7

VarScreen Manual

This chapter contains a user's manual for the *VarScreen* program. Of necessity there is some duplication of text that appears earlier in the book. The reason is that I want to make the manual available for free download, and as such it needs to be self-contained, with every algorithm explained in sufficient detail to allow the user to understand how to perform experiments and understand the output. It would be unfair for me to force users of the program to buy the book in order to understand proper operation.

About the VarScreen Program

VarScreen contains in one easy-to-use program a variety of software tools useful for the developer of predictive models. These tools screen and evaluate candidates for predictors and targets. More on this later. But first, we need to issue a vitally important disclaimer:

This program is an experimental work in progress. It is provided free of charge to interested users for educational purposes only. In all likelihood this program contains errors and omissions. If you use this program for a purpose in which loss is possible, then you are fully responsible for any and all losses associated with use of this program. The developer of this program disclaims all responsibility for losses which the user may incur.

Okay, enough of that. You've been warned. The *VarScreen* program is being developed with two major goals in mind:

1) The program should be exceptionally easy to learn and use. Results should be obtainable with no more than a few intuitive mouse clicks and key presses. Detailed study of an exhaustive manual should not be required.

2) The software should provide cutting-edge statistical information, employing tests and algorithms not readily available in any standard analysis software.

I believe that these goals have been and will continue to be obtained.

Finally, understand that VarScreen is a work in progress. New screening algorithms will likely be added on a regular basis. Stay tuned. Updates will be reported on the author's website: TimothyMasters.info.

Features of the Program

In keeping with the goals of simplicity plus mathematical sophistication, the following items are noteworthy:

- Most operations involve just two quick steps: read the data and select the test to be performed. Program-supplied defaults are often satisfactory, and adjusting them is easy. The next section will describe reading the data, and subsequent sections will describe the tests that can be performed.

- The program is fully multi-threaded, enabling it to take maximum advantage of modern multiple-core processors. As of this writing, many over-the-counter computers contain a CPU with at least six cores, each of which is hyperthreaded to perform two sets of operation streams simultaneously. *VarScreen* keeps all of these threads busy as much as possible, which tremendously speeds operation compared to single-threaded programs.

- The most massively compute-intensive algorithms make use of any CUDA-enabled nVidia card in the user's computer. These widely available video cards (standard hardware on many computers) turn an ordinary desktop computer into a super-computer, accelerating computations by several orders of magnitude. Enormously complex algorithms that would require days of compute time on an ordinary computer with ordinary software can execute in several minutes using the *VarScreen* program on a computer with a modern nVidia display card.

- Rather than printing results on the screen, the program writes a log file called VARSCREEN.LOG. This way a 'permanent' copy of all results is available for optional printing and archiving.

About CUDA Processing

CUDA stands for Compute Unified Device Architecture. It is the interface system by which nVidia makes the massive parallel processing hardware of its video display cards available to applications. The power of this hardware is breathtaking; the GTX Titan video card contains nearly 3000 processors that can execute programs simultaneously. *VarScreen* makes use of this capability for especially time-consuming tasks.

There is an annoying quirk, however, which users of *VarScreen* should be aware of. Microsoft, in its infinite wisdom, forbids any Windows program from executing a CUDA application for longer than two seconds. Moreover, Windows makes it almost impossible for most users to increase or disable this limit; doing so involves tampering with the Registry, a frightening endeavor. Unfortunately, some large problems can require far more than two seconds of CUDA time.

In order to get around this issue, *VarScreen* breaks up large tasks into multiple small tasks. Each such task is called a *Launch*. An ugly tradeoff is involved in this breakup. Each launch incurs a significant overhead, so one should minimize the number of launches. On the other hand, increasing the workload of each launch increases the probability that the deadly two-second limit will be reached, with the result that Windows terminates the program, and somewhere, behind some closed door, a Microsoft programmer snickers. Due to the large variety of CUDA hardware available, it is not practical to predict in advance how long a launch will tie up the CUDA processing, so one must be conservative.

The reason I am making such an issue of this is to allow the user of *VarScreen* to understand a bit of output written to the screen by some algorithms. Whenever an unusually large test involving CUDA computations is running, a progress bar is displayed. This bar also includes text similar to the following:

```
Max CUDA time = 23 ms in 2 launches
```

What this means is that each task had to be broken up into two launches, and the maximum CUDA processing time for those two launches was 23 milliseconds. There is one reason why this may be important to the user: if the time approaches 2000 (two seconds) you are near crashing (a brief black screen followed by a message that the video card has been reset). I would be grateful if you contacted me at my email: tim@TimothyMasters.info and reported this so I can continue to tweak the program.

Reading a Dataset

VarScreen reads data files that are in a common data format: the first record names the fields, and each subsequent record is a single case. For example, the first few lines of a dataset might look like this:

```
X1 X2 X3 Y
3.14 0.21 -5.33 4.01
-1.02 -0.45 2.12 -7.02
...
```

Variable names may be at most 15 characters long. Spaces, commas, and tabs may be used as delimiters. One implication of this fact is that variable names must not contain spaces. In place of a space, the underscore character (_) may be used. Numeric values must be strictly numeric; scientific notation (i.e. 3.14e-9) is illegal in the current version of the program. If users scream loudly enough, this feature may be added later.

Files exported from Microsoft Excel as comma-delimited (.CSV) files are generally readable by *VarScreen*, although if dates with slashes appear, or other text fields appear, trouble may be encountered. (Text variables or otherwise non-numeric fields will typically be assigned the value 0.0.) If exporting from Excel, also beware of column headers that contain spaces. CSV files strictly use commas as delimiters, so spaces in column names are legal in Excel, but since *VarScreen* treats spaces as delimiters, the single variable name in Excel will be mistakenly treated as two or more variables in *VarScreen* if the name contains spaces.

Missing data is not allowed; every data record must have a numeric value present for every field. Note that if a file exported from Excel contains missing data, this will be represented in the file as contiguous commas, which will cause problems for *VarScreen*.

After the file is read, the log file VARSCREEN.LOG will contain a table of the mean and standard deviation of every variable in the file. Users should get in the habit of skimming this table as a quick sanity check of the validity of the data; a wild value in the table may indicate an unexpected flaw in the data file.

One additional variable is computed: _SEQNUM_. For each case this is the sequence number of the case within the dataset. The value of _SEQNUM_ is 1 for the first case, 2 for the second, and so forth. One interesting use for this variable arises when the data is a time series. A relationship such as mutual information between _SEQNUM_ and a variable indicates that the variable is probably nonstationary.

Univariate Mutual Information

The Univariate Mutual Information test computes the mutual information between a specified target variable and each of a specified set of predictor candidates. The predictors are then listed in the VARSCREEN.LOG file in descending order of mutual information. Along with each candidate, a specialized probability described later, as well as the *Solo pval* and *Unbiased pval*, are printed if Monte-Carlo replications are requested.

The *Solo pval* is the probability that a candidate that has a strictly random (no predictive power) relationship with the target could have, by sheer good luck, had a mutual information at least as high as that obtained. If this quantity is not small, the developer should strongly suspect that the candidate is worthless for predicting the target. Of course, this logic is, in a sense, accepting a null hypothesis, which is well known to be a dangerous practice. However, if a reasonable number of cases are present and a reasonable number of Monte-Carlo replications have been done, this test is powerful enough that failure to achieve a small p-value can be interpreted as the candidate having little or no predictive power.

The problem with the *Solo pval* is that if more than one candidate is tested (the usual situation!), then there is a large probability that some truly worthless candidate will be lucky enough to achieve a high level of mutual information, and hence achieve a very small *Solo pva*l. In fact, if all candidates are worthless, the *Solo pvals* will follow a uniform distribution, frequently obtaining small values by random chance. This situation can be remedied by conducting a more advanced test which accounts for this selection bias. The *Unbiased pval* for the best performer in the candidate set is the probability that this best performer could have attained its exalted level of performance by sheer luck if all candidates were truly worthless.

The *Unbiased pval* is printed for all candidates, not just the best. For those other, lesser candidates, the *Unbiased pva*l is an upper bound (a conservative measure) for the true unbiased p-value of the candidate. Thus, a very small *Unbiased pval* for any candidate is a strong indication that the candidate has true predictive power. Unfortunately, unlike the *Solo pva*l, large values of the *Unbiased pval* are not necessarily evidence that the candidate is worthless. Large values, especially near the bottom of the sorted list, may be due to over-estimation of the true p-value. The author is not aware of any algorithm for computing correct unbiased p-values for any candidate other than the best. However, because this measure is conservative, it does have great utility in selecting promising predictors.

The user must be aware of a vital caveat to this procedure: The *Solo pval* and *Unbiased pval* computations fall apart if there is significant serial correlation (or any other dependency) among both the target variable as well as one or more of the predictor candidates. In most practical applications, the predictor candidates are hopelessly dependent, so the key is the target variable. If it has anything beyond tiny dependency (typically serial correlation), the test will become anti-conservative: the computed p-values will be smaller than the correct values. This is dangerous. *VarScreen* contains a *cyclic permutation* option that somewhat helps in this situation, but it is not a complete cure.

The final column printed is inspired by a research report titled "The Probability of Backtest Overfitting" by David Bailey, Jonathan Borwein, Marcos Lopez de Prado, and Jim Zhu. Like the permutation test, it assumes that there is no significant serial correlation among both the target variable and one or more predictor candidates, although it tends to be fairly robust in this regard. I heavily modified their clever algorithm to apply to mutual information.

When one examines a pool of candidates and selects a predictor based on its having the maximum value of some criterion such as mutual information, one hopes that this superiority will carry over to data not yet seen (out-of-sample or OOS data). In particular, consider the (unknown at test time) median OOS performance of all predictor candidates. At a minimum, one would hope that the OOS performance of the candidate selected based on its having maximum in-sample performance would exceed the median OOS performance of all candidates. If not, the selection process is useless; no superiority is obtained by choosing the best in-sample performer.

The rightmost figure printed for the first row (the best candidate) is the estimated probability that the OOS mutual information of this selected candidate will be less than or equal to the median OOS performance of all of the other candidates. Obviously, we want this probability to be small.

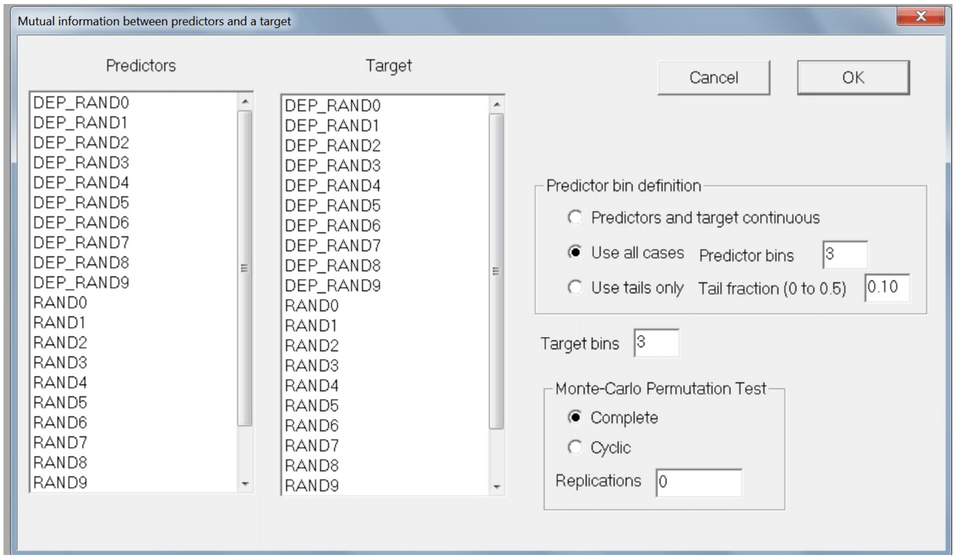
The figures printed for subsequent rows are the equivalent probabilities for lower rank orders. For example, the figure for the second row is the probability that the second best in-sample candidate will have OOS performance less than or equal to the median. This is subtly different from the probability for the particular candidate that was selected; it's a more theoretical figure. Nonetheless, equating the two should not be unreasonable.

Ideally, one would see low probabilities near the top (the best in-sample candidates outperform OOS) and high probabilities near the bottom (the worst underperform OOS). A large quantity of worthless candidates will make the distribution more random.

The algorithms in this mutual information section are described in detail, including source code, in my book "Data Mining Algorithms in C++" published by the Apress division of Springer.

Specifying the Test Parameters

When the user clicks Tests / Univariate Mutual Information, a dialog similar to that shown below will appear. The various parameters are described on the following page.


Mutual information between predictors and a target

Predictors: DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8, RAND9

Target: DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8, RAND9

Predictor bin definition
- ○ Predictors and target continuous
- ● Use all cases — Predictor bins: 3
- ○ Use tails only — Tail fraction (0 to 0.5): 0.10

Target bins: 3

Monte-Carlo Permutation Test
- ● Complete
- ○ Cyclic

Replications: 0

The leftmost column is used to specify the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to select a single target variable.

Three methods for computing mutual information are available, and the method to use is chosen by selecting one of the three buttons in the Predictor bin definition block:

Predictors and target continuous uses the Darbellay-Vajda algorithm (fully described in "Assessing and Improving Prediction and Classification" by Timothy Masters) to compute continuous mutual information. This method is appropriate (and almost always the preferred approach) when all variables are continuous or nearly so. It's main disadvantage is that it is much slower to compute than the bin methods. Also, candidates that have tiny mutual information with the target will have their computed mutual information reduced to exactly zero by the algorithm. This will produce a sudden discontinuity in p-values, which may appear unusual but which in fact is perfectly reasonable.

Use all cases partitions each predictor into bins that are as equal in size as possible. The user must specify the number of bins to employ, and unless the dataset is huge the default of three bins is frequently appropriate.

Use tails only computes mutual information based on only the maximum and minimum collection of values of each predictor. The tail fraction specified by the user is the fraction of cases in each tail. So, for example, the default tail fraction of 0.1 would use the cases having the smallest ten percent and the largest ten percent of predictor values. The 80 percent of cases having intermediate values of the predictor candidate would be completely ignored in the mutual information calculation. This method is especially useful in high-noise situations, such as prediction of financial markets. The probability that superior mutual information will hold up out-of-sample cannot be computed when this option is selected.

Target bins must be specified if *Use all cases* or *Use tails only* is chosen. This is the number of approximately equal-size bins into which the target variable is distributed. The default value of 3 is appropriate for a wide variety of applications. This field is ignored if the *Predictors and target continuous* option is selected.

Replications defaults to zero, in which case no Monte-Carlo Permutation Test is performed. However, it is usually best to set this to at least 100, and perhaps as much as 1000, so that solo and unbiased p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutations. So, for example, if the user specifies 100 permutations, the minimum p-value that can appear is 0.01. Run time of this test is linearly related to the number of permutations.

The user must choose either ***Complete*** or ***Cyclic*** permutations. If the user is confident that there is no dependency as described earlier, then *Complete* should be used; it is the traditional approach which does a complete random shuffle for each permutation. However, if there is dependency, this type of shuffling will produce underestimation of p-values, a very dangerous situation. If the dependency is serial (the data is a time series and the dependency is among samples close in time) then a slight improvement in the situation can be obtained by using *Cyclic* permutation. In this type of shuffle, the time order of the target is kept intact except at the ends by rotating the targets with end-point wraparound. Shuffling this way preserves most of the serial dependency in the permutated targets, which makes the algorithm more accurate. The p-values computed this way will generally be larger than those computed with complete shuffling, and hence less likely to lead to false rejection of the null hypothesis of no predictive power. But be warned that the cure is far from complete; computed p-values will still underestimate the true values, just not as badly.

Note that in most cases it is legitimate to use *Cyclic* permutation instead of *Complete* when there is no dependency. However, if the dataset is small, Cyclic permutation will limit the number of unique permutations and hence increase the random error inherent in the process. As long as the dataset is large, some users may prefer to use Cyclic permutation even if it is assumed that there is no serial dependency; in case there really is hidden serial dependency, this is a cheap insurance policy. Still, the best practice is to make sure that the data does not contain dependency and then use Complete permutation. Relying on Cyclic permutation to take care of dependency problems is living dangerously. And if the dataset contains fewer than 1000 or so cases, use of Cyclic permutation is not recommended unless it is necessary to handle dependency.

Examples of Univariate Mutual Information

This section demonstrates three situations, all using synthetic data to clarify the presentation. The variables in the dataset are as follows:

RAND0 - ***RAND9*** are independent (within themselves and with each other) random time series.

DEP_RAND0 - ***DEP_RAND9*** are derived from ***RAND0*** - ***RAND9*** by introducing strong serial correlation up to a lag of nine observations. They are independent of one another.

SUM12 = RAND1 + RAND2

SUM34 = RAND3 + RAND4

SUM1234 = SUM12 + SUM34

The first test run attempts to predict ***SUM1234*** from ***RAND0*** - ***RAND9***, ***SUM12***, and ***SUM34***. The output looks like this:

```
***********************************************************
*                                                         *
* Univariate mutual information (1 predictor, 1 target)   *
*    12 predictor candidates                              *
*     5 predictor bins                                    *
*     5 target bins                                       *
* 10000 replications of complete Permutation Test         *
*                                                         *
***********************************************************


The bounds that define bins are now shown

Target bounds are based on the entire dataset...
    -0.97362      -0.27795       0.31417       1.00879

Variable  Bounds...

   RAND0  -0.59427      -0.18805       0.20723       0.60549
   RAND1  -0.58905      -0.18795       0.22570       0.62047
   RAND2  -0.59430      -0.18090       0.21697       0.61045
   RAND3  -0.62008      -0.20843       0.19894       0.59159
   RAND4  -0.59696      -0.18753       0.21087       0.61077
   RAND5  -0.59819      -0.21468       0.18130       0.56676
   RAND6  -0.61150      -0.21273       0.19102       0.59680
   RAND7  -0.61383      -0.22039       0.18521       0.58843
   RAND8  -0.59055      -0.19032       0.20591       0.59859
   RAND9  -0.60422      -0.19932       0.20315       0.58792
   SUM12  -0.67798      -0.17129       0.22588       0.74242
   SUM34  -0.73810      -0.21209       0.21164       0.74363

The marginal distributions are now shown.
If the data is continuous, the marginals will be nearly
equal.
Widely unequal marginals indicate potentially problematic
ties.

Target marginals are based on the entire dataset...
0.19987     0.20003     0.20003     0.20003     0.20003

Variable    Marginal...

RAND0    0.19987    0.20003    0.20003    0.20003    0.20003
RAND1    0.19987    0.20003    0.20003    0.20003    0.20003
RAND2    0.19987    0.20003    0.20003    0.20003    0.20003
RAND3    0.19987    0.20003    0.20003    0.20003    0.20003
RAND4    0.19987    0.20003    0.20003    0.20003    0.20003
RAND5    0.19987    0.20003    0.20003    0.20003    0.20003
RAND6    0.19987    0.20003    0.20003    0.20003    0.20003
RAND7    0.19987    0.20003    0.20003    0.20003    0.20003
RAND8    0.19987    0.20003    0.20003    0.20003    0.20003
RAND9    0.19987    0.20003    0.20003    0.20003    0.20003
SUM12    0.19987    0.20003    0.20003    0.20003    0.20003
SUM34    0.19987    0.20003    0.20003    0.20003    0.20003

--------> Mutual Information with SUM1234 <--------

Variable      MI      Solo pval Unbiased pval P(<=median)

SUM34       0.2877      0.0001       0.0001   0.0000
SUM12       0.2610      0.0001       0.0001   0.0000
RAND3       0.1307      0.0001       0.0001   0.0000
RAND4       0.1263      0.0001       0.0001   0.0000
RAND1       0.1129      0.0001       0.0001   0.0000
RAND2       0.1085      0.0001       0.0001   0.0000
RAND8       0.0015      0.2994       0.9828   1.0000
RAND5       0.0014      0.3673       0.9950   1.0000
RAND6       0.0012      0.5303       1.0000   1.0000
RAND7       0.0010      0.7384       1.0000   1.0000
RAND0       0.0008      0.8332       1.0000   1.0000
RAND9       0.0006      0.9605       1.0000   1.0000
```

The bounds that define the target and predictor bins are shown, along with the marginal probabilities. If any marginal is far from being equal, that variable has significant ties and the situation should be investigated.

As expected, the best predictors of SUM1234 are SUM12 and SUM34. RAND1 - RAND4 are the next best. All other predictors are obviously worthless. Note how dramatically the unbiased p-value delineates the break.

The next example shows what happens when worthless and serially correlated predictors are tested with a serially correlated target. We use DEP_RAND1 - DEP_RAND9 to predict DEP_RAND0, a situation which should demonstrate no predictive power whatsoever. The mutual information table is as follows:

```
-------> Mutual Information with DEP_RAND0 <-------

Variable       MI      Solo pval  Unbiased pval   P(<=median)

DEP_RAND2   0.0044       0.0001       0.0002        0.6944
DEP_RAND4   0.0030       0.0018       0.0175        0.6190
DEP_RAND3   0.0025       0.0110       0.0881        0.6270
DEP_RAND6   0.0023       0.0249       0.2004        0.5516
DEP_RAND9   0.0023       0.0242       0.2062        0.5397
DEP_RAND8   0.0023       0.0287       0.2284        0.5079
DEP_RAND1   0.0022       0.0317       0.2494        0.4960
DEP_RAND5   0.0019       0.0883       0.5509        0.4325
DEP_RAND7   0.0008       0.8682       1.0000        0.5317
```

The mutual information figures are all tiny, yet the p-values show extreme significance. The careless user would surely be fooled by this, because not only are the solo p-values mostly small, but even the unbiased p-value has been fooled for one or two of the candidates.

It should be emphasized that this phenomenon is not an artifact of just the Monte-Carlo Permutation Test. This is a universal phenomenon, which is why Statistics 101 courses always emphasize the importance of independent observations. The simple explanation of why this occurs is that any sort of dependence reduces the effective degrees of freedom of the test. The testing procedure looks at the number of cases and proceeds accordingly, but the dependence in the data increases the variance of the test statistic beyond what would be expected from a sample of the given size. Thus we are more likely to falsely reject the null hypothesis.

Observe that in this ‘no predictive power' case, despite the serial correlation, the probabilities in the final column are distributed around 0.5, which would be expected when none of the candidates has predictive power. This is because the best in-sample candidate is random, and hence its associated out-of-sample performance has about a 50-50 chance of lying above or below the median. This is the pattern usually seen when all candidates are worthless.

The final example shows how the cyclic modification of the Monte-Carlo Permutation Test can at least partially remedy the situation. We repeat the same test as that just shown, except that instead of using *Complete* permutation we use *Cyclic* permutation. The results are shown below:

```
-------> Mutual Information with DEP_RAND0 <------

Variable        MI      Solo pval  Unbiased pval   P(<=median)

DEP_RAND2   0.0044       0.0513       0.3529        0.6944
DEP_RAND4   0.0030       0.2408       0.9316        0.6190
DEP_RAND3   0.0025       0.3976       0.9918        0.6270
DEP_RAND6   0.0023       0.5007       0.9976        0.5516
DEP_RAND9   0.0023       0.5237       0.9982        0.5397
DEP_RAND8   0.0023       0.4719       0.9988        0.5079
DEP_RAND1   0.0022       0.5344       0.9990        0.4960
DEP_RAND5   0.0019       0.6643       1.0000        0.4325
DEP_RAND7   0.0008       0.9920       1.0000        0.5317
```

Now observe that even the largest random relationship is not significant at the 0.05 level on a solo basis, and the unbiased p-value is far from significant.

Bivariate Mutual Information / Uncertainty Reduction

Sometimes a single variable acting alone has little or no predictive power, but in conjunction with another it becomes useful. The classic example is the height and weight of an individual, predicting coronary health. Either predictor alone has relatively little predictive power, but the two taken together can have great power.

Also, sometimes we have several equally useful candidates for the target variable, and we are not sure which will be most predictable. One example of this situation is when the application is predicting future movement of a financial market with the goal of taking a position and then hopefully closing the position with a profit. Should we employ a tight stop to discourage severe losses? Or should we use a loose stop to avoid being closed out by random noise? We might test multiple targets corresponding to various degrees of stop positioning, and then determine which of the competitors is most predictable.

The *Bivariate Mutual Information* test handles both of these situations. It computes the mutual information or uncertainty reduction between each of one or more specified target variables and each possible pair of predictors taken from a specified set of predictor candidates. The predictor pairs and associated targets are then listed in the VARSCREEN.LOG file in descending order of mutual information. Along with each such set, the *Solo pval* and *Unbiased pval* are printed if Monte-Carlo replications are requested.

The *Solo pval* is the probability that a pair of candidates that has a strictly random (no predictive power) relationship with the target could have, by sheer good luck, had a relationship at least as high as that obtained. If this quantity is not small, the developer should strongly suspect that the candidate is worthless for predicting the target. Of course, this logic is, in a sense, accepting a null hypothesis, which is well known to be a dangerous practice. However, if a reasonable number of cases are present and a reasonable number of Monte-Carlo replications have been done, this test is powerful enough that failure to achieve a small p-value can be interpreted as the candidate having little or no predictive power.

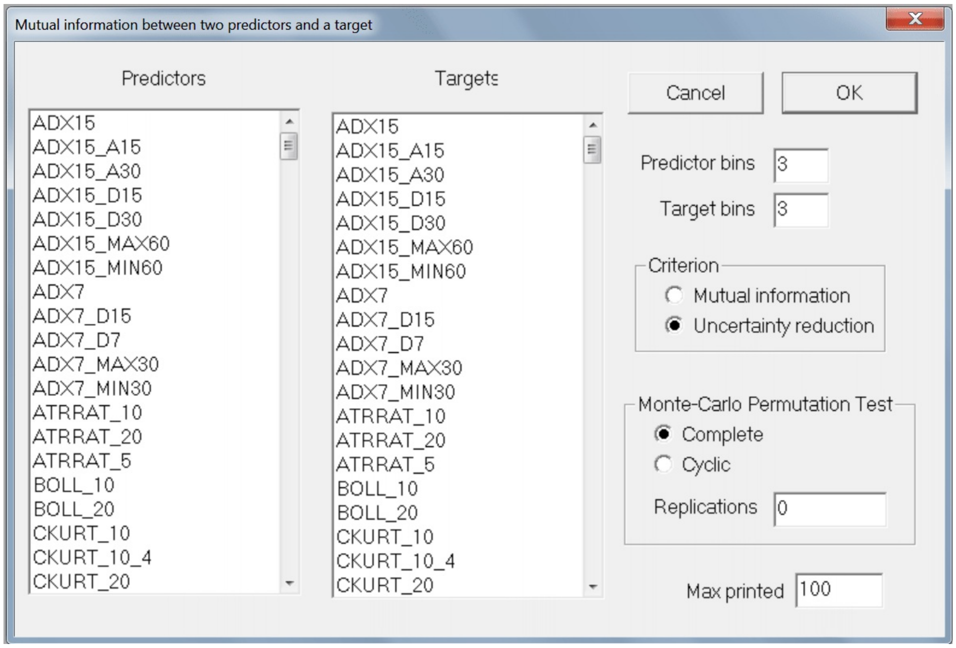
The problem with the *Solo pval* is that if more than one candidate set (a set being two predictors and a target) is tested (the usual situation!), then there is a large probability that some truly worthless candidate set will be lucky enough to achieve a high level of the relationship criterion, and hence achieve a very small *Solo pval*. In fact, if all candidate sets are worthless, the Solo pvals will follow a uniform distribution, frequently obtaining small values by random chance. This situation can be remedied by conducting a more advanced test which accounts for this selection bias. The *Unbiased pval* for the best performing candidate set is the probability that this best performer could have attained its exalted level of performance by sheer luck if all candidate sets were truly worthless.

The *Unbiased pval* is printed for all candidate sets, not just the best. For those other, lesser candidates, the *Unbiased pval* is an upper bound (a conservative measure) for the true unbiased p-value of the candidate set. Thus, a very small *Unbiased pval* for any candidate set is a strong indication that the pair of predictors has true predictive power for the target. Unfortunately, unlike the *Solo pval*, large values of the Unbiased pval are not necessarily evidence that the candidate set is worthless. Large values, especially near the bottom of the sorted list, may be due to over-estimation of the true p-value. The author is not aware of any algorithm for computing correct unbiased p-values for any candidate set other than the best. However, because this measure is conservative, it does have great utility in selecting promising predictors.

The user must be aware of a vital caveat to this procedure: The *Solo pval* and *Unbiased pval* computations fall apart if there is significant serial correlation (or any other dependency) among one or more target variables as well as one or more of the predictor candidates. In most practical applications, the predictor candidates are hopelessly dependent, so the key is the target variable. If it has anything beyond tiny dependency (typically serial correlation), the test will become anti-conservative: the computed p-values will be smaller than the correct values. This is dangerous. *VarScreen* contains a cyclic permutation option that somewhat helps in this situation, but it is not a complete cure.

Specifying the Test Parameters

When the user clicks Tests / Bivariate Mutual Information, a dialog similar to that shown below will appear. The various parameters are described after the dialog.


Mutual information between two predictors and a target
Predictors
Targets
Cancel
OK
Criterion
Monte-Carlo Permutation Test

Predictors: ADX15, ADX15_A15, ADX15_A30, ADX15_D15, ADX15_D30, ADX15_MAX60, ADX15_MIN60, ADX7, ADX7_D15, ADX7_D7, ADX7_MAX30, ADX7_MIN30, ATRRAT_10, ATRRAT_20, ATRRAT_5, BOLL_10, BOLL_20, CKURT_10, CKURT_10_4, CKURT_20

Targets: ADX15, ADX15_A15, ADX15_A30, ADX15_D15, ADX15_D30, ADX15_MAX60, ADX15_MIN60, ADX7, ADX7_D15, ADX7_D7, ADX7_MAX30, ADX7_MIN30, ATRRAT_10, ATRRAT_20, ATRRAT_5, BOLL_10, BOLL_20, CKURT_10, CKURT_10_4, CKURT_20

Predictor bins 3

Target bins 3

Criterion: Mutual information [✓ Uncertainty reduction]

Monte-Carlo Permutation Test: [✓ Complete] Cyclic

Replications 0

Max printed 100

The leftmost column is used to specify the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to select one or more target variables, with multiple selections obtained as described for predictors.

The predictors and the targets are partitioned into bins that are as equal in size as possible. The user must specify the number of bins to employ for each, and unless the dataset is huge the default of three bins is frequently appropriate.

There can be an annoying problem when using mutual information as a measure of relationship when more than one target is in competition. Mutual information is highly related to the entropy of the predictor and target. If there is only one target in play, the mutual information between it and each predictor candidate will have the same rank order as the uncertainty reduction. But if there are several targets in competition and they have widely disparate entropies, then mutual information is not a good measure of their relationship because the target entropies can confound the rank ordering.

What you are really interested in is the degree to which uncertainty about a target is reduced by having knowledge of a predictor. It can be thought of as their mutual information divided by the entropy of the target. Equivalently, it is the fraction of the target's entropy which is mutual information. For example, if they have zero mutual information, there will be zero uncertainty reduction (about the target) by knowing the predictor. At the other extreme, if their mutual information equals the target entropy, then knowing the predictor will provide perfect (1.0) uncertainty reduction regarding the target.

Thus, a target with high entropy will need high mutual information in order to have a high relationship score. For this reason, uncertainty reduction is the default for this test. Much more detail on this important concept can be found in "Assessing and Improving Prediction and Classification" by Timothy Masters.

Replications defaults to zero, in which case no Monte-Carlo Permutation Test is performed. However, it is usually best to set this to at least 100, and perhaps as much as 1000, so that solo and unbiased p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutations. So, for example, if the user specifies 100 permutations, the minimum p-value that can appear is 0.01. Run time of this test is linearly related to the number of permutations.

The user must choose either *Complete* or *Cyclic* permutations. If the user is confident that there is no dependency as described earlier, then *Complete* should be used; it is the traditional approach which does a complete random shuffle for each permutation. However, if there is dependency, this type of shuffling will produce underestimation of p-values, a very dangerous situation. If the dependency is serial (the data is a time series and the dependency is among samples close in time) then a slight improvement in the situation can be obtained by using *Cyclic* permutation. In this type of shuffle, the time order of the target is kept intact except at the ends by rotating the targets with end-point wraparound. Shuffling this way preserves most of the serial dependency in the permutated targets, which makes the algorithm more accurate. The p-values computed this way will generally be larger than those computed with complete shuffling, and hence less likely to lead to false rejection of the null hypothesis of no predictive power. But be warned that the cure is far from complete; computed p-values will still underestimate the true values, just not as badly.

Note that in most cases it is legitimate to use *Cyclic* permutation instead of *Complete* when there is no dependency. However, if the dataset is small, *Cyclic* permutation will limit the number of unique permutations and hence increase the random error inherent in the process. As long as the dataset is large, some users may prefer to use *Cyclic* permutation even if it is assumed that there is no serial dependency; in case there really is hidden serial dependency, this is a cheap insurance policy. Still, the best practice is to make sure that the data does not contain dependency and then use *Complete* permutation. Relying on *Cyclic* permutation to take care of dependency problems is living dangerously. And if the dataset contains fewer than 1000 or so cases, use of *Cyclic* permutation is not recommended unless it is necessary to handle dependency.

Examples of Bivariate Mutual Information

This section demonstrates three situations, all using synthetic data to clarify the presentation. The variables in the dataset are as follows:

RAND0* - *RAND9 are independent (within themselves and with each other) random time series.

DEP_RAND0* - *DEP_RAND9 are derived from ***RAND0* - *RAND9*** by introducing strong serial correlation up to a lag of nine observations. They are independent of one another.

SUM12 = RAND1 + RAND2

SUM34 = RAND3 + RAND4

SUM1234 = SUM12 + SUM34

The first test run attempts to predict ***SUM1234*** from ***RAND0* - *RAND9***, ***SUM12***, and ***SUM34***. Two predictors at a time will be used. The output is shown below. For bin boundaries and marginals, the predictor candidates are shown first, followed by a single blank line, and then the target candidates (just one in this example) appear.

```
***********************************************************
*                                                         *
* Bivariate mutual information (2 predictors, 1 target)   *
*      12 predictor candidates                            *
*       1 target candidates                               *
*      66 predictor/target combinations to test           *
*     100 best combinations will be printed               *
*       5 predictor bins                                  *
*       5 target bins                                     *
*   10000 replications of complete Permutation Test       *
*                                                         *
***********************************************************


The bounds that define bins are now shown

RAND0     -0.59427      -0.18805       0.20723       0.60549
RAND1     -0.58905      -0.18795       0.22570       0.62047
RAND2     -0.59430      -0.18090       0.21697       0.61045
RAND3     -0.62008      -0.20843       0.19894       0.59159
RAND4     -0.59696      -0.18753       0.21087       0.61077
```

```
RAND5     -0.59819      -0.21468       0.18130       0.56676
RAND6     -0.61150      -0.21273       0.19102       0.59680
RAND7     -0.61383      -0.22039       0.18521       0.58843
RAND8     -0.59055      -0.19032       0.20591       0.59859
RAND9     -0.60422      -0.19932       0.20315       0.58792
SUM12     -0.67798      -0.17129       0.22588       0.74242
SUM34     -0.73810      -0.21209       0.21164       0.74363
SUM1234   -0.97362      -0.27795       0.31417       1.00879

The marginal distributions are now shown.
If the data is continuous, the marginals will be nearly
equal.
Widely unequal marginals indicate potentially problematic
ties.

   RAND0   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND1   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND2   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND3   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND4   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND5   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND6   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND7   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND8   0.19987   0.20003   0.20003   0.20003   0.20003
   RAND9   0.19987   0.20003   0.20003   0.20003   0.20003
   SUM12   0.19987   0.20003   0.20003   0.20003   0.20003
   SUM34   0.19987   0.20003   0.20003   0.20003   0.20003

 SUM1234   0.19987   0.20003   0.20003   0.20003   0.20003


------------------> Mutual Information <-----------------

Pred 1      Pred 2    Target     MI  Solo pval  Unbiased pval

SUM12      SUM34      SUM1234   1.0781   0.0001   0.0001
RAND1      SUM34      SUM1234   0.5363   0.0001   0.0001
RAND3      SUM12      SUM1234   0.5356   0.0001   0.0001
RAND2      SUM34      SUM1234   0.5333   0.0001   0.0001
RAND4      SUM12      SUM1234   0.5242   0.0001   0.0001
RAND3      RAND4      SUM1234   0.3094   0.0001   0.0001
RAND3      SUM34      SUM1234   0.2994   0.0001   0.0001
RAND4      SUM34      SUM1234   0.2985   0.0001   0.0001
RAND6      SUM34      SUM1234   0.2947   0.0001   0.0001
RAND9      SUM34      SUM1234   0.2946   0.0001   0.0001
RAND8      SUM34      SUM1234   0.2944   0.0001   0.0001
RAND5      SUM34      SUM1234   0.2939   0.0001   0.0001
RAND0      SUM34      SUM1234   0.2937   0.0001   0.0001
RAND7      SUM34      SUM1234   0.2925   0.0001   0.0001
RAND2      RAND3      SUM1234   0.2881   0.0001   0.0001
RAND1      RAND3      SUM1234   0.2879   0.0001   0.0001
RAND1      RAND4      SUM1234   0.2861   0.0001   0.0001
RAND2      RAND4      SUM1234   0.2811   0.0001   0.0001
RAND1      RAND2      SUM1234   0.2755   0.0001   0.0001
```

```
RAND2      SUM12      SUM1234   0.2709   0.0001   0.0001
RAND1      SUM12      SUM1234   0.2705   0.0001   0.0001
RAND5      SUM12      SUM1234   0.2697   0.0001   0.0001
RAND6      SUM12      SUM1234   0.2692   0.0001   0.0001
RAND0      SUM12      SUM1234   0.2673   0.0001   0.0001
RAND8      SUM12      SUM1234   0.2664   0.0001   0.0001
RAND7      SUM12      SUM1234   0.2661   0.0001   0.0001
RAND9      SUM12      SUM1234   0.2656   0.0001   0.0001
RAND3      RAND7      SUM1234   0.1371   0.0001   0.0001
RAND3      RAND5      SUM1234   0.1369   0.0001   0.0001
RAND3      RAND9      SUM1234   0.1363   0.0001   0.0001
RAND0      RAND3      SUM1234   0.1362   0.0001   0.0001
RAND3      RAND6      SUM1234   0.1361   0.0001   0.0001
RAND3      RAND8      SUM1234   0.1358   0.0001   0.0001
RAND4      RAND6      SUM1234   0.1344   0.0001   0.0001
RAND0      RAND4      SUM1234   0.1341   0.0001   0.0001
RAND4      RAND5      SUM1234   0.1328   0.0001   0.0001
RAND4      RAND9      SUM1234   0.1322   0.0001   0.0001
RAND4      RAND7      SUM1234   0.1321   0.0001   0.0001
RAND4      RAND8      SUM1234   0.1313   0.0001   0.0001
RAND1      RAND6      SUM1234   0.1207   0.0001   0.0001
RAND1      RAND5      SUM1234   0.1205   0.0001   0.0001
RAND1      RAND7      SUM1234   0.1191   0.0001   0.0001
RAND1      RAND9      SUM1234   0.1185   0.0001   0.0001
RAND1      RAND8      SUM1234   0.1183   0.0001   0.0001
RAND0      RAND1      SUM1234   0.1180   0.0001   0.0001
RAND2      RAND5      SUM1234   0.1162   0.0001   0.0001
RAND2      RAND8      SUM1234   0.1154   0.0001   0.0001
RAND2      RAND6      SUM1234   0.1153   0.0001   0.0001
RAND2      RAND7      SUM1234   0.1150   0.0001   0.0001
RAND2      RAND9      SUM1234   0.1144   0.0001   0.0001
RAND0      RAND2      SUM1234   0.1131   0.0001   0.0001
RAND6      RAND7      SUM1234   0.0091   0.0952   0.9775
RAND7      RAND8      SUM1234   0.0090   0.1081   0.9905
RAND0      RAND8      SUM1234   0.0088   0.1563   0.9982
RAND5      RAND9      SUM1234   0.0086   0.1904   0.9994
RAND0      RAND9      SUM1234   0.0084   0.2327   0.9997
RAND5      RAND6      SUM1234   0.0083   0.2549   0.9998
RAND0      RAND5      SUM1234   0.0080   0.3693   1.0000
RAND8      RAND9      SUM1234   0.0079   0.3949   1.0000
RAND0      RAND6      SUM1234   0.0074   0.5647   1.0000
RAND5      RAND8      SUM1234   0.0074   0.5734   1.0000
RAND7      RAND9      SUM1234   0.0074   0.5830   1.0000
RAND0      RAND7      SUM1234   0.0069   0.7550   1.0000
RAND6      RAND8      SUM1234   0.0065   0.8598   1.0000
RAND5      RAND7      SUM1234   0.0064   0.8652   1.0000
RAND6      RAND9      SUM1234   0.0058   0.9657   1.0000
```

It should be no surprise that the best pair of predictors for SUM1234 are SUM12 and SUM34. Mutual information trails off according to how many components of the sum are present. Note the sharp transition in the unbiased p-value when we reach the point of having no component present!

The next example shows what happens when worthless and serially correlated predictors are tested with a serially correlated target. We use ***DEP_RAND1 - DEP_RAND9*** to predict ***DEP_RAND0***, a situation which should demonstrate no predictive power whatsoever. The mutual information table is as follows:

```
----------> Mutual Information with DEP_RAND0 <----------

Predictor 1  Predictor 2  Target     MI   Solo p Unbiased p

DEP_RAND2     DEP_RAND7  DEP_RAND0   0.0159   0.0001   0.0001
DEP_RAND2     DEP_RAND3  DEP_RAND0   0.0145   0.0001   0.0001
DEP_RAND2     DEP_RAND9  DEP_RAND0   0.0138   0.0001   0.0001
DEP_RAND2     DEP_RAND6  DEP_RAND0   0.0132   0.0001   0.0005
DEP_RAND4     DEP_RAND8  DEP_RAND0   0.0132   0.0001   0.0005
DEP_RAND3     DEP_RAND4  DEP_RAND0   0.0132   0.0001   0.0005
DEP_RAND2     DEP_RAND4  DEP_RAND0   0.0132   0.0001   0.0005
DEP_RAND5     DEP_RAND7  DEP_RAND0   0.0131   0.0001   0.0005
DEP_RAND1     DEP_RAND2  DEP_RAND0   0.0131   0.0001   0.0005
DEP_RAND2     DEP_RAND5  DEP_RAND0   0.0129   0.0001   0.0011
DEP_RAND2     DEP_RAND8  DEP_RAND0   0.0129   0.0001   0.0011
DEP_RAND4     DEP_RAND9  DEP_RAND0   0.0127   0.0002   0.0016
DEP_RAND1     DEP_RAND3  DEP_RAND0   0.0125   0.0001   0.0020
DEP_RAND3     DEP_RAND6  DEP_RAND0   0.0125   0.0001   0.0022
DEP_RAND1     DEP_RAND5  DEP_RAND0   0.0123   0.0001   0.0038
DEP_RAND3     DEP_RAND5  DEP_RAND0   0.0122   0.0002   0.0056
DEP_RAND6     DEP_RAND8  DEP_RAND0   0.0121   0.0003   0.0074
DEP_RAND1     DEP_RAND6  DEP_RAND0   0.0117   0.0010   0.0213
DEP_RAND6     DEP_RAND9  DEP_RAND0   0.0115   0.0006   0.0323
DEP_RAND4     DEP_RAND6  DEP_RAND0   0.0110   0.0021   0.0893
DEP_RAND1     DEP_RAND4  DEP_RAND0   0.0110   0.0027   0.0904
DEP_RAND5     DEP_RAND8  DEP_RAND0   0.0110   0.0032   0.0906
DEP_RAND5     DEP_RAND9  DEP_RAND0   0.0108   0.0044   0.1298
DEP_RAND7     DEP_RAND9  DEP_RAND0   0.0108   0.0051   0.1442
DEP_RAND7     DEP_RAND8  DEP_RAND0   0.0107   0.0060   0.1584
DEP_RAND4     DEP_RAND5  DEP_RAND0   0.0107   0.0063   0.1610
DEP_RAND3     DEP_RAND9  DEP_RAND0   0.0107   0.0051   0.1620
DEP_RAND1     DEP_RAND9  DEP_RAND0   0.0104   0.0096   0.2819
DEP_RAND6     DEP_RAND7  DEP_RAND0   0.0103   0.0132   0.3179
DEP_RAND8     DEP_RAND9  DEP_RAND0   0.0102   0.0147   0.3827
DEP_RAND3     DEP_RAND7  DEP_RAND0   0.0101   0.0181   0.4380
DEP_RAND5     DEP_RAND6  DEP_RAND0   0.0099   0.0249   0.5409
DEP_RAND1     DEP_RAND8  DEP_RAND0   0.0098   0.0294   0.5901
DEP_RAND3     DEP_RAND8  DEP_RAND0   0.0097   0.0347   0.6486
DEP_RAND4     DEP_RAND7  DEP_RAND0   0.0087   0.1757   0.9908
DEP_RAND1     DEP_RAND7  DEP_RAND0   0.0084   0.2498   0.9983
```

Notice how many truly worthless predictive pairs have tiny p-values, even in the unbiased case. This is a severe problem that affects all common statistical tests, not just Monte-Carlo Permutation Tests.

The final example shows how the cyclic modification of the Monte-Carlo Permutation Test can at least partially remedy the situation. We repeat the same test as that just shown, except that instead of using *Complete* permutation we use *Cyclic* permutation. The results are shown below:

```
----------> Mutual Information with DEP_RAND0 <----------

Predictor 1  Predictor 2  Target     MI   Solo p Unbiased p
DEP_RAND2     DEP_RAND7  DEP_RAND0   0.0159   0.0261   0.4007
DEP_RAND2     DEP_RAND3  DEP_RAND0   0.0145   0.0813   0.8015
DEP_RAND2     DEP_RAND9  DEP_RAND0   0.0138   0.1404   0.9240
DEP_RAND2     DEP_RAND6  DEP_RAND0   0.0132   0.1968   0.9761
DEP_RAND4     DEP_RAND8  DEP_RAND0   0.0132   0.1660   0.9776
DEP_RAND3     DEP_RAND4  DEP_RAND0   0.0132   0.1859   0.9792
DEP_RAND2     DEP_RAND4  DEP_RAND0   0.0132   0.1768   0.9804
DEP_RAND5     DEP_RAND7  DEP_RAND0   0.0131   0.2354   0.9837
DEP_RAND1     DEP_RAND2  DEP_RAND0   0.0131   0.2077   0.9858
DEP_RAND2     DEP_RAND5  DEP_RAND0   0.0129   0.2329   0.9915
DEP_RAND2     DEP_RAND8  DEP_RAND0   0.0129   0.2162   0.9925
DEP_RAND4     DEP_RAND9  DEP_RAND0   0.0127   0.2594   0.9949
DEP_RAND1     DEP_RAND3  DEP_RAND0   0.0125   0.3104   0.9972
DEP_RAND3     DEP_RAND6  DEP_RAND0   0.0125   0.3243   0.9977
DEP_RAND1     DEP_RAND5  DEP_RAND0   0.0123   0.3545   0.9978
DEP_RAND3     DEP_RAND5  DEP_RAND0   0.0122   0.3621   0.9982
DEP_RAND6     DEP_RAND8  DEP_RAND0   0.0121   0.3613   0.9984
DEP_RAND1     DEP_RAND6  DEP_RAND0   0.0117   0.4874   0.9998
DEP_RAND6     DEP_RAND9  DEP_RAND0   0.0115   0.5108   0.9998
DEP_RAND4     DEP_RAND6  DEP_RAND0   0.0110   0.6064   1.0000
DEP_RAND1     DEP_RAND4  DEP_RAND0   0.0110   0.5907   1.0000
DEP_RAND5     DEP_RAND8  DEP_RAND0   0.0110   0.5737   1.0000
DEP_RAND5     DEP_RAND9  DEP_RAND0   0.0108   0.6308   1.0000
DEP_RAND7     DEP_RAND9  DEP_RAND0   0.0108   0.6902   1.0000
DEP_RAND7     DEP_RAND8  DEP_RAND0   0.0107   0.6681   1.0000
DEP_RAND4     DEP_RAND5  DEP_RAND0   0.0107   0.6274   1.0000
DEP_RAND3     DEP_RAND9  DEP_RAND0   0.0107   0.6552   1.0000
DEP_RAND1     DEP_RAND9  DEP_RAND0   0.0104   0.7349   1.0000
DEP_RAND6     DEP_RAND7  DEP_RAND0   0.0103   0.7587   1.0000
DEP_RAND8     DEP_RAND9  DEP_RAND0   0.0102   0.7330   1.0000
DEP_RAND3     DEP_RAND7  DEP_RAND0   0.0101   0.7944   1.0000
DEP_RAND5     DEP_RAND6  DEP_RAND0   0.0099   0.8103   1.0000
DEP_RAND1     DEP_RAND8  DEP_RAND0   0.0098   0.8036   1.0000
DEP_RAND3     DEP_RAND8  DEP_RAND0   0.0097   0.8085   1.0000
DEP_RAND4     DEP_RAND7  DEP_RAND0   0.0087   0.9581   1.0000
DEP_RAND1     DEP_RAND7  DEP_RAND0   0.0084   0.9731   1.0000
```

This time, the unbiased p-values are not fooled at all by the serial correlation, and even the solo p-values behave well.

Predictors having Max Relevance, Min Redundancy

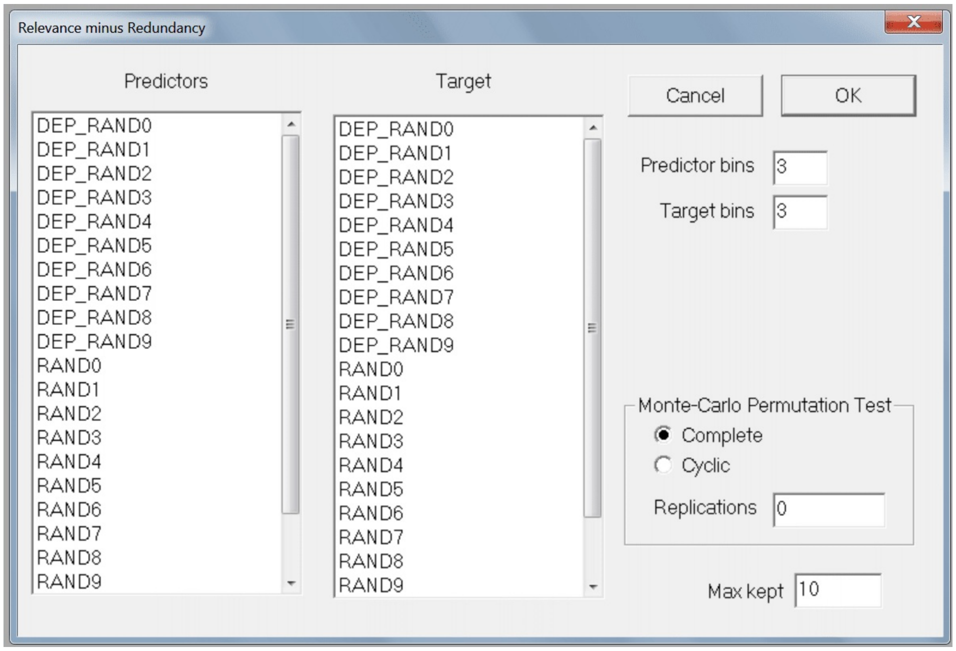
Selection of predictors by examining individual or even pairwise performance is useful for quickly identifying the most promising candidates. However, this simplistic approach suffers from redundancy. If two predictor candidates are nicely related to a target, chances are good that they are also closely related to each other; they may provide similar if not identical predictive information. Thus, if one examines a large number of candidates and chooses a subset of predictors that are all good at predicting the target, this subset will in most cases be unnecessarily large; many of them will provide nearly or exactly the same predictive information as other candidates in the subset. A much more efficient approach to selecting a good subset of predictor candidates would be to consider not only the relevance of the members at predicting the target, but also their redundancy with other members of the subset.

Peng, Long and Ding (2005) provide a fabulous algorithm for handling this redundancy problem in their paper "Feature Selection Based on Mutual Information: Criteria of Max-Dependency, Max-Relevance, and Min Redundancy". An intuitive summary of the algorithm, along with C++ code, appears in my book "Assessing and Improving Prediction and Classification," so details will be omitted here. However, it must be stressed that this algorithm has a powerful optimality property: suppose one were to consider the mutual information between a set of predictors (taken as a group) and a target. This is called joint dependency. A reasonable method for choosing an optimal subset of predictors is to use forward stepwise selection to maximize the joint dependency of the subset with the target. Unfortunately, this quantity is difficult if not impossible to compute in practical applications. But the Pen, Long, and Ding algorithm is an elegant work-around that produces the same subset of predictors as stepwise selection based on maximizing joint dependency, but it does so in a computationally feasible way.

At each step, the algorithm considers the relevance of a candidate for predicting the target, as well as the redundancy of the candidate with predictors already in the chosen subset. These quantities are subtracted to provide a selection criterion. The candidate with the maximum relevance-minus-redundancy criterion is chosen.

Specifying the Test Parameters

When the user clicks Tests / Relevance minus Redundancy, a dialog similar to that shown will appear. The various parameters are described below.



Predictor bins 3

Target bins 3

Monte-Carlo Permutation Test: [✓ Complete] Cyclic

Replications 0

Max kept 10

The leftmost column is used to specify the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to select the target variable.

The predictors and the target are partitioned into bins that are as equal in size as possible. The user must specify the number of bins to employ for each, and unless the dataset is huge the default of three bins is frequently appropriate.

Replications defaults to zero, in which case no Monte-Carlo Permutation Test is performed. However, it is usually best to set this to at least 100, and perhaps as much as 1000, so that solo and group p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutations. So, for example, if the user specifies 100 permutations, the minimum p-value that can appear is 0.01. Run time of this test is linearly related to the number of permutations.

The user must choose either ***Complete*** or ***Cyclic*** permutations. If the user is confident that there is no dependency as described earlier in this document, then *Complete* should be used; it is the traditional approach which does a complete random shuffle for each permutation. However, if there is dependency, this type of shuffling will produce underestimation of p-values, a very dangerous situation. If the dependency is serial (the data is a time series and the dependency is among samples close in time) then a considerable improvement in the situation can be obtained by using *Cyclic* permutation. In this type of shuffle, the time order of the target is kept intact except at the ends by rotating the target with end-point wraparound. Shuffling this way preserves most of the serial dependency in the permutated target, which makes the algorithm more accurate. The p-values computed this way will generally be larger than those computed with complete shuffling, and hence less likely to lead to false rejection of the null hypothesis of no predictive power. But be warned that the cure is far from complete; computed p-values will still underestimate the true values, just not as badly.

Note that in most cases it is legitimate to use *Cyclic* permutation instead of *Complete* when there is no dependency. However, if the dataset is small, *Cyclic* permutation will limit the number of unique permutations and hence increase the random error inherent in the process. As long as the dataset is large, some users may prefer to use *Cyclic* permutation even if it is assumed that there is no serial dependency; in case there really is hidden serial dependency, this is a cheap insurance policy. Still, the best practice is to make sure that the data does not contain dependency and then use *Complete* permutation. Relying on *Cyclic* permutation to take care of dependency problems is living dangerously. And if the dataset contains fewer than 1000 or so cases, use of *Cyclic* permutation is not recommended unless it is necessary to handle dependency.

Max kept is the maximum size of the selected subset. Execution time is approximately linearly related to this quantity, so it should be kept as small as possible if run time is critical.

Note that this algorithm employs CUDA processing if available. However, unless there are many hundreds of predictor candidates, its overhead may actually slow execution.

An Example of Relevance Minus Redundancy

This section demonstrates a revealing example of the algorithm using synthetic data to clarify the presentation. The variables in the dataset are as follows:

RAND0 - ***RAND9*** are independent (within themselves and with each other) random time series.

SUM12 = RAND1 + RAND2

SUM34 = RAND3 + RAND4

SUM1234 = SUM12 + SUM34

The test run attempts to predict ***SUM1234*** from ***RAND0 - RAND9, SUM12***, and ***SUM34***. The output is shown below. Brief explanatory comments are interspersed.

```
***********************************************************
*                                                         *
* Relevance minus redundancy for optimal predictor subset *
*      12 predictor candidates                            *
*      12 best predictors will be printed                 *
*       5 predictor bins                                  *
*       5 target bins                                     *
*     100 replications of complete Permutation Test       *
*                                                         *
***********************************************************
```

```
Initial candidates, in order of decreasing mutual information
with SUM1234

       Variable         MI

          SUM34       0.2877
          SUM12       0.2610
          RAND3       0.1307
          RAND4       0.1263
          RAND1       0.1129
          RAND2       0.1085
          RAND8       0.0015
          RAND5       0.0014
          RAND6       0.0012
          RAND7       0.0010
          RAND0       0.0008
          RAND9       0.0006

Predictors so far   Relevance   Redundancy   Criterion

          SUM34       0.2877       0.0000       0.2877
```

We see from the table above that the first candidate chosen is the one which has maximum mutual information with the target. Naturally this would be either ***SUM12*** or ***SUM34***, and it happens to be the latter. Then, in the table below we see that ***SUM12*** has the largest relevance (its mutual information with the target) and essentially no redundancy with SUM34 (again, no surprise). This gives it the highest selection criterion and it is chosen.

```
Additional candidates, in order of decreasing relevance minus
redundancy

       Variable     Relevance   Redundancy   Criterion

          SUM12       0.2610       0.0014       0.2596
          RAND1       0.1129       0.0016       0.1112
          RAND2       0.1085       0.0009       0.1076
          RAND6       0.0012       0.0007       0.0005
          RAND0       0.0008       0.0009      -0.0000
          RAND8       0.0015       0.0017      -0.0002
          RAND5       0.0014       0.0016      -0.0002
          RAND9       0.0006       0.0008      -0.0002
          RAND7       0.0010       0.0012      -0.0003
          RAND3       0.1307       0.3154      -0.1847
          RAND4       0.1263       0.3158      -0.1895
```

```
Predictors so far   Relevance   Redundancy   Criterion

        SUM34        0.2877       0.0000       0.2877
        SUM12        0.2610       0.0014       0.2596
```

Now we come to an important observation. One might think that the next candidate selected would be either ***RAND1***, ***RAND2***, ***RAND3***, or ***RAND4***, the four components of the ***SUM1234*** target. However, the table on the next page shows that these four candidates actually fall at the bottom of the list! This is because they have so much redundancy with ***SUM12*** and ***SUM34*** (taken as a group) that they will not be chosen next. In fact, ***RAND6***, which has no relationship whatsoever with any of the other variables, is chosen based only on its tiny random relevance and slightly smaller random redundancy.

```
Additional candidates, in order of decreasing relevance minus
redundancy

      Variable     Relevance   Redundancy   Criterion

         RAND6       0.0012       0.0009       0.0003
         RAND0       0.0008       0.0008       0.0000
         RAND8       0.0015       0.0015       0.0000
         RAND9       0.0006       0.0008      -0.0002
         RAND5       0.0014       0.0017      -0.0003
         RAND7       0.0010       0.0013      -0.0004
         RAND3       0.1307       0.1581      -0.0274
         RAND4       0.1263       0.1585      -0.0322
         RAND1       0.1129       0.1527      -0.0398
         RAND2       0.1085       0.1485      -0.0399

Predictors so far   Relevance   Redundancy   Criterion

        SUM34        0.2877       0.0000       0.2877
        SUM12        0.2610       0.0014       0.2596
        RAND6        0.0012       0.0009       0.0003
```

But now that the selected set's redundancy with the remaining candidates has been ‘diluted' by the inclusion of the unrelated ***RAND6***, ***RAND1-RAND4*** jump to the top of the list due to their relatively large relevance but lessened redundancy.

```
Additional candidates, in order of decreasing relevance minus
redundancy

       Variable     Relevance   Redundancy   Criterion

          RAND3       0.1307       0.1058       0.0249
          RAND4       0.1263       0.1061       0.0202
          RAND1       0.1129       0.1021       0.0107
          RAND2       0.1085       0.0995       0.0090
          RAND0       0.0008       0.0010      -0.0002
          RAND9       0.0006       0.0009      -0.0003
          RAND5       0.0014       0.0017      -0.0003
          RAND8       0.0015       0.0018      -0.0004
          RAND7       0.0010       0.0015      -0.0006


Predictors so far   Relevance   Redundancy   Criterion

          SUM34       0.2877       0.0000       0.2877
          SUM12       0.2610       0.0014       0.2596
          RAND6       0.0012       0.0009       0.0003
          RAND3       0.1307       0.1058       0.0249
```

There is little point in continuing to show the inclusion steps. We now jump to the final table that lists all candidates in the order in which they were selected, along with associated p-values.

```
----------> Final results predicting SUM1234 <----------

Preds Relevance Redundancy Criterion  Solo pval  Group pval

SUM34    0.2877    0.0000     0.2877      0.010      0.010
SUM12    0.2610    0.0014     0.2596      0.010      0.010
RAND6    0.0012    0.0009     0.0003      0.570      0.010
RAND3    0.1307    0.1058     0.0249      0.010      0.010
RAND4    0.1263    0.0797     0.0465      0.010      0.010
RAND1    0.1129    0.0617     0.0511      0.010      0.010
RAND2    0.1085    0.0505     0.0581      0.010      0.010
RAND8    0.0015    0.0014     0.0001      0.320      0.010
RAND5    0.0014    0.0014    -0.0001      0.340      0.010
RAND7    0.0010    0.0014    -0.0004      0.650      0.010
RAND0    0.0008    0.0013    -0.0004      0.850      0.010
RAND9    0.0006    0.0012    -0.0006      0.980      0.010
```

Two different p-values are printed for each predictor candidate. The *Solo pval* is the same quantity printed in the Univariate Mutual Information test. This is the probability that, if the predictor has no actual mutual information with the target, a mutual information (Relevance here) as large as that obtained could have occurred. Understand that this quantity considers each candidate in isolation, not involving any other candidates.

Note how nicely this reveals the uselessness of the third candidate chosen, ***RAND6***.

The *Group pval* considers the associated candidate along with every prior candidate. It tests the null hypothesis that the group of candidates selected so far, on average, has no mutual information with the target.

Regrettably, I am not aware of any way of computing what would be an especially useful p-value, that which tests the null hypothesis that selecting the candidate provides no additional (non-redundant) relevance. Such a p-value would be valuable for determining when to stop including additional candidates in the selected subset. The problem appears to be that the test statistic at any step is strongly dependent on the relevance of those predictors already selected. If anyone knows of a way around this problem, I would love to hear about it.

Hidden Markov Models with Target Correlation

When working with time series data, the developer need not assume a direct relationship between predictors and a target. Sometimes it is better to posit an underlying condition, the *state* of the process under study, which impacts both the predictors and the target. This process is assumed to exist at all times in exactly one of two or more possible states. The state at any given time impacts the distribution of associated variables. Some of these variables may be observable (predictors), while others may be unknown but be of great interest (targets). Our goal is to use measured values of the observable variables to determine (or make an educated guess at) the state of the process, and then use this knowledge to estimate the value of an unobservable variable which interests us.

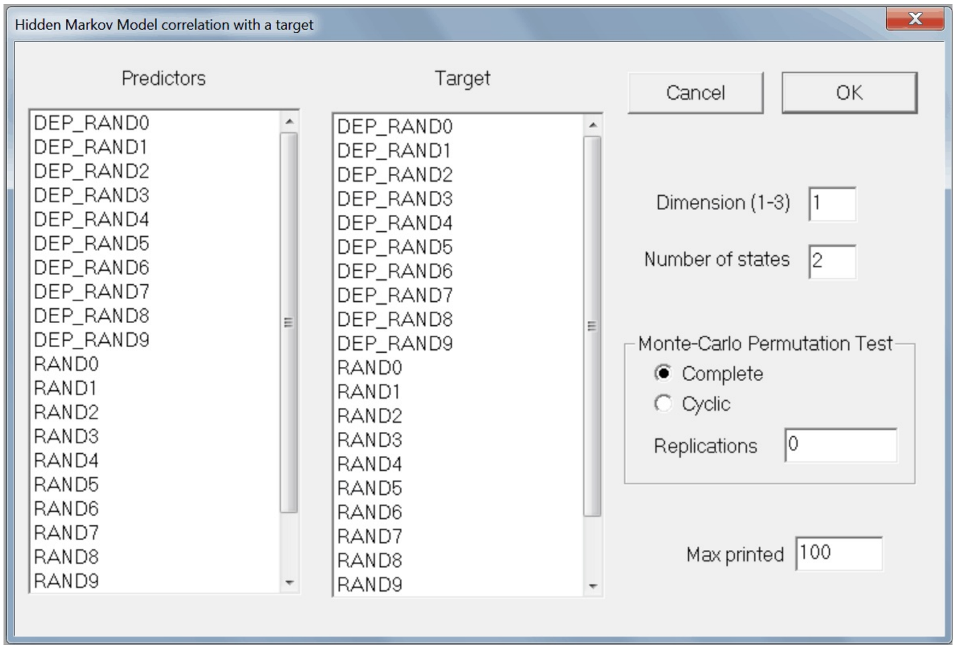
This is different from ordinary classification methods which are not restricted to time series data. In simple classification, one measures some predictor variable(s) and makes a class decision, which in turn implies values of other unmeasurable variables. But a hidden Markov model assumes a sequential process with this property: the probability of being in a given state at an observed time depends on the process's state at the prior observed time. In other words, unlike ordinary classification, a hidden Markov model has memory.

This memory is immensely useful in some applications. For example, it may prevent whipsaws. Suppose a certain state tends to be persistent in real life. Ordinary classification will suffer if there is large random noise in the observed variables, which may snap the decision back and forth at the whim of chance. But the memory inherent in a hidden Markov model will tend to hold its decision in a persistent state even as noise in the measured variables tries to whip the decision back and forth. Of course, the downside of this memory is a tendency toward delayed decisions; the model may need several observed values to confirm a state change. But this is often a price well worth paying, especially in high-noise situations.

One application of a hidden Markov model is the prediction of a financial market. Perhaps the developer assumes that it is always in either a bull market (a long-term up-trend), a bear market (a long-term down-trend) or a flat market (no long-term trend). By definition, bull and bear markets cover an extended time period; one does not go from a bull to a bear market in one day, and then return to a bull market the next day. Such direction changes are just short-term fluctuations in a more extensive move. If one were to use frequent observations to make daily predictions of whether the market is in a bull or bear state, these decisions could reverse ridiculously often. One is better off taking advantage of the memory of a hidden Markov model to stabilize behavior.

Specifying the Test Parameters

When the user clicks *Tests / Hidden Markov model - Target*, a dialog similar to that shown will appear.



Predictors: DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8, RAND9

Target: DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8, RAND9

Dimension (1-3) 1

Number of states 2

Monte-Carlo Permutation Test: [✓ Complete] Cyclic

Replications 0

Max printed 100

The leftmost column is used to specify the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to specify the target variable. This variable is ignored when the models are computed; rather it plays a role in selecting the 'best' model.

The ***Dimension*** must be 1, 2, or 3. This is the number of predictor variables that will be used by the hidden Markov model.

The ***Number of states*** is exactly that, the number of states in which the process can exist. It must be at least two, and it typically is small, rarely more than four. Execution time blows up rapidly as the number of states increases.

The user must choose either ***Complete*** or ***Cyclic*** permutations and the number of replications to perform. Please refer to the discussions of this issue earlier in this document. However, because hidden Markov models virtually always are applied to serially correlated data, cyclic permutation is the default.

Max printed is the maximum number of models printed in the log file.

WARNING... This test can be extremely slow. While threads are being initialized for the first set of models, the ESCape key is ignored. After that, ESCape is polled only at widely spaced intervals. Then, when waiting for the final threads to complete, ESCape is again ignored. For a few thousand cases, 2 dimensions, and 2 states, the complete test should run in a few minutes or less on modern computers. But if there are many thousands of cases, 3 dimensions, and 4 or more states, the test could require several hours to complete. If you get in over your head, you may need to use Task Manger to force a shutdown of the program. Sorry about that, but as of yet I have not been able to figure out an efficient way to interrupt threads that are in the middle of extensive computation without inducing significant overhead, which just makes the situation worse.

Operation of This Test

The *hidden Markov model* test operates in two completely separate steps. In the first step, every possible combination of predictor candidates is used to fit a hidden Markov model. Let *N* be the number of candidates specified by the user (selected from the list in the left column of the dialog). If the dimension is specified to be 1, then each candidate is used alone, resulting in *N* models, one for each candidate. If the dimension is 2, then there are *N**(*N*-1)/2 models, one for each possible pair of candidates. If the dimension is 3, then there are *N**(*N*-1)*(*N*-2)/6 models, one for each possible trio. It must be emphasized that these models are optimized without regard to the target variable; the target plays no role whatsoever in the development of the models.

After this (potentially large!) set of hidden Markov models has been found, the relationship between each of them and the user-specified target variable is found. The relationship between a model and the target is defined as the multiple-R (the multivariate correlation coefficient) between the vector of state probabilities and the target. In other words, for a given model, each case will have associated with it a vector giving the probability that this observation is in each possible state. These state probability vectors are regressed on the target variable using ordinary multiple linear regression.

Details of the best (most highly correlated) model are printed. Then the models (up to *Max printed* of them) are listed in descending order of relationship with the target. The multiple-R is printed for each. If Monte-Carlo replications were specified, solo and unbiased p-values are printed for each model. The *solo p-value* is the probability that, if there were actually no relationship between the state (as defined by that model) and the target, we could have obtained a multiple-R at least as large as we did obtain. The *unbiased p-value* for the best model is the probability that if *none* of the models were related to the target, the best among them would have a multiple-R at least as large as that obtained. Subsequent unbiased p-values are upper bounds on similarly defined probabilities.

Note that exact results will not in general be replicated if runs are repeated. This is because training a hidden Markov model relies on random number generation, and Windows' scheduling of training threads is rarely consistent. The competing models will receive their random numbers in different orders during different runs, resulting in slightly different solutions being obtained. In rare cases, a 'satisfactory' solution will not be obtained at all. But the probability of this happening depends on how well the data is explained by a hidden Markov model. Data which is almost entirely random noise will have the highest probability of leading to disappointing or unstable models.

A Contrived and Inappropriate Example of This Test

This section demonstrates a revealing example of the algorithm using synthetic data to clarify the presentation. We will also explore what happens when we attempt to use this technique on data that is not well fit by a hidden Markov model. The variables in the dataset are as follows:

RAND0 - ***RAND9*** are independent (within themselves and with each other) random time series. These are the predictor candidates.

SUM12 = ***RAND1*** + ***RAND2***. This is the target variable.

I chose to use two predictors and allow four states in the models. The program fits a hidden Markov model to each of the (10-9)/2=45 pairs of predictor candidates. Not surprisingly, the model based on ***RAND1*** and ***RAND2*** has the highest correlation with ***SUM12***. Its means and standard deviations for each state are printed first:

```
Means (top number) and standard deviations (bottom number)

State           RAND1            RAND2

   1          0.06834         -0.66014
              0.48729          0.21358

   2         -0.73466          0.07687
              0.17187          0.54038

   3         -0.02272          0.35902
              0.39033          0.39555

   4          0.73542          0.08884
              0.17546          0.52133
```

RAND1 and ***RAND2*** are totally random (they exist in only one state), so attempting to fit a hidden Markov model to them should be extremely unstable. Indeed, in ten runs of this test, twice the program found solutions in which the means of the states were all nearly zero, indicating no differentiation between states. But most of the time it came up with a pattern essentially identical to the one shown above. This solution is remarkably similar to a sort of principal components decomposition: ***RAND1*** distinguishes between State 2 and State 4, while ***RAND2*** distinguishes between State 1 and State 3. Thus, knowledge of which of the four states the process is in provides great information about ***SUM12***.

Next we see the transition probabilities. The figure in Row *i* and Column *j* is the probability that the process will transition from State *i* to State *j*. Not surprisingly, they are almost all identical. The relatively small discrepancies are just due to random variation in the data.

```
Transition probabilities...

          1         2         3         4
  1   0.2638    0.2037    0.3494    0.1830
  2   0.2438    0.1945    0.3638    0.1979
  3   0.2130    0.1682    0.4174    0.2014
  4   0.2404    0.2148    0.3272    0.2176
```

Further properties of each state are then printed:

Percent is the percentage of cases in which this state has the highest probability. The sum of these quantities across all states may not reach 100 percent, because cases in which there is a tie for the highest probability are not counted. If the data is continuous, this should almost never happen.

Correlation is the ordinary correlation coefficient between the target and the membership probability for this state. On first consideration it might be thought that the beta weight in the linear equation predicting the target from the state probabilities would be the better quantity to print. But the beta weight is not printed at all due to the fact that such weights are notoriously unstable and hence uninformative. Suppose there is very high correlation between the membership probabilities of two states, a situation which is especially likely to happen if the user specifies more states than actually exist in the process. Then both of these probabilities could be highly correlated with the target, while they might actually have opposite signs for their beta weights!

Target mean is the mean of the target when this state has the highest membership probability. Cases in which there is a tie for maximum (almost impossible for continuous data) do not enter into this calculation.

Target StdDev is the standard deviation of the target when this state has the highest membership probability. Cases in which there is a tie for maximum (almost impossible for continuous data) do not enter into this calculation.

```
State   Percent   Correlation  Target mean  Target StdDev

  1      23.76     -0.53350     -0.54538       0.45473
  2      21.73     -0.52368     -0.71809       0.56342
  3      34.03      0.38210      0.35674       0.47747
  4      20.48      0.62840      0.92173       0.49069
```

The reader should look back at the table of ***RAND1*** and ***RAND2*** means for each of the four states and confirm that the correlations and target means shown in the table above make sense. We also see that the state membership probabilities conform with the transition matrix. As expected for random series, the target standard deviations are all about the same.

Last but not least is the list of models, sorted in descending order of their multiple-R with the target. As expected (or at least hoped), the models involving either ***RAND1*** or ***RAND2*** appear first, and they are all extremely significant. As soon as these two variables are exhausted, multiple-R plunges and significance is lost. The remainder of this table is not shown here, but this situation continues.

```
------> Hidden Markov Models correlating with SUM12 <------

Predictor 1  Predictor 2  Multiple-R  Solo pval  Unbiased

   RAND1        RAND2        0.8896     0.0010     0.0010
   RAND1        RAND3        0.6937     0.0010     0.0010
   RAND1        RAND5        0.6680     0.0010     0.0010
   RAND0        RAND1        0.6619     0.0010     0.0010
   RAND1        RAND9        0.6604     0.0010     0.0010
```

```
RAND1          RAND8        0.6590      0.0010      0.0010
RAND2          RAND5        0.6579      0.0010      0.0010
RAND0          RAND2        0.6554      0.0010      0.0010
RAND2          RAND9        0.6493      0.0010      0.0010
RAND1          RAND7        0.5870      0.0010      0.0010
RAND1          RAND4        0.5845      0.0010      0.0010
RAND2          RAND4        0.5756      0.0010      0.0010
RAND2          RAND3        0.5721      0.0010      0.0010
RAND2          RAND7        0.5667      0.0010      0.0010
RAND2          RAND6        0.5648      0.0010      0.0010
RAND2          RAND8        0.5623      0.0010      0.0010
RAND1          RAND6        0.3938      0.0010      0.0010
RAND3          RAND9        0.0307      0.1110      0.8760
```

A Sensible and Practical Example

This section demonstrates an example of hidden Markov models using actual data, in this case an application that predicts future movement of a financial market. There are five candidates for predictor variables and a single target:

CMMA_5 is the current closing price of the market, minus its 5-day moving average. This shows the degree to which the market just (as of the end of the current day) departed from its recent price level.

CMMA_10 is a similar quantity, but based on the 10-day moving average.

CMMA_20 is a similar quantity, but based on the 20-day moving average.

LIN_ATR_7 is the slope of the best-fit straight line connecting the prices over the most recent 7 days, normalized by average true range. This indicates the short-term price trend in the market.

LIN_ATR_15 is a similar quantity, but based on the 15-day trend.

DAY_RETURN_1 is the market change over the next day, normalized by average true range. This variable serves as the target, as it represents the future change of the market price.

This example specifies that two predictors will be used by the model, and three states are possible. The model that correlates most highly with the target uses ***CMMA_5*** and ***CMMA_20*** as predictors. The means and standard deviations of these variables are shown for each of the three states:

```
Means (top number) and standard deviations (bottom number)

State          CMMA_20            CMMA_5

   1         -20.81845         -15.87819
               9.42582          16.57821

   2          24.57826          17.83951
               8.25328          15.22672

   3           3.57633           2.36846
               7.27092          17.76842
```

The three states are highly distinct in terms of their predictor distributions. ***CMMA_20***, in particular, has means that are widely separated relative to their standard deviations. We see that State 1 is characterized by today's price being much lower than recent prices, State 2 is characterized by today's price being much higher than recent prices, and State 3 is characterized by today's price being about the same as recent prices. This sounds almost too 'sensible' to be believed, but numerous reruns of the test consistently produced similar results.

The transition probability matrix, shown below, reveals several interesting properties. First, we see that states have considerable persistence; there is about a 90 percent probability that tomorrow will remain in the same state as today. What is also interesting is that it is nearly impossible for the market to transition between States 1 and 2 without going through State 3, and in fact probably staying in State 3 for some time. In fact, the probability of going from State 1 to State 2 is zero to at least four digits!

```
Transition probabilities...

          1        2        3
  1   0.8978   0.0000   0.1022
  2   0.0014   0.9095   0.0890
  3   0.0711   0.0747   0.8542
```

The table of additional properties shows how these states relate to the target, the price change of the market the next day. We see that State 3, that corresponding to prices remaining fairly constant, is the most common, occurring almost 40 percent of the time. We also see at least one-day persistence of price movements into the future, as State 1, which corresponds to a pattern of today's closing price being far below recent prices, is associated with a negative price movement tomorrow. Similarly, State 1, which corresponds to a pattern of today's closing price being far above recent prices, is associated with an upward price movement tomorrow. Finally, it is noteworthy that the standard deviation of the target when in State 1 is almost fifty percent higher than when in the other two states. Thus, we can expect unusually large market turbulence when we have been in a pattern of prices closing far below their recent values. This agrees well with intuition, but it is nice to see it corroborated numerically.

```
State    Percent   Correlation  Target mean  Target StdDev

  1       27.75      -0.07034     -0.05099       0.86047
  2       32.41       0.06831      0.08906       0.60901
  3       39.84      -0.00049      0.02438       0.64007
```

Finally, we have the list of models sorted according to their relationship to the target. The major take-away from this list is that the CMMA variables are much more important to predicting tomorrow's price movement than the linear trend variables. Also, the degree of significance of these relationships is impressive, usually the minimum obtainable from the 1000 Monte-Carlo replications performed.

```
Predictor 1  Predictor 2   Multiple-R  Solo pval  Unbiased

  CMMA_20     CMMA_5         0.0807      0.0010     0.0010
  CMMA_5      LIN_ATR_7      0.0762      0.0010     0.0010
  CMMA_10     CMMA_5         0.0689      0.0010     0.0010
  CMMA_10     CMMA_20        0.0686      0.0010     0.0010
  CMMA_20     LIN_ATR_7      0.0650      0.0010     0.0010
  CMMA_20     LIN_ATR_15     0.0442      0.0010     0.0010
  CMMA_10     LIN_ATR_7      0.0408      0.0010     0.0010
  CMMA_10     LIN_ATR_15     0.0330      0.0020     0.0080
  CMMA_5      LIN_ATR_15     0.0227      0.0480     0.1500
  LIN_ATR_15  LIN_ATR_7      0.0168      0.1790     0.4750
```

Assessing HMM Memory in a Time Series

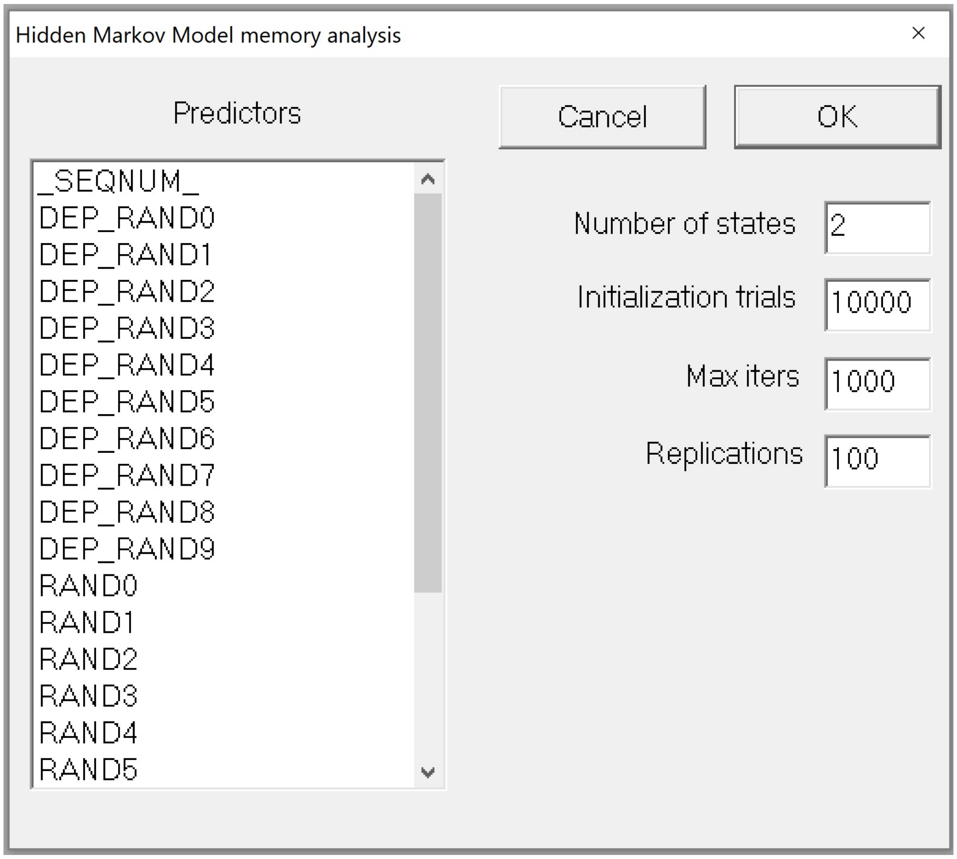
The prior section described a test for linking measurable feature variables to an unmeasurable target variable by means of an underlying hidden Markov model. But it makes no sense doing that if our candidate features do not have memory that can be modeled by a hidden Markov model. Thus, if we have doubts, our preliminary step should be to assess whether our feature variables, alone or in small groups, have memory that can be explained by a hidden Markov model.

Alternatively, and especially if we have an unwieldy quantity of candidate variables, we may wish to reverse this order: first, perform the linkage test, and then confirm that the selected features conform satisfactorily to a hidden Markov model explanation. Of course, if they do not, the linkage test will often fail, and if they do, the linkage test will often succeed (if such linkage is actually present!). However, conflicts do arise and can be quite revealing. If the linkage test shows a strong relationship but the memory test described in this section shows a poor HMM explanation, we should be inclined to largely disregard the linkage results and focus on more traditional data mining techniques. Conversely, if the linkage test fails but the HMM memory test succeeds, we have pretty good evidence that the features have little predictive power for the target variable, stronger evidence than what could be obtained by most traditional tests alone. Thus, it behooves us to perform *both* tests, ideally but not necessarily doing the memory test first.

This memory test gives us a simple Monte-Carlo p-value for the null hypothesis that the data cannot be explained by a hidden Markov model. If this null hypothesis is true (the data has no HMM memory), we would expect that the fitting criterion of the original data would be about the same as those of the permuted datasets, which by definition have no HMM memory. But if the data is well fitted by a hidden Markov model, we would expect its fitting criterion to be greater than that of most or all of the permuted datasets, leading to a very small p-value. In fact, the computed p-value is the probability that, if the null hypothesis is true (the data has no HMM memory), we could have gotten a fitting criterion as great as we observed by pure luck. When we perform this test, we really want a probability no greater than 0.05, and a cutoff of 0.01 is nicely conservative.

Specifying the Test Parameters

When we select *Hidden Markov Model Memory* from the *Test* menu, a dialog similar to that shown below appears.


Hidden Markov Model memory analysis

SEQNUM
DEP_RAND0
DEP_RAND1
DEP_RAND2
DEP_RAND3
DEP_RAND4
DEP_RAND5
DEP_RAND6
DEP_RAND7
DEP_RAND8
DEP_RAND9
RAND0
RAND1
RAND2
RAND3
RAND4
RAND5

Number of states: 2
Initialization trials: 10000
Max iters: 1000
Replications: 100

The following items must be specified:

The ***Predictors*** column is used to specify the set of predictor variables. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual variables can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to specify the target variable. This variable is ignored when the models are computed; rather it plays a role in selecting the 'best' model.

The ***Number of states*** is exactly that, the number of states in which the process can exist. It must be at least two, and it typically is small, rarely more than four. Execution time blows up rapidly as the number of states increases.

Initialization trials is the number of trials used to find a starting point for the iterative fitting process. Run time is approximately linearly proportional to this number, but it really should be as large as possible, because having a sufficient number of trials is critical to correct operation.

Max iters is an insurance policy against unending iterations. Leaving it set at the default 1000 should virtually always be good. Unless the data is pathological, the number of iterations will never get even close to this limit.

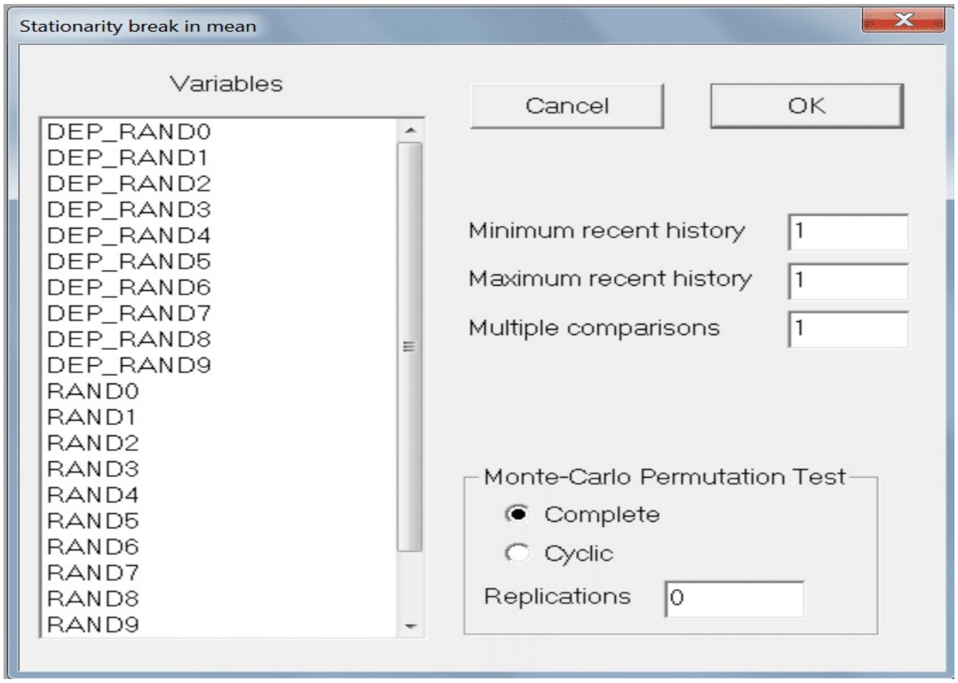
Replications is the number of Monte-Carlo permutation test replications. Values in the range 100-1000 are reasonable, with larger being better. Complete permutation is always done, as cyclic permutation would not simulate the null hypothesis of no memory.

The only thing printed by this test is a p-value for the null hypothesis that the data cannot be explained by a hidden Markov model.

Stationarity Test for Break in Mean

Stationarity in the mean is vital to most prediction schemes. If a predictor or target significantly changes its mean in the midst of a data stream, it would be foolish to assume that a prediction model will perform well on both sides of this break. Thus, we should always check for this sort of nonstationarity in all predictors and targets.

Even for applications in which series being evaluated are not being used as predictors or targets, this test is also useful. We may have a process whose performance is indicated by a numerical value. It may be the error rate of a prediction system, or cost savings achieved by a new manufacturing process. A classic example is following the performance of a market trading system. Suppose a previously profitable system suddenly deteriorates. We naturally wish to determine whether this falloff in performance is within historical norms or signifies something serious. This test is performed by clicking *Test / Stationarity break in mean*. The dialog box shown below will appear:


Stationarity break in mean

Variables
DEP_RAND0
DEP_RAND1
DEP_RAND2
DEP_RAND3
DEP_RAND4
DEP_RAND5
DEP_RAND6
DEP_RAND7
DEP_RAND8
DEP_RAND9
RAND0
RAND1
RAND2
RAND3
RAND4
RAND5
RAND6
RAND7
RAND8
RAND9

Cancel OK

Minimum recent history 1
Maximum recent history 1
Multiple comparisons 1

Monte-Carlo Permutation Test
[✓ Complete] Cyclic
Replications 0

The user must select one or more variables. The user also specifies the range of recent history which will be searched for a break in the mean. The default of doing no search at all, but rather looking at only the most recent observation, allows the fastest detection of a change. However, it is also the least sensitive test, being based on a single observation relative to the rest of history. Employing a wider search range greatly increases the sensitivity of the test, at the price of delayed confirmation of a change in the mean.

The ***multiple comparisons*** field has a subtle but important function. Suppose you are performing a one-time test. You have one or more series which you plan to employ as predictors and/or targets in a modeling operation. You simply want to test whether any of them have a significant break in their mean. Then you can leave the *multiple comparisons* field at its default of one.

But now suppose you are monitoring incoming data from a series. For example, you may be assessing quarterly returns of a market trading system. Every time a new quarter rolls around you repeat the test. The statistical term for this repetition of the same test with different data is *multiple comparisons*. Its effect is to increase the chance that you will observe a statistically significant result, even though the effect you are looking for is not present. Sooner or later, random chance is going to present a significant result due to nothing more than luck.

The user can compensate for this effect by having the program adjust its p-values under the assumption that a specified number of tests will be performed. Of course, in real life it would be difficult to make an honest assessment in advance of exactly how much testing will be done. Still, this capability is better than blithely ignoring this vital issue! At a minimum, the user can see the effect of multiple tests on the computed p-values, and make a good-faith assessment of the number of tests that will be performed.

Because there will be huge correlation between successive test statistics due to overlap of the testing regions, ordinary multiple-comparison tests are invalid. For this special application, an ad hoc but reasonable methodology is followed. Look at Figure 7.7 on the next page.

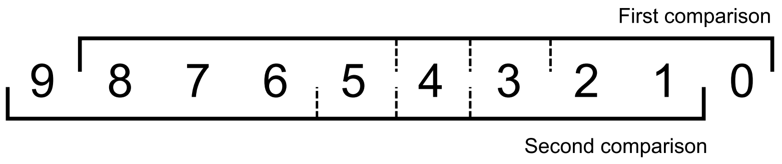


Figure 7.7: Testing for break in mean

Figure 7.7 illustrates the simple situation of testing with a range of 3 (*minimum recent history*) to 5 (*maximum recent history*) cases on the 'recent' side of the hypothetical break. It also shows 2 multiple comparisons. The dotted lines show the breakpoints tested.

For the original, unpermuted data only the 'First comparison' would be performed. Whichever of the three trial breakpoints produces the largest break will be the score as of the most current observation.

For all permutations, both comparisons will be performed. The null-hypothesis score will be the greatest of the six scores (three for each of the two comparisons). We then count how many of these null hypothesis scores equal or exceed the obtained score for each test. As per the usual Monte-Carlo permutation test, let there be *m* permutations, with *k* of them having a score equaling or exceeding the greatest score among the tests (which, strictly speaking, is not known until all tests are complete!). Then the p-value is (*k*+1)/(*m*+1). This is the approximate probability that, if there were no break in the mean, we would have obtained a maximum break score across all tests that is at least as large as that actually observed.

There are several theoretical problems with this multiple-comparison test. Foremost, it is not strictly correct to keep re-evaluating the p-value on each test. By rights we should wait until all tests are complete and examine the maximum break across all tests. The computed p-value relates to this maximum break. Of course, in real life this would defeat the whole purpose of the test! We want to test on an ongoing basis. But I strongly suspect that, compared to other sources of random error, this is of minor consequence.

Also, the shifting of test windows probably does a good job of accounting for serial correlation in the test statistics, but I have no rigorous proof. Because each sequential test involves a massive overlap in the data that goes into the test, the test statistics will have similarly massive serial correlation. The algorithm illustrated in Figure 7.7 simulates what would happen in real life, but rigorous justification would be nice.

In short, the mathematical foundations of this test are shaky. Nonetheless, in a multiple-comparisons situation, this test is almost certainly far superior to failing to compensate in any way, and I have reasonable confidence that it is actually quite good. But be warned.

If the user sets the Monte-Carlo permutation test replications to zero or one, no MCPT will be performed, and only one column of results will be printed. This column is labeled Z(U), and it is the absolute z-score corresponding to the Mann-Whitney U-test statistic for the difference in means between the data before and after the break point.

In the more usual situation of the user specifying a large number of replications (100-1000 or so), two additional columns are printed. The *Solo pval* for a variable is the approximate p-value for that variable considered in isolation; it is the probability that if the variable had no break in its mean we would have obtained a test statistic at least as large as was actually obtained.

If this quantity is not small, the developer should be inclined to believe that the variable does not have a significant mean break. Of course, this logic is, in a sense, accepting a null hypothesis, which is well known to be a dangerous practice. However, if a reasonable number of cases are present and a reasonable number of Monte-Carlo replications have been done, this test is powerful enough that failure to achieve a small p-value can be interpreted as the variable being decently stationary in its mean.

If more than one variable is specified, then the *Unbiased pval* column has a useful interpretation. When several variables are tested, chances are that one or more of them will, by sheer chance, have an usually large apparent mean break, even if in truth no such break exists. The *Unbiased pval* compensates for this effect.

The *Unbiased pval* is printed for all variables. For the first variable, the one having the greatest observed mean break, this is the approximate probability that, if none of the variables had a mean break, we could get a greatest mean break among them at least as large as that observed. For those other, lesser candidates, the *Unbiased pval* is an upper bound for the true unbiased p-value of the variable. Thus, a very small *Unbiased pval* for any candidate is a strong indication that the candidate has a significant mean break. Unfortunately, unlike the *Solo pval*, large values of the *Unbiased pval* are not necessarily evidence that the candidate is break-free. Large values, especially near the bottom of the sorted list, may be due to over-estimation of the true p-value. The author is not aware of any algorithm for computing correct unbiased p-values for any candidate other than that having the largest break. However, because this measure is conservative, it does have great utility in discovering nonstationary variables.

On a final note, be aware that having a *statistically* significant mean break does not equate to having a *practically* significant mean break. If the dataset is large, even a trivial mean break, something of no practical consequence, may show statistical significance. This test should be treated as a tool, a supplementary source of information, as opposed to the final arbiter of stationarity.

Serial Correlation and Cyclic Permutation

Like many other tests in *VarScreen*, the user may select either *complete* or *cyclic* permutation. In other tests, the cyclic method is useful for variables having significant serial correlation. However, with a test for a break in the mean, one must be cautious, as positive serial correlation can, in and of itself cause the mean to wander. Thus, any situation in which cyclic permutation is warranted is likely to also be a situation in which the mean will be inherently nonstationary, or at least appear so! The user will nearly always employ complete permutation. Still, in some special situations, cyclic permutation is appropriate. This should become more clear as we work through several examples.

We begin with the most basic situation in which the variables have negligible serial correlation and the user wishes to search nearly the entire extent of the data stream for a break in the mean. We'll use the same ***RAND0*** through ***RAND9*** that have appeared in prior demonstrations. These are random series, independent within themselves and with one another. There are 6300 cases. We decide to keep at least 50 cases on each side of the sought-after break in order to provide decent sensitivity, realizing that if a mean break happens in the outer 50 cases we will miss it. Thus, we specify a minimum of 50 and a maximum of 6250 recent cases. Since this is a one-shot test we leave the multiple comparisons parameter at its default of one. The following results are obtained with 100 iterations:

```
Variable        Z(U)       Solo pval  Unbiased pval

   RAND6       3.3257        0.0600       0.3900
   RAND8       2.8718        0.1200       0.8100
   RAND5       2.5216        0.2800       0.9900
   RAND9       2.5128        0.3600       0.9900
   RAND0       2.4845        0.3900       1.0000
   RAND3       2.3591        0.3900       1.0000
   RAND2       2.0359        0.6600       1.0000
   RAND7       2.0212        0.7500       1.0000
   RAND1       1.9353        0.7400       1.0000
   RAND4       1.3772        0.9800       1.0000
```

One of these variables, RAND6, manages through luck to get a solo p-value of 0.06. But its unbiased p-value of 0.39 tells us that this was almost certainly just a fluke from having tested ten variables.

But what if our variables have substantial positive serial correlation? It is vital that we do not attempt to perform an 'across the extent' test for a mean break, such as the 50-6250 test just shown. If we were to use complete permutation, the serial correlation in the null hypothesis runs would be destroyed, while the serial correlation in the unpermuted data would cause the mean to wander, making it virtually certain that a significant (probably *highly* significant!) break would be found, whether one truly exists or not. But cyclic permutation would not work here either. The only effect of the data rotation by cyclic permutation would be to shift the position of the break; the search for a break would still find it in nearly every permutation. So the null hypothesis distribution would be too large, resulting in overly large p-values.

The only sort of test we can do when the data has substantial serial correlation is to limit the searched range to a very small fraction of the number of cases, so that when the null hypothesis distribution is computed via cyclic permutation, only a tiny fraction of that distribution will find the original mean break in the search. The most common such situation is when we suspect that a series we are measuring has recently suffered a shift in mean beyond that which can be expected from any positive serial correlation.

So let's suppose that we want to examine only the most recent ten cases out of 6300. We'll use the ***DEP_RAND0*** through ***DEP_RAND9*** variables seen in other tests. These variables, while independent of one another, have large positive serial correlation. The incorrect approach is to use complete permutation, as this destroys the serial correlation in the null hypothesis. If we were to make this foolish mistake, we would get the result shown on the next page. Remember that these variables have no break in the mean other than the wandering that is to be expected from positive serial correlation. It's obvious that this is a crazy test. Even most of the unbiased p-values are tiny.

```
 Variable         Z(U)       Solo pval  Unbiased pval

DEP_RAND3       4.2651         0.0100        0.0100
DEP_RAND5       3.8843         0.0100        0.0100
DEP_RAND9       3.6829         0.0100        0.0100
DEP_RAND0       3.4434         0.0100        0.0100
DEP_RAND6       3.3758         0.0100        0.0100
DEP_RAND8       3.1717         0.0100        0.0200
DEP_RAND7       3.1334         0.0100        0.0500
DEP_RAND4       2.5973         0.0300        0.2300
DEP_RAND1       2.1878         0.0900        0.6300
DEP_RAND2       0.7855         0.9500        1.0000
```

But if we switch to cyclic permutation, we will be testing whether the mean of the most recent few cases differs from the mean of the earlier cases more than is usual in a series with this level of serial correlation. The results are as follows:

```
 Variable         Z(U)       Solo pval  Unbiased pval

DEP_RAND3       4.2651         0.1000        0.6600
DEP_RAND5       3.8843         0.2000        0.8900
DEP_RAND9       3.6829         0.1600        0.9600
DEP_RAND0       3.4434         0.2400        0.9800
DEP_RAND6       3.3758         0.2000        0.9800
DEP_RAND8       3.1717         0.3400        0.9900
DEP_RAND7       3.1334         0.3000        0.9900
DEP_RAND4       2.5973         0.4500        1.0000
DEP_RAND1       2.1878         0.6600        1.0000
DEP_RAND2       0.7855         0.8800        1.0000
```

This is more reasonable. Even the solo p-value of the 'worst' variable is not terribly significant, and its unbiased p-value correctly confirms that nothing unusual is going on here.

Finally, suppose we want to perform this exact same test but with the understanding that a few more observations will be coming in and we will want to repeat the test. In particular, we agree that we will be performing this same test a total of five times, each time a new case arrives. So we specify 5 multiple comparisons and observe the results shown on the next page.

```
 Variable        Z(U)       Solo pval  Unbiased pval

DEP_RAND3       4.2643       0.1300       0.8900
DEP_RAND5       3.8833       0.2600       0.9800
DEP_RAND9       3.6887       0.3600       0.9800
DEP_RAND0       3.4422       0.3300       0.9900
DEP_RAND6       3.3811       0.4800       0.9900
DEP_RAND8       3.1702       0.4600       1.0000
DEP_RAND7       3.1319       0.4300       1.0000
DEP_RAND4       2.5955       0.6200       1.0000
DEP_RAND1       2.1892       0.7600       1.0000
DEP_RAND2       0.7836       1.0000       1.0000
```

As a result of allowing multiple comparisons, the p-values have all increased somewhat. Also note that the Z(U) values have changed slightly. This is because the number of cases tested is slightly reduced for multiple comparisons, as illustrated in Figure 7.7 on Page 293.

FREL: Feature Weighting as Regularized Energy-Based Learning

The FREL algorithm (Yun Li et al, 'FREL: A Stable Feature Selection Algorithm', *IEEE Transactions on Neural Networks and Learning Systems*, July 2015) is a useful method for ranking, and even weighting, predictor variables in a classification application which is relatively low noise but is plagued by high dimensionality (numerous predictors) and small sample size. The implementation in *VarScreen* is strongly based on their innovative algorithm, but with significant modifications that I believe improve on the original version by providing more accurate and stable weights (at the cost of slower execution). My implementation also includes an approximate Monte-Carlo permutation test (MCPT) of the null hypothesis that all predictors have equal value, as well as an MCPT of the null hypothesis that the predictors, taken as a group, are worthless. Sadly, I am unable to devise a FREL-based MCPT of any null hypothesis concerning individual predictors taken in isolation. Complete source code and a detailed discussion can be found in my book "Data Mining Algorithms in C++", published by the Apress division of Springer.

The 'model' which inspires FREL is weighted nearest-neighbor classification. The distance between a test case having predictors $\boldsymbol{x} = \{x_1, ..., x_K\}$ and a training-set case $\boldsymbol{t} = \{t_1, ..., t_K\}$ is defined as the city-block distance between these cases, with each dimension having its own weight. This is defined as:

$$D(\boldsymbol{x},\boldsymbol{t}) = \sum_k w_k \left| x_k - t_k \right| \qquad (7.1)$$

Then, if one wishes to classify an unknown test case $\boldsymbol{x}$ based on a training set, one would compute the distance between the test case and each member of the training set. The chosen class for the test case would be the class of the training case having minimum distance from the test case.

Of course, performing this classification presupposes that we know appropriate weights. The procedure can be inverted and used to find optimal weights, and we could then interpret the weights as measures of importance of the predictors (assuming that the predictors have commensurate scaling!). All we would do is define a measure of classification quality and then find weights that maximize this quality measure.

An approach to machine learning that is becoming more and more popular is *energy-based modeling*. One has a set of random variables, which in the current context would be predictors, and a prediction target or class membership. The model defines a scalar *energy* as a function of the values of these variables, sometimes called their *configuration*. This energy is a measure of the compatibility of the configuration, with small values of energy corresponding to compatible configurations. If we have a known energy-based model and we wish to make an inference (a prediction or classification) based on specified values of the predictors, we fix the predictors and vary the target or class variable to identify the configuration that minimizes the energy.

In order to find a good energy-based model, we tune the parameters of the model in such a way that 'correct' configurations (as indicated by the training set) have small energy and 'incorrect' configurations have large energy.

Once the structure of the model is specified, in order to find optimal parameters we define a loss functional (a function of a function). The model is a function which maps configurations of variables to energy values, and the loss functional maps models to scalar loss values. In order to train the model, we find the version (parameters for the model family) which minimizes the loss functional.

The most common version of this latter operation, which we will do here, is to define a per-sample loss functional as a function of the model and a single case, and then average this per-sample measure across the entire training set.

This is a good time for a brief digression to make sure that two crucial issues are clear. First, many models, such as nearest-neighbor classification and some types of kernel regression, implicitly include the entire training set (or some other dataset) as a key component of the model. Do not confuse this with discussions of the training set related to training. It's still just the model, and we need not explicitly mention the presence of the training set as part of the model. Second, do not confuse energy with loss. Energy is a measure of the compatibility of a given variable configuration with a model, and it is used to make a prediction. Loss is a measure of the quality of a model in a way that generally includes a training set, and it is used to find an optimal model.

The energy that a model $\boldsymbol{M}$ assigns to a hypothetical variable configuration $\{\boldsymbol{x}, \boldsymbol{y}\}$ can be conveniently written as $E(\boldsymbol{M}, \boldsymbol{x}, \boldsymbol{y})$. An extremely common and useful way to express the per-sample loss for a single training case $\{\boldsymbol{x}^i, \boldsymbol{y}^i\}$ is $L(\boldsymbol{y}^i, E(\boldsymbol{M}, \boldsymbol{x}^i, \Upsilon\ )$, in which the term $E(\boldsymbol{M}, \boldsymbol{x}^i, \Upsilon\ )$ actually stands for multiple energy values, one for each possible value of $\boldsymbol{y}$. In other words, the per-sample loss for a single case is a function of the true value of $\boldsymbol{y}$ for that case, and the energies given by the model for $\boldsymbol{x}$ associated with every possible $\boldsymbol{y}$.

Note, by the way, that the distinction between *function* and *functional* become a bit murky here, depending on whether we think in terms of E being a hypothetical function or an observed number. In any case, the idea should be clear.

We are almost done presenting a general form of an effective loss function(al) for training an optimal (in the sense of the loss) model. We have seen the form of a per-sample loss, and stated that averaging this quantity over every sample in the training set is reasonable. The only remaining issue is that of *regularization*. This enables us to embed prior knowledge about the model in the final solution. Typically, this involves limiting the size of weights involved in the expression of the model, although other approaches are possible. With these things in mind, we can express the loss of a given model *M* for a given training set *T* and regularization function *R* as shown below. This is a scalar quantity which we will minimize in order to develop a good model.

$$L(M,T) = \frac{1}{K} \sum_k L\left[y^k, E\left(M, x^k, \Upsilon\right)\right] + R(M) \qquad (7.2)$$

To review, a good model will fulfill two requirements: it will have low energy for correct configurations and high energy for incorrect configurations. Looked at another way, when a good model is presented with a set of predictors $\boldsymbol{x}$, its energy will be low when it is simultaneously presented with the correct $\boldsymbol{y}$ for that $\boldsymbol{x}$, and its energy will be high when it is simultaneously presented with any incorrect $\boldsymbol{y}$.

It is tempting, and often appropriate, to consider only the first half of this two-part requirement: the model will have low energy for correct configurations. This is especially true for models in which fulfilling the first half automatically fulfills the second half. For example, suppose we have a regression equation as the model, and we define the energy associated with the model and a training case as the squared difference between the correct answer and the answer provided by the regression function. If the loss is just this energy, then averaged across the entire training set, the loss is the mean squared error (MSE). The optimal model is produced by minimizing the MSE, a venerable approach.

But for many model architectures, this halfway method is not a good approach. It is much better, if not mandatory, to explicitly take into account the second half of the requirement: the energy of incorrect answers should be large. And intuitively, we don't much care about easy situations, those incorrect answers that have huge energy. Even a weak model will do well with them. What we must worry about is those situations in which an incorrect answer has dangerously low energy. We want our model to be able to raise the energy of these problematic cases as much as possible above the energy of the correct answer.

This intuition leads to the following definition:

The *most offending incorrect answer* for a case, which we will call $\ddot{y}$, is the incorrect answer that has the lowest energy. This is the answer most likely to cause an error, because it is the incorrect answer that is most difficult for the model to distinguish from the correct answer. The second half of the training procedure discussed earlier, that incorrect answers should have large energy, is more general than is necessary. All we really care about is that the most offending incorrect answer has energy as large as possible, compared to the energy of the correct answer. The other incorrect answers are of relatively minor importance because they are easier for the model to avoid.

In particular, what we often want to maximize is the difference between the energy of the most offending incorrect answer and the energy of the correct answer. This will give us a model that is optimal in the sense of effectively handling the most difficult cases, while letting the easy cases slide.

A popular per-sample loss criterion, and which is used in *VarScreen*, is the log loss shown below. Note how it is a monotonic function of the difference between the two energies, so optimizing either is equivalent to optimizing the other (for a single case, not averaged across the training set!).

$$Loss\left(M, x^i, y^i\right) = \log\left(1 + \exp\left[E\left(M, x^i, y^i\right) - E\left(M, x^i, \ddot{y}^i\right)\right]\right) \qquad (7.3)$$

Now that a theoretical foundation is laid, we can apply these ideas to the specific model used in the FREL paper and *VarScreen*. Recall from the beginning of this section that we use weighted nearest-neighbor classification. Thus, in order to compute E($\boldsymbol{M}$, $\boldsymbol{x}^i$, $\boldsymbol{y}^i$) for training case *i*, we check all other training cases in the correct class, $\boldsymbol{y}^i$. The smallest distance is the energy for the correct class. Similarly, to compute E($\boldsymbol{M}$, $\boldsymbol{x}^i$, $\ddot{y}^i$) we search all other training cases in an incorrect class and find the distance to the nearest. Of course, although this is simple to describe and implement, it can be horrendously slow to compute. The quantity being minimized is the average across the training set of the per-sample losses shown in the equation above. If there are *n* training cases and *K* predictors, a single evaluation of the grand loss function requires on the order of Kn^2 operations. Yikes! Luckily, FREL is most useful for situations in which the training set is small relative to the number of predictor candidates, so that squared term will hopefully not be a serious problem.

All that remains to be settled is the regularization. In any reasonable application, the energy of the incorrect answers will, on average, exceed that of the correct answers; otherwise the model would be worthless! For the loss function just shown applied to weighted nearest-neighbor classification, increasing the weights together will decrease the loss, because the term being exponentiated will become increasingly negative. Thus, naive minimization of the loss will result in the weights blowing up without bound. Thus, we are inspired to penalize large weights. This is common practice, even in situations in which this blowup is not natural. The reason is that in many models, large weights are associated with overfitting and poor out-of-sample performance. In *VarScreen* we use the common method of penalizing by the sum of the squares of the weights. The sum of their absolute values is also common and may be implemented in a future version of the program.

The optimal weights determined by minimizing regularized loss can be interpreted as measures of importance of the individual predictors. However, two issues must be considered. First, the scaling of the predictors obviously impacts the weights, so their scaling should be commensurate. *VarScreen* takes care of this by internally scaling per their standard deviation. Second, interpretation by the user is aided by normalizing the weights in some way for display. In *VarScreen* they are linearly normalized so as to sum to 100.

A frequently useful variation is to take many bootstrap samples from the dataset and compute the final weight estimate by averaging the estimates produced from each bootstrap sample. The sampling must be done without replacement, as nearest-neighbor algorithms are irreparably damaged when the dataset contains exact replications of cases. Bootstrapping FREL has at least two major advantages over doing one FREL analysis of the entire dataset:

1) Stability is usually improved. A critical aspect of any weighting scheme is that the computed optimal weights should be affected as minimally as possible by small changes in the dataset. Such changes might be inclusion or exclusion of a few training cases, or change might be effected by the addition of noise to the data. An average of bootstraps is much more

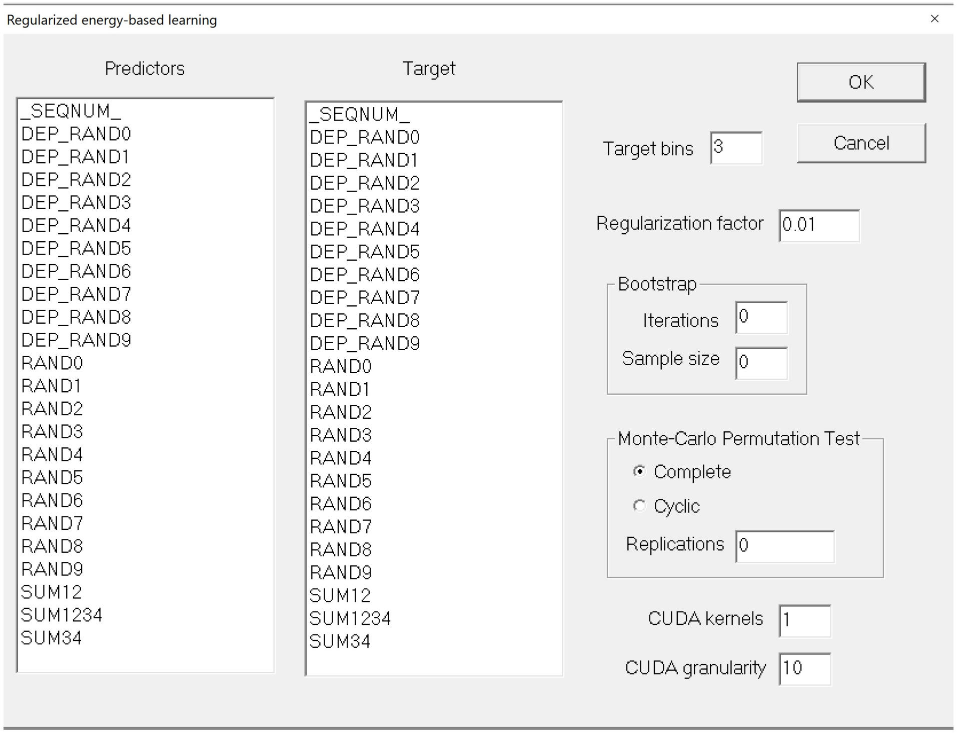
robust against data changes compared to a single complete FREL processing.

2) Because run time of the FREL algorithm is proportional to the square of the number of cases, we can greatly decrease the run time by performing many iterations of a small sample.

For these reasons, bootstrapping is generally recommended.

FREL Operation in VarScreen

We've already discussed the mathematics behind the FREL implementation in *VarScreen*. This section covers the user interface. When the user clicks *Test / Regularized energy-based*, a dialog box similar to that shown below appears.



Regularized energy-based learning

Predictors: _SEQNUM_, DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8, RAND9, SUM12, SUM1234, SUM34

Target: _SEQNUM_, DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8, RAND9, SUM12, SUM1234, SUM34

Target bins: 3

Regularization factor: 0.01

Bootstrap
Iterations: 0
Sample size: 0

Monte-Carlo Permutation Test
[✓ Complete] Cyclic
Replications: 0

CUDA kernels: 1

CUDA granularity: 10

The following information must be supplied by the user:

The leftmost (***Predictors***) column is used to specify the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to specify the target variable. This variable will be partitioned into two or more classes based on it values. FREL does not permit continuous targets.

Target bins specifies the number of bins into which the target will be categorized. The number of cases in each bin will be made as equal as possible.

Regularization factor traditionally prevents model weights from running away to problematic large values. However, in *VarScreen* this is a fairly non-critical parameter; even zero is acceptable. This is because the optimization algorithm in *VarScreen* inherently prevents weight runaway as part of its stability enforcement. In practical terms, the effect of the regularization factor is to control the relative spread of weights. Suppose that predictability is concentrated in just one or a few candidates. If the user specifies a small or zero value for this parameter, the computed weights will strongly reflect this focus. However, if a very large regularization factor is specified, the focus will be less intense; some of the weight will be redistributed away from the dominant predictors and given to predictors of lesser value. Intense focus on one or a few dominant predictors can, in some cases, be seen as a form of overfitting.

Bootstrap operation usually increases robustness of the weight estimates and also decreases runtime, a happy confluence of outcomes. By default, no bootstrapping is done. But the user can specify that a given number of *iterations* are performed, each having a specified *sample size*. The sample size must be large enough that each sample is virtually guaranteed to have a significant number of representatives from each target class. For the number of iterations, my own rough rule of thumb is that the product of the number of iterations times the sample size should be about twice the number of training cases.

A ***Monte-Carlo permutation test*** is a useful, though time consuming, way to test certain null hypotheses about the predictor candidates. It is vital to understand that these tests are radically different from the other permutation tests in *VarScreen*. For one thing, I am not aware of any way of performing a perfect individual-candidate MCPT with FREL; the best I can do is come up with a rough approximation that appears to work well in practice. More importantly, in other tests, the candidate predictors are handled individually, so the p-values (at least the solo tests) are independent. But FREL considers all candidates simultaneously. This dependence changes the nature of MCPT. One effect is for dominant candidates to 'suck' weight out of lesser candidates, thus reducing their apparent significance. But the important effect is to radically change the nature of the null and alternative hypotheses of the test.

In other *VarScreen* tests, the null hypothesis for each solo p-value is that the *individual* candidate is *worthless*, and that for the unbiased p-values is that *all* candidates are *worthless*, and the power of the test is in identifying *individual* candidates which have predictive power. But for FREL, the individual MCPT tests have no useful power in situations in which all candidates have equal predictive power, regardless of whether that power is tiny or large. The null hypothesis is still generated by making all candidates worthless, exactly as in other tests. But because of the joint estimation of weights, it is more intuitive (though not strictly correct!) to think of the null hypothesis as being that *all candidates have equal predictive power*, with the unbiased p-values compensating for the fact that we are testing numerous candidates, and any of them may be outstanding by random luck. In other words, these individual tests are related to the predictive power of each candidate *relative to their competitors*. Their *individual* predictive powers play no easily identifiable role in determining p-values.

With this in mind, we can look at the p-values of candidates at the top of the list, those ranked highest in terms of predictive power and having the largest weights, and consider the p-values as being the probability that if all candidates were truly *equal* in predictive power, the top-ranked candidates would have *outperformed the others* to the degree shown. Suppose we see a highly significant result for the single best candidate. It may be that this best candidate is *almost* worthless, and its competitors are *completely* worthless. Or it may be that this single candidate is *excellent*, while its competitors are merely *very, very good*. In either case we may see the best candidate having a highly significant p-value. Again, I emphasize that this interpretation is not strictly correct, but I believe that it is close enough, especially the unbiased p-values, to be effective indicators of the validity of the obtained results.

The sucking of weight from relatively poor predictors to good predictors has a peculiar and potentially confusing effect on the solo p-values. As we drop down the sorted list to the low-ranked candidates, we can see the solo p-values cover a wide range, jumping up and down between high and low significance randomly. This is illustrating in an exaggerated manner the fact that the p-values for worthless candidates in any statistical test have a uniform distribution, with all values being equally likely. This is yet another reason why we should focus on the unbiased p-values, ignoring the solo p-values except perhaps (and with great caution) for the few top-ranked candidates.

VarScreen does print one additional p-value, called the *Loss p-value*. This is a 'grand' measure of the ability of all predictors taken together to be effective at correct classification. The null hypothesis is that none of the candidates have any predictive power, and the Loss p-value is the probability that if this were so, we would have achieved a loss at least as low as that obtained. This p-value being small is a necessary condition for any of the individual p-values to be meaningful. If we cannot be reasonable certain that at least one of the candidates has predictive power, then there is no point in considering their relative power!

The user may specify several parameters for the MCPT:

Replications defaults to zero, in which case no Monte-Carlo permutation test is performed. However, if computer time permits, it is usually best to set this to at least 100, and perhaps as much as 1000, so that solo and unbiased p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutations. So, for example, if the user specifies 100 permutations, the minimum p-value that can appear is 0.01. Run time of this test is linearly related to the number of permutations.

The user must choose either ***Complete*** or ***Cyclic*** permutations. If the user is confident that there is no dependency as described earlier in this document, then *Complete* should be used; it is the traditional approach which does a complete random shuffle for each permutation. However, if there is dependency, this type of shuffling will produce underestimation of *p-values*, a very dangerous situation. If the dependency is serial (the data is a time series and the dependency is among samples close in time) then a slight improvement in the situation can be obtained by using *Cyclic* permutation. In this type of shuffle, the time order of the target is kept intact except at the ends by rotating the targets with end-point wraparound. Shuffling this way preserves most of the serial dependency in the permutated targets, which makes the algorithm more accurate. The *p-values* computed this way will generally be larger than those computed with complete shuffling, and hence less likely to lead to false rejection of the null hypothesis of no predictive power. But be warned that the cure is far from complete; computed *p-values* will still underestimate the true values, just not as badly.

Note that in most cases it is legitimate to use *Cyclic* permutation instead of *Complete* when there is no dependency. However, if the dataset is small, *Cyclic* permutation will limit the number of unique permutations and hence increase the random error inherent in the process. As long as the dataset is large, some users may prefer to use *Cyclic* permutation even if it is assumed that there is no serial dependency; in case there really is hidden serial dependency, this is a cheap insurance policy. Still, the best practice is to make sure that the data does not contain dependency and then use *Complete* permutation. Relying on *Cyclic* permutation to take care of dependency problems is living dangerously. And if the dataset contains fewer than 1000 or so cases, use of *Cyclic* permutation is not recommended unless it is necessary to handle dependency.

CUDA Considerations

First, be aware that the default CUDA parameters (*Kernels* and *Granularity*) should be fine for nearly all applications and hardware. However, for users who wish to tweak operation (or those who must do so because of timeouts) the FREL dialog allows the user to specify two parameters.

Computation of the loss function entails two nested loops. The outer loop performs cross validation, letting each training case play the role of a test case, with these individual losses averaged across the entire training set. The inner loop passes through all cases other than the test case and finds the energy of the correct answer and that of the most offending incorrect answer. Since this latter operation also involves finding the weighted distance between cases, this results in a *lot* of mathematical operations.

Microsoft Windows has the infamous 'feature' of limiting the time during which CUDA computation can monopolize the video display in a contiguous stretch, typically two seconds. Therefore, the ***CUDA Kernels*** parameter lets the outer loop be broken up into multiple kernel launches. By default all computation is performed in a single launch, which is good, because launches have considerable overhead. But if the screen goes black and a message pops up that the display adapter has been reset, you will have to increase (as little as possible!) the CUDA Kernels parameter.

The ***Granularity*** parameter is more subtle and requires an understanding of CUDA hardware to be fully appreciated. If the granularity is set to 1, each outer-loop case is assigned to a thread, and this single thread handles the entire inner loop. But CUDA devices prefer much finer granularity so that they can run thousands or even millions of threads simultaneously. Otherwise, vast amounts of hardware resources sit idle, a grievous waste. To avoid this, the inner loop for each outer-loop case is broken up into *Granularity* sub-tasks, where this parameter cannot exceed the number of cases. The bottom line is that a total of ***Number of cases*** times ***Granularity*** separate threads are executed. Users with a late-model extremely powerful CUDA processor may benefit from increasing the granularity beyond the default, perhaps even to its limit of the number of cases.

FSCA: Forward Selection Component Analysis

The algorithms provided here are greatly inspired by the paper "Forward Selection Component Analysis: Algorithms and Applications" by Luca Puggini and Sean McLoone, published in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, December 2017, and widely available for free download on various Internet sites. However, I have made several modifications that make it more practical for real-life applications.

The technique of principal components has been used for centuries (or so it seems!) to distill the information (variance) contained in a large number of variables down into a smaller, more manageable set of new variables. Sometimes the researcher is interested only in the *nature* of the linear combinations of the original variables that provide new 'component' variables having the property of capturing the maximum possible amount of the total variance inherent in the original set of variables. In other words, principal components analysis can be viewed as an application of descriptive statistics. Other times the researcher wants to go one step further, computing and employing the principal components as predictors in a modeling application.

However, with the advent of extremely large datasets, several shortcomings of traditional principal components analysis have become problematic. The root cause of these problems is that traditional principal component analysis computes the new variables as linear combinations of *all* of the original variables. If you have been presented with thousands of variables, there can be issues with using all of them.

One possible issue is the cost of obtaining all of these variables going forward. Maybe the research budget allowed for collecting a huge dataset for initial study, but the division manager would look askance at such a massive endeavor on an ongoing basis. It would be a lot better if, after an initial analysis, you could request updated samples from only a much smaller subset of the original variable set.

Another issue is interpretation. Being able to come up with descriptive names for the new variables (even if the 'name' is a paragraph long!) is always good. It's hard enough putting a name to a linear combination of a dozen or two variables; try understanding and explaining the nature of a linear combination of two thousand variables! So if you could identify a much smaller subset of the original set, such that this subset encapsulates the majority of the independent variation inherent in the original set, and then compute the new component variables from this smaller set, you are in a far better position to understand, name, and explain what these new variables represent.

Yet another issue with traditional principal components when applied to an enormous dataset arises is the all too common situation of groups of variables having large mutual correlation. For example, in the analysis of financial markets for automated trading systems, we may measure many families of market behavior: trends, departures from trends, volatility, and so forth. We may have many hundreds of such indicators, and among them we may have several dozen different measures of volatility, all of which are highly correlated. When we apply traditional principal components analysis to such correlated groups, an unfortunate effect of the correlation is to cause the weights within each correlated set to be evenly dispersed among the correlated variables in the set. So, for example, suppose we have a set of 30 measures of volatility that are highly correlated. Even if volatility is an important source of variation (potentially useful information) across the dataset (market history), the computed weights for each of these variables will be small, each measure garnering a small amount of the total 'importance' indication. As a result, we may examine the weights, see nothing but tiny weights for the volatility measures, and erroneously conclude that volatility does not carry much importance. When there are many such groups, and especially if they do not fall into obvious families, the possibility of intelligent interpretation becomes hopeless.

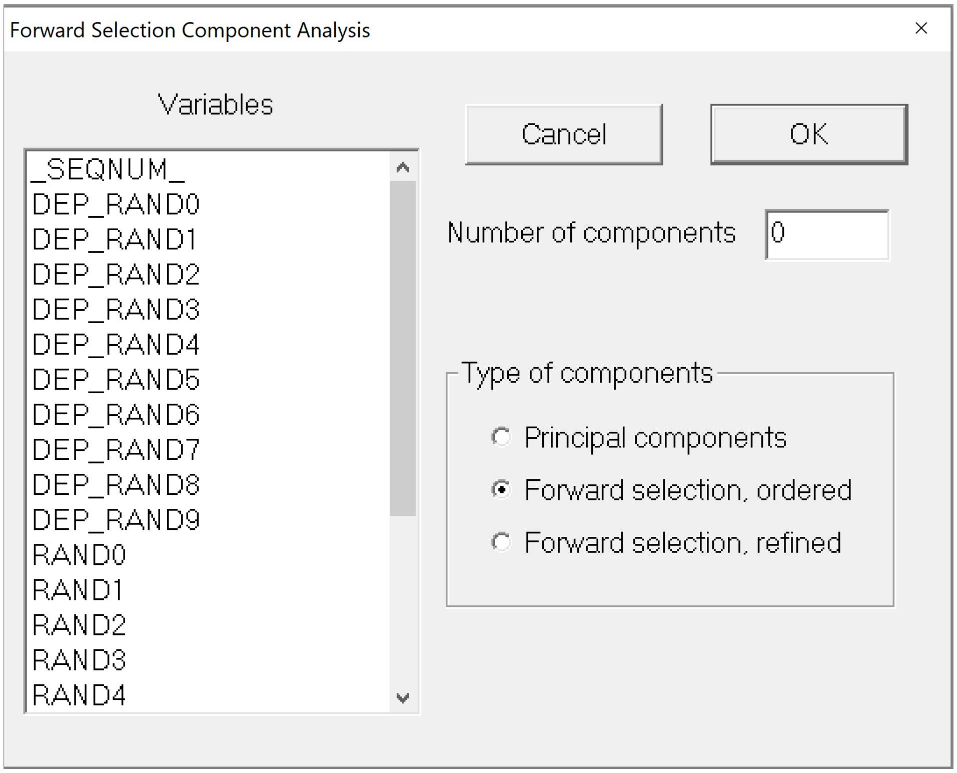
The algorithms presented here go a long way toward solving all of these problems. They work by first finding the single variable that does the best job of 'explaining' the total variability (all original variables) observed in the dataset. Roughly speaking, we say that a variable does a good job of explaining the total variability if knowledge of the value of that variable tells us a lot about the values of all of the other variables in the original dataset. So the best variable is the one that lets us predict the values of all other variables with maximum accuracy.

Once we have the best single variable, we consider the remaining variables and find the one that, in conjunction with the one we already have, does the best job of predicting all other variables. Then we find a third, and a fourth, et cetera. Application of this simple algorithm gives us an ordered set of variables selected from the huge original set, beginning with the most important, and henceforth with decreasing but always optimal importance (conditional on prior selections).

It is well known that a greedy algorithm such as the strictly forward selection just described can produce a sub-optimal set of variables. It is always optimal in a certain sense, but only in the sense of being conditional on prior selections. It can (and often does) happen that when some new variable is selected, a previously selected variable suddenly loses a good deal of its importance. Thus, the algorithms here optionally allow for continual refinement of the set of selected variables by regularly testing previously selected variables to see if they should be removed and replaced with some other candidate. Unfortunately, we then lose the strict ordering-of-importance property that we have with strict forward selection, but we gain a more optimal final subset of variables. Of course, even with backward refinement we can still end up with a set of variables that is inferior to what could be obtained by testing every possible subset. However, the combinatoric explosion that results from anything but a very small universe of variables makes exhaustive testing impossible. So in practice, backward refinement is pretty much the best we can do.

When *FSCA* is selected from the *Create* menu, a dialog box similar to that shown on the next page will appear, from which the user makes the following specifications:

The leftmost (***Variables***) column is used to specify the universe of variables from which a subset will be selected. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.


Forward Selection Component Analysis

Variables

SEQNUM
DEP_RAND0
DEP_RAND1
DEP_RAND2
DEP_RAND3
DEP_RAND4
DEP_RAND5
DEP_RAND6
DEP_RAND7
DEP_RAND8
DEP_RAND9
RAND0
RAND1
RAND2
RAND3
RAND4

Cancel OK

Number of components 0

Type of components

Principal components [✓ Forward selection, ordered] Forward selection, refined

The ***Number of Components*** specifies how many variables will be selected, although if the dataset contains extreme colinearity this number will be reduced as needed to prevent colinearity in the computed components. Setting this value to zero causes all variables to be selected. This, of course, runs counter to the primary purpose of this algorithm. On the other hand, it does let us see the universe of variables rank-ordered according to ability to reconstruct the complete dataset. This information is often interesting and useful. The number of components computed will always equal the number of variables selected in the absence of collinerity.

Three algorithms for variable selection and corresponding component generation are available:

Principal Components of the traditional variety can be computed. This is a rather uninteresting option, but it is included for comparison purposes.

Forward selection, ordered uses strict forward selection; no backward refinement is done. As a result, the order in which variables are printed when the program is finished represents their order of importance in reproducing the entire dataset. In other words, the first variable in the list is the single most important. The second variable in the list is the one that, *given the value of the first variable selected*, is the most important among the remaining variables. The third is the one that, *given the values of the first two variables selected*, is best at reproducing the dataset. Et cetera.

Forward selection, refined combines forward selection with backward refinement. This generally improves the quality of the final subset of variables compared to the prior option, but backward refinement destroys the ordering of the variables. It can happen that the first variable selected, the single best, doesn't even make it to the final subset! At this time, this option (the slowest of the three) is the only one of the three that is multi-threaded for full use of multi-core CPUs.

All three of these options create a new set of variables in the database which can then be used in subsequent studies. If the user specified principal components, the variable names will be in the form ***PrinCo_n_m***, while the other two options will produce variables named ***FSCA_n_m***. In both cases, *n* refers to the sequence number in which they were computed as separate operations. The first time you run the algorithm, *n*=1. The second time, *n*=2, and so forth. In both cases, *m* is the component number, ranging from 1 through the number of variables in the selected subset.

For all three options, the newly computed variables will have zero mean, unit standard deviation, and they will be uncorrelated. The *VarScreen.log* file will provide information to allow the user to recreate the components with other data and programs, if desired.

For the *ordered* (no refinement) option, the log file will list the actual coefficients needed to convert *standardized* (zero mean, unit standard deviation) values of the original variables to the newly created component variables, also standardized. For the other two options, the log file will list the correlations between each component and the original variable, with the first column being the component that captures the most variance from the subset, the second column capturing the second-most variance, and so forth. If you require coefficients for computing standardized values of the components, just divide each correlation by the eigenvalue shown at the top of the table. Or you can use the correlations directly, without dividing by the eigenvalues, in which case you will get the same components, but they will not have unit standard deviations.

For all three options, the eigenvalues and eigenvectors of the correlation matrix of the universe will be printed first, with as many columns as variables/components specified by the user. This is followed by a list of the mean squared correlation of each variable in the universe with all other variables. Finally, the table of coefficients or component/variable correlations as described above is printed.

Here is an example of each of the two FSCA algorithms. For this example, the following variables are employed:

RAND1 - RAND6 are independent (within themselves and with each other) random time series.
SUM12 = RAND1 + RAND2
SUM34 = RAND3 + RAND4
SUM1234 = SUM12 + SUM34

When we run the FSCA algorithm using the option for strict ordering (no refinement), we first see the following results printed:

```
There are 6 unique (non-redundant) sources of variation
The number of components computed is therefore being reduced
to this value.

Eigenvalues, cumulative percent, and principal component
factor structure

Eigenvalue    2.988   1.986   1.052   1.015   0.987   0.972
Cumulative   33.195  55.263  66.957  78.240  89.203 100.000

RAND1        0.4835  0.4964 -0.6476 -0.1497 -0.1080 -0.2576
RAND2        0.4597  0.5206  0.6390  0.1478  0.1037  0.2770
RAND3        0.5246 -0.4808 -0.0470 -0.2077  0.6690 -0.0271
RAND4        0.5175 -0.4859  0.0620  0.2194 -0.6661  0.0240
RAND5       -0.0198 -0.0198 -0.4669  0.4999  0.1474  0.7139
RAND6        0.0020  0.0260  0.0233  0.7937  0.2265 -0.5635
SUM12        0.6800  0.7331 -0.0090 -0.0021 -0.0036  0.0128
SUM1234      0.9997  0.0239  0.0012  0.0040  0.0003  0.0073
SUM34        0.7331 -0.6800  0.0104  0.0076  0.0039 -0.0023
```

We have 9 variables in the universe, but the program notes that there are only 6 unique sources of variation. This is not surprising, because the 3 sum variables are just combinations of the other variables. Since by definition the computed components must be independent, the program limits us to just 6 components.

The first eigenvector accounts for one-third of the total variation in the dataset, and it correlates almost perfectly with SUM1234, very highly with SUM12 and SUM34, and moderately highly with RAND1-RAND4. None of this should be surprising.

The second component is just the contrast between RAND1 and RAND2 versus RAND3 and RAND4. In conjunction with the first component, it gives us over 55 percent of the total variation. The remaining components are other contrasts as well as RAND5 and 6.

Next, we get a list of the mean squared correlation of each variable in the universe with all other variables:

```
Mean squared correlation of each variable with all others

          RAND1    0.091
          RAND2    0.088
          RAND3    0.096
          RAND4    0.095
          RAND5    0.000
          RAND6    0.000
          SUM12    0.181
        SUM1234    0.248
          SUM34    0.191
```

It is not surprising that RAND1-RAND4, along with their various sums, have positive mean squared correlations, while RAND5 and RAND6 have zero correlations.

Last of all we get the table of coefficients needed to compute the 6 components from the chosen 6 variables in the subset. Note that each component depends on only the corresponding ordered variable and all previously selected variables.

```
Variable     1         2         3         4         5         6

SUM1234   1.0000  -0.9730   0.0181   0.0106   0.0047  -1.4045
  SUM12  -0.0000   1.3953  -0.9696  -0.0091  -0.0131   0.9888
  RAND2   0.0000  -0.0000   1.3842  -0.0129   0.0380  -0.0081
  RAND6   0.0000   0.0000  -0.0000   1.0001  -0.0169  -0.0071
  RAND5   0.0000  -0.0000   0.0000   0.0000   1.0007  -0.0017
  RAND4  -0.0000   0.0000  -0.0000  -0.0000  -0.0000   1.4188
```

Observe that the best single variable for reproducing the entire universe of values is SUM1234, the sum of four other variables in the universe, and the first component is just this one variable (its coefficient is 1.0 and all other coefficients are 0.0).

The second variable selected is another sum variable, and the corresponding component's value is computed as that sum variable times 1.3953, minus the prior selected variable times 0.9730.

The third variable selected is a similar weighted sum, primarily based on RAND2. The next two components are essentially equal to the two completely independent variables, RAND6 and RAND5. Note that their coefficients are almost exactly 1, and all other coefficients are almost exactly 0. And the last component is a complex mix of other variables.

We use this same universe of variables to demonstrate the other FSCA option, forward selection combined with backward refinement. The initial information (eigenstructure and mean squared correlations) are the same as in the prior example, so we will skip straight to the interesting part, the log of variables being added and replaced:

```
Commencing stepwise construction with SUM1234
Added SUM12 for criterion=4.973085
  Replaced SUM1234 with SUM34 to get criterion = 4.973123
Added RAND2 for criterion=6.011605
  Replaced SUM12 with RAND1 to get criterion = 6.011623
Added RAND6 for criterion=7.011701
Added RAND5 for criterion=8.010402
Added RAND4 for criterion=8.999919
  Replaced SUM34 with RAND3 to get criterion = 8.999940
```

As in the prior example, the first variable selected is SUM1234. We then add SUM12, as in the prior example. (Both options will always select the same first two variables.) But then something interesting happens:

SUM1234 is replaced by SUM34, giving us a two-variable set of SUM12 and SUM34. To me, this is prettier than SUM1234 and SUM12.

We then add RAND2, which immediately triggers the replacement of SUM12 with RAND1. After that we add the two totally independent variables, RAND6 and RAND5. Finally, we add RAND4, which triggers the replacement of SUM34 with RAND3. The final results are shown below:

```
Eigenvalues, cumulative percent, and selected principal
component factor structure

Eigenvalue     1.056   1.023   1.002   0.988   0.983   0.948
Cumulative    17.600  34.646  51.343  67.811  84.194 100.000

RAND3         0.2269  0.5167  0.2772  0.5747 -0.5221 -0.0431
RAND1         0.5751  0.0508 -0.1047  0.3593  0.5829  0.4322
RAND2        -0.6877  0.0094  0.1597  0.2264  0.0044  0.6709
RAND6        -0.0862 -0.6254  0.4725  0.4844  0.1757 -0.3357
RAND5         0.4228 -0.5366  0.1187 -0.2014 -0.5355  0.4381
RAND4         0.1210  0.2723  0.8070 -0.4496  0.2300  0.0702
```

The final set of selected variables is intuitively more appealing than what we got with the strict ordering option, because it's just the individual random variables, without their various sums. Because replacement has destroyed any ordering of the subset, it makes the most sense to me to just compute the components as the principal components of the final subset. Note that the eigenvalues are all nearly equal, meaning that the components have no strong ordering either. Also note that the values in the table are the correlations between the components and the variables, and they can be converted to weights by dividing each column by the eigenvalue at the top of the column.

LFS: Local Feature Selection

Most common feature selection algorithms are primarily oriented toward favoring features that are at least somewhat predictive over the *entire domain* of the feature set. This predictivity may be nonlinear, and it may interact with other features, but such a predictor will be at a significant advantage over *more powerful but only locally predictive candidates* if the nature of its relationship to a target variable is at least somewhat consistent across the domain of all possible values of all candidate features.

This 'global favoritism' can be a major problem, because modern nonlinear models can obtain a lot of useful predictive information from variables whose power is limited to small areas of the domain, or whose predictive relationship changes significantly over the domain. But if our predictor selection algorithm fails to find such variables, focusing instead on more global candidates, we lose out on what may be valuable information.

For example, consider the XOR problem. Suppose we have two variables symmetric around zero, and we define two classes. A case is a member of Class 1 if both of our variables are positive or both negative, and it is in Class 2 if one is positive and the other negative. This classification problem can be solved with 100 percent accuracy by a simple rule, and modern nonlinear models should have no trouble achieving nearly perfect performance. Yet if we were to augment these two variables with numerous worthless predictor candidates and then try to identify the two true predictors, an amazing number of otherwise sophisticated predictor selection algorithms would fail to find them. Not only are the marginal distributions of both variables identical in both classes, but the relationship of each variable to the class depends completely on the value of the other variable, with the relationship reversing across the domain. This is a tough problem.

This same issue arises in applications that are closer to reality. For example, a common phenomenon in equity market prediction is that certain families of indicators have considerable predictive power in times of low market volatility, but become useless in times of high volatility. The presence of a large amount of high-volatility data in the dataset dilutes the predictive power of such variables and may put otherwise excellent indicators at a competitive disadvantage. And this problem arises in many other applications. The effectiveness of medical treatments can vary according to age, weight, and a potentially large number of other unknown conditions. Identification of vehicles and pedestrians by a self-driving car's control system can depend on features that are vital in some contexts and distracting clutter in others. We need a feature selection algorithm that is sensitive to predictive power that comes, goes, and even reverses, according to location in the feature domain.

In terms of modeling we can deal with inconsistent behavior by using sophisticated nonlinear models (which are prone to overfitting!), or by using different models in differing regimes (assuming that we know how to define these regimes!). But consider the pre-modeling stage, when we are searching for predictor candidates. We would like to have a predictor selection algorithm that can *automatically* find such regime-dependent behavior and identify powerful predictors, even if this power is localized.

The feature selection algorithm described in "Local Feature Selection for Data Classification" by Narges Armanfard, James P. Reilly, and Majid Komeili (*IEEE Transactions on Pattern Analysis and Machine Intelligence*, June 2016) fits the bill nicely. We'll now present a condensed and intuitive overview of how it operates.

There are a large number of possible approaches to feature selection. We've seen some based on mutual information and uncertainty reduction, techniques that are effective at detecting highly nonlinear relationships. Some techniques actually train predictive models, and perform their feature selection by intelligently choosing inputs for these models. Early discriminant analysis methods involve the use of Mahalanobis distances to find dimensions of maximum separation when the predictors are highly correlated, optimally taking correlation into account. The *LFS* algorithm presented here is based on yet another approach, a concept akin to nearest neighbor classification, but taken to a much higher level of sophistication.

We begin with a simple example: we want to predict success in college, with students divided into two classes: those who graduate, and those who drop out. We measure four candidate predictors for each student in our study dataset, and standardize the values of these predictors (mean zero and standard deviation one) to put their variation on a level playing field. These candidate predictors are:

1) SAT score
2) High school grade point average
3) Circumference of thumb divided by circumference of index finger
4) Day of month student was born

Suppose we randomly choose two students, both in the *Success* class. For each of these four features, think about the average difference in predictor value we would see for these two students. But now suppose we randomly choose two students, one in the *Success* class, and one in the *Dropout* class. The expected difference between these two students would be about the same as it was for the 'same class' students for the third and fourth candidate predictors, but much larger for the first and second candidate predictors: a person who graduated would probably have a higher GPA and SAT score than a dropout, leading to a large difference, while these two students would probably have similar finger sizes and birthdays, at least relatively speaking.

If we effectively estimated these expected differences throughout the dataset, looking at every pair of students, we would conclude that the first two candidate predictors are the ones we want, because the expected difference in these two features for students in different classes greatly exceeds the expected difference for students in the same class, while for the third and fourth candidates we observe about the same difference, regardless of whether the two students are in the same or different classes.

Now, instead of looking at candidate predictors individually, let's look at them in pairs: 1 and 2, 1 and 3, et cetera. A good measure of the difference between two cases is the Euclidean distance between them. Let $x_k^{(i)}$ represent the value of variable *k* as measured for case *i*, and let $\boldsymbol{x}^{(i)}$ represent the vector of all variables for this case. Then the distance between case *i* and case *j* is given by the following equation:

$$d_{ij} = \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\| = \sqrt{\sum_k \left(x_k^{(i)} - x_k^{(j)}\right)^2} \qquad (7.4)$$

It should be clear that the pair of variables consisting of the first two competitors will have the greatest expected inter-class distance between cases, the pair consisting of the last two competitors will have the least expected inter-class distance, and mixed pairs will have intermediate values.

Intuition can now guide us toward a good way to choose an effective set of candidate predictors. We look for a set that has a high contrast between expected intra-class distance (which we want to be small) and inter-class distance (which we want to be large). Neither quality alone is good. For example, if we find a set of candidates that produces large average inter-class separation between cases, but the expected separation between cases in the same class is also large, we have gained nothing; we cannot look at either quality in isolation. We must find a balance, a way to trade off the desirable quality of low intra-class separation with the also desirable quality of high inter-class separation. The LFS algorithm has an automated way to find the optimal tradeoff, a topic which we will return to later.

All that we've seen so far is good, and the algorithm just outlined would work fairly well in practice. However, it is missing the 'Local' component of the 'Local Feature Selection' algorithm. We still need a way to handle the problem of predictive power that varies across the domain of all features. For example, the distribution shown in Figure 7.10 would foil the algorithm just described.

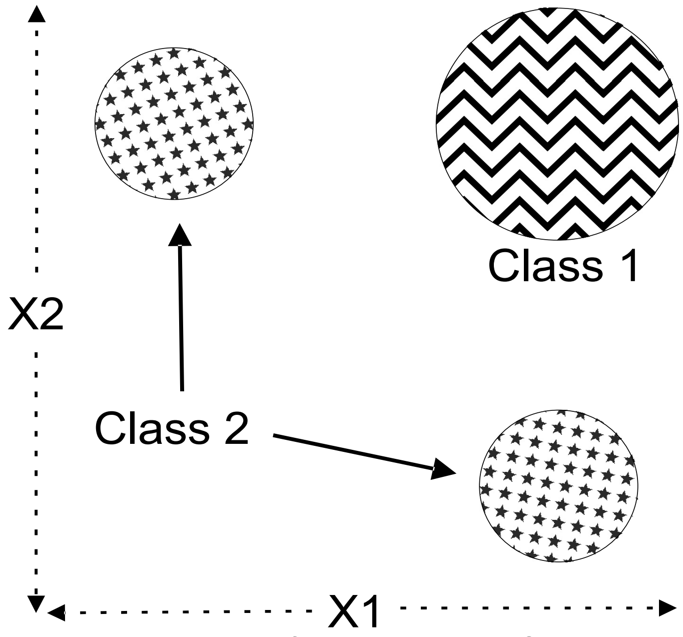


Figure 7.10: A job for Local Feature Selection

In this example, we have two classes, one of which is split into two distinct subsets. Think about how the variable selection algorithm just described would perform when presented with this problem. Half of the cases in Class 2 would have excellent inter-class separation from Class 1 via X1, though no separation at all via X2. The other half would experience the opposite behavior, gaining great separation via X2 but none via X1. If inter-class separation were the only consideration, the algorithm would pick up X1 and X2 easily.

The problem lies with the intra-class separation. Cases that lie within either of the subsets of Class 2 would have nicely small separation. But if one case in Class 2 lies in one subset, and the other case lies in the other subset, the distance between them would be enormous, larger even than the inter-class separation! So the average intra-class separation for Class 2 would be so large that it would be nearly commensurate with the inter-class separation. It's unlikely that (X1,X2) would stand out as a set of effective predictors, even though this figure shows that they are fabulous.

The key element of the paper cited at the beginning of this section is that the problem shown in Figure 7.10 can be alleviated by weighting the distances with intelligently computed weights. The primary focus in the weighting scheme is that pairs of cases which are close are given higher weights than pairs which are distant, with the weighting dropping off exponentially with distance. It's somewhat more complicated than that, because the class memberships of the cases are taken into account, as well as global behavior of the distance metrics. The details are far too complex to get into here; see Chapter 3 of my "Extracting and Selecting Features for Data Mining" if you are interested.

In order to get an idea of what's happening in regard to weights, the four histograms in Figure 7.11 show the weights generated from a test with data having the pattern shown in Figure 7.10.

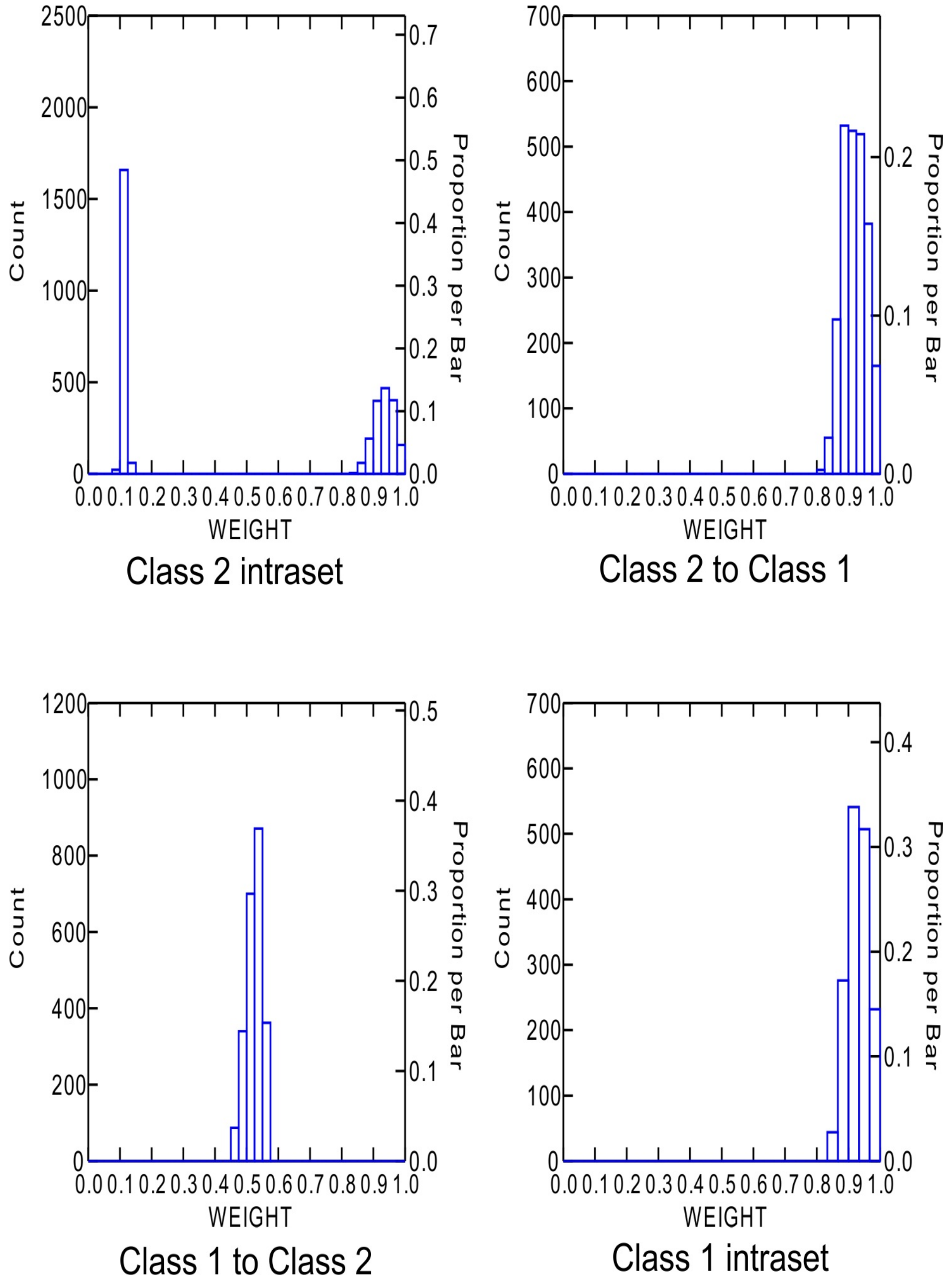


Figure 7.11: LFS weights for split-class example

The most interesting of these four histograms is the upper-left, which shows the weights for pairs of cases that are both in Class 2. We see that half of the weights are clustered near the maximum possible weight, one. These are the pairs of cases that are both in the same subset of Class 2. The other half of the weights are clustered near zero, the minimum possible. These are the pairs of cases that, while both in Class 2, are in different subsets of this class. So we see that when the intra-class separation (mean distance separating cases both in Class 2) is computed with weighted distances, pairs that span the two subsets are downplayed, thus providing a more realistic estimate of the intra-class separation.

The Class 1 intra-class weights are all close to one because this class is not split into subsets. Also, when we are considering cases in Class 2 and looking at their distances from cases in Class 1, we have full weighting. The weights are about 0.5 when we consider cases in Class 1 and look at the distances to cases in Class 2 (the weighting algorithm is asymmetric). Very roughly speaking, this is because there are two possible ways the difference can go. You can study the weight equations in the cited paper to see exactly how this comes to be.

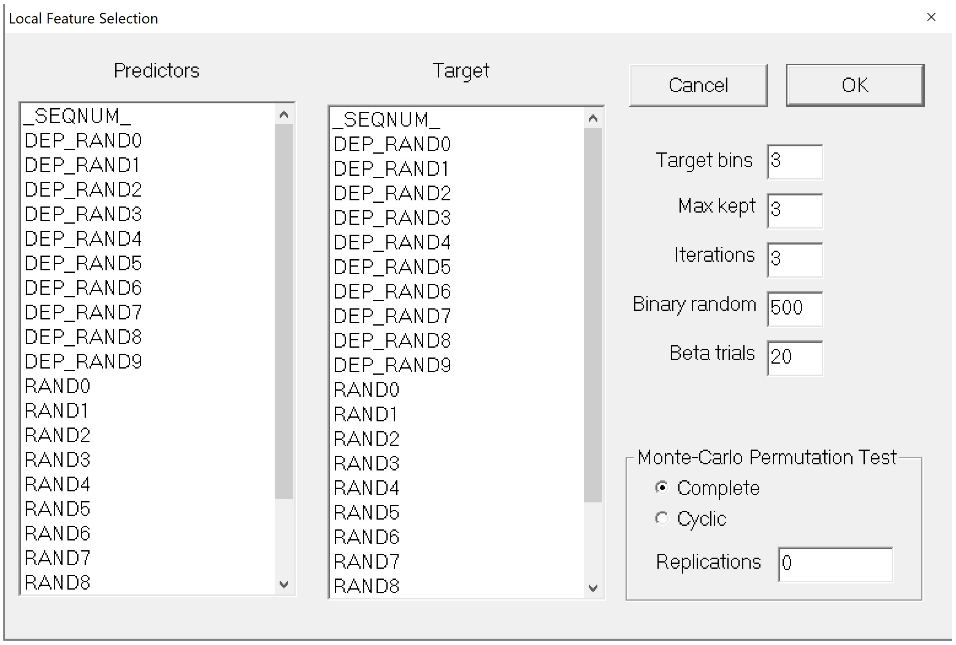
What This Algorithm Reports

Because the algorithm performs optimal-candidate selection separately for each case, there is no practical way to report a single optimal candidate set, let alone a sorted list of subsets like we were able to achieve with some prior algorithms. Instead, it counts the number of times each candidate predictor appears in an optimal subset. For example, we might see that X2, X7, and X35 form an optimal subset for some region; X3, X7, and X21 form another optimal subset, X7 and X94 form another, and so forth. X7 appeared in an optimal subset 3 times, while each of the other subset members appeared just once. So it looks like X7 is on its way to becoming popular and heading up the popularity list.

This does not mean that X7 alone is valuable. In fact, it may be (and often is) that X7 alone is worthless; it's value is only in conjunction with other candidates. This is why LFS is superior to many other feature selection algorithms, which often rely on some form of stepwise selection and hence ignore individually worthless candidates. But *this property of reliance is not a problem*. The reason is that most modern prediction models, if given a list of the most popular predictors, can sort out the complex relationships between them and perform well. All they need is preprocessing to weed out the worthless candidates, so they are not overwhelmed.

Specifying the Test Parameters

When LFS is selected from the *Tests* menu, a dialog box similar to that shown below appears.


Local Feature Selection

Predictors: _SEQNUM_, DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8

Target: _SEQNUM_, DEP_RAND0, DEP_RAND1, DEP_RAND2, DEP_RAND3, DEP_RAND4, DEP_RAND5, DEP_RAND6, DEP_RAND7, DEP_RAND8, DEP_RAND9, RAND0, RAND1, RAND2, RAND3, RAND4, RAND5, RAND6, RAND7, RAND8

Target bins 3
Max kept 3
Iterations 3
Binary random 500
Beta trials 20

Monte-Carlo Permutation Test
[✓ Complete] Cyclic
Replications 0

The following items must be specified:

The leftmost ***Predictors*** column specifies the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Single candidates can be toggled on/off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to select the target variable. The target is partitioned into bins that are as equal in size as possible. The user must specify the number of bins to employ for each, and unless the dataset is huge the default of three bins is frequently appropriate.

Max kept is the maximum number of variables ever employed in a metric space (subset of candidates). In general it is best to make this as small as possible, consistent with having enough variables simultaneously present to provide predictive power. In my experience, setting this to more than 5 is rarely, if ever needed. The default is 3.

Iterations is the number of LFS algorithm iterations to obtain good weight estimates. Run time is heavily impacted by this number. The point of diminishing returns is reached quickly; in many cases 2 is sufficient, and 3 is almost certainly more than enough for all but the most critical applications. The default is 3.

Binary random is the number of random trials employed to convert the floating-point usage flags to binary flags. More is better, but the default of 500 should be plenty for most applications, although if there are a great many variables this should be increased. It has a modest but not severe impact on run time for most applications.

Beta trials specifies the number of search points for optimizing the relative importance of intra-class versus inter-class separation discussed in Chapter 3 of this book. The default of 20 should be sufficient for the vast majority of applications. It has a modest but not severe impact on run time for most applications.

Replications defaults to zero, in which case no Monte-Carlo Permutation Test is performed. However, it is usually best to set this to at least 100, and perhaps as much as 1000, so that solo and unbiased group p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutations. So, for example, if the user specifies 100 permutations, the minimum p-value that can appear is 0.01. Run time of this test is linearly related to the number of permutations.

The user must choose either *Complete* or *Cyclic* permutations if a Monte-Carlo Permutation Test is to be performed. If the user is confident that there is no dependency as described earlier in this document, then *Complete* should be used; it is the traditional approach which does a complete random shuffle for each permutation. However, if there is dependency, this type of shuffling will produce underestimation of *p*-

values, a very dangerous situation. If the dependency is serial (the data is a time series and the dependency is among samples close in time) then a considerable improvement in the situation can be obtained by using *Cyclic* permutation. In this type of shuffle, the time order of the target is kept intact except at the ends by rotating the target with end-point wraparound. Shuffling this way preserves most of the serial dependency in the permutated target, which makes the algorithm more accurate. The *p-values* computed this way will generally be larger than those computed with complete shuffling, and hence less likely to lead to false rejection of the null hypothesis of no predictive power. But be warned that the cure is far from complete; computed *p-values* will still underestimate the true values, just not as badly.

Note that in most cases it is legitimate to use *Cyclic* permutation instead of *Complete* when there is no dependency. However, if the dataset is small, *Cyclic* permutation will limit the number of unique permutations and hence increase the random error inherent in the process. As long as the dataset is large, some users may prefer to use *Cyclic* permutation even if it is assumed that there is no serial dependency; in case there really is hidden serial dependency, this is a cheap insurance policy. Still, the best practice is to make sure that the data does not contain dependency and then use *Complete* permutation. Relying on *Cyclic* permutation to take care of dependency problems is living dangerously. And if the dataset contains fewer than 1000 or so cases, use of *Cyclic* permutation is not recommended unless it is necessary to handle dependency.

Important note: If you perform a Monte-Carlo permutation test, please see the discussion of solo and unbiased p-values that begins on Page 250 and continues onto the next page. (Also see the first section of this manual.) That discussion covers vital issues related to what these figures mean, as well as when they are and are not valid.

CUDA note: LFS will by default use CUDA-capable video hardware if present. This results in a speed increase of 1 or even 2 orders of magnitude if there are several thousand cases and not more than a few hundred variables. In other situations, CUDA may slow processing due to its overhead, and might better be disabled by clicking File/Use CUDA to make the check mark disappear.

An Example of Local Feature Selection

I created a dataset consisting of about 4000 cases and 10 variables, X0 through X9. Each random variable is uniformly distributed on [–1, 1]. Variables X3 and X4 determine the class. A case is in one class if X3 and X4 are both positive or both non-positive. The case is in the other class if one of these variables is positive and the other is not. This is a very difficult problem for many feature selection algorithms because the marginal distributions of these variables are identical for both classes, and the nature of the relationship between one of the variables with the class is determined by the value of the other variable. Here is the output of the LFS algorithm:

```
**********************************************************
*                                                        *
* Computing Local Feature Selection for predictor subset *
*      10 predictor candidates                           *
*       5 predictors at most will define a metric space  *
*       2 target bins                                    *
*       3 iterations of LFS algorithm                    *
*     500 random trials for real-to-binary f conversion  *
*      20 trial values for beta optimization             *
*     100 replications of complete Permutation Test      *
*                                                        *
**********************************************************


------------> Percent of times selected <-----------

        Variable        Pct      Solo pval  Unbiased pval

             X3        96.26       0.0100       0.0100
             X4        69.62       0.0100       0.0100
             X0         4.66       1.0000       1.0000
             X1         2.94       1.0000       1.0000
             X6         2.29       1.0000       1.0000
             X7         1.76       1.0000       1.0000
             X9         1.13       1.0000       1.0000
             X8         0.58       1.0000       1.0000
             X2         0.53       1.0000       1.0000
             X5         0.39       1.0000       1.0000
```

It's a little curious that X3 was selected somewhat more often than X4, when they have identical roles in predicting the class, but I've seen this happen often. It's undoubtedly a random occurrence that would change with a different random set of cases. What is certainly clear is that these two variables are selected vastly more often than their worthless competitors. Also, the computed solo and unbiased p-values are impressive, leaving no doubt about the conclusion reached by the algorithm.

A Note on Run Time

This local feature selection algorithm does have one downside that can make it unusable in some situations. Its run time is proportional to the cube of the number of cases. On modern computers, especially those containing CUDA-capable video hardware, handling several thousand cases should be manageable. But if you get up to the range of many thousand cases, run time will become so slow as to be impractical.

Enhanced Stepwise Lin-Quad

It's likely that everyone reading this book is familiar with stepwise selection. Typically, you have a large set of candidates for some task, often prediction or classification. You test each individual candidate and select the one candidate that performs the task best. Then you test the remaining candidates, seeking the one that, in conjunction with the one already selected, performs best. This is repeated as desired. It is a fast, efficient, and usually fairly effective way of selecting a respectable subset of features from a potentially large population.

Unfortunately, this venerable and widely used algorithm suffers from several serious weaknesses. The most obvious and problematic is that very often an application can be handled only when we have multiple features available simultaneously. As a crude example, suppose we wish to evaluate the intelligence of a person. We could give this person a test involving sophisticated logical reasoning. Suppose the person got half of the problems correct. That score would mean one thing if the person were 25 years old, and it would mean something else entirely if the person were 3 years old. Or suppose we want to measure a risk of cardiac disease. Neither height alone nor weight alone would be very good, but the two together would provide significant predictive power. When we are dealing with such an application, simple stepwise selection could easily miss a predictor that is immensely powerful when used in conjunction with another predictor but that is nearly worthless when used alone.

Another issue with stepwise selection that can be a problem if not properly handled is the fact that a naive selection criterion results in the performance steadily increasing as we add more variables (features). This is due to the fact that random noise is mistaken for legitimate information. The selection algorithm gets better and better at learning the properties of the noise as more features are examined, all the while blissfully unaware that the supposedly valuable 'features' do not represent repeatable patterns. If we judge quality on a simplistic measure such as in-sample performance, we are very likely to add more variables than are appropriate and actually decrease out-of-sample performance.

Yet another potential problem with naive stepwise selection is failure to distinguish between seemingly good performance versus statistically sound performance. An apparently great performance figure means nothing if there is a substantial probability that it could have done that well by nothing more than good luck. These are the key issues that will be addressed in this chapter.

In particular, *VarScreen* provides a generic, broadly applicable stepwise selection algorithm. The three aspects of this stepwise selection algorithm that set it apart from simple, traditional methods are:

- It significantly overcomes the problem of neglecting important variables that have little value when used alone, while avoiding the combinatoric explosion arising from exhaustive testing of all possible subsets. It does this by saving multiple promising subsets at every step, and evaluating future candidates in conjunction with these multiple subsets.

- It avoids the 'more variables means better performance' issue by judging the quality of a feature set according to its cross-validation performance. This tremendously reduces the likelihood that random noise will be misinterpreted as valid predictive information. It also provides a simple and effective automatic way to stop adding new features to the feature set.

- As each new feature variable is added, it computes two probabilities. The most important is the probability that, if all currently selected features are truly worthless, the performance criterion achieved by the current feature set could be as good as it is by pure luck. A less important but still useful measure is the probability that, if all current features are truly worthless, the performance *increase* provided by adding the most recently selected feature to those already selected could have been as large as we observed.

The Feature Evaluation Model

In order to implement the enhanced stepwise selection algorithm, we need a basis model with which to assess the predictive power of feature variables. One of my favorite prediction algorithms fits the bill nicely. This is what is sometimes called linear-quadratic regression, or perhaps quadratic-linear regression. In this model, the input vector is expanded to include not just the feature variables, but their squares and all possible cross products. These variables are supplied to an ordinary linear regression model. This hybrid approach gives us the speed and stability of simple linear regression while still supplying significant nonlinear capabilities, including complete reversal of the predictor/target relationship across the feature domain, as well as complex interactions between features. It's really a wonderful model.

Mathematically, standardization of the input variables is not required and makes no theoretical difference in performance. However, for real-life computing, as well as easy human interpretation of model coefficients, it is important to standardize the inputs so that their means are all zero and their variances are equal (one in this code). Therefore, *VarScreen* automatically standardizes all variables, including the target.

The Cross-Validated Performance Measure

The naive and traditional way of selecting features for a task is to maximize an *in-sample* performance criterion. In other words, we use a single dataset to compute the performance criterion, and select those features that provide the most optimal criterion. Of course, an even modestly responsible developer will then go on to use a second, independent data sample to evaluate the quality of that feature set in conjunction with the model that was employed. But by then it's too late. That feature selection method will almost always produce a sub-optimal feature set.

The reason this naive selection method is sub-optimal is that any dataset is a mixture of legitimate information and random noise. Unfortunately, in virtually any application, there is no way for the optimization algorithm to distinguish between noise and legitimate information when it has only one dataset to examine. Thus, whatever algorithm associates features in the dataset with correct target values in order to compute a performance measure will, to at least some degree, confuse noise with features. By definition, noise will not repeat in other data, and so some features will be selected based on their ability to relate noise to the target, a dangerous error.

Many ways to deal with this serious issue have been devised. Most are based on some sort of complexity penalty. The performance criterion may be based on something simple like applying a penalty that increases as more features are added. Others may try to evaluate the degree to which features contribute to performance within the dataset and reject those features that appear to make relatively little contribution. Still others may use sophisticated measures of complexity and penalize feature sets that produce a model with high complexity. These are all worthy endeavors, but they all *indirectly* address the issue of feature selection confusing nonrepeatable noise with repeatable information.

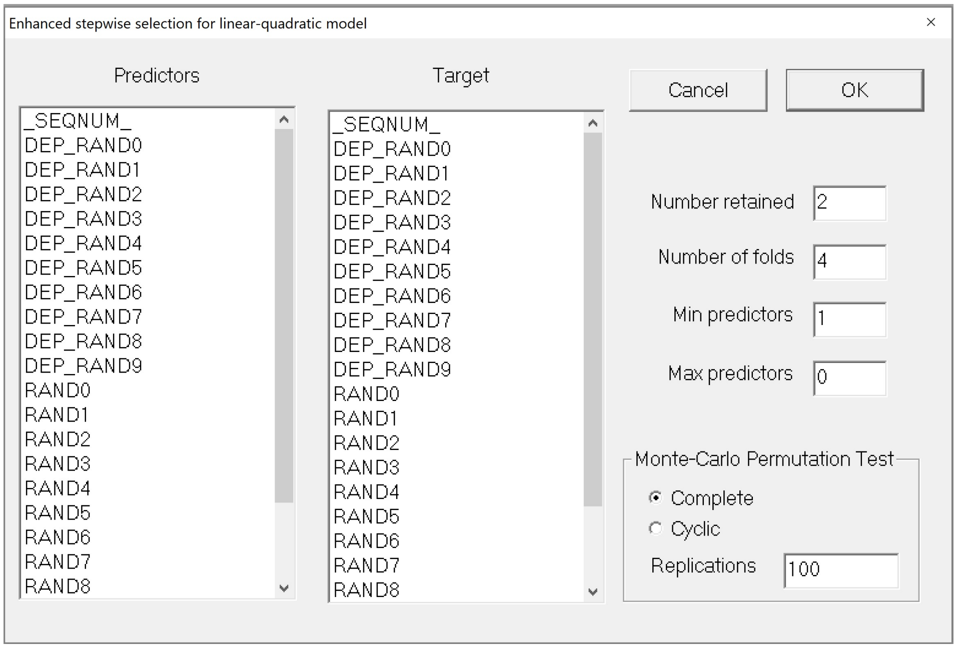
In my opinion, we are best off taking a direct approach to solving this problem: use one dataset to optimize our core model's performance with a trial feature set, and then evaluate the quality of this feature set by measuring its performance on a *different* dataset. This way, features that capture legitimate information will also perform well on the second dataset, in which the legitimate information also appears. But features that mistake random noise for legitimate information will perform poorly on the second dataset, because those phony patterns will likely not appear.

We would waste a lot of potentially expensive data if we simply divided our dataset in half for this purpose. So instead we use cross validation. A fraction of the dataset is withheld from optimizing the model, and that withheld portion is used to test the trained model. Then that portion is returned to the dataset and a different fraction is withheld. This repeats in such a way that each case in the dataset appears in a withheld portion exactly once.

One unavoidable disadvantage of cross validation is that it requires a sometimes annoying tradeoff. If we hold out only a few cases at a time (with each in/out split being called a *fold*), processing time will be huge, because we have to re-optimize the model for each fold. Thus, we are encouraged to minimize the number of folds (hold out many cases each time). But if we hold out a lot of cases, we reduce the number of cases used for optimization, which makes the model less accurate and less stable, leading to less accurate results. The rule of thumb is that we should use as many folds as possible, consistent with being able to run the program in a manageable length of time.

Specifying the Test Parameters

When the user clicks Tests / Enhanced stepwise lin-quad, a dialog similar to that shown below will appear.



Number retained 2
Number of folds 4
Min predictors 1
Max predictors 0
Monte-Carlo Permutation Test: [✓ Complete] Cyclic
Replications 100

The leftmost ***Predictors*** column is used to specify the set of predictor candidates. Multiple candidates can be selected by dragging the mouse cursor across a block, or by clicking the first candidate in a block, holding the Shift key, and clicking the last candidate in the block. Individual candidates can be toggled on and off by holding the Ctrl key while clicking on the variable.

The ***Target*** column is used to select a single target variable.

Number retained is the number of best models retained for further testing at each step. Traditional stepwise selection has this equal to 1. If you set this to an enormous number (perhaps 99999999), exhaustive testing of all combinations is attempted. Larger values generally provide superior results, but run time blows up fast as this parameter increases.

Number of folds is the number of cross-validation folds used in performance evaluation. Generally, larger is better, but runtime increases approximately linearly as this parameter increases.

Min predictors is the minimum number of predictors in the final model. As soon as this quantity is reached, addition of new variables will stop when such addition results in a performance decrease. Setting this to zero will force all selected predictor candidates to be included.

Max predictors is the maximum number of predictors in the final model. Addition of new variables will stop when this limit is reached. Setting it to zero imposes no upper limit.

Replications is the number of Monte-Carlo permutation test replications. It is usually best to set this to at least 100, and perhaps as much as 1000, so that p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutations. So, for example, if the user specifies 100 permutations, the minimum p-value that can appear is 0.01. Run time of this test is linearly related to the number of permutations.

The user should select ***Complete*** if the targets are independent, the usual case. If the targets have serial correlation, ***Cyclic*** should be selected to reduce anti-conservative behavior. This topic has been discussed in detail elsewhere.

Demonstrating the Algorithm Three Ways

This section presents three examples of the enhanced stepwise selection algorithm. For the first two, the following 11 variables are employed:

RAND0 - ***RAND9*** are independent (within themselves and with each other) random time series.

SUM1234 = ***RAND1*** + ***RAND2 + RAND3 + RAND4***

I specified a minimum and maximum number of variables to both be the number of predictor candidates. This forces testing of all candidates. The algorithm produces the output shown below, slightly reformatted to fit.

```
***********************************************************
*                                                         *
* Computing enhanced stepwise linear-quadratic model      *
*                                                         *
*         SUM1234 is the target                           *
*      10 predictor candidates                            *
*       5 candidates retained for each iteration          *
*       4 folds for cross validation performance          *
*      10 minimum predictors in final model               *
*      10 maximum predictors in final model               *
*     100 replications of complete Monte-Carlo Test       *
*                                                         *
***********************************************************

Stepwise inclusion of variables...
R-sqr  MOD pval CHG pval Predictors...
0.2811   0.010   0.010  RAND3
0.5183   0.010   0.010  RAND3 RAND4
0.7497   0.010   0.010  RAND2 RAND3 RAND4
1.0000   0.010   0.010  RAND1 RAND2 RAND3 RAND4
1.0000   0.010   0.690  RAND0 RAND1 RAND2 RAND3 RAND4
1.0000   0.010   0.850  RAND0 RAND1 RAND2 RAND3 RAND4 RAND5
1.0000   0.010   0.970  RAND0 RAND1 RAND2 RAND3 RAND4 RAND5
                        RAND6
1.0000   0.010   1.000  RAND0 RAND1 RAND2 RAND3 RAND4 RAND5
                        RAND6 RAND7
1.0000   0.010   1.000  RAND0 RAND1 RAND2 RAND3 RAND4 RAND5
                        RAND6 RAND7 RAND8
1.0000   0.010   1.000  RAND0 RAND1 RAND2 RAND3 RAND4 RAND5
                        RAND6 RAND7 RAND8 RAND9

STEPWISE successfully completed
Final XVAL criterion = 1.00000
In-sample mean squared error = 0.00000
```

Observe the following:

- The R-square criterion jumps up by about 0.25 as each of the four 'true' predictors is added, reaching and remaining at 1.0 thereafter.

- Beginning with the first predictor, the model p-value is at the minimum (most significant) possible, 1/*replications*=0.01.

- As the three additional 'true' predictors are added, the p-value for the added variable remains at 0.01. But as soon as an irrelevant variable is added, the change p-value jumps up to extreme insignificance. The boundary between important and worthless could not be more clear.

I won't show the results here, but I reran this test with the minimum number of predictors set to 1, the default. It accepted the four 'true' predictors exactly as shown above but stopped with a 'performance decrease' caused by addition of a worthless variable.

Finally, here is a more practical example. I computed 19 common indicators used in analyzing equity markets, as well as a measure of the market change over the trading day following availability of the indicators. Here is the output produced by this test:

```
************************************************************
*                                                          *
* Computing enhanced stepwise linear-quadratic model      *
*                                                          *
*         Z_DAY_RET is the target                          *
*      19 predictor candidates                             *
*      10 candidates retained for each iteration           *
*       4 folds for cross validation performance           *
*       1 minimum predictors in final model                *
*      19 maximum predictors in final model                *
*     100 replications of complete Monte-Carlo Test        *
*                                                          *
************************************************************

Stepwise inclusion of variables...

R-square  MOD pval  CHG pval  Predictors...
  0.0049    0.040     0.040   CMMA_10
  0.0079    0.020     0.090   ADX15 CMMA_10

STEPWISE terminated early because adding a new variable
caused performance degradation
```

```
Final XVAL criterion = 0.00793
In-sample mean squared error = 0.98768

Regression coefficients for standardized data:
        0.035689  ADX15
       -0.025106  CMMA_10
       -0.001000  ADX15 Squared
        0.032440  CMMA_10 Squared
       -0.079148  ADX15 * CMMA_10
       -0.030700  CONSTANT
```

The variable first selected, CMMA_10, is the close of the current bar minus the moving average of the prior 10 bars. (All prices are converted to logs before indicator computation is performed.) This variable measures the direction and degree of price departure from recent history. The second variable, ADX15, is the ordinary ADX indicator with a 15-day lookback. This indicates strength of trend, though without specifying a direction.

Even CMMA_10 alone has a p-value of 0.04, meaning that, if CMMA_10 had no day-ahead predictive power, there is only a 0.04 probability that it would have performed as well as it did in predicting market movement the next day. Adding ADX15 lowers this probability to 0.02.

Here's a quick note for mathematically inclined readers. It may superficially appear as if this 0.02 p-value suffers from selection bias and hence is overly optimistic. After all, the program first picked CMMA_10 as the best performer, and then picked ADX15 to best complement it. But remember that the permutation replications do exactly the same optimal selections, thus correctly accounting for any selection bias. So this is a fair and unbiased p-value.

The p-value for adding ADX15 is 0.09, decent but not excellent. And after that addition, despite having 17 more industry-standard candidates to choose from, it terminates with the observation that performance deteriorates with the addition of a third indicator.

Finally, I printed the fascinating model coefficients. CMMA_10 has a negative coefficient, alone and in the cross product, which indicates that regression to the mean is at work. And the fact that the cross product has the largest coefficient says that this effect is strongest when it happens in the presence of a strong trend. Very, very interesting!

Nominal-to-Ordinal Conversion

A nominal variable is one that identifies a class membership, as opposed to having a numerical meaning. Nominal variables can have numeric values yet have no numeric meaning, no sense of quantity or order. The classic example is the month of the year. We may say that June has the value 6 and November has the value 11. Certainly 11 is greater than 6, but this does not mean that November is greater than June.

Very few prediction or classification models can directly accept a nominal value as an input, which presents a problem if one or more variables in our application are nominal. There are some awkward ways around this. The most popular method is to recode a single nominal variable as a set of binary variables, with as many binary variables as the nominal variable has classes, and assign one of these binary variables to each class. Then, for each case, we set the single binary variable corresponding to the case's class to 1, and set all others to 0. This works well if there are just two classes, and it works fairly well for three classes. But if there are more than a few classes, not only can this generate an impractical number of input variables, but the information provided by any given class membership can be diluted.

If we have a variable that takes on meaningful numeric values, and that is equal to or shares substantial information with our ultimate target variable, we can often use our training data to elevate the level of an ordinal variable. In theory at least we can elevate it to the same measurement level as our (possibly surrogate) target variable. However, it has been my experience that raising it to just ordinal level, so that order (greater/less than) is meaningful, accomplishes the goal of converting a nominal variable so that it is suitable for model input, without introducing excessive random noise. This will be what is implemented in *VarScreen*.

Some nice bells and whistles are added, but to start let's discuss the basic idea. The user supplies a dataset containing values of the nominal variable to be converted, as well as values for what we will here call a target variable. In many applications this will be the actual target variable that will be used by a prediction model. However, all it really needs to be is some variable of at least ordinal level that is significantly related to the ultimate target variable.

As a perhaps overly simplistic example, suppose our ultimate goal is to be able to look at a set of properties of a patient's disease and decide whether a particular treatment should be used as a follow-up to surgery, or if the side effects are too severe to justify it. So this is a binary classification problem: use the treatment or do not use it. Also suppose that we have available as an input variable the ethnic heritage of the patient, perhaps broken down into a dozen or so categories. In an ideal world we would produce 12 different classification models, using a different model for each ethnic category. But the world being what it is, we don't have nearly enough data to take that extravagant approach. So instead we treat ethnicity as a nominal variable and associate it with a synthetic target variable, such as each patient's personal rating of quality of life after treatment, perhaps on a scale of 1 to 10.

We have a training set of patients, all of whom had this treatment. (We may have in our dataset many patients who did not have this treatment. These patients do not concern us now.) In addition to many other measured variables that are not relevant to this discussion, for each patient we have the nominal variable *Ethnicity* and the target variable *Quality of life*. We wish to compute a new variable to substitute for *Ethnicity* that will have a level of measurement higher than nominal so that we can use it as a direct input to our ultimate classification model.

A reasonable approach, which is almost but not quite the approach used here, is to find the mean of the target for each ethnic class and substitute this target mean for the *Ethnicity* variable. For example, suppose people of *Vulcan* ethnicity report a very high quality of life after this treatment, while people of *Romulan* ethnicity report a very low quality of life after the treatment. Then we would recode the dataset, substituting the (large) mean of Vulcans for the *Ethnicity* variable for each Volcan patient, and the (small) mean of Romulans for each Romulan patient, and similarly for all other ethnicities. This gives us a numeric value for the formerly nominal variable *Ethnicity*, and this new variable can be directly input to a classification or prediction model.

In this particular example, the synthetic target variable is well behaved because it has just ten possible values. But suppose the synthetic target can have heavy tails. For example, perhaps the target is the number of days before death after treatment. Perhpas most patients have about 10-50 days of life remaining, while a very few go on to live many hundreds of days. Taking the mean to use as our substitute value would likely give poor results, because one or a few crazy outliers would skew the results.

To avoid this, in my code I pass through the entire dataset and convert the target values to percentiles. Thus, the case having the smallest target value would have a score of 0, the case with the largest would have a score of 100, and all others would lie in between these extremes. This gives us a new target having ordinal scale; order in the sense of one number being greater than another is preserved, but outliers are tamed. In my own work I have found that this preserves nearly all useful information for the conversion, yet outliers have no impact.

There are three other improvements to this basic technique that I have found to be useful. When I first became involved in predicting movement in financial markets, I soon learned that some techniques work only in times of low volatility, and others (fewer!) work only in times of high volatility. I nearly always devised prediction models that specialized in one or the other of these two market states. The same applies to nominal-to-ordinal conversion. It will often be the case that we will want to employ two different conversions, with the choice being dependent on the value of some binary state variable. This binary state variable is often called the *gate*.

A second enhancement to the basic technique is the ability to ignore some cases when devising the nominal-to-ordinal mapping. In some applications we may have reason to believe that the class membership of some cases is irrelevant, with the decision depending on some other variable. Consider the prior example that involved converting the nominal variable *Ethnicity* to an ordinal variable that captures self-assessed quality of life. Suppose that some patients had medical disabilities that prevented them from providing this assessment, and a relative substituted his or her own judgement of the patient's quality of life. We may not trust that third-person view and decide, based on a 'who answered the question?' variable, whether we want to disregard this case for the purposes of creating the mapping. Of course, we want to avoid creating 'missing data' whenever possible, so it is almost always in our best interest to assign some number to such cases. The most reasonable number to assign is the median of the synthetic target, something right in the middle. Because we are mapping to percentiles of the synthetic target, this median will be very close to 50, departing only due to ties in the dataset.

The third desirable enhancement is the ability to decide whether our mapping is based on a legitimate relationship, as opposed to being based on random variation. If the nominal variable that we wish to convert has no legitimate relationship with the synthetic target we are using to compute the mapping, then the whole operation is pointless. We might as well assign random numbers to the cases. This subject is discussed in an upcoming section.

Implementation Overview

The *VarScreen* program implements gating (being able to compute two separate mappings according to the value of a gate variable) as well as ignoring cases according to a gate value. It handles both options by means of a single gate variable. This does impose some generally minor limitations on the developer. On the other hand, it also simplifies operation of the program. Any reasonably competent programmer should be able to easily modify the supplied code to separate these operations, and even to include the possibility of several gate variables.

This implementation treats the optional gate variable as trinary: positive, negative, or zero. (Any value whose magnitude is less than 1.e-15 is considered to be zero.) A positive value of the gate variable places this case in one mapping category, a negative value places it in another mapping category, and zero causes the case to be ignored. It is legal to use only positive and zero values, or only negative and zero values; either situation will result in only one effective mapping to be generated. And of course, having only positive and negative values means that two maps are produced and no cases are ignored.

Testing For A Legitimate Relationship

We can use a Monte-Carlo permutation test to provide a broadly applicable method for estimating the probability that an apparently decent mapping we obtained could have been nothing more than the product of random, meaningless relationships between our nominal variable and our target. There are many different tests that we could perform. All of them share the null hypothesis that there is no relationship between the nominal variable and the target. But we can test this null hypothesis against a variety of alternative hypotheses. I have chosen the tests shown below. The first test is the only one done if there is no gate variable. If there is a gate variable, three categories are possible for the cases: positive gate mapping, negative gate mapping, and case ignored. If the gate variable takes only positive nonzero values, or only negative nonzero values, the unused 'mapping' maps all values of the nominal variable to the median rank of the target, very close to 50 except in pathological cases of extreme ties. Here are the tests performed:

- The minimum mean target rank (across all categories of the nominal variable) is subtracted from the maximum. We test whether a difference this extreme could have arisen by random chance from an unrelated nominal variable and target.

- Separately for each nominal variable category, we compute the absolute difference in mean target rank between the positive gate and the negative gate. We test whether a difference this extreme could have arisen by random luck.

- The maximum of the differences computed above is considered. We test whether a maximum difference this extreme could be just the product of random luck. The prior test, which looks at each category separately, is subject to selection bias because multiple p-values are computed. This test is immune to this particular selection bias. If you attain a significant p-value in the test above, discount its importance if you do not also attain a significant p-value for this test.

- Considering only cases having a negative gate value, compute the minimum mean target rank across all categories and subtract it from the maximum. We test whether a difference as extreme as what we observed could have arisen by random chance.

- This same test is performed by considering only cases having a positive gate value.

- We look at the greater of the two differences computed in the prior two tests and test whether a maximum difference as large as what we observed could have arisen by random chance. The prior two tests have a small but significant selection bias because we look at each gate category (positive and negative) and pay attention to whichever is more significant. This 'greater of the two differences' does not suffer from this particular selection bias.

Of course, in most applications we will be looking at a multitude of p-values, so selection bias is unavoidable. But in order to be able to examine a variety of ways in which the mapping could demonstrate that the null hypothesis (no nominal variable - target relationship) is false, we need to perform multiple tests. Thus, some degree of selection bias is unavoidable.

An Example From Equity Price Changes

The test described in this section is based on over 8500 days of closing prices of OEX, the S&P 100 index. I wondered whether the order of the most recent three day's closing prices could be used as an input to a model that predicts price movement the next day. In other words, prices increasing steadily over the prior three days might mean one thing, and a steady decrease might mean another, and a price rise followed by a price drop might mean another, and so forth. There are 3!=6 ways in which three different prices can be ordered. (I made arbitrary decisions for ties, which are not terribly common). Let C be the closing price two days ago, B be the closing price yesterday, and A be the closing price today. I assigned the six categories as shown on the next page. To accommodate price ties, the class is assigned to the last category in which it falls.

```
0:  C <= B <= A
1:  C <= A <= B
2:  B <= C <= A
3:  B <= A <= C
4:  A <= B <= C
5:  A <= C <= B
```

This is clearly a nominal variable, as there is no apparent way to sensibly assign numeric values to these categories.

It is well known that market price patterns can take very different forms in times of high volatility as opposed to times of low volatility. I decided that I wanted to compute separate mappings for exceptionally high volatility regimes and exceptionally low volatility regimes, and ignore price order when volatility is just average. (This may or may not be a wise plan in real life, but it ideally suits demonstrating this mapping technique.) This test produced the following output:

```
***********************************************************
*                                                         *
* Computing nominal-to-ordinal conversion                 *
*                                                         *
*     ORDER_CLASS is the sole predictor                   *
*        VOLATILE is the gate                             *
*       Z_DAY_RET is the target                           *
*    1000 replications of complete Monte-Carlo Test       *
*                                                         *
***********************************************************

Class bin counts...

Class    Gate-    Gate0    Gate+
    0      874      707      710
    1      471      324      378
    2      473      325      360
    3      380      324      350
    4      687      552      581
    5      412      327      313

Class bin mean percentiles...

 Class      Gate-      Gate0      Gate+
     0      47.40      50.09      48.13
     1      48.77      49.68      51.97
     2      48.04      46.74      50.96
     3      50.03      52.56      50.26
     4      50.87      50.39      54.61
     5      51.73      49.73      50.34
```

```
For each class individually, p-value for positive gate versus
negative gate...
 Class    p-value
     0    0.614
     1    0.097
     2    0.145
     3    0.939
     4    0.018
     5    0.532

p-value for max across classes of the gate +/- difference =
0.254
p-value for max class mean percentile minus min, for negative
gate  = 0.162
p-value for max class mean percentile minus min, for positive
gate  = 0.016
p-value for max of the above two = 0.024
```

By examining the bin counts, we see (not surprisingly) that for all volatility regimes, the pattern of steadily increasing price is by far the most common. A fairly distant second is the pattern of steadily decreasing prices.

The target is the log price change the next day. The table of mean target percentiles shows an interesting pattern. For both extremes of volatility, the category of steadily increasing prices shows the smallest mean target percentile. For exceptionally high volatility, the category of steadily decreasing prices shows the largest mean target percentile, and for exceptionally low volatility this category is second-largest. The largest is still a category in which the most recent price is the lowest of the three. This is evidence that mean reversion is in control, as opposed to trend following, at least for these two extremes of volatility.

Note that the 'Gate 0' category, which means 'ignore this case' still has mean target percentiles printed for the edification of the user. When the new nominal variable is generated, it will be assigned the target mean percentile, which of course will be very close to 50.

Now let's look at the p-values. We see that the difference in target mean percentiles is highly significant (0.018) for only category 4, steadily decreasing prices. However, we are picking the most significant out of six p-values, so selection bias is at work. We see that when the max across categories is considered, the p-value is an unimpressive 0.254. This tells us that we should not pay much attention to that one attention-grabbing p-value. It could easily be the product of random variation.

The maximum difference across classes for a negative gate (exceptionally low volatility) is an uninteresting 0.162. But with high volatility, we get a much more impressive p-value of 0.016. Moreover, we are inclined to take it seriously, since the selection-bias-free p-value for categories is 0.024, quite respectable.

After this test is run, a new variable is created. If this is the first time during this run of the *VarScreen* program that we performed nominal-to-ordinal conversion, this variable will be named NomOrd_1. The second time, the new variable will be NomOrd_2, and so forth. This variable can be used in subsequent tests, and it can also be written to a text file using the 'File / Write variables' menu command.