 2.0, or is greater than the ‘class size’, it is replaced by the class size. It is used to indicate that following parameter values have uncertainties of order $(1 / \sqrt{sample\ size})$.)

7d The rest of the record lists the distribution parameters for the variable distribution within the class:

7d (continuous): # $< Mean >$ # $< Standard\ Deviation >$

(for a continuous variable)

7d (multistate) # $< prob\ 1 >$ # $< prob\ 2 >$... # $< prob\ n >$

(For a multistate with n states.) These probabilities will be normalized by “**rprep**”.

The serial numbers of classes must be integers at least 1, all distinct but not necessarily consecutive or in order.

The age is ignored (taken as 2).

The sizes determine the relative abundances of the various classes, and must all be at least 1.0. They also determine the supposed accuracy of the parameter values, small sizes implying wide uncertainty.

When the model is applied to a sample, a POP class is added to the model with parameters estimated from the sample. These parameters, and the input class 'sizes', will affect the 'Parameter Cost' (Pcost) ascribed to the model, so the model may show different Pcosts when applied to different samples. None of this is important if the model is just used as the starting point for a model search using "doall".

Of course, if "rprep" reads a .prep file produced by the "prep" command, the presence of a POP class (serial 0) as the first class section causes Snob to take the leaf ages, class sizes, variable samplesizes etc. seriously, and it does not fiddle them.

10.6 Commands about Samples

samps This command lists the names and indices of the samples held in the sample store. One of these will be the current sample to which work applies.

readsample $\langle F \rangle$ Reads a sample from file " $F.samp$ " and installs it in the sample store at the lowest available index. The sample name is taken from the first line of the file, and is not necessarily $\langle F \rangle$.

The vset used for the sample must be the current vset.

The work model is unaffected, and the current sample unchanged.

select $\langle D \rangle$ Command **select** $\langle D \rangle$ requires D to be either the name or the index of some sample in the sample store. If D is the current sample, the command says so and does nothing.

Otherwise, D is selected as the current sample, and work is replaced by a one-leaf model of the new sample, or, if a BST_ model is known, by it.

Just in case, the old work model is copied to the model store with the name OldWork.

As often the use of select will be to see how the selected sample is modeled by an existing model, select turns off adjustment of class parameters and model structure.

10.7 Commands for Reporting

sum Just shows the current classes.

prclass $\langle S \rangle \langle 0 \text{ or } 1 \rangle$ **prclass** prints the properties of class serial S , briefly if 0 and giving the parameters for each variable if 1. If the latter, a "significance" indication is shown for each variable. This shows if the distribution of the variable within class S differs significantly from its distribution in POP, and if so, a percentage significance level which needs to be taken with a grain of salt.

If $S = -1$, POP and all leaves are reported. If $S = -2$, subclasses and TC are included.

prep $\langle P \rangle$ Writes a description of model P onto a file named $P.prep$. The description names the current sample then in effect does a "**prclass** -1 1" report writing to the file.

If P is "work", prep requires a second parameter which will be used as the name of the filed model instead of "work". The form then is "**prep** work $\langle newname \rangle$ " and the file will be called $newname.prep$.

A .prep file may be read back in by the "rprep" command and will store a new model with the name being taken from the .prep file, overwriting any previous model of the same name. A .prep file may be prepared manually using some text editor, containing a suggested population model. The required format need not follow exactly that of files produced by "prep". See the "rprep" command.

mrep <*F*> Writes to a file named *F.mrep*. It shows the current sample name, the number of leaves, the leaf serials and sizes, and then lists for each leaf the identifiers of those things assigned to the leaf by Most_Likely assignment. The thing list for each leaf is terminated by -1.

A .mrep file may be read in later provided the current sample is the one for which the file was written, and will then replace work by a rough reconstruction of the work model at the time it was written. See the “**rmrep**” command.

trep <*F*> Writes to a file named *F.trep*. It shows the current sample name, number of leaves, leaf serials and sizes, and then writes a line for each thing in the sample, in identifier order. The line for a thing gives its identifier, then the serials of up to 5 leaves to which the thing might belong, in order of decreasing probability.

The probability follows each leaf serial as a percentage in brackets. Probabilities greater than 98.5% are shown as (99). Leaves with probability less than 0.5% are not shown.

A .trep file cannot be used as input.

crosstab <*P*> Displays a cross-tabulation or contingency table between work and stored model *P*. The table has a row for each leaf of work and a column for each leaf of *P*.

A table entry for leaf serial *Sw* of work and leaf serial *Sp* of *P* shows the permillage² of all active things which are in both leaves. Crosstab <work> is also meaningful. Here, an entry for leaves *S1*, *S2* shows the permillage of things which are partially assigned to both classes.

tree This command draws a (possibly multi-rooted) binary tree with the leaf classes of work as its leaves. The tree is based on a sort of Bhattacharyya Coefficient of similarity among the leaf distributions. The classic Bhattacharyya Coefficient between two probability distributions $f(x)$ and $g(x)$ over a discrete set X , where ‘ x ’ ranges over the members of X , is defined as

$$BC(f, g) = \sum_{x \in X} \sqrt{f(x) \cdot g(x)} \quad (1)$$

$BC(f, g) = 1$ iff $f(x) = g(x)$ for all x , and is zero iff $f(x) * g(x) = 0$ for all x .

To build the tree, each leaf is regarded as defining a distribution over the things in the current sample. In these distributions, the relative abundances of the classes are ignored, so if $f(x)$ is the distribution for class *F*, $f(x)$ is the probability that a random instance of a thing in class *F* would have the attribute values of thing x , but the distribution is normalized so that the sum of $f(x)$ over all things in the sample is one.

For every pair of classes, the BC is calculated. Then the pair of classes with the highest BC is joined together in the tree, and effectively replaced by a “parent” class. The distribution defined by the (size-weighted) union of the pair is then used to calculate the BC coefficient between the parent and all remaining unjoined classes. The new lowest BC is then found, and that pair (which may or may not include the new parent) is joined.

This process continues until either only one class remains, or no pair of remaining classes has a BC greater than zero (to double-precision floating point accuracy). The remaining class(es) form the root(s) of the hierarchy of leaves and parents.

Sample output:

```
# tree

Table of class similarities
SER      7      8      10      11      12
  8 0.20310
 10 2.4e-05 0.34549
 11 4.1e-04 0.51563 0.65246
```

²Like percentage. “Rate per thousand; an amount reckoned as so much in the thousand,” as in, “1900 Fortn. Rev. Jan. 62 It should, perhaps, be remarked that I have reduced Dr. Galton’s results to permillages.” –Ed., OED

```

12 1.2e-09 0.04868 0.00159 6.9e-04
13 2.1e-07 0.14094 0.18114 0.07729 0.05320

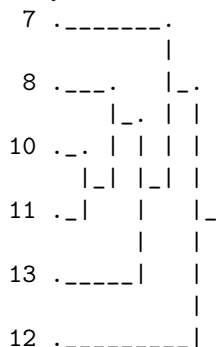
```

```

Join Leaf 10 and Leaf 11 into dad 1 at sim 6.525e-01
Join Leaf 8 and Dad 1 into dad 2 at sim 5.090e-01
Join Dad 2 and Leaf 13 into dad 3 at sim 1.407e-01
Join Leaf 7 and Dad 3 into dad 4 at sim 5.548e-02
Join Dad 4 and Leaf 12 into dad 5 at sim 3.279e-02

```

Binary tree(s) of classes. There are 1 roots



```

>> Cycle      340 Pop 1      6 leaves Cost 53412.6
>> Sample mcdat
>> Adjust PT Assign P

```

10.8 Run Control Commands

file $\langle F \rangle$ After a **file** F command, Snob will read F to obtain future commands. As each command from F is started, it is displayed preceded by **===**.

The file F may contain (sensibly only as its last line) a “**file**” command, which will switch the source of commands to the named file. The named file may be F itself, in which case the commands in F will be repeated over and over. This looping is actually useful. Snob’s automatic search command “**doall**” is not guaranteed to find the global optimum model, so a loop of commands such as :

```

doall 200
save BST_
ranclass 3
file xxx

```

in a file named xxx will generate a variety of models, saving the best in a file named BSTP $\langle \text{samplename} \rangle$.save More elaborate loops, using Random assignment mode for some **doalls** and using several **ranclass** numbers, may be used to advantage.

If any command in F fails, or if the end of F is reached without another file command, command source reverts to normal keyboard input.

If Snob was started in indirect mode, typing any command or a blank line while Snob is busy in a file of commands will cause Snob to revert to normal input after completing its current action or a **doall** cycle.

stop stop stops a run. This command cannot be abbreviated.

10.9 The run log

Snob writes a file called run.log on which it records every command obeyed. The format is such that if the run.log file is renamed or copied to, say, “timber”, then a new run in which the first command

entered is “file timber” should reproduce the original run. Useful in reproducing crashes.

11 References

- [1] C.S. Wallace and D.M. Boulton. 1968. An information measure for classification, *Computer Journal*, 11:2, pp.185-194.
- [2] C.S. Wallace. 1990. Classification by Minimum-Message-Length Inference, S.G. Akl *et al* (eds.) *Advances in Computing and Information - ICCI'90*, Niagara Falls, LNCS 468, Springer-Verlag, pp.72-81.
- [3] C.S. Wallace and D.L. Dowe. 1994. Intrinsic classification by MML - the Snob program, *Proc. 7th Australian Joint Conference on Artificial Intelligence* (UNE, Armidale, NSW, Australia, November 1994), World Scientific, pp.37-44.
- [4] C.S. Wallace and D.L. Dowe. 1996. MML Mixture Modelling of Multi-State, Poisson, von Mises Circular and Gaussian Distributions. *Proc. Sydney International Statistical Congress (SISC-96)*, Sydney, Australia, July. p.197.
- [5] C.S. Wallace and D.L. Dowe. 1997. MML mixture modelling of multi-state, Poisson, von Mises circular and Gaussian distributions, *Proc. 6th International Workshop on Artificial Intelligence and Statistics*, Ft. Lauderdale, Florida, U.S.A., 4-7 Jan., pp.529-536.