

# Screening Algorithms for Model Predictors and Financial Indicators

---

An Expanded Manual and How-To Guide  
for the Free *VarScreen* Program

For Version 4.4

**Timothy Masters**

Great effort has been undertaken in ensuring that the content of this book, as well as the associated *VarScreen* program, is as close to correct as possible. However, errors and omissions are inevitable in a work of this extent; they are surely present. Neither this book nor the *VarScreen* program, are meant as professional advice. No guarantee is made that this material is free of errors and omissions, and the reader assumes full liability for any losses associated with use of this material. The algorithms described in this book and implemented in the *VarScreen* program are experimental and not vetted by any outside experts or tested in the crucible of time. Please treat them accordingly.

About the author:

Timothy Masters received a PhD in mathematical statistics with a specialization in numerical computing. Since then he has continuously worked as an independent consultant for government and industry. His early research involved automated feature detection in high-altitude photographs while he developed applications for flood and drought prediction, detection of hidden missile silos, and identification of threatening military vehicles. Later he worked with medical researchers in the development of computer algorithms for distinguishing between benign and malignant cells in needle biopsies. For the last twenty-five years he has focused primarily on methods for evaluating automated financial market trading systems. He has authored numerous books on practical applications of predictive modeling:

*Practical Neural Network Recipes in C++* (Academic Press, 1993)  
*Signal and Image Processing with Neural Networks* (Wiley, 1994)  
*Advanced Algorithms for Neural Networks* (Wiley, 1995)  
*Neural, Novel, and Hybrid Algorithms for Time Series Prediction* (Wiley, 1995)  
*Assessing and Improving Prediction and Classification* (Apress, 2018)  
*Deep Belief Nets in C++ and CUDA C: Volume I: Restricted Boltzmann Machines and Supervised Feedforward Networks* (Apress, 2018)  
*Deep Belief Nets in C++ and CUDA C: Volume II: Autoencoding in the Complex Domain* (Apress, 2018)  
*Deep Belief Nets in C++ and CUDA C: Volume III: Convolutional Nets* (Apress, 2018)  
*Data Mining Algorithms in C++* (Apress, 2018)  
*Testing and Tuning Market Trading Systems* (Apress, 2018)  
*Modern Data Mining Algorithms in C++ and CUDA C* (Apress, 2020)  
*Statistically Sound Indicators for Financial Market Prediction* (KDP, 2020)  
*Permutation and Randomization Tests for Trading System Development* (KDP, 2020)

The *VarScreen* program may be downloaded for free from the author's website:

**TimothyMasters.info**

Copyright © 2025 Timothy Masters

All rights reserved

ISBN: 9798272551153

# Table of Contents

Preface .....	1
About the VarScreen Program .....	3
Features of the Program .....	4
About CUDA Processing .....	5
Reading a Dataset .....	7
Reading a Market History File .....	9
Univariate Mutual Information .....	10
Definition of Mutual Information .....	10
Solo p-Values via Monte-Carlo Permutation .....	11
Compensating for Selection Bias .....	12
The Stepwise Permutation Test Option .....	13
Stepwise Algorithm Details .....	15
Accelerating the Stepwise Algorithm .....	20
Serial Correlation and Cyclic Permutation .....	22
Combinatorially Symmetric Cross Validation .....	23
Specifying the Test Parameters .....	26
Contrived Examples of Univariate Mutual Information .....	30
A Small Practical Example .....	34
A Larger Practical Example .....	35
Bivariate Mutual Information .....	36
Uncertainty Reduction .....	36
Specifying the Test Parameters .....	40
Contrived Examples of Bivariate Mutual Information .....	43
A Practical Multiple-Target Example .....	48
Indicator Selection Based On Profit Factor .....	50
Demonstrating the Stepwise Option .....	54

Max Relevance / Min Redundancy Predictors. . . . .	56
Specifying the Test Parameters . . . . .	58
A Contrived Example of Relevance Minus Redundancy. . . . .	60
A Practical Example of Relevance Minus Redundancy . . . . .	64
Optimal Transforms . . . . .	68
Specifying the Test Parameters . . . . .	70
Testing Financial Market Indicators . . . . .	73
Hidden Markov Models with Target Correlation. . . . .	74
Detailed Operation of This Test. . . . .	77
Specifying the Test Parameters . . . . .	79
A Contrived Example of Hidden Markov Models . . . . .	81
A Practical Example of Hidden Markov Models . . . . .	85
Assessing HMM Memory in a Time Series . . . . .	88
Specifying the Memory Test Parameters . . . . .	90
Stationarity Test for Break in Mean. . . . .	92
The Multiple Comparisons Test. . . . .	94
The Serial Correlation Test. . . . .	98
A Major Caveat, Again . . . . .	100
A Multiple-Comparisons Demonstration. . . . .	102
Testing Correlated Market Returns. . . . .	104
Testing an Indicator. . . . .	105
Multiple Mean Breaks . . . . .	107
FREL for Feature-Weighted Classification. . . . .	108
Very Basic Theory . . . . .	108
FREL Classification in VarScreen. . . . .	112
CUDA Considerations. . . . .	117
An Inappropriate but Illustrative FREL Example. . . . .	118
A Difficult Zero-Noise Problem, Solved. . . . .	121
FSCA: Forward Selection Component Analysis . . . . .	123
Intuitive Justification of the Algorithm. . . . .	125



Desirable Properties of Principal Components . . . . .	127
Computing Principal Components . . . . .	128
A Contrived Example of Ordered Forward Selection . . . . .	131
A Contrived Example of Refined Selection . . . . .	135
A Practical Example With Indicators . . . . .	137
 LFS: Local Feature Selection . . . . .	 140
What This Algorithm Computes and Reports . . . . .	142
Specifying the Test Parameters . . . . .	144
A Contrived Example of Local Feature Selection . . . . .	148
A Practical Exploration of Local Feature Selection . . . . .	149
 Enhanced Stepwise Lin-Quad . . . . .	 152
Improving the Stepwise Process . . . . .	153
Controlling Runaway Candidate Inclusion . . . . .	154
Ensuring Statistically Sound Selection . . . . .	155
The Feature Evaluation Model . . . . .	156
Specifying the Test Parameters . . . . .	157
A Contrived Example of Enhanced Stepwise Selection . . . . .	159
A Practical Example of Enhanced Stepwise Selection . . . . .	161
 RANSAC for Devaluing Noisy Cases . . . . .	 164
The Algorithm . . . . .	166
Running the Test . . . . .	168
An Example From Market Prediction . . . . .	171
 Nominal-to-Ordinal Conversion . . . . .	 175
Measurement Level and Modeling . . . . .	177
Basic Elevation from Nominal to Ordinal . . . . .	178
Enhancing the Basic Algorithm With Gating . . . . .	180
Testing the Authenticity of the Mapping . . . . .	181
Creation of a New Variable . . . . .	184
Invoking Nominal-to-Ordinal Conversion . . . . .	185
Market Prediction Example 1: No Gate . . . . .	187
Market Prediction Example 2: Ignoring Cases . . . . .	189
Market Prediction Example 3: Full Gating . . . . .	191

Mean Reversion .....	193
Preprocessing .....	194
The Algorithm .....	196
Expected Reversions For a Random Series .....	197
Mean Reversion in the Bond Market?.....	198
Mean Reversion in the Equity Market? .....	199
Entropy Tests for Structure .....	201
Shannon Entropy .....	203
Singular Value Decomposition Entropy.....	209
Kolmogorov K2 Entropy.....	212
AP (Approximate) Entropy .....	216
Plotting .....	221
Series .....	221
Histogram .....	221
Density, Inconsistency, and Mutual Information Plots .....	222
Appendix: Version Updates.....	229

---

## Preface

Several decades ago I recognized that databases were growing at a breathtaking rate. Whereas in the early days of prediction and classification software, researchers might consider a dozen or so candidate predictor variables, hundreds of candidates for analysis were becoming common, with sets of thousands of candidates appearing on the horizon. The situation grew even more explosively when financial market traders began computing dozens or even hundreds of indicator families, each of them having dozens or hundreds of lookback lengths. Wannabe algorithmic market traders, armed with powerful desktop computers, happily threw thousands of pieces of spaghetti at the wall, eager to see which stuck. Most of these people learned the hard way that just because they hit on something good at the moment, it was unlikely that their enticing performance would continue.

Thus, it became clear to me that the prediction and classification communities, and especially financial market traders, needed statistically sound algorithms for rapidly sifting through hundreds or thousands of predictor candidates, finding those that would most likely be useful for a particular application. Of special importance is that these algorithms should correctly account for the fact that when one searches through a large number of competitors, some are bound to be lucky and hence perform unjustifiably well. This painful fact is ignored all too often, with occasionally catastrophic results.

With this motivation in mind, twenty or so years ago I began writing a program for performing large-scale variable sifting in a way that not only separates the wheat from the chaff, but also provides indications to the user of how reliable this sifting appears to be. I called this program *VarScreen*, and made it as well as a user's manual freely available for download. Since then, new algorithms have been regularly added. As of this writing, I have released 28 versions of the program, each having new features.

As is easily imagined, numerous users wrote to me, asking for source code for the algorithms. I knew that if I emailed code back to them, I would be bombarded with questions about how I do what, how to modify the code, and so forth. So I wrote several books, most currently published by the Apress imprint of Springer Science, in which I delve into the theory and mathematics behind the algorithms. These books also include C++ source code, as well as CUDA C code for the most compute-intensive algorithms.

The book you are reading has no computer code and only minimal mathematics; it serves strictly as a comprehensive user's manual for the *VarScreen* program, along with detailed examples of practical use of the program. Much of this book is available (and has been for decades!) as a free pdf download from my website, *TimothyMasters.info*. However, this book contains more examples than the web version, as well as documentation for several algorithms added since the most recent web version of the documentation.

For your convenience, here is a list of my other published books in which you can find theoretical and mathematical details of the *VarScreen* algorithms, along with source code:

*Assessing and Improving Prediction and Classification*

Mutual information and uncertainty reduction

Maximum relevance / minimum redundancy predictor sets

*Data Mining Algorithms in C++*

Mutual information and uncertainty reduction

Maximum relevance / minimum redundancy predictor sets

Combinatorially Symmetric Cross Validation (CSCV)

FREL (Feature Weighting as Regularized Energy-Based Learning)

*Modern Data Mining Algorithms in C++ and CUDA C*

*Testing and Tuning Market Trading Systems*

## About the VarScreen Program

*VarScreen* contains in one easy-to-use program a variety of software tools useful for the developer of predictive models, including those used for financial market prediction. 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 has been 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 standard analysis software.

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

## 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, nearly all over-the-counter computers contain a CPU with six or more 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; modern video cards contain thousands of 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. As of this writing, 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 work 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. Whenever a large test involving CUDA computations is running, a progress bar is displayed. This bar 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). This is no fun.



## Reading a Dataset

*VarScreen* reads data files that are in a fairly standard 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, a 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.

---

## Reading a Market History File

*VarScreen* can also read a financial market price history file in standard format, thus eliminating the need to insert a header in the file. The program creates variables with the following names: Date, Open, High, Low, Close, Volume. Each bar must contain the values in this order. Volume is optional, and if omitted will default to a value of 1. A typical market price history file might look like this:

```
19880211 122.32 122.89 121.42 121.92 3272
19880212 121.92 123.37 121.82 122.72 3155
19880216 122.72 123.92 122.14 123.92 2708
19880217 123.92 124.88 123.00 124.14 3905
...
```

The date must not contain any special characters, such as a slash or dash; it must be strictly numeric and expressed as a single number, typically YYYYMMDD as shown in the example above.

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.

## 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. These algorithms, with source code, are in my book “Data Mining Algorithms in C++”, and the stepwise permutation test for unbiased p-values is described, along with source code, in my book “Permutation and Randomization Tests for Trading System Development”.

In a few pages we will demonstrate how the various parameters for this test are specified by the user. But first, we will define mutual information, discuss the parameters of the test, and describe the results of the test that will be printed for the user.

### Definition of Mutual Information

Roughly stated, information that is shared between two random variables is called their *mutual information*. We can make this slightly more rigorous. Suppose  $A$  and  $B$  are two random variables. If knowledge of the value of one of these variables provides information about the distribution of the other, information which we would not possess in the absence of this knowledge, then these variables have mutual information.

This may be made even more clear if we look at Equation (1), which is the definition of discrete mutual information, and Equation (2), which defines continuous mutual information.

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log \frac{p(x, y)}{p(x) p(y)} \quad (1)$$

$$I(X; Y) = \iint f_{X,Y}(x, y) \log \frac{f_{X,Y}(x, y)}{f_X(x) f_Y(y)} dx dy \quad (2)$$

Two random variables are independent if and only if their joint distribution (the numerator) is equal to the product of their marginal distributions (the terms in the denominators). In this case, the fraction whose log is taken is 1. Because the log of 1 is 0, the mutual information for a pair of independent variables is 0. But suppose the two variable have a relationship. When the joint distribution exceeds the product of the marginals, the log is positive and the probability multiplier will be relatively large, while in the opposite situation the log will be negative and the multiplier will be relatively small. So any imbalance will result in a net positive mutual information.

## Solo p-Values via Monte-Carlo Permutation

My books referenced on the prior page describe several useful permutation tests whose details are beyond the scope of this text. Here, I will discuss only their applications. The user would normally specify a large number (at least 1000) of *Monte-Carlo replications*. Two or three columns of probabilities are printed for each candidate predictor. The first, *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. *Each candidate is tested individually, ignoring other competitors.*

## Compensating for Selection Bias

The problem with the *Solo pval* is that if more than one candidate is tested (the usual situation!), then it can easily happen that some truly worthless candidate will have a lucky run, and thereby exhibit large but unjustified mutual information. In this case, the candidate will attain a very small *Solo pval*, and likely fool the developer into thinking that this worthless candidate is actually valuable. As a matter of fact, if all candidates are worthless, the *Solo pvals* will follow a uniform distribution; a candidate will obtain a p-value of 0.1 or less with probability 0.1, a p-value of 0.01 or less with probability 0.01, and so forth. So, for example, if you are screening 100 independent candidates, all of which happen to be worthless, it is nearly certain that at least one of them will attain a solo p-value of 0.01 or less, despite being worthless.

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 done at least as well as it did by luck alone if all candidates were truly worthless.

The *Unbiased pval* is printed for all candidates, not just the best. The interpretation of unbiased p-values for the other, lesser candidates depends on whether the *Stepwise* method is selected by the user. If it is not selected, the *Unbiased pval* is an upper bound for the true unbiased p-value of the candidate. Because it is an upper bound, a very small *Unbiased pval* for a candidate is good evidence that the candidate has legitimate predictive power. However, 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 p-value. The fact that it is only an upper bound on the true p-value reduces its power; it can easily fail to identify effective predictors.

---

## The Stepwise Permutation Test Option

The stepwise permutation test described in this section is my own adaptation of the algorithm presented in Romano and Wolf (2016) “Efficient Computation of Adjusted p-Values for Resampling-Based Stepdown Multiple Testing” and described in detail in my book “Permutation and Randomization Tests for Trading System Development”. This algorithm should be applicable to *any* ‘best-of’ MCPT. It’s just that the programming is more complex than the traditional algorithm, and it runs much more slowly, so I have implemented this particular screening algorithm for only a few of the fastest screening techniques. I may implement it for other screening algorithms later.

Why consider this stepwise option? For all its utility, the traditional (not stepwise) ‘best-of’ algorithm suffers from two annoying weaknesses:

- 1) The null hypothesis is that *all* competitors are unrelated to the target. This is a significant restriction, at least theoretically. In practice, this restriction seems to have little or no apparent ill effect when violated, but it makes me uncomfortable.
- 2) The computed probability is strictly correct only for whichever competitor has the greatest relationship with the target. All other best-of probabilities are upper bounds on the true probabilities. This fact was discussed in the prior section.

The second problem is not always devastating, because *all* competitors for which the computed unbiased probability is less than or equal to the desired alpha level (maximum tolerable p-value) for the test can be considered to be probably related to the target, at least relative to the alpha level. That joint statement should satisfy the alpha level because if the poorest of those that satisfy alpha does so, then certainly all those superior to it do as well.

On the other hand, even this result is not ideal because we could easily miss some competitors that are truly related to the target. If their computed probabilities overestimate the true probabilities under the null hypothesis to a degree that causes the computed probability to exceed  $\alpha$ , despite there being a relationship with the target, then we have missed this competitor. This is a significant problem, and the stepwise algorithm solves it.

The stepwise procedure is almost entirely better than the traditional one-shot best-of method, which pools all candidates into a single batch with the null hypothesis that they are *all* unrelated to the target. The stepwise method tests each null hypothesis *individually*, but with the familywise error rate (FWE) controlled by our desired  $\alpha$ . The FWE is the probability of rejecting *one or more* of the *true* individual null hypotheses. More loosely speaking, FWE is the probability of making even one mistake in identifying individual null hypotheses to reject.

FWE comes in two forms. An FWE with *weak control* is one that requires that *all* null hypotheses be true. This is what we have in the traditional best-of test. Far more desirable is an FWE with *strong control*, which means that it holds regardless of which or how many of the null hypotheses are true. This, of course, corresponds better to real life. In my own professional work I have always acted as if the traditional best-of test has strong control even though it does not, and it's never come back to bite me. Much heuristic evidence supports that use. Still, a method with strong control would be better when it can be achieved.

An even more desirable property of a best-of test is that it have as much power as possible. There are many possible definitions of power. At one extreme we might want to maximize the probability of rejecting *at least* one false null hypothesis. At the other extreme we might want to maximize the probability of rejecting *all* false null hypotheses. Those are both too extreme, one with too little demanded and one with too much. More reasonably we might want to maximize some measure of average rejection probability. This intermediate goal, perhaps maximizing the average probability of rejecting



false null hypotheses, is doubtless the best, and is a property that I believe is possessed by the stepwise algorithm.

It's important to understand this property of maximum power, because it is very important in practice. Recall that the traditional best-of algorithm provides only upper bounds for the p-values for all competitors except the best. This makes it possible that it will fail to reject null hypotheses (decide that there is a relationship) for competitors that truly have a relationship with the target. That's the beauty of this new stepwise algorithm: it can often flag competitors that would have been missed by the traditional algorithm due to overestimation of p-values, while still maintaining a user-specified familywise error rate.

In summary, we want to be able to test each individual competitor's null hypothesis while having strong control of the FWE and maximizing average power. The traditional best-of algorithm has only weak control of the FWE and it has excellent power only for whichever competitor is the best (maximum relationship with the target).

I believe that the stepwise algorithm provides these superior properties. The algorithm is shown soon. But first I want to discuss the general philosophy of the procedure so as to make the algorithm more clear.

## Stepwise Algorithm Details

Because the stepwise MCPT algorithm is not well known in the modeling community, I will now provide details that will interest some readers. If you are not interested, you may safely skip this long section.

This is a stepwise procedure, with hypotheses being rejected one at a time, in order starting with the best competitor (largest target relationship) and working downward until no more null hypotheses can be rejected at the user-specified FWE,  $\alpha$ . As each null hypothesis is tested, we

approximate the null hypothesis distribution of that relationship statistic by permuting the target as in the traditional algorithm but finding the maximum of *only the populations that have not yet been rejected*. This is the critical difference between this improved algorithm and the traditional ‘best-of’ algorithm. If, for each step, we were to approximate the null hypothesis distribution by finding the maximum relationship statistic of *all* permuted populations we would have an algorithm that is essentially identical to the prior, more traditional algorithm, just re-ordered as stepwise instead of all at once. However, in the stepwise algorithm, the number of populations that go into the computation of the maximum relationship statistic is reduced by one for each step, thus shrinking the null distribution.

In summary, this algorithm is almost identical to the traditional algorithm, except that instead of testing all null hypotheses at once we test them one at a time, and as we do successive tests we keep shrinking the number of competing distributions that go into approximating the null distribution.

I’ll now walk through the algorithm listed on the next page, and continue the walkthrough after the listing. The user has specified that there are  $n$  competitors (indicators here), and the test will employ  $m$  permutations (hopefully thousands) to estimate the null hypothesis distributions. A desired alpha level (maximum FWE that the user can accept) for the test has also been specified.

The first step is to compute the relationship statistic for each competitor and store them in the `original` array. We’ll also need to sort them so that the stepwise procedure can proceed from best (largest) to smallest. But we must not disturb the order of `original`, so we copy that array to a `work` array and sort it ascending. We also initialize `sort_indices` to an identity array, and when we do the sorting we simultaneously move the elements of this array. Thus, after sorting, `sort_indices[0]` will be the index of the competitor having the smallest relationship, `sort_indices[1]` the next smallest, and so forth. Later, the stepwise procedure will work backwards through this array to test the competitors in order from best to worst.

---

```

// Compute the relationship for each competitor and sort

For i from 0 through n-1
    sort_indices[i] = i ;
    original[i] = relationship of competitor i with Y
    work[i] = original[i] ;

Sort work ascending, moving sort_indices simultaneously

// Initialize: no competitors yet passed (null rejected)

For i from 0 through n-1
    passed[i] = FALSE ;

// The stepwise accumulation loop begins here

For step from n-1 through 0, backwards (best to worst)
    this_i = sort_indices[step] Index of best remaining
    count[this_i] = 1 ; Counts right-tail probability

    // Permutation loop estimates null distribution

    For irep from 1 through m    Do all replications
        Shuffle Y
        max_f = minus infinity
        For i from 0 through n-1    All competitors
            if (NOT passed[i]) Do only those not rejected
                this_f = relationship of competitor i with Y
                if (this_f > max_f) Keep track of maximum
                    max_f = this_f ;

        If (max_f >= original[this_i])
            ++count[this_i] ; Count right-tail probability
        } // For irep

// See if this new competitor passed (NULL rejected).

If count[this_i] / (m+1) <= alpha
    passed[this_i] = TRUE

Else
    Break out of step loop; we are done

```

As competitors have their null hypotheses rejected, we keep track of which have been rejected via the `passed` array, where a `TRUE` value means that its null hypothesis has been rejected; it passed the test for having a relationship with the target. Prepare for the stepwise accumulation loop by initializing `passed` to `FALSE` for all competitors.

The stepwise accumulation loop now begins. It moves backwards through the competing indicators because they have been sorted ascending and we want to begin with the best. Recall that `sort_indices` contains the indices of the sorted competitors, so we place in `this_i` the index of the competitor that is about to be tested for inclusion in the set of rejected null hypotheses. Initialize the counter of right-tail probability to 1 before performing the loop that approximates the null hypothesis distribution of the relationship statistic.

The permutation loop is now executed. Shuffle the target and initialize `max_f` to any number that is smaller than the smallest possible relationship statistic. This variable will keep track of the maximum relationship statistic in this replication. We now come to the part of the algorithm that distinguishes it from the traditional best-of algorithm. In that prior algorithm we found the maximum relationship statistic across *all* competing populations. But in this algorithm we exclude those competitors whose null hypotheses have already been rejected. So inside the loop that passes through all populations we process only those for which `passed` is `FALSE`. After we find the maximum we compare it to the original value of the competitor being tested and increment the right-tail probability counter if this null hypothesis value equals or exceeds the original value.

After all permutation replications are complete we have an estimate of the right-tail probability of the relationship statistic for competitor `this_i`. All we need to do at this point is compare this probability to the user-specified `alpha`. If it is less than or equal to `alpha` we add it to the accumulated collection of passing competitors (those that we conclude have a relationship

with the target). But if it did not pass, we are done, so break out of the accumulation loop.

Here is a rough overview for why this algorithm has an FWE of  $\alpha$  with strong control, and also maximizes the average probability of rejecting false null hypotheses. Consider the best competitor, the one having the greatest relationship statistic and hence the one that we test first. Suppose its null hypothesis is true. By implication its relationship statistic will have the same distribution as that for all permutations (under the usual assumption that the target values are independent and identically distributed). Thus we will erroneously reject this null hypothesis with probability  $\alpha$ . If we do so, it does not matter what errors we may subsequently make for other populations, because the definition of FWE is the probability that we will make *one or more* rejection errors. On the other hand, if we do not reject this null hypothesis, we are finished testing populations, so there is no more opportunity to make an error.

Now suppose the first null hypothesis is false. I claim (without rigorous proof) that the permutation test as described is the most powerful possible test for detecting this false null hypothesis. This should be a no-brainer, because we are testing the observed statistic against an asymptotically exact estimate of its actual distribution. If we declare this null hypothesis true (incorrectly, but not affecting FWE), we are finished testing populations for inclusion, so there is no more opportunity to make an error. If we declare it to be false we are correct and we advance to the next candidate.

When we advance to the next candidate, we are in exactly the same situation we were in with the first candidate, but now that first candidate is entirely removed from further computation. Its relationship statistic is no longer referenced, and that population no longer takes part in estimating the null hypothesis distribution of this next candidate. So if this second candidate's null hypothesis is true, we have probability  $\alpha$  of incorrectly rejecting it. All other logic is exactly as it was for the first candidate.

This repeats until eventually we do not reject a null hypothesis, at which point we stop. We have alpha probability of having erroneously rejected a true null hypothesis at least once along the way, and thus we have a FWE of alpha, as desired. This fact holds regardless of how many null hypotheses are true, so our FWE has strong control, as desired. Finally, each time we encounter a false null hypothesis we employ the most powerful test possible to test that hypothesis, and so we have maximum average probability of correctly rejecting false null hypotheses.

These assertions are distressingly heuristic, with little in the way of rigor to back them up. However, the intuition seems sound to me. Moreover, I have run massive quantities of Monte-Carlo simulations, using multiple alpha levels, multiple numbers of cases, multiple numbers of candidate populations, and various proportions of the candidates (from 0 to most) having false null hypotheses. In every case, the FWE came in at almost exactly the specified alpha level, well within normal variation tolerances. And this test has amazing apparent power to detect even minuscule degrees of relationship between X candidates and Y. So I am confident enough in its practical utility to use it in my own work and recommend it to others.

## Accelerating the Stepwise Algorithm

The algorithm shown on Page 17 is the best way to present the stepwise method, because it is a straightforward implementation of the mathematical statement. However, it is unnecessarily slow. This is because the block of permutations does not need to be repeated each time a new competitor is tested for inclusion. We need to do it only once, estimating all null hypothesis distributions simultaneously. Then we can do the stepwise inclusion after the permutations are complete. To collect all distributions at once, we work from worst to best, updating the 'maximum so far' as each increasingly good competitor is added to the mix. Here is the fast but mathematically identical algorithm:

```

For i from 0 through n-1
    sort_indices[i] = i ;
    original[i] = relationship of competitor i with Y
    work[i] = original[i] ;

```

Sort work ascending, moving sort\_indices simultaneously

*Step 1 of 2: do the random replications and count right tail*

```

For i from 0 through n-1
    count[i] = 1 ; Counts right-tail probability

```

```

For irep from 1 through m
    Shuffle Y

```

```

max_f = number smaller than smallest possible
For i from 0 through n-1    Work from worst to best
    this_i = sort_indices[i]
    this_f = relationship between this_i and Y
    if (this_f > max_f) Keep track of maximum
        max_f = this_f

    If (max_f >= original[this_i])
        ++count[this_i] ; Count right-tail probability
    } // For irep

```

*Step 2 of 2: Do the stepwise inclusion*

```

For i from n-1 through 0    Work from best to worst
    this_i = sort_indices[i] Index of best remaining
    If count[this_i] / (m+1) <= alpha
        Accept this competitor
    Else
        Break out of step loop; we are done

```

## Serial Correlation and Cyclic Permutation

The user must be aware of a vital caveat to the permutation tests: 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 an option called *cyclic permutation* that somewhat helps in this situation, but it is not a complete cure.

Cyclic permutation works by rotating the target series by a random amount for each replication. The rotation wraps around the end. So, for example, if the target series is 10 observations long (ridiculously few!), one permutation might be 3, 4, 5, 6, 7, 8, 9, 10, 1, 2. This type of permutation preserves any serial correlation in the target series, except near the wraparound (10 to 1 in this example). This greatly reduces the anti-conservative nature of the test, though it does not eliminate it entirely. At the same time, it destroys the original pairing of observations in the two series, so we are able to simulate a pretty decent approximation to the null hypothesis distribution.

Cyclic permutation slightly reduces the power of the test in that unless there are a huge number of cases, there will likely be some duplication of pairings across replications. But as long as there are a large number of cases relative to the number of replications, the reduction in power will generally be negligible. For this reason, if you have any doubts about the possibility of serial correlation, you should almost certainly use cyclic permutation.

It should be noted that serial correlation is not a problem for only permutation tests. It is a deal killer for nearly all traditional statistical tests such as t-tests and most bootstraps.



---

## Combinatorially Symmetric Cross Validation

The final column printed, labeled **P( $\leq$ median)** 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 and some other *VarScreen* tests.

Here is the motivation for this test. 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 with the target, 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) true 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 performance in our dataset would exceed the median OOS performance of all candidates*. If not, the selection process is useless; no ultimate superiority is obtained by choosing the best performer in the dataset used for selection. It’s a low bar, but reasonable nonetheless.

The test employs a complex form of cross validation called *combinatorially symmetric cross validation* (CSCV) to estimate the probability of each competitor achieving this minimal standard of performance. For each fold it finds the best in-sample competitor, finds its OOS performance, finds the OOS performance of all competitors, and compares the former to the median of the latter. These test results are averaged over all folds in order to compute the probabilities printed in the rightmost column.

The figure printed for the first row (the best candidate) is the estimated probability that the OOS mutual information of the best candidate in a given partition will be less than or equal to the median OOS performance of all of

the other candidates in that partition. Obviously, if this probability is small we can be confident in the superior quality of this candidate relative to the others.

Note, by the way, that *the best candidate in each partition will not in general always be the best candidate for the entire dataset*. If the relationship is strong, there will be a strong tendency for all partitions to have the same best candidate as the best in the complete dataset, but this need not be the case, especially if the relationship is weak.

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 for a partition will have OOS performance less than or equal to the median for that partition. 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.

This is such an important concept that it bears repeating and amplification. The table is printed in decreasing order of relationship between the candidate and the target, as measured in the entire dataset. But for the CSCV median test, the relationship between the named variable and its printed probability criterion is looser. For example, consider the CSCV probability printed for the third-best candidate in the entire dataset, the third row. The printed probability refers to the median OOS performance of whichever candidate was third-best in a fold, averaged over all folds. When we associate this figure with the third-best candidate in the entire dataset, as we do in this table, we are making a somewhat loose statement: averaged across all folds, the third-best candidate as measured in-sample performed this well out-of-sample, so we assume that this performance will also apply to the third-best candidate measured for the entire dataset. In general, this is a reasonable assumption for good performers, and more questionable for poor performers. But of course, for poor performers it doesn't matter anyway!

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). If you have a large quantity of equally powerful candidates, the distribution of probabilities will be more random. Which brings us to the final important point...

It's vital to understand that the CSCV median test is not so much a test of *individual* quality as it is of quality of each competitor *relative* to its fellows. It tells us whether one or more competitors that apparently rank superior to the others are truly superior, or if they could have obtained their exalted position in the ranking by sheer good luck. It may be that one competitor is *very poor* while all of the others are *extremely poor*. In this case, the *very poor* candidate will score a small probability, which tells us that it is truly superior to the others, but which gives no indication that its absolute power is very poor. On the other hand, all competitors may be *extremely good*. In this case we may find that no competitor gives a nicely small p-value. All this tells us is that no competitor stands out; it tells us nothing about absolute power.

***This test is all about relative power, not absolute power.***

The implication of this fact is that in order for this test to be most effective, we need for the collection of competitors to be, in some sense, a 'fair' representation of the possible competitors; it should be 'complete' and 'representative'. By this I mean that your list of competitors should contain most or all of the possible competitors envisioned by the model designer, and no competitors that are inappropriate for the application. In practice, all that we can reasonably ask is that you not include any competitors that you know in advance will be very poor, and that you do not omit candidates just because you know in advance that they will be good. For example, suppose you want to look back in recent market history to predict tomorrow's move, so you choose a large number of candidate indicators. Suppose you know from past experience that a 10-day RSI has great power in this application. Don't omit it just because you know it will be good. Let it compete with the others. And don't include the number of millimeters of rain that fell in Portland yesterday.

## Specifying the Test Parameters

When the user clicks *Tests / Univariate Mutual Information*, a dialog similar to that shown below will appear. Parameter descriptions follow.

Mutual information between predictors and a target

Predictors	Target
_SEQNUM_	_SEQNUM_
CMMA_10	CMMA_10
CMMA_20	CMMA_20
CMMA_5	CMMA_5
DAY_RETURN	DAY_RETURN
LIN_ATR_15	LIN_ATR_15
LIN_ATR_5	LIN_ATR_5
LIN_ATR_7	LIN_ATR_7
LINDEV_10	LINDEV_10
LINDEV_20	LINDEV_20
LINDEV_5	LINDEV_5
PVFIT_15	PVFIT_15
PVFIT_7	PVFIT_7
RSI_10	RSI_10
RSI_20	RSI_20
RSI_5	RSI_5
RTVY_12	RTVY_12
RTVY_25	RTVY_25
RTVY_6	RTVY_6

Predictor bin definition

☐ Predictors and target continuous

☒ Use all cases Predictor bins

☐ Use tails only Tail fraction (0 to 0.5)

Target bins

Monte-Carlo Permutation Test

☒ Complete

☐ Cyclic

Replications

Stepwise

☐ Stepwise method

Alpha

Cancel OK

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 only 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 are completely ignored in the mutual information calculation. This method is especially useful in high-noise situations, such as prediction of financial markets. The CSCV probability that superior mutual information will hold up out-of-sample is not 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 or more, 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 permuted 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.

This topic is also discussed on Page 22.

If the *Stepwise* box is checked, the stepwise permutation test described starting on Page 13 is used. In this case the user must also specify an alpha level. This is the required familywise error rate. Selection of candidates will cease before this error rate is exceeded. This alpha level is the probability that we will make one or more errors in selecting indicators.

## Contrived 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:

```
*****
*
* Computing univariate mutual information (one predictor, one target) *
* 12 predictor candidates *
* 5 predictor bins *
* 5 target bins *
* 10000 replications of complete Monte-Carlo 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 and the CSCV median test delineate the break.

The next example shows what happens when worthless, serially correlated predictors are tested with a serially correlated target. We use DEP\_RANDOM1 - DEP\_RANDOM9 to predict DEP\_RANDOM0, a situation which should demonstrate no predictive power whatsoever. The mutual information table is as follows:

```
-----> Mutual Information with DEP_RANDOM0 <-----
```

Variable	MI	Solo pval	Unbiased pval	P(<=median)
DEP_RANDOM2	0.0044	0.0001	0.0002	0.6944
DEP_RANDOM4	0.0030	0.0018	0.0175	0.6190
DEP_RANDOM3	0.0025	0.0110	0.0881	0.6270
DEP_RANDOM6	0.0023	0.0249	0.2004	0.5516
DEP_RANDOM9	0.0023	0.0242	0.2062	0.5397
DEP_RANDOM8	0.0023	0.0287	0.2284	0.5079
DEP_RANDOM1	0.0022	0.0317	0.2494	0.4960
DEP_RANDOM5	0.0019	0.0883	0.5509	0.4325
DEP_RANDOM7	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. Only the CSCV median test probabilities catch the situation correctly.

This table illustrates an interesting occasional behavior of the p-values and the CSCV median test. Superficially, one would think that the probabilities would appear in ascending order. For example, one would think that the p-values and CSCV median probabilities for the best performer would be smaller than those for the second-best performer. And this would almost certainly be the case if the quality ordering is strong. But when performances are nearly tied, random variations can impact ordering.

It should be emphasized that underestimation of p-values when there is serial correlation in the target and one or more predictors 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 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_RANDOM <-----
```

Variable	MI	Solo pval	Unbiased pval	P(<=median)
DEP_RANDOM2	0.0044	0.0513	0.3529	0.6944
DEP_RANDOM4	0.0030	0.2408	0.9316	0.6190
DEP_RANDOM3	0.0025	0.3976	0.9918	0.6270
DEP_RANDOM6	0.0023	0.5007	0.9976	0.5516
DEP_RANDOM9	0.0023	0.5237	0.9982	0.5397
DEP_RANDOM8	0.0023	0.4719	0.9988	0.5079
DEP_RANDOM1	0.0022	0.5344	0.9990	0.4960
DEP_RANDOM5	0.0019	0.6643	1.0000	0.4325
DEP_RANDOM7	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-values are far from significant. So in at least this example, the cyclic modification largely solves the problem of serially dependent data.

## A Small Practical Example

I created a database of normalized indicators and a target from the S&P100 index day bars, OEX. For all indicators, the number of bars to look back terminates the indicator name. These indicators and target are:

**CMMA\_N** Today's close minus moving average  
**LIN\_ATR\_N** Slope of straight line fit to log prices, normalized by ATR  
**RSI\_N** Traditional RSI (relative strength index)  
**LINDEV\_N** Today's price minus that predicted by linear extrapolation  
**PVFIT\_N** Slope of straight line predicting log close from log volume  
**RTVY\_N** Gietzen's reactivity  
**DAY\_RETURN** Log ratio of open-to-open returns one day ahead

I ran a univariate mutual information test for these indicators predicting the target, and received these somewhat surprising and interesting results:

```
*****
*
* Computing univariate mutual information (one predictor, one target)
* 17 predictor candidates
* Predictors and target are continuous
* 1000 replications of complete Monte-Carlo Permutation Test
*
*****

-----> Mutual Information with DAY_RETURN <-----

Variable          MI          Solo pval    Unbiased pval    P(<=median)
CMMA_20           0.0372          0.0010          0.0010          0.0556
RSI_10            0.0358          0.0010          0.0010          0.0556
RTVY_25           0.0350          0.0010          0.0010          0.0794
CMMA_10           0.0348          0.0010          0.0010          0.1429
RSI_20            0.0342          0.0010          0.0010          0.1865
RSI_5             0.0319          0.0010          0.0010          0.2143
RTVY_12           0.0308          0.0010          0.0010          0.2143
CMMA_5            0.0292          0.0010          0.0010          0.4087
LIN_ATR_7         0.0259          0.0010          0.0010          0.5556
LIN_ATR_5         0.0248          0.0010          0.0010          0.6310
LIN_ATR_15        0.0219          0.0010          0.0010          0.7222
RTVY_6            0.0206          0.0010          0.0010          0.7738
PVFIT_7           0.0136          0.0010          0.0010          0.9643
PVFIT_15          0.0126          0.0010          0.0010          0.9960
LINDEV_20          0.0089          0.0020          0.0030          1.0000
LINDEV_5           0.0030          0.0040          0.0870          1.0000
LINDEV_10          0.0029          0.0100          0.0880          1.0000
```

It's only a little surprising that the solo p-values are all tiny, most of them at the minimum possible p-value of 1/1000 replications. It doesn't take a lot of relationship for solo p-values to be small, though being this small is a bit surprising. What did amaze me is how all but two or three of the unbiased p-values are also tiny. Clearly, there is a mutual information relationship between most or all of these indicators and the target. But note that none of the CSCV median test probabilities are tiny, and in fact only two or three of them are even modestly small. This means that even the best (largest relationship) indicators fail to stand out from their competitors.

## A Larger Practical Example

To demonstrate a larger application of the univariate mutual information test, I created a file containing 220 candidate indicators, along with the same one-day-ahead return target as in the prior example. I looked for a relationship with only the tail values, used 10,000 replications, and used the stepwise method with the tiny alpha of 0.0002. Almost shockingly, even with this ridiculously small familywise error rate it picked up five extremely significant indicators. There is a probability of only 2/10,000 that one or more of these indicators were selected in error. I'm impressed.

```
*****
*
* Computing univariate mutual information (one predictor, one target)
* 220 predictor candidates
* 0.100 predictor tails used
* 3 target bins
* 10000 replications of complete Monte-Carlo Permutation Test
* 0.0002 is user-specified alpha level
*
*****

-----> Mutual Information with DAY_RETURN <-----

Variable      MI      Solo pval  Unbiased pval
PVI_20        0.0691      0.0001      0.0001
CMAA_20N      0.0474      0.0001      0.0001
CMAA_20        0.0465      0.0001      0.0001
ATRRAT_10     0.0431      0.0001      0.0001
LIN_ATR_5     0.0403      0.0001      0.0002
Best remaining p-value=0.0003 while alpha=0.0002, so quitting
```

## Bivariate Mutual Information

Examining one candidate predictor at a time is quick and simple, but it has a major drawback: a candidate may have little predictive power alone, but be very useful when used in combination with another variable. For example, imagine an application in which we have two variables that, under normal conditions, track each other closely. But if these variables diverge, something has gone wrong. In this situation, either variable alone may be worthless, while having both of them is highly predictive.

We can expand the testing procedure by examining target candidates in addition to predictor candidates. It may be that we have an application in which being able to successfully predict any of several competing targets is good enough; it doesn't matter which target we predict, as long as we can predict at least one of them well. In such an application it may benefit us to place several targets in competition, in case it happens that some targets are easier to predict than others. For example, we may be testing a financial market application in which we don't know the best length of time for keeping a trade position open. So we might create several targets, each corresponding to a different holding time, and test to see which holding time provides the most predictable outcome.

## Uncertainty Reduction

When we have several targets in competition, we face an issue that is of little or no consequence when only predictors compete. Look at Equation (3), which is one of many equivalent definitions for mutual information. In this equation,  $H(X)$  is the entropy of the predictor  $X$ , and  $H(X|Y)$  is the entropy of  $X$  if we know  $Y$ . The other two terms are similarly defined. Mutual information is symmetric; it measures only the information in common between  $X$  and  $Y$ .

$$I(X; Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) \quad (3)$$

When we have only one target, the entropy of  $Y$  is the same for all competing predictors, so the predictors supply the only variation in mutual information. But when we have multiple competing targets, this symmetry is a problem, because in nearly all cases our application is asymmetric: we are interested only in the degree to which each  $X$  lets us predict  $Y$ , not the degree to which each  $Y$  lets us predict  $X$ . If the different  $Y$  variables, the competing targets, have different entropies, the usual case, this will pollute our measurements. In particular, a given amount of mutual information would signify a lot less predictive power for a target with high entropy than for a target with low entropy.

The solution to this problem is simple: we normalize the mutual information by the entropy in the target, as shown in Equation (4). This quantity is usually called the *Uncertainty Reduction*, because it measures the degree to which knowledge of  $X$  reduces the uncertainty (entropy) of  $Y$ . Consider the term  $H(Y|X)$  in this equation. If  $X$  provides no information about  $Y$ , that term will equal  $H(Y)$ , so the uncertainty reduction will be zero. On the other hand, if knowing  $X$  gives us  $Y$ , perfect prediction, that term will be zero, and the uncertainty reduction will be one. Because of this nice normalization, we may wish to employ uncertainty reduction as our criterion even when there is only one target. This will produce exactly the same quality ordering of predictors as straight mutual information, but the number will be more meaningful.

$$\text{Uncertainty reduction} = \frac{H(Y) - H(Y|X)}{H(Y)} \quad (4)$$

Note, by the way, that these concepts are described in much greater detail and rigor in my two books “Assessing and Improving Prediction and Classification” and “Data Mining Algorithms in C++”.

The *Bivariate Mutual Information* test 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. Please see Page 11 for more details.

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. Please see Page 12 for more details.

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.

By choosing the *Stepwise* option for the permutation test, an alternative algorithm can be used to eliminate this problem of overly conservative unbiased p-values. Please see Page 13 for more details.

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 an option, *Cyclic Permutation*, that somewhat helps in this situation, but it is not a complete cure. Please see Page 22 for details.

## 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
_SEQNUM_	_SEQNUM_
CMAA_10	CMAA_10
CMAA_20	CMAA_20
CMAA_5	CMAA_5
DAY_RETURN	DAY_RETURN
LIN_ATR_15	LIN_ATR_15
LIN_ATR_5	LIN_ATR_5
LIN_ATR_7	LIN_ATR_7
LINDEV_10	LINDEV_10
LINDEV_20	LINDEV_20
LINDEV_5	LINDEV_5
PVFIT_15	PVFIT_15
PVFIT_7	PVFIT_7
RSI_10	RSI_10
RSI_20	RSI_20
RSI_5	RSI_5
RTVY_12	RTVY_12
RTVY_25	RTVY_25
RTVY_6	RTVY_6

Cancel OK

Predictor bins 3

Target bins 3

Criterion

- ☐ Mutual information
- ☒ Uncertainty reduction

Monte-Carlo Permutation Test

- ☒ Complete
- ☐ Cyclic

Replications 0

Max printed 100

Stepwise

- ☐ Stepwise

Alpha 0.10

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 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 be a meaningful measure of their relationship. 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. (Note that if the targets are all continuous or nearly so, they will all have about the same entropy, due to the bins being the same size. But if there are numerous ties, the targets can have widely disparate entropies.)

What we 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. This was discussed in more detail on Page 36.

*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 several thousand, 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. Please see Page 22 for more details.

If the *Stepwise* box is checked, the alternative stepwise permutation test described on Page 13 is used. In this case the user must also specify an alpha level. This is the required familywise error rate. Selection of candidates will cease before this error rate is exceeded. This alpha level is the probability that we will make one or more errors in selecting predictor/target sets.

## Contrived 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.

```
*****
*
* Computing bivariate mutual information (two predictors, one 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 Monte-Carlo 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 <-----

Predictor 1	Predictor 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 is 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 pval	Unbiased pval
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. Serial dependency 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\_RANDOM0 <-----

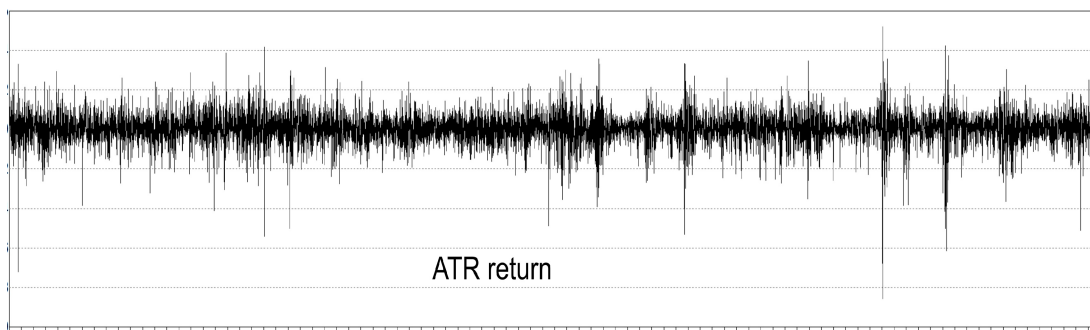
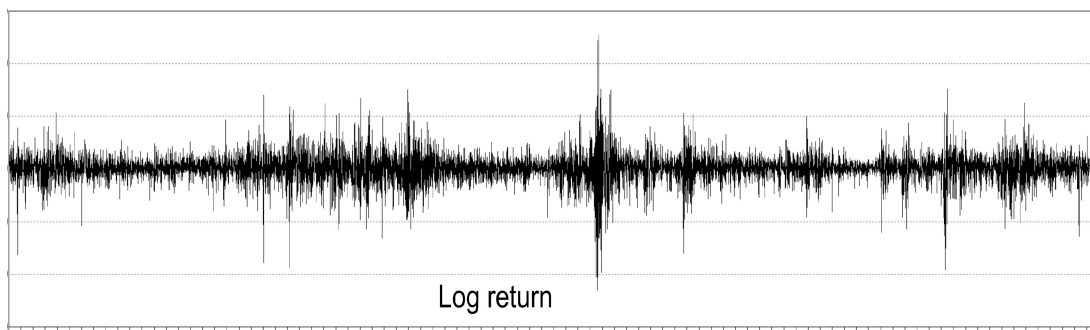
Predictor 1	Predictor 2	Target	MI	Solo pval	Unbiased pval
DEP_RANDOM2	DEP_RANDOM7	DEP_RANDOM0	0.0159	0.0261	0.4007
DEP_RANDOM2	DEP_RANDOM3	DEP_RANDOM0	0.0145	0.0813	0.8015
DEP_RANDOM2	DEP_RANDOM9	DEP_RANDOM0	0.0138	0.1404	0.9240
DEP_RANDOM2	DEP_RANDOM6	DEP_RANDOM0	0.0132	0.1968	0.9761
DEP_RANDOM4	DEP_RANDOM8	DEP_RANDOM0	0.0132	0.1660	0.9776
DEP_RANDOM3	DEP_RANDOM4	DEP_RANDOM0	0.0132	0.1859	0.9792
DEP_RANDOM2	DEP_RANDOM4	DEP_RANDOM0	0.0132	0.1768	0.9804
DEP_RANDOM5	DEP_RANDOM7	DEP_RANDOM0	0.0131	0.2354	0.9837
DEP_RANDOM1	DEP_RANDOM2	DEP_RANDOM0	0.0131	0.2077	0.9858
DEP_RANDOM2	DEP_RANDOM5	DEP_RANDOM0	0.0129	0.2329	0.9915
DEP_RANDOM2	DEP_RANDOM8	DEP_RANDOM0	0.0129	0.2162	0.9925
DEP_RANDOM4	DEP_RANDOM9	DEP_RANDOM0	0.0127	0.2594	0.9949
DEP_RANDOM1	DEP_RANDOM3	DEP_RANDOM0	0.0125	0.3104	0.9972
DEP_RANDOM3	DEP_RANDOM6	DEP_RANDOM0	0.0125	0.3243	0.9977
DEP_RANDOM1	DEP_RANDOM5	DEP_RANDOM0	0.0123	0.3545	0.9978
DEP_RANDOM3	DEP_RANDOM5	DEP_RANDOM0	0.0122	0.3621	0.9982
DEP_RANDOM6	DEP_RANDOM8	DEP_RANDOM0	0.0121	0.3613	0.9984
DEP_RANDOM1	DEP_RANDOM6	DEP_RANDOM0	0.0117	0.4874	0.9998
DEP_RANDOM6	DEP_RANDOM9	DEP_RANDOM0	0.0115	0.5108	0.9998
DEP_RANDOM4	DEP_RANDOM6	DEP_RANDOM0	0.0110	0.6064	1.0000
DEP_RANDOM1	DEP_RANDOM4	DEP_RANDOM0	0.0110	0.5907	1.0000
DEP_RANDOM5	DEP_RANDOM8	DEP_RANDOM0	0.0110	0.5737	1.0000
DEP_RANDOM5	DEP_RANDOM9	DEP_RANDOM0	0.0108	0.6308	1.0000
DEP_RANDOM7	DEP_RANDOM9	DEP_RANDOM0	0.0108	0.6902	1.0000
DEP_RANDOM7	DEP_RANDOM8	DEP_RANDOM0	0.0107	0.6681	1.0000
DEP_RANDOM4	DEP_RANDOM5	DEP_RANDOM0	0.0107	0.6274	1.0000
DEP_RANDOM3	DEP_RANDOM9	DEP_RANDOM0	0.0107	0.6552	1.0000
DEP_RANDOM1	DEP_RANDOM9	DEP_RANDOM0	0.0104	0.7349	1.0000
DEP_RANDOM6	DEP_RANDOM7	DEP_RANDOM0	0.0103	0.7587	1.0000
DEP_RANDOM8	DEP_RANDOM9	DEP_RANDOM0	0.0102	0.7330	1.0000
DEP_RANDOM3	DEP_RANDOM7	DEP_RANDOM0	0.0101	0.7944	1.0000
DEP_RANDOM5	DEP_RANDOM6	DEP_RANDOM0	0.0099	0.8103	1.0000
DEP_RANDOM1	DEP_RANDOM8	DEP_RANDOM0	0.0098	0.8036	1.0000
DEP_RANDOM3	DEP_RANDOM8	DEP_RANDOM0	0.0097	0.8085	1.0000
DEP_RANDOM4	DEP_RANDOM7	DEP_RANDOM0	0.0087	0.9581	1.0000
DEP_RANDOM1	DEP_RANDOM7	DEP_RANDOM0	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.

## A Practical Multiple-Target Example

I created an OEX prediction database consisting of 220 indicator candidates (massive amounts of spaghetti thrown at the wall!) and two target candidates. The primary purpose of this demonstration is not so much to find effective indicators as it is to find the most predictable target.

The target `LOG_RET` is simple, the log of the return the next day, the return being the ratio of the price closing the trade to the price opening the trade. The target `ATR_RET` is the price change (difference) during the span of the trade, divided by the Average True Range over the prior year. This scales the target according to volatility. Plots of these competing targets are shown below for a time span of several decades. Note that except for a few spikes, the variance of the ATR-scaled return is much more stable over time than the simple log return. This might lead us to believe that ATR scaling aids predictability. The test results on the next page show otherwise.



```
*****
*
* Computing bivariate uncertainty reduction (two predictors, one target)
*   220 predictor candidates
*   2 target candidates
* 48180 predictor/target combinations to test
*   100 best combinations will be printed
*   3 predictor bins
*   3 target bins
* 10000 replications of complete Monte-Carlo Permutation Test
*
*****
```

-----> Uncertainty reduction <-----

Predictor 1	Predictor 2	Target	UR	Solo pval	Unbiased pval
CMMA_20	VMAMA_20	LOG_RET	0.0163	0.0001	0.0001
CMMA_20	MNRTVY_25	LOG_RET	0.0162	0.0001	0.0001
CMMA_20	VMAMA_10	LOG_RET	0.0159	0.0001	0.0001
CMMA_20	DVMAMA_5_10	LOG_RET	0.0157	0.0001	0.0001
BOLL_20	CMMA_20	LOG_RET	0.0156	0.0001	0.0001
CMMA_20	MNRTVY_12	LOG_RET	0.0155	0.0001	0.0001
CMMA_5	RTVY_25	LOG_RET	0.0154	0.0001	0.0001
CMMA_5	MOM_20_100	LOG_RET	0.0154	0.0001	0.0001
BOLL_10	CMMA_20	LOG_RET	0.0153	0.0001	0.0001
CMMA_20	PVFIT_15	LOG_RET	0.0153	0.0001	0.0001
CMMA_20	DVMAMA_10_40	LOG_RET	0.0152	0.0001	0.0001
CMMA_10	RTVY_25	LOG_RET	0.0151	0.0001	0.0001
CMMA_20	PVFIT_7	LOG_RET	0.0149	0.0001	0.0001
RTVY_25	VMAMA_10	LOG_RET	0.0149	0.0001	0.0001
PPV_5	RTVY_25	LOG_RET	0.0149	0.0001	0.0001
CMMA_10N	RTVY_25	LOG_RET	0.0147	0.0001	0.0001
MOM_20_100	VMAMA_20	LOG_RET	0.0147	0.0001	0.0001
CMMA_5N	RTVY_25	LOG_RET	0.0147	0.0001	0.0001
CMMA_20	RTVY_25	LOG_RET	0.0147	0.0001	0.0001
CMMA_20	DVMAMA_10_20	LOG_RET	0.0146	0.0001	0.0001
MOM_20_100	VMAMA_10	LOG_RET	0.0145	0.0001	0.0001
CMMA_5N	MOM_20_100	LOG_RET	0.0143	0.0001	0.0001
BOLL_10	CMMA_10N	ATR_RET	0.0143	0.0001	0.0001
MNRTVY_25	RTVY_25	LOG_RET	0.0143	0.0001	0.0001
CMMA_20	DVMAMA_20_80	LOG_RET	0.0142	0.0001	0.0001
CMMA_20	DVMAMA_20_40	LOG_RET	0.0142	0.0001	0.0001
CMMA_20	PENT_2	LOG_RET	0.0142	0.0001	0.0001
RTVY_25	VMAMA_20	LOG_RET	0.0142	0.0001	0.0001
RTVY_25	RTVY_6	LOG_RET	0.0142	0.0001	0.0001

We see that even the unbiased p-values are all at their minimum possible value, indicating that we have highly significant predictability. But note that except for one indicator pair, the simple log return is more predictable than the ATR-normalized return. To save space I terminated this list early, but this pattern continues through the entire 100 pairs printed.

### Indicator Selection Based On Profit Factor

The prior sections, as well as several upcoming sections, present very general predictor selection algorithms based on mutual information or similar measures. In this section we examine a more specialized performance criterion, one that is ideal for indicators used in prediction of financial markets. As is done for most algorithms in *VarScreen*, not only are indicator candidates ranked in order of performance, but *solo* and *unbiased* p-values are computed and printed for each candidate. Please see Pages 11 and 12 for a more detailed explanation of these two types of p-values. Also, as explained on Page 22, it is crucial that the target variable have negligible serial correlation. If it is serially correlated, the *cyclic* permutation option will help compensate for this problem, but all computed p-values will still be somewhat anti-conservative, a very serious problem.

When the *Optimal profit factor* test is selected, a dialog similar to that shown on the next page appears. 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.

Permutation test parameters are discussed on Page 11, and the *Stepwise* option is described in detail starting on Page 13.

The *Min fraction kept* parameter is a smallish value, usually around 0.1 or so, which plays an important role in computation of the performance criterion. It is discussed on the next page.

## Indicator Selection Based On Profit Factor 51

Optimal profit factor ×

Predictors	Target	
<div style="border: 1px solid black; padding: 2px;">           _SEQNUM_            ADX15            ADX15_A15            ADX15_A30            ADX15_D15            ADX15_D30            ADX15_MAX60            ADX15_MIN60            ADX7            ADX7_D15            ADX7_D7            ADX7_MAX30            ADX7_MIN30            CMMA_10            CMMA_10N            CMMA_20            CMMA_20N            CMMA_5            CMMA_5N            CUB_ATR_15         </div>	<div style="border: 1px solid black; padding: 2px;">           _SEQNUM_            ADX15            ADX15_A15            ADX15_A30            ADX15_D15            ADX15_D30            ADX15_MAX60            ADX15_MIN60            ADX7            ADX7_D15            ADX7_D7            ADX7_MAX30            ADX7_MIN30            CMMA_10            CMMA_10N            CMMA_20            CMMA_20N            CMMA_5            CMMA_5N            CUB_ATR_15         </div>	<div style="border: 1px solid black; padding: 5px;"> <div style="display: flex; justify-content: space-between; margin-bottom: 10px;"> <span>Cancel</span> <span>OK</span> </div> <div style="margin-bottom: 10px;">             Min fraction kept (0-5) <input style="width: 50px;" type="text" value="0.1"/> </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">             Stepwise              Use stepwise <input type="checkbox"/>              Alpha <input style="width: 50px;" type="text" value="0.1"/> </div> <div>             Monte-Carlo Permutation Test  <input checked="" type="radio"/> Complete  <input type="radio"/> Cyclic              Replications <input style="width: 50px;" type="text" value="0"/> </div> </div>

This performance criterion is computed as follows for each indicator. First, the nonparametric correlation Spearman rho is computed for the indicator with the target. If the correlation is negative the sign of the indicator is flipped, resulting in a modified indicator that has positive (or, rarely, zero) correlation with the target and that contains exactly the same predictive information. Then an upper threshold for the indicator is computed, with an important optimality property: of all possible thresholds such that at least the *min fraction kept* of cases have an indicator value that equals or exceeds this threshold, those cases that satisfy the threshold have the maximum profit factor. These would correspond to long trades taken in response to large indicator values. A lower threshold is similarly computed, and those indicator values that are strictly below this lower threshold would define short trades, and the threshold is chosen such that the profit factor from these short trades is maximized, subject to the same restriction that at least the specified minimum fraction of potential trades satisfy the short

## 52 Indicator Selection Based On Profit Factor

---

threshold. The final performance criterion is the greater of the long or short optimal profit factors.

Note that there is a tradeoff involved in your choice of the minimum fraction parameter. Nearly always, the best predictability appears in the extreme values of an indicator, meaning that there will be a strong tendency for the threshold optimizer to drive the upper (long) and lower (short) thresholds to extreme values. This is why the user must set a floor under the number of 'trades' executed. If there were no minimum fraction enforced, it would often be possible for the optimizer to drive the threshold to such an extreme value that almost no trades are taken, and the resulting profit factor would be huge or even infinite, yet be a poor, unstable measure of real-life performance. My own rule-of-thumb is to set the minimum fraction such that at least 100 trades are taken, and usually many more. In other words, the minimum fraction should be (in my opinion) at least 100 divided by the number of cases. And I find that anything under 0.05 leads to instability too often. My default is 0.1, which is usually excellent.

I computed a set of indicators and a target for OEX from early 1988 through the end of 2024. The test was performed with the 'Use stepwise' box *not* checked. The stepwise option (Page 13) is discussed after this example. The indicators are as follows:

**DAY\_RETURN** - Next-day log return divided by 252-day average true range for volatility normalization.

**CMMA\_N** - Current close minus N-day moving average, scaled and normalized.

**LIN\_ATR\_N** - Linear trend over the prior N days, normalized by 252-day average true range.

**RSI\_N** - Ordinary RSI with N-day lookback

**LINDEV\_N** - Current close's deviation from its value predicted by linear projection over the prior N days, suitably normalized.

**PVFIT\_N** - Price-volume fit over the prior N days.

**RTVY\_N** - Reactivity computed over the prior N days.

## Indicator Selection Based On Profit Factor 53

---

If you are interested in exactly how these indicators were computed, they are discussed in detail in my book "Statistically Sound Indicators for Financial Market Prediction".

I ran this test using the *DAY\_RETURN* target and these indicators computed with several lookbacks, and with 1000 replications. The optimal thresholds and profit factors are as follows:

Variable		Rho	Long thr	Long pf	Short thr	Short pf
CMMA_5	(-)	-0.050	7.7419	1.423	-6.5944	1.017
CMMA_10	(-)	-0.054	9.1368	1.597	-3.9037	0.981
CMMA_20	(-)	-0.043	12.3398	1.397	-5.5540	0.982
LIN_ATR_5	(-)	-0.036	24.7723	1.303	-19.7236	0.988
LIN_ATR_7	(-)	-0.045	21.4862	1.394	-17.3554	1.026
LIN_ATR_15	(-)	-0.021	15.1428	1.244	-18.2686	0.945
RSI_5	(-)	-0.052	-28.8175	1.544	-71.5579	0.989
RSI_10	(-)	-0.046	-36.1751	1.475	-57.0598	0.978
RSI_20	(-)	-0.036	-41.4523	1.419	-65.5329	1.012
LINDEV_5	(-)	-0.008	26.2327	1.185	-26.9793	0.993
LINDEV_10	(-)	-0.016	33.3522	1.168	-6.8519	0.920
LINDEV_20	(-)	-0.039	32.9259	1.387	-16.3927	0.983
PVFIT_7	(+)	0.001	14.3582	1.279	-17.2509	1.022
PVFIT_15	(-)	-0.010	21.7791	1.178	5.0897	0.924
RTVY_6	(-)	-0.049	16.0264	1.462	-7.1658	0.994
RTVY_12	(-)	-0.029	15.1335	1.228	-13.7269	0.955
RTVY_25	(-)	-0.028	18.7179	1.259	-20.7411	1.027

It's worth noting that all but one of these indicators was negatively correlated with the target and hence had its sign flipped, as shown by the (-) after the variable name. That's all the more interesting because these are all, to some degree, traditionally known as momentum indicators. So it appears that, at least for these relatively short lookbacks and looking ahead just one day, they are signaling counter-trend situations. This is less surprising when one notes that the thresholds are rather extreme. For example, with RSI\_5 we take a long position when RSI is less than 28.8 (negative RSI greater than -28.8), a clearly oversold condition.

It's also worth noting that these indicators, while quite powerful for suggesting long trades, are more or less worthless for short trades.

## 54 Indicator Selection Based On Profit Factor

---

The solo and unbiased p-values for these indicators, sorted from largest criterion to smallest, are as shown below. Interpretation of these values is discussed in prior sections as previously noted and will not be repeated here.

Variable	profit factor	solo pval	unbiased pval
CMMA_10	1.597	0.000	0.006
RSI_5	1.544	0.001	0.020
RSI_10	1.475	0.006	0.093
RTVY_6	1.462	0.009	0.119
CMMA_5	1.423	0.022	0.230
RSI_20	1.419	0.021	0.245
CMMA_20	1.397	0.036	0.338
LIN_ATR_7	1.394	0.037	0.356
LINDEV_20	1.387	0.047	0.390
LIN_ATR_5	1.303	0.179	0.843
PVFIT_7	1.279	0.252	0.926
RTVY_25	1.259	0.329	0.969
LIN_ATR_15	1.244	0.414	0.987
RTVY_12	1.228	0.495	0.996
LINDEV_5	1.185	0.768	1.000
PVFIT_15	1.178	0.806	1.000
LINDEV_10	1.168	0.874	1.000

The unbiased p-values for the two best indicators, CMMA\_10 and RSI\_5, are clearly outstanding. For CMMA\_10, 0.006 is the probability that, if all of our competitors were worthless, the best profit factor among them could have been at least the 1.597 that we observed. I'll hang my hat on that any day. Recall from our earlier discussion that once we go below the single best indicator, all subsequent unbiased p-values are upper bounds for the true unbiased p-values. So even the third and possibly the fourth competitors are in contention for being useful. This is reinforced by the fact that their solo p-values, while not taking into account selection bias, are under 0.01.

### Demonstrating the Stepwise Option

On Page 52 we saw a demonstration of indicator selection by optimal profit factor, using the traditional Monte-Carlo permutation test. Please keep handy the table of final results from that test. We now run exactly the same test, except using the stepwise option with  $\alpha=0.1$ . These results are as follows:



Variable	profit factor	unbiased pval
CMMA_10	1.597	0.005
RSI_5	1.544	0.019
RSI_10	1.475	0.078
RTVY_6	1.462	0.098

Best remaining p-value=0.1960, so quitting

For the best two indicators the p-values are almost the same in both tests. (Theoretically, the first should be exactly the same, because the two versions of the test are identical for the best performer. But these tests use random numbers, so small variation is likely. This is why it's important to use a large number of MCPT replications.) By the third, the p-value has dropped from 0.093 for the traditional test (which is an upper bound for the true value) to the true value of 0.078. For the fourth it drops from 0.119 to 0.098. This makes it just under my specified alpha of 0.1 so we pick up one more indicator at this alpha level, a clear demonstration of the increased power of the stepwise version. The fifth p-value, 0.1960, blows far past my alpha, so inclusion ceases.

It is tempting to use a larger alpha in order to see more computed p-values, but there is a serious potential problem with this if you are not careful. ***You must stop considering candidates as soon as the p-value passes your preset alpha.*** This is because raw p-values may actually decrease later. The Romano-Wolf reference cited in the section that begins on Page 13 solves this problem by forcing each successive p-value to be at least equal to the prior p-value, and they explain why this is necessary if p-values beyond that for the best competitor are to be used as actual p-values. I do the same in *VarScreen*. The explanation is far too complex for this text, so please see that paper for details.

Note that these p-values are computed using random numbers, so if you do not perform a large number of replications (thousands) you may occasionally find that the stepwise test produces a p-value greater than that of the traditional test, which in theory should never happen. This is just random variation, easily fixed by using more replications.

## 56 Max Relevance / Min Redundancy Predictors

---

### Max Relevance / Min Redundancy Predictors

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 unless we take steps to alleviate this problem.

This gets to the crux of the problem. Our goal is not to just find a subset of predictors all of which are good, but we want this subset to be as small as reasonably possible. Why would we want ten predictors when we can get the same predictive power from four predictors? And as long as we are talking about predictive power, let us define it as the *joint mutual information* or *joint dependency* between our subset of predictors, taken as a group (which takes into account all of their interactions), and the target.

If we had a nearly infinite quantity of data, finding a good such subset, one which has high joint mutual information while not being excessively large, could be done with ordinary forward stepwise selection. First we find the single candidate which has maximum mutual information with the target. Then we find, from the remaining candidates, the one which *together with the one already selected* has the maximum joint mutual information with the target. With each step we pick one more candidate so as to maximize their joint mutual information with the target.

Why did I begin that paragraph by requiring a nearly infinite number of cases? It's because the combinatoric explosion is catastrophic. Suppose that to compute the joint mutual information we divide the range of values into ten bins for each predictor. Then with just two predictors and a target we

## Max Relevance / Min Redundancy Predictors 57

---

would have  $10^3=1000$  bins. If we had 1000 cases, that would average just 1 case per bin. Add a third predictor and we have 10,000 bins. No can do.

A much more efficient and practical approach to selecting a good subset of your 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. In other words, we want to find a group of relatively few candidates that *jointly* have high mutual information with the target.

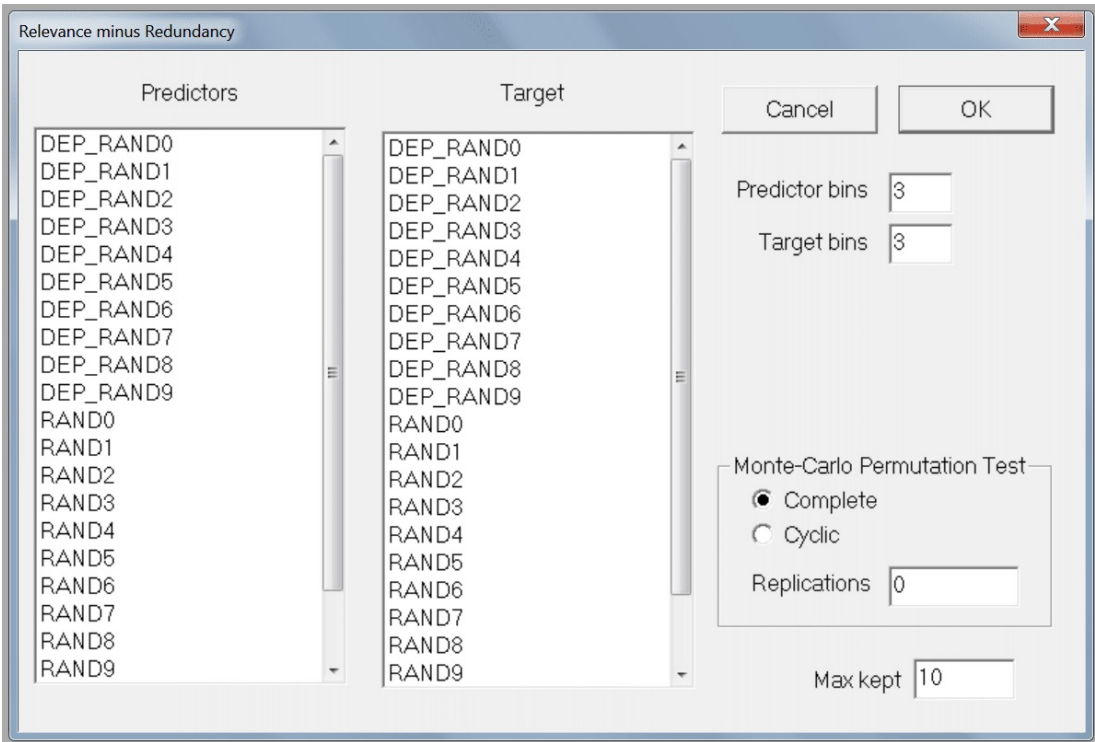
Peng, Long and Ding (2005) provide a fabulous algorithm for handling this redundancy problem in their paper “Feature Selection Based on Mutual Information: Criteria of Max-Dependency, Max-Relevance, and Min Redundancy”. An intuitive summary of the algorithm, along with C++ code, appears in my book “Assessing and Improving Prediction and Classification,” so details will be omitted here. However, it must be stressed that this algorithm has a powerful optimality property: the Peng, Long, and Ding algorithm is an elegant work-around to combinatoric explosion that produces the *same subset of predictors* as stepwise selection based on maximizing joint mutual information, 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. Peng, Long, and Ding prove that this candidate is the same one that would be selected if we were to somehow maximize the joint mutual information, a task that we previously showed to be virtually impossible.

# 58 Max Relevance / Min Redundancy Predictors

## Specifying the Test Parameters

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



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.

## Max Relevance / Min Redundancy Predictors 59

---

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 (Page 63) 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. This option and algorithm is discussed in detail starting on Page 22.

*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.

# 60 Max Relevance / Min Redundancy Predictors

---

## A Contrived 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:

*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.

```
*****
*
* Computing 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 Monte-Carlo 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

## Max Relevance / Min Redundancy Predictors 61

---

It should be no surprise that the single best candidate for predicting SUM1234 would be either SUM12 or SUM34, and it turns out to be the latter. Its relevance is its mutual information with the target, and of course its redundancy is zero, because it's the only candidate selected so far. Then we search for the next candidate to select:

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

Again, the table above contains no surprises. If we are predicting SUM1234, and we already are in possession of SUM34, we would expect SUM12 to be selected next, which is exactly what happens. This happens here because SUM12 has the highest mutual information of the remaining candidates, and it has essentially no redundancy with SUM34. Now look at how things pan out when the next candidate is to be selected:

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

## 62 Max Relevance / Min Redundancy Predictors

---

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

In the table on the prior page, we see that the remaining candidates fall into two classes. RAND0 and RAND5 through RAND9 have neither relevance (mutual information with SUM1234) nor redundancy with either SUM12 or SUM34. These candidates have a criterion (relevance minus redundancy) of about zero. The other class is RAND1 through RAND4. These remaining candidates obviously have relatively high relevance with SUM1234, but they also have high redundancy with SUM12 or SUM34, the two candidates already selected. So when we subtract their redundancy from their relevance we also get values near zero, even slightly negative. Visual inspection of this table clearly tells us that we are finished; we have selected all useful candidates.

It's instructive to examine one more step. The redundancy of a candidate is the *average* relationship with the selected set's members. Now that the irrelevant RAND6 has been selected, the average relationship for RAND1-RAND4 is reduced, so now they look attractive despite being quite redundant.

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



## Max Relevance / Min Redundancy Predictors 63

---

We can safely ignore subsequent steps and show the final table, which lists the candidates in their selection order, along with the solo and group p-values, which are described after the table.

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

Final predictors	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 p-values are printed for each candidate. The *Solo pval* is the probability that, if the predictor, considered alone, has no actual mutual information with the target, its mutual information (Relevance) as large as that obtained could have occurred. This is described in more detail starting on Page 11. We saw earlier that this relevance/redundancy algorithm selected the useless RAND6 as the third candidate. Its large solo p-value is a great indication that this variable is worthless.

The *Group pval* has not been previously discussed in this book, but the name should make its definition pretty clear. Its null hypothesis is that none of the candidates selected so far has any mutual information with the target. This is tested against the alternative that their average mutual information with the target is positive. The fact that the test statistic for this p-value is an average rather than a maximum means that this p-value can be subject to dilution effects if a large number of irrelevant candidates are included. Note that the group p-value is *not* unbiased against selection bias; it is not like the unbiased p-values discussed in prior sections. It is more like a solo p-value for the group of selected predictors as of each step.

## 64 Max Relevance / Min Redundancy Predictors

---

### A Practical Example of Relevance Minus Redundancy

I computed over 200 indicators for OEX, nearly every indicator described in my book “Statistically Sound Indicators for Financial Market Prediction”, each at several lookback lengths. I also computed DAY\_RET as the log ratio of the open two days from now to the open tomorrow morning. This is a real spaghetti-on-the-wall approach to indicator selection. Here follows an overview of the results obtained. Because there are well over 200 indicator candidates, it is impractical to list them all at any step, so I will list only the first few for each step. Here is the output; comments follow.

```
*****
*
* Computing relevance minus redundancy for optimal predictor subset
*   219 predictor candidates
*   10 best predictors will be printed
*     3 predictor bins
*     3 target bins
*  10000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Initial candidates, in order of decreasing mutual information with DAY\_RET

Variable	MI
CMMA_20N	0.0183
PPV_20	0.0160
CMMA_20	0.0152
MOM_20_100	0.0142
PVI_20	0.0137
PPV_10	0.0136
SPV_20	0.0135
RTVY_12	0.0125
PVI_10	0.0123
CMMA_10	0.0123
RTVY_25	0.0118
MOM_10_100	0.0115
PPV_5	0.0113
CMMA_10N	0.0111
CMMA_5N	0.0109
SPV_10	0.0108
CMMA_5	0.0107
OBV_20	0.0106
LIN_ATR_15N	0.0104

... Several hundred more candidates

Predictors so far	Relevance	Redundancy	Criterion
CMMA_20N	0.0183	0.0000	0.0183

# Max Relevance / Min Redundancy Predictors 65

Additional candidates, in order of decreasing relevance minus redundancy

Variable	Relevance	Redundancy	Criterion
IMORLET_20	0.0030	0.0003	0.0027
DPKURT_10	0.0028	0.0007	0.0021
QUODDEV_5	0.0023	0.0002	0.0021
ADX15_A15	0.0022	0.0006	0.0017
IPMORLET_20	0.0015	0.0003	0.0012
DKURT_10	0.0028	0.0017	0.0011
VENT_3	0.0025	0.0018	0.0007
LINDEV_5	0.0016	0.0008	0.0007
PKURT_10	0.0015	0.0008	0.0007
VMUTINF_3	0.0011	0.0006	0.0006
MNRTVY_25	0.0034	0.0030	0.0003
VENT_2	0.0024	0.0022	0.0003
CKURT_10	0.0011	0.0009	0.0002
VENT_4	0.0009	0.0008	0.0001
CKURT_10_4	0.0015	0.0014	0.0001
CKURT_20_4	0.0014	0.0014	0.0000
CUBEDEV_10	0.0002	0.0002	-0.0000
PMUTINF_1	0.0009	0.0009	-0.0001
DKURT_20_4	0.0028	0.0029	-0.0001
ADX7_MAX30	0.0010	0.0012	-0.0002
ADX15_D15	0.0006	0.0009	-0.0003
PKURT_20	0.0010	0.0013	-0.0003
IMORLET_5	0.0005	0.0008	-0.0003
DPSKEW_10	0.0002	0.0005	-0.0004
PMUTINF_2	0.0004	0.0010	-0.0007
DKURT_10_4	0.0006	0.0012	-0.0007
RD MORLET_20	0.0007	0.0015	-0.0008
ADX15	0.0004	0.0013	-0.0010
PKURT_20_4	0.0022	0.0033	-0.0010
QUODDEV_10	0.0008	0.0019	-0.0011
PSKEW_10	0.0002	0.0013	-0.0012
DPSKEW_20	0.0014	0.0026	-0.0012
RMORLET_20	0.0008	0.0020	-0.0012
CUBEDEV_20	0.0009	0.0022	-0.0013
RMORLET_5	0.0014	0.0027	-0.0013
PKURT_10_4	0.0011	0.0025	-0.0014
IPMORLET_5	0.0006	0.0020	-0.0014
VMUTINF_2	0.0009	0.0024	-0.0014
MXPVARRAT_20	0.0008	0.0023	-0.0014
DKURT_20	0.0002	0.0017	-0.0015
RD MORLET_10	0.0005	0.0022	-0.0016
DCSKEW_20_4	0.0015	0.0032	-0.0017
CKURT_20	0.0019	0.0036	-0.0017
PHMORLET_10	0.0004	0.0021	-0.0017
CSKEW_10_4	0.0015	0.0034	-0.0019
RMORLET_10	0.0003	0.0022	-0.0019
PHMORLET_20	0.0003	0.0023	-0.0020
DCSKEW_20	0.0002	0.0022	-0.0020

... Several hundred more candidates

Predictors so far	Relevance	Redundancy	Criterion
CMA_20N	0.0183	0.0000	0.0183
IMORLET_20	0.0030	0.0003	0.0027

## 66 Max Relevance / Min Redundancy Predictors

---

Additional candidates, in order of decreasing relevance minus redundancy

Variable	Relevance	Redundancy	Criterion
QUODDEV_5	0.0023	0.0002	0.0020
DPKURT_10	0.0028	0.0011	0.0018
DCKURT_10	0.0028	0.0010	0.0017
LINDEV_5	0.0016	0.0008	0.0007
CKURT_10	0.0011	0.0006	0.0005
CKURT_20_4	0.0014	0.0011	0.0003
CKURT_10_4	0.0015	0.0012	0.0002
PKURT_10	0.0015	0.0014	0.0001
VENT_3	0.0025	0.0025	0.0000
PKURT_20	0.0010	0.0011	-0.0001
DCKURT_20_4	0.0028	0.0030	-0.0002
IMORLET_5	0.0005	0.0008	-0.0003
VENT_2	0.0024	0.0028	-0.0003
VENT_4	0.0009	0.0013	-0.0004
PMUTINF_1	0.0009	0.0013	-0.0004
CUBEDEV_20	0.0009	0.0014	-0.0005
PKURT_20_4	0.0022	0.0028	-0.0005
CUBEDEV_10	0.0002	0.0007	-0.0005
PENT_3	0.0034	0.0040	-0.0006
VMUTINF_3	0.0011	0.0018	-0.0006
ADX15_A15	0.0022	0.0029	-0.0006
QUODDEV_10	0.0008	0.0014	-0.0006
ADX15_D30	0.0016	0.0023	-0.0007
ADX15	0.0004	0.0011	-0.0007
VMUTINF_2	0.0009	0.0017	-0.0007
DCSKEW_20_4	0.0015	0.0023	-0.0007
ADX7_MAX30	0.0010	0.0019	-0.0009
DCKURT_20	0.0002	0.0011	-0.0009
DCKURT_10_4	0.0006	0.0015	-0.0009
RPMORLET_5	0.0014	0.0024	-0.0010
CSKEW_10_4	0.0015	0.0025	-0.0010
PMUTINF_2	0.0004	0.0014	-0.0011
PSKEW_20	0.0012	0.0023	-0.0011
DPSKEW_20	0.0014	0.0028	-0.0013
DPKURT_10_4	0.0014	0.0028	-0.0014
IPMORLET_5	0.0006	0.0020	-0.0014
PVFIT_15_60	0.0024	0.0039	-0.0015
ADX15_A30	0.0032	0.0048	-0.0016
ADX15_D15	0.0006	0.0023	-0.0017
PKURT_10_4	0.0011	0.0028	-0.0017
RMORLET_5	0.0009	0.0028	-0.0019
CKURT_20	0.0019	0.0038	-0.0019
PSKEW_10	0.0002	0.0021	-0.0020
DPSKEW_10	0.0002	0.0022	-0.0020
PMUTINF_3	0.0011	0.0033	-0.0022
RPMORLET_10	0.0006	0.0028	-0.0022
IDMORLET_5	0.0004	0.0027	-0.0022
Predictors so far			
CMMA_20N	0.0183	0.0000	0.0183
IMORLET_20	0.0030	0.0003	0.0027
QUODDEV_5	0.0023	0.0002	0.0020

## Max Relevance / Min Redundancy Predictors 67

---

I'll skip seven more steps and go directly to the final results:

-----> Final results predicting DAY\_RET <-----

Predictors	Relevance	Redundancy	Criterion	Solo pval	Group pval
CMMA_20N	0.0183	0.0000	0.0183	0.000	0.000
IMORLET_20	0.0030	0.0003	0.0027	0.013	0.000
QUODDEV_5	0.0023	0.0002	0.0020	0.045	0.000
DPKURT_10	0.0028	0.0008	0.0021	0.017	0.000
DCKURT_10	0.0028	0.0008	0.0020	0.021	0.000
VENT_3	0.0025	0.0013	0.0012	0.028	0.000
ADX15_A30	0.0032	0.0026	0.0006	0.009	0.000
PVFIT_15_60	0.0024	0.0021	0.0003	0.036	0.000
RPMORLET_5	0.0014	0.0013	0.0001	0.196	0.000
DCSKEW_20_4	0.0015	0.0020	-0.0005	0.169	0.000

The group p-value is essentially zero throughout, because the first variable selected is a solid predictor, and we never get so many candidates selected that dilution of the average mutual information occurs. In fact, we get as many candidates selected as we do because there are so many completely different families in the candidate list that it is easy to find new additions that have essentially no redundancy with those already selected. But pay attention to the solo p-values. If any are other than really tiny, that candidate is suspicious.

## Optimal Transforms

In many cases, especially for experienced researchers, examination of a histogram of a candidate predictor will suggest a transform that is likely to improve its utility. Sometimes there will even be theoretical justification for a transform, such as the use of logs for strictly positive ratios. But for less experienced developers, and for large-scale use, it can be helpful to automate the process. *VarScreen* includes the option of applying multiple common transforms to each candidate predictor and then computing and reporting the univariate mutual information of each with a target variable. Solo (Page 11) and unbiased (Page 12) p-values are computed. If the entire dataset is tested, as opposed to just the predictor tails, CSCV (Page 23) median-test p-values are printed as well. The following transforms are tested (in addition to the untransformed value):

### *Log* ( *X* )

```
If X > 1.e-10
    Return loge ( X )
Else
    Return loge ( 1.e-10 )
```

### *Sqrt* ( *X* )

```
If X >= 0.0
    Return Sqrt ( X )
Else
    Return -Sqrt ( -X )
```

### *Htan* ( *X* ) (Hyperbolic tangent)

```
If X > 50
    Return 1
Else If X < -50
    Return -1
Else
    Return [ Exp ( X ) - Exp ( -X ) ] / [ Exp ( X ) + Exp ( -X ) ]
```

*Hsin* ( *X* ) (Hyperbolic sine)

Limit =  $\log_e ( 50.0 )$

If  $X > \text{Limit}$

$X = \text{Limit}$

If  $X < -\text{Limit}$

$X = -\text{Limit}$

Return [  $\text{Exp} ( X ) - \text{Exp} ( -X )$  ] / 2

For this test, discrete mutual information is computed somewhat differently from how it is computed in other tests. In all prior tests described in this book, discrete mutual information is based on partitions defined in such a way that for each predictor candidate, each bin contains the same number of cases, or at least as close as possible in case of ties. But because these transforms are all monotonic, defining bins this way would mean that the mutual information would be the same for transformed values as for original values. We cannot define bins this way.

Here, bins are defined by dividing the range into equal intervals. Because predictors almost never have a uniform distribution, but usually have at least a roughly bell shape even after transformation, central bins will tend to have the highest concentration of cases. This tends to give extra emphasis to truly extreme values, which is almost always a good thing, as it identifies the ‘oddballs’.

Another implication of equal range division is that because some bins may be nearly empty, p-values may not always monotonically increase as mutual information decreases. Naturally there will still be a strong tendency toward monotonic increase. But the candidates may have such different distributions after different transformations that the location of the unpermuted statistic in the permutation distribution can vary greatly. So you can expect to occasionally see a low p-value sandwiched between two much higher p-values, or a high between lows. This is legitimate and to be expected.

## Specifying the Test Parameters

When the user clicks *Tests / Univariate Opt transform*, a dialog similar to that shown will appear. The various parameters are described below.

Optimal predictor transformation for a target

Predictors	Target
_SEQNUM_	_SEQNUM_
CMMA_10	CMMA_10
CMMA_20	CMMA_20
CMMA_5	CMMA_5
DAY_RETURN	DAY_RETURN
LIN_ATR_15	LIN_ATR_15
LIN_ATR_5	LIN_ATR_5
LIN_ATR_7	LIN_ATR_7
LINDEV_10	LINDEV_10
LINDEV_20	LINDEV_20
LINDEV_5	LINDEV_5
PVFIT_15	PVFIT_15
PVFIT_7	PVFIT_7
RSI_10	RSI_10
RSI_20	RSI_20
RSI_5	RSI_5
RTVY_12	RTVY_12
RTVY_25	RTVY_25
RTVY_6	RTVY_6

Predictor bin definition

☐ Predictors and target continuous  
☒ Use all cases Predictor bins   
☐ Use tails only Tail fraction (0 to 0.5)

Target bins

Monte-Carlo Permutation Test

☒ Complete  
☐ Cyclic  
 Replications

Cancel OK

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.



If *Predictors and target are continuous* is selected, the performance criterion will be R-squared from a linear fit, as opposed to mutual information. It is perfectly legitimate to use this option with discrete data, but it nearly always makes more sense to use it for continuous data. And of course it is common and legitimate to use the other two mutual information options with continuous data. So in a way, this labeling is less than perfect, more of a suggestion than a firm choice.

Most often, the user will select *Use all cases* and specify the number of predictor bins for computing mutual information. The predictors are partitioned into bins that equally split the *range* of each variable (rather than equally splitting the *count* of cases). 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.

*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 the *total range* in each tail. So, for example, the default *tail fraction* of 0.1 would use the cases in the smallest ten percent and the largest ten percent of the complete range of each candidate predictor. The cases having intermediate values of the predictor candidate are completely ignored in the mutual information calculation. This method is especially useful in high-noise situations, such as prediction of financial markets. Note that for other tests, the tails are based on case count, while for this test the tails are based on the complete range of each predictor. The CSCV probability that superior mutual information will hold up out-of-sample is not 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 or more, so that solo and unbiased p-values (Page 12) 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. This option and algorithm is discussed in detail starting on Page 22.

The *Stepwise* option provided for many other tests is not available for the Optimal Transform test. This is because this test is not generally used to select an optimal set of predictors. Rather, it is used to identify performance improvements that could be obtained with transforms, and hence only the ranking is usually of interest.

## Testing Financial Market Indicators

I generated a handful of indicators known to often be useful for predicting future movement in the OEX index, and I also computed the one-day-ahead log return for the market. These indicators were all computed using the algorithms shown in my book “Statistically Sound Indicators for Financial Market Prediction.” Because for financial market prediction, the most information is usually present in the tails, I selected the *Tails only* option. Here are the results of this test, abbreviated to only the first few selections.

```
*****
*
* Computing optimal transforms (one predictor, one target)
* 17 predictor candidates
* 0.100 predictor tails used
* 5 target bins
* 10000 replications of complete Monte-Carlo Permutation Test
*
*****

-----> Mutual Information with DAY_RETURN <-----

Variable      MI      Solo pval  Unbiased pval
CMMA_20       0.3071      0.0004      0.0010
RSI_20        0.2627      0.0019      0.0029
Sqrt RSI_20   0.1175      0.0747      0.2448
RSI_10        0.1068      0.0056      0.3455
CMMA_5        0.1065      0.1077      0.3652
CMMA_10       0.1005      0.0167      0.3984
RSI_5         0.0827      0.0001      0.5880
RTVY_6        0.0825      0.0001      0.5883
Sqrt CMMA_20  0.0724      0.0001      0.7195
LIN_ATR_7     0.0707      0.0001      0.7431
Sqrt CMMA_10  0.0601      0.0003      0.9023
LIN_ATR_5     0.0564      0.0001      0.9538
Sqrt CMMA_5   0.0534      0.0040      0.9809
Sqrt RTVY_6   0.0526      0.0001      0.9810
Sqrt RTVY_25  0.0507      0.0001      0.9844
```

Several things stand out. First, we see that the raw (untransformed) values occupy the top positions. This is a testimony to the quality of the normalization used in computing these indicators. It's tough to improve on it. Also, although most of these solo p-values are excellent, when we look at the unbiased p-values we see that only two candidates are trustworthy.

## 74 Hidden Markov Models with Target Correlation

---

### Hidden Markov Models with Target Correlation

Most 'traditional' modeling is explicitly or implicitly based on several common assumptions (which may not be strictly correct but which are harmless nonetheless) about how the process works and how we model it to make classification or numeric prediction decisions. In particular, one or more *predictor* variables, which we typically think of as being at least somewhat independent (whether they are or not!) exert influence on the distribution of a *target* variable, which we typically think of as being *dependent* on the predictor(s). For example, a person's Zip code may predict his or her income level. The temperature and pressure in a chemical reactor vessel may predict the time to completion of the reaction. The degree to which an equity's price just departed from normalcy may predict its next price move.

In other words, whether our assumption is strictly correct or not, we are acting as if there is an immediate and quantifiable relationship between the variables: we observe the value of a predictor variable  $X$ , which may be a vector, and we say that this observed value allows us to make a statement about the likely value of a currently unobservable target variable  $Y$ . In general terms, we assume  $Y = f(X) + \text{random error}$  for some function  $f()$ , and we try to approximate  $f()$ .

We can approach the classification or prediction problem from an entirely different direction. We may assume that the underlying process is always in exactly one of several possible states, and that the current state determines the distribution of various variables, some of which we can measure, and some of which we cannot measure but which are of great interest.

For example, we may believe that a financial market may be in a bull state, a bear state, or a flat state. The distributions of (measurable) historical indicators, as well as the distribution of (unmeasurable) near-term future returns, will depend on the current state.

## Hidden Markov Models with Target Correlation 75

---

There is one more assumption that we can include in this alternative approach to prediction and classification: we can assume that the process has memory. In other words, empirical evidence may indicate that certain states tend to be persistent, while other states may be highly transient. Also, certain states may tend to change to certain other states while avoiding changing to still other states. For example, it's more common for a failing bull market to transition to a flat state for a while, rather than instantly shifting to a bear market. If we can estimate these transition probabilities using historical data and incorporate them into our final model of the process, the quality of our modeling improves.

In summary, our modeling works like this. We specify in advance the number of possible states that we are admitting. We then use extensive measurable historical data (such as indicators in a financial application) to fit a *Hidden Markov Model* to the data. The details of an HMM are far beyond the scope of this text; they are discussed many places online and in standard textbooks. Also, my book "Modern Data Mining Algorithms in C++ and CUDA C" presents a modest mathematical explanation as well as complete source code for the modeling procedure.

A good HMM of the process takes as its moment-to-moment input the measured values of the predictor variables as well as the current state of the process. It then makes a decision as to whether to stay with the current state or transition to a different state. The version in *VarScreen* goes a step further in that it produces a vector of probabilities for being in each of the possible states, rather than just providing a simple decision of which state it most likely is in.

This memory inherent in the process (its dependency on the state it was in before the most current predictor values arrive) is a two-edged sword. It's a good property, because in real life states often tend to be persistent. For example, a financial market would not be in a bull state one day, a bear state the next, and a bull state again the day after. Memory can prevent the presumed state from rapidly whipping back and forth unrealistically.

## 76 Hidden Markov Models with Target Correlation

---

On the other hand, memory can also delay crucial decisions. If a state is persistent (high probability of remaining in that state) it can take many observations of the predictor variables before the model gathers enough new evidence to justify changing state. This delay may or may not be a deal-killer in your application. The deciding factor is often noise level. If you have very clean, consistently accurate data, you may wish to avoid an HMM. Or you may find that your HMM learns to have a degree of persistence that is in accord with the process. But if your application is noisy, such as most financial markets, you may find that the delay in state changes works to your benefit by reducing deadly whipsaw decisions.

So far we've considered only predictor variables and states, ignoring the target variables. Indeed, *the target variables play no role whatsoever in training the HMM*. We use historical values of the predictor variables to construct a Hidden Markov Model that hopefully does a good job of approximating the behavior of the underlying process. Once we have the trained model in hand, every time a new observation (set of current predictor variables) is obtained, we can decide whether to stay in the current state or transition to a new state.

This knowledge of the current state does have some utility on its own. *VarScreen* prints, among other things, the mean values of each predictor for each state, and this information often lets us label states. For example, suppose one state has a large positive mean for a trend indicator, and another state has a large negative value for this same indicator. Then we would be inclined to label the former state as a bull market and the latter state as a bear market. Given that *VarScreen* also computes and saves the probability of being in each state, we can then get a lot of useful information. For example, we would find it useful to see that as of the most current observation, we have, say, 0.92 probability of being in a bear state. But we can do even better...

It's time to talk about the target variable, which is a currently unmeasurable variable whose value interests us. For example, in a financial market

## Hidden Markov Models with Target Correlation 77

---

prediction application, we might let the target be the one-day-ahead log price change in a market of interest. What we would like to do is link the state to likely values of the target variable. It's one thing to know the probability of being in some particular state, but it's a much more valuable thing to know that when we are in some specified state, the expected value of the target can be computed. And we can take it even further. Why limit ourselves to just computing this prediction based on the most likely state, the state with the highest current probability? Surely, the magnitude of this probability in the context of probabilities of other possible states provides even more information about the target. It's one thing when we have two possible states with probabilities of 0.51 and 0.49, and another very different thing when the probabilities are 0.99 and 0.01.

An easy and effective way to link states to a target variable is multiple regression: we use the vector of state probabilities to predict the target. Then, the multiple-R of the regression is a good measure of the degree to which states can be associated with target values. And remember that *VarScreen* is fundamentally a variable screening program. So we can give it a list of many predictor candidates and let it screen for the best predictors. In particular...

### Detailed 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 yet again that these models are optimized without

## 78 Hidden Markov Models with Target Correlation

---

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* (Page 11) 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* (Page 12) 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. The stepwise FWE algorithm (Page 13) is not available for this test.

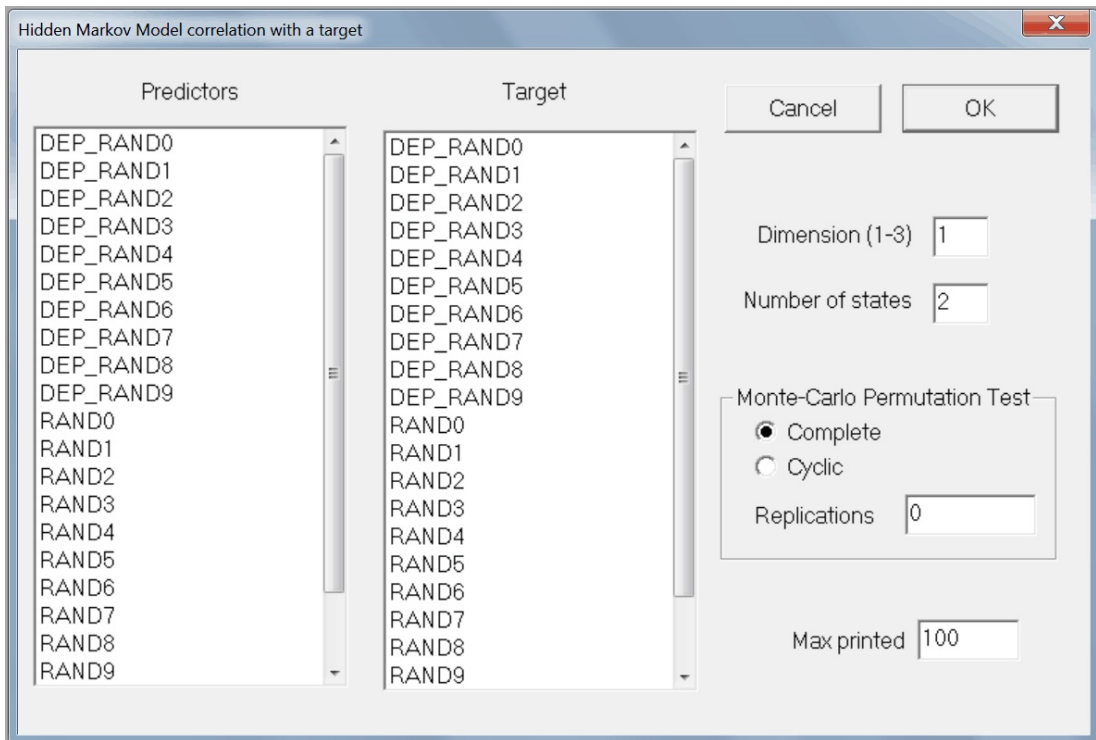
Note that exact results will not in general be replicated if runs are repeated. This is because training a Hidden Markov Model relies on random number generation, and Windows' scheduling of training threads is rarely consistent. The competing models will receive their random numbers in different orders 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.



# Hidden Markov Models with Target Correlation 79

## Specifying the Test Parameters

When the user clicks *Tests / Hidden Markov model*, a dialog similar to that shown will appear. The various parameters are described below.



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.

## 80 Hidden Markov Models with Target Correlation

---

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 Page 22 for more details. Because Hidden Markov Models are often applied to serially correlated targets, cyclic permutation is the default. However, if your target has minimal serial correlation, you should probably prefer complete permutation.

*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 Manager 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.

# Hidden Markov Models with Target Correlation 81

---

## A Contrived Example of Hidden Markov Models

The example in this section uses data that has no inherent memory; each observation is independent of all prior and subsequent observations. Thus, a Hidden Markov Model would probably not be the best way to model the data. Nonetheless, the results are educational, and hence make a good example. The variables in the dataset are as follows:

*RAND1 - RAND5* are independent, identically distributed time series. These are used as the predictor candidates.

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

I specified two predictors and four permissible states for the HMM. *VarScreen* fits a Hidden Markov Model to each of the  $(5 \times 4)/2 = 10$  pairs of predictor candidates. Here are the results, and an explanation follows.

```
*****
*
* Computing hidden Markov model
*      2 dimensions
*      4 states
*      5 predictor candidates
*      10 predictor combinations to test
*      10 best combinations will be printed
*      1000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Specifications of the best HMM model correlating with SUM12...

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

State	RAND1	RAND2
1	-0.01390 0.54844	-0.72842 0.17455
2	0.63195 0.23486	0.01414 0.46806
3	-0.58391 0.37407	0.04409 0.39134
4	-0.04108 0.53717	0.75599 0.15993

## 82 Hidden Markov Models with Target Correlation

---

Transition probabilities...

	1	2	3	4
1	0.2346	0.2444	0.3522	0.1688
2	0.1956	0.2646	0.3580	0.1817
3	0.1903	0.2732	0.3613	0.1751
4	0.1835	0.2945	0.3222	0.1999

Further properties of each state...

Percent of cases state is highest (tied cases are ignored)  
Correlation of state probability with target  
Mean of target when in this state (tied cases are ignored)  
Standard deviation of target when in this state (tied cases are ignored)

State	Percent	Correlation	Target mean	Target StdDev
1	22.45	-0.59891	-0.75988	0.55522
2	25.52	0.58424	0.69198	0.45257
3	31.65	-0.42018	-0.37622	0.47034
4	20.38	0.46704	0.67455	0.56119

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

Predictor 1	Predictor 2	Multiple-R	Solo pval	Unbiased pval
RAND1	RAND2	0.8865	0.0010	0.0010
RAND1	RAND5	0.6678	0.0010	0.0010
RAND1	RAND4	0.6667	0.0010	0.0010
RAND2	RAND3	0.6507	0.0010	0.0010
RAND2	RAND5	0.6505	0.0010	0.0010
RAND1	RAND3	0.6353	0.0010	0.0010
RAND2	RAND4	0.5743	0.0010	0.0010
RAND3	RAND5	0.0230	0.3550	0.8880
RAND4	RAND5	0.0224	0.3490	0.9130
RAND3	RAND4	0.0074	0.9450	1.0000

I'll walk through these results now. After printing the user's specifications for the test, it prints details for the model that has the highest correlation (multiple-R) with the target variable, SUM12. It should come as no surprise that the maximally correlated HMM is the one based on RAND1 and RAND2. Examination of the table of means for each state tells us that knowledge of the value of RAND1 tells us a lot about whether we are in State 2 (large positive value of RAND1), versus State 3 (large negative value of RAND1), versus either State 1 or State 4 (intermediate value of RAND1). Knowledge of the value of RAND2 tells us about whether we are in State 4 (large positive value of RAND2), versus State 1 (large negative value of RAND2), versus either State 2 or State 3 (intermediate value of RAND2). Thus, if we know both RAND1 and RAND2, we can have a pretty good idea

## Hidden Markov Models with Target Correlation 83

---

of which state we are in. Note, by the way, that because training an HMM involves random numbers, exact replication of these results will never be obtained, but results should usually be similar to these.

The transition probability matrix follows the means. The entry in Row  $i$  and Column  $j$  is the probability that the HMM will transition from State  $i$  to State  $j$  from one observation to the next. The probabilities are almost all identical here; the relatively small discrepancies are due to random variation in the data. This should not be unexpected, as the data has no serial dependency.

The next block of parameters shows properties of each of the four states.

The **Percent** column is the percentage of cases for which this state is most probable. Their sum may not equal 100 percent, because cases for which the highest probability is tied for two or more states are not included in this calculation. Here we see that each of the four possible states are almost equally likely. The observed discrepancy is just random variation in the data.

**Correlation** is the Pearson correlation coefficient between the target value and this state's membership probability. Again, there's nothing surprising in this table. For example, State 1 has a RAND2 mean of  $-0.72842$ , so it's expected that SUM12 would probably be unusually negative when there is high probability of being in State 1. The same logic applies to the other three states.

**Target mean** is the mean of the target for cases in which this state is the most probable. Cases in which two or more states are tied for maximum probability are ignored. As already described for the Correlation, all results here are to be expected.

**Target StdDev** is the standard deviation of the target for cases in which this state is the most probable. Cases in which two or more states are tied for maximum probability are ignored.

## 84 Hidden Markov Models with Target Correlation

---

The results shown so far in the output are all for the single HMM that has the highest correlation with the target. However, it is useful to see at least basic information about the other models that were considered. Because it is conceivable that we may have an enormous number of models, the user specifies the maximum number that will be printed. This table of results for models other than the most highly correlated includes the following columns:

*Predictor 1* and subsequent predictors (1, 2, or 3, according to the user-specified dimension) show the predictors employed by the HMM.

*Multiple-R* is the R-square multivariate correlation between the target and the state probability vector for that HMM.

*Solo p-value* (Page 11) 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.

*Unbiased p-value* (Page 12) for the best model (the model listed first in the table) 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 by this best model. Subsequent unbiased p-values are upper bounds on similarly defined probabilities. The stepwise FWE algorithm (Page 13) is not available for this test.

The Multiple-R, solo p-value, and unbiased p-values in this table, like everything else so far, is completely unsurprising. We see that every model that contains either RAND1 or RAND2 has a significant Multiple-R and tiny p-values. When models that contain neither RAND1 nor RAND2 appear, there is a sharp break in the results, with correlation dropping to nearly zero and p-values becoming large.

# Hidden Markov Models with Target Correlation 85

---

## A Practical Example of Hidden Markov Models

This section demonstrates the use of Hidden Markov Models using indicators and a target based on OEX. These variables are:

*CMMA\_N* is today's close, minus its N-day moving average, thus giving an indication of how today's close relates to recent closes. N takes the lookback lengths of 5, 10, and 20 days.

*RSI\_N* is the ordinary RSI indicator with a lookback of N days. This indicator ranges from 0 to 100, with 50 being 'flat' market movement. N takes the lookback lengths of 5, 10, and 20 days

*DAY\_RETURN* is the log price change from tomorrow's open to the next day's open, divided by the 252-day average true range. This is the target.

Here are the results of this test; a discussion follows:

```
*****
*
* Computing hidden Markov model
*   2 dimensions
*   3 states
*   6 predictor candidates
*   15 predictor combinations to test
*   15 best combinations will be printed
*   1000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Specifications of the best HMM model correlating with DAY\_RETURN...  
Means (top number) and standard deviations (bottom number)

State	CMMA_10	RSI_5
1	-10.97968 6.25266	28.70829 10.36783
2	8.81260 4.04592	74.51744 9.13200
3	0.05922 3.91065	50.79946 11.57959

## 86 Hidden Markov Models with Target Correlation

---

Transition probabilities...

	1	2	3
1	0.8288	0.0000	0.1712
2	0.0002	0.8668	0.1330
3	0.0644	0.0923	0.8433

Further properties of each state...

Percent of cases state is highest (tied cases are ignored)

Correlation of state probability with target

Mean of target when in this state (tied cases are ignored)

Standard deviation of target when in this state (tied cases are ignored)

State	Percent	Correlation	Target mean	Target StdDev
1	17.75	-0.06153	-0.06052	1.12803
2	33.60	0.06896	0.10239	0.62334
3	48.65	-0.01935	0.01838	0.77593

-----> Hidden Markov Models correlating with DAY\_RETURN <-----

Predictor 1	Predictor 2	Multiple-R	Solo pval	Unbiased pval
CMMA_10	RSI_5	0.0784	0.0010	0.0010
CMMA_5	RSI_5	0.0679	0.0010	0.0010
CMMA_10	RSI_10	0.0637	0.0010	0.0010
CMMA_20	RSI_5	0.0465	0.0030	0.0030
CMMA_20	RSI_10	0.0404	0.0030	0.0220
CMMA_20	CMMA_5	0.0310	0.0280	0.1820
CMMA_10	CMMA_5	0.0275	0.0600	0.3140
CMMA_5	RSI_10	0.0273	0.0650	0.3260
CMMA_10	CMMA_20	0.0259	0.0790	0.3900
CMMA_5	RSI_20	0.0255	0.0760	0.4200
CMMA_10	RSI_20	0.0232	0.1180	0.5630
CMMA_20	RSI_20	0.0213	0.1680	0.6830
RSI_10	RSI_5	0.0188	0.2690	0.8130
RSI_20	RSI_5	0.0154	0.4050	0.9490
RSI_10	RSI_20	0.0081	0.7740	1.0000

Recall that RSI has a central value of 50, so the RSI\_5 mean of 28.70829 for State 1 is very low; the market is dropping hard, at least very recently. Also, CMMA\_10 has a strong negative value, meaning that the current close is much lower than recent closes. Clearly, State 1 corresponds to a plunge in the market, at least in recent history. If we skip ahead to the target mean table, we see that when in this state, the next-day-ahead price move also has a negative mean, so trend following is going on. Also, the target standard deviation for State 1 is by far the largest of the three states, meaning that market turbulence is unusually high when in this state.



## Hidden Markov Models with Target Correlation 87

---

State 2 is just the opposite: RSI\_5 indicates a strong upward trend, and CMMA\_10 indicates that today's close is well above its recent level. The target mean for this state is not only quite positive, but it is well above its standard deviation in this state. Once again we see strong trend following, even stronger than we saw for a (short-term!) bear market.

State 3 is a flat state. CMMA\_10 is essentially zero compared to its standard deviation, and RSI\_5 is essentially 50 compared to its standard deviation. The target mean for this state is slightly positive, but insignificant compared to its standard deviation in this state.

Examining the transition matrix shows that all three states are fairly persistent, staying unchanged over 80 percent of the time. Not surprisingly we see that there is a probability of zero to four decimal places for going directly from a bear state to a bull state, and practically zero for going instantly from a bull to a bear state. A transition to a flat state is virtually mandatory. And once in a flat state, it is slightly more likely (0.0923) to transition to a bull state than to a bear state (0.0644).

The table of all models is interesting when we look at the relationship between lookback lengths and the performance figures of multiple-R and unbiased p-values. Only lookbacks of 5 or 10 appear in the three top spots, those having the minimum possible p-value of 1/1000. When we get a lookback of 20 in the fourth spot, the multiple-R drops precipitously from 0.0637 to 0.0465, although the p-value stays decent. From there on, any time a lookback of 20 appears we have poor performance. Also, having two predictors from the same family (both CMMA or both RSI) leads to poor performance. Of course there are a couple oddballs, pairs from different families having lookbacks less than 20, but this is not common.

## 88 Hidden Markov Models with Target Correlation

---

### 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 the big advantage of an HMM over other more traditional models is its ability to take advantage of memory in the data. If certain states are more or less likely to transition to certain other states, and if we can take advantage of that learned knowledge, an HMM may be better than models that have no memory. Thus, it may behoove us to assess whether our data has memory that can be exploited by a Hidden Markov Model.

The HMM memory test in *VarScreen* is more limited than the target linking test in that it does not allow testing of subsets from a larger set of candidates. Rather, the user must specify the exact predictor variables, and this set of predictors is tested for memory. There is no target, as this test is strictly for memory in the vector of predictor values.

The HMM fitting algorithm computes a fitting criterion similar to a likelihood, with the value of this criterion quantifying the degree to which an HMM explains the data. This criterion will be small if the data has little or no memory, and large if the data has substantial memory and its behavior can be well explained by an HMM. The criterion will be computed for the original data as well as for a large number of permutations. Obviously, permutation will destroy any memory inherent in the data. If the data is well described by an HMM we will find that the fitting criterion for the original data will exceed most or all of the criteria for the permuted data. Thus we can compute a p-value for the null hypothesis that the data cannot be explained by an HMM. If we plan to use an HMM to link states to a target, we would want to see a very small p-value for this memory test.

It might seem counterintuitive, but in most cases we would want to perform the linkage test first, and then the memory test. There are two reasons for this. First, the linkage test has the ability to screen a large number of predictor candidates, while the memory test cannot do screening; the exact

## Hidden Markov Models with Target Correlation 89

---

predictor set must be specified. But more importantly, our ultimate goal is to be able to predict a target variable; fitting the data with an HMM is secondary. So we are most interested in finding predictor subsets that give us good target prediction ability. Only after we have one or more such subsets would we try to confirm that an HMM is a good way to model the data.

In the ideal situation, the linkage test will give us good linkage performance, and then the memory test will confirm that an HMM is a good way to model the data. If we find insignificant linkage (unbiased p-values other than tiny), there is no point in even going on to a memory test. But what if we get good linkage but a poor (not tiny p-value) memory test result? This outcome almost certainly means that our predictors are good at predicting the target, but we would be better off using a conventional model rather than an HMM.

I ran this memory test for the example shown in the prior example of HMM linkage. These are the results, so clearly this dataset is well modeled by an HMM.

```
*****
*
*   Computing Hidden Markov Model memory in time series
*       2 predictors
*       3 states
*       100 MCPT replications
*      10000 initialization tries
*       1000 convergence iterations
*        16 max threads used
*
*****
```

The following variable(s) were tested:

```
  CMMA_10
  RSI_5
```

p-value for null hypothesis 'The data has no memory' = 0.010

## 90 Hidden Markov Models with Target Correlation

### Specifying the Memory 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

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

Number of states: 2

Initialization trials: 10000

Max iters: 1000

Replications: 100

Cancel OK

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

## Hidden Markov Models with Target Correlation 91

---

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. Remember that only a single model will be tested, that employing *all* of the selected variables. This test does not do any screening for predictor subsets.

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. Make it as large as your time allows, because the fitting criterion surface has numerous local maxima; better initialization reduces the probability of landing in a false optimum.

*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, and is generally the most critical of stationarity properties. 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. Even a slow, steady drift is a serious problem. Thus, we should always check for this sort of nonstationarity in all predictors and targets.

For applications in which the 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 or even slowly 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 on the next page will appear.

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.

Two families of tests are available, the multiple comparisons option, and the serial correlation option. They are mutually incompatible in that if the data has significant serial correlation, a statistically sound multiple comparisons test cannot be done, at least not in *VarScreen*, and probably not in any practical way in any program.

Stationarity break in mean ×

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

Cancel OK

Minimum recent history

Maximum recent history

Multiple comparisons or max correlation lag

Multiple vs. dependence

☐ Multiple comparisons

☒ Serial correlation

MCPT replications

If you leave the default multiple comparisons option selected, and leave the numeric field above it set to the default of 1, a single, straightforward test as just described is performed. The program simply searches the user-specified range of recent history, minimum to maximum, moving a hypothetical boundary, and finds the greatest difference in means on the two sides of the boundary.

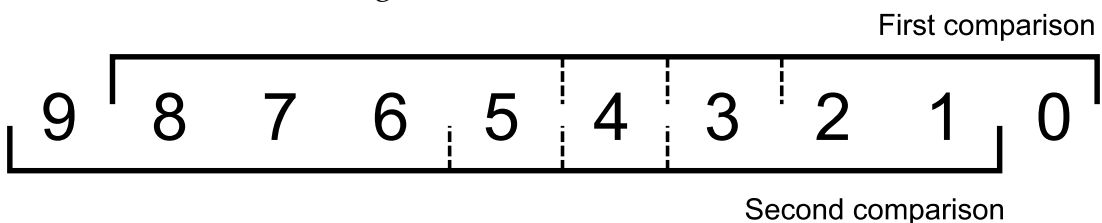
But two other alternative tests are provided. We first explore the test performed when the default multiple comparisons option is selected, but the numeric field above it is set to a value greater than 1.

## The Multiple Comparisons Test

Suppose you are monitoring incoming data from a series *that has no serial correlation (This is vital!)*. For example, you may be assessing monthly returns of a market trading system. These should be uncorrelated, because if they are correlated your system can be improved! Every time a new month 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 if 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 have been performed. Of course, in real life it would be difficult to make an honest assessment in advance of exactly how much testing has been done in the past and will be done in the future. 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 estimate of the total number of tests 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 the figure below.



This figure 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 in real life 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. This single 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, not just once after all of the multiple results are in. 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 1.2 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 two columns of results for each variable will be printed. The first 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. The second column reports how many indicator values lie on the recent side of the break boundary. You can count backwards this many cases from the end of your dataset to locate the case on which the maximum difference occurs.

In the more usual situation of the user specifying a large number of replications (100-1000 or so), one or 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. Note that this is not the p-value associated with the printed  $Z(U)$ . That  $Z(U)$  is the Z-score for the *greatest* break encountered across the user-specified search range. Thus, it is a 'best-of' statistic, though limited to that single variable. The solo p-value takes into account that a whole range of boundaries has been searched for the greatest break.

If this solo p-value 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 stepwise FWE algorithm (Page 13) is not available for this test.

My general recommendation for the use of the multiple-comparisons version of the mean break test is to set the *Multiple Comparisons* count to 1 the first time the test is done, 2 the second time, and so on. This is not a perfect solution to the multiple comparisons problem, but it is a good approximate solution.

On a final note, be aware that having a *statistically* significant mean break does not automatically 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.

## The Serial Correlation Test

It will often be the case that the variable we are testing for a mean break will have substantial serial correlation. This is particularly true for indicators derived from financial markets. Most indicators examine recent market history using a moving window, computing each indicator value from the market prices in the window. Thus, when we advance from one bar to the next, most historical prices will remain in the window. For example, suppose we compute an indicator based on the most recent 20 bars of price history. Then, adjacent indicator values will have 19 historical price bars in common, with only the oldest price being replaced by a new price as time marches forward. This leads to massive serial correlation.

Such dependency among the observations in a statistical test will, almost without exception, completely invalidate any standard statistical test. Testing for a break in the mean is no exception. Even a fully nonparametric test such as the combination of complete permutation and the Mann-Whitney U-test, as performed in the algorithm described in the prior section, is not immune to this nearly universal problem. Roughly stated, the issue is that when observations are dependent on one another, the dataset's statistical behavior is very different from the behavior that would be observed if there were no dependencies, and the latter situation is the assumption in this and nearly all other statistical tests.

In many of the tests available in *VarScreen*, serial correlation is accommodated by performing cyclic permutation instead of complete permutation. This largely preserves the serial dependencies inherent in the unpermuted data, and it is a reasonable if not perfect solution to the problem. But when testing for a break in the mean, cyclic permutation is much less applicable. All that data rotation does is move the locations of any mean breaks, so if a wide area is searched (the usual situation in indicator analysis), the generated null hypothesis distribution is worthless for testing purposes. And even if a narrow region is searched, the distortion induced by serial correlation has a much stronger impact than for the other tests in

*VarScreen*. For these reasons, only complete permutation is available for the mean break test.

But what can we do if our data is serially correlated, as will nearly always be the case for any variables whose computation is based on a moving window? Unfortunately, if the correlation extends back in history for a substantial distance, there is little or nothing that can be done. However, if the extent of serial correlation is a small fraction of the total number of observations, we can modify the test in a way that is almost certainly valid, though at the price of somewhat blurring the location of the break. (In practice, visual inspection of a plot of the series would show the location of any sudden break in most situations.)

To modify the mean break test, we take advantage of the fact that, by definition, observations separated by at least the extent of the serial dependency will be independent. For example, suppose we compute a financial market indicator by considering the most recent 20 historical price changes. Then the current computed value will be independent of the value that was computed 20 time intervals ago, because the two computation windows are completely disjoint. This, of course, assumes that the underlying price changes are independent, which in general is almost but not quite the case. So we would be wise to actually examine a table of lagged correlations to confirm the dropoff in correlation. This also assumes freedom from any significant dependencies that do not show up as correlation, but such situations are rare enough that we can almost always ignore that possibility.

This naturally leads to the following modified mean-break test, under the assumption that serial dependency extends for  $k$  prior observations. When we test for the presence of a mean break at a particular boundary, in the traditional test we compare all observations on one side of the trial boundary with all observations on the other side. But in this modified test we compare every  $k$ 'th observation on one side of the trial boundary with every  $k$ 'th on the other side, and we do so for every possible offset from 0 through  $k-1$ .

The 'score' is the maximum U-statistic over all trial boundaries and offsets. For generating the null hypothesis distribution, each offset is permuted separately, and all trial boundaries are tested for each permutation.

An example may make this admittedly vague description more clear. Suppose our indicator's historical lookback window is five observations, and we confirm that, as normally expected, serial correlation drops to essentially zero for observations five time samples apart. Number the observations beginning with 0 as the most recent.

Start the test with an offset of 0. Collect observations 0, 5, 10, 15, 20, and so forth. These are independent (at least as far as overlap-induced correlation goes, and probably entirely). If we are in a permutation replication, permute this set, which is legal since there are no dependencies to destroy. Test all trial boundaries that are within the user's specified range. For example, suppose the trial boundary is between observations 12 and 13. Then we will use the U-test to compare observations 0, 5, and 10 to observations 15, 20, 25, and so forth.

That takes care of offset 0. Now we move on to offset 1. Collect observations 1, 6, 11, 16, 21, and so forth. Permute these if we are in a permutation replication. Again test all trial boundaries. For a trial boundary between 12 and 13, test observations 1, 6, and 11 against 16, 21, and so forth.

Offsets of 2, 3, and 4 are treated similarly. The final test statistic is the maximum U-statistic taken over all offsets and trial boundaries.

## A Major Caveat, Again

At the risk of being overly pedantic, I'll repeat the warning given earlier: a *statistically significant* break in the mean does not necessarily imply a *problematic* break in the mean. A key part of any investigation of the properties of a time-series variable is visual examination of a plot of the

series. If the series contains a mean break large enough to be a problem, you will almost certainly see it in the plot, either as a sharp discontinuity or as a steady drift upward or downward. You won't need any printed number to tell you where it is. This is one of those rare times in life in which what you don't see really won't hurt you. Most of the time, anyway.

## 102      Stationarity Test for Break in Mean

---

### A Multiple-Comparisons Demonstration

We begin with a very simple test. Suppose we have a market trading system, and we monitor its monthly net profit. We decide that we want to see how the latest month stacks up compared to prior months. Is it an oddball win or loss? We know, perhaps from theory, or perhaps from empirical evidence, or even both, that our monthly returns have negligible serial correlation, so we don't have to compensate for that. With these things in mind, we specify a minimum and maximum size of recent history to be just one observation and run the test. The following results are obtained:

```
*****
*
* Computing stationarity test for break in means
*   1 predictor candidates
*   1 Minimum recent history cases
*   1 Maximum recent history cases
*   1 Multiple comparisons
*  100 replications of Monte-Carlo Permutation Test
*
*****
```

-----> Mean break test <-----

Z(U) refers to the maximum break across the user's range.  
Nrecent is the number of most recent cases this side of the break.  
Solo pval takes into account all tries across the range.

Variable	Z(U)	Nrecent	Solo pval
PROFIT	0.4811	1	0.7000

This large p-value of 0.70 is evidence that the mean profit has not just shifted. Of course, as any graduate of Statistics 101 will know, failing to achieve a significant p-value does not mean that the null hypothesis is true. Making that leap would be a cardinal sin called *accepting the null hypothesis*. So to be strictly correct, we cannot say that the average monthly return is still about the same as it has historically been, especially as this suspect conclusion is based on a single observation relative to history. But the most recent profit is obviously on par with other results, and as long as we have a reasonably long history of monthly profits we can treat this result as circumstantial but strong evidence that the mean profit is still the same.



The test just shown has yet another statistical flaw that we should be aware of. This procedure is fine as long as we perform a single test, examining the monthly return for just this one month and never doing it again. But that's pretty doubtful. Chances are we'll be repeating this test the next month, and the month after that, and so forth. Suppose the mean never changes. If we keep testing every month, it's likely that eventually we'll erroneously get a significant p-value simply by random chance. We can compensate for that to some degree. This was discussed earlier in this chapter, so we won't pursue the general issue here. But we will repeat the example just shown, this time assuming that we have already done this test 11 times (11 months of monthly returns) and we are now doing it the 12'th time. Here are the results:

```
*****
*
* Computing stationarity test for break in means
*   1 predictor candidates
*   1 Minimum recent history cases
*   1 Maximum recent history cases
*  12 Multiple comparisons
* 100 replications of Monte-Carlo Permutation Test
*
*****
```

```
-----> Mean break test <-----
```

Z(U) refers to the maximum break across the user's range.  
 Nrecent is the number of most recent cases this side of the break.  
 Solo pval takes into account all tries across the range.

Variable	Z(U)	Nrecent	Solo pval
PROFIT	0.4832	1	1.0000

Note that Z(U) changes slightly because the number of tested cases changes slightly, and the function that maps U to Z uses this quantity. More importantly, observe that the p-value rises to its maximum value. P-values will never decrease in a multiple-comparisons test, and will usually increase.

## 104 Stationarity Test for Break in Mean

---

### Testing Correlated Market Returns

Before investing a lot of time developing a trading system for some market, it would be wise to investigate whether the market returns (change in the market over some time period) are stable across the time period over which we will devise our system. As an illustration of this, I computed returns of OEX (the S&P 100 index) over time intervals of 1 day, 20 days, and several intermediate intervals. I looked at trial break boundaries ranging from 100 days ago to 5000 days ago. Computations are based on day-bars, so returns for any time interval greater than 1 day are serially correlated due to overlapped sharing of daily returns. Thus we must use the *serial correlation test* with a lag of 20. The following results are obtained:

```
*****
*
* Computing stationarity test for break in means
*   5 predictor candidates
*   100 Minimum recent history cases
*   5000 Maximum recent history cases
*   20 Maximum serial correlation lag
*   100 replications of Monte-Carlo Permutation Test
*
*****
```

-----> Mean break test <-----

Z(U) refers to the maximum break across the user's range.  
Nrecent is the number of most recent cases this side of the break.  
Solo pval takes into account all tries across the range.  
Unbiased pval takes into account all variables tested.

Variable	Z(U)	Nrecent	Solo pval	Unbiased pval
DAY_RETURN_20	3.5216	4361	0.1200	0.4000
DAY_RETURN_10	3.4939	4605	0.1900	0.4400
DAY_RETURN_1	3.1224	1926	0.4200	0.8200
DAY_RETURN_2	3.0492	1402	0.5600	0.9000
DAY_RETURN_5	3.0157	4517	0.5000	0.9000

None of the p-values are significant, especially when we look at the unbiased values that take into account our testing of several variables. Despite skating on the edge of illegally accepting a null hypothesis, given that we have over 6000 days represented we are reasonably safe in concluding that the returns of this market do not suffer a mean break.

## Testing an Indicator

We conclude these demonstrations by testing a standard indicator for a break in its mean. This indicator is ADX with a lookback window of 7 days, so we will need to perform the serial correlation test with a lag of 7. But it's always good to verify that serial correlation has fallen off to essentially zero by the time we think it will. The autocorrelation test provides these results:

```
*****
*
*   Computing autocorrelation analysis
*       20 maximum lag
*           ADX7 is the tested variable
*
*****
```

Lag	AutoCorr	Partial
1	0.951	0.951
2	0.837	-0.717
3	0.685	0.020
4	0.517	-0.086
5	0.348	-0.015
6	0.192	0.037
7	0.066	0.138
8	-0.014	0.235
9	-0.059	-0.365
10	-0.084	0.009
11	-0.096	-0.026
12	-0.098	-0.014
13	-0.096	0.047
14	-0.089	0.100
15	-0.077	0.149
16	-0.065	-0.277
17	-0.052	0.016
18	-0.042	-0.017
19	-0.033	-0.015
20	-0.027	0.012

We see that the autocorrelation is practically zero by a lag of 7 days. If we were fanatic we might want to go to a lag of 8 days, but we know that the computation of ADX7 uses only the last 7 days' prices, so we are safe in assuming that the observed autocorrelation of 0.066 is just random variation. If we had observed a significant value at this lag we would know that something is wrong with our ADX computation and the problem must be investigated.

## 106 Stationarity Test for Break in Mean

---

We proceed with the mean break test and observe the following result, which lets us cautiously (one cannot accept a null hypothesis) conclude that the mean of our indicator does not drift or shift precipitously.

```
*****
*
* Computing stationarity test for break in means
*   1 predictor candidates
*   100 Minimum recent history cases
*   5000 Maximum recent history cases
*   7 Maximum serial correlation lag
*   100 replications of Monte-Carlo Permutation Test
*
*****
```

-----> Mean break test <-----

Z(U) refers to the maximum break across the user's range.  
Nrecent is the number of most recent cases this side of the break.  
Solo pval takes into account all tries across the range.

Variable	Z(U)	Nrecent	Solo pval
ADX7	2.4502	2616	0.5500

What if we were to incorrectly use an ordinary mean break test, ignoring the large serial correlation? We would get a vivid demonstration of why we must use the serial correlation test when the data has serial correlation. Note the massive Z(U) and the minimum possible p-value.

```
*****
*
* Computing stationarity test for break in means
*   1 predictor candidates
*   100 Minimum recent history cases
*   5000 Maximum recent history cases
*   1 Multiple comparisons
*   100 replications of Monte-Carlo Permutation Test
*
*****
```

-----> Mean break test <-----

Z(U) refers to the maximum break across the user's range.  
Nrecent is the number of most recent cases this side of the break.  
Solo pval takes into account all tries across the range.

Variable	Z(U)	Nrecent	Solo pval
ADX7	6.1463	2619	0.0100

## Multiple Mean Breaks

The tests we have described look for a single boundary such that the observed values prior to this boundary are maximally larger or smaller than those on the other side of the boundary. It does not require a sharp break; if the observed values have a slow, steady increase or decrease in their central tendency, this test will discover this property by finding a boundary at which this tendency is most evident, even if this discovered boundary is itself not pronounced at all.

But what if the mean of the series has multiple changes, going up and down several times? This is certainly dangerous nonstationary behavior, but the algorithm may have difficulty finding a break that optimally splits the entire series. In such a situation, the user may be tempted to visually examine the series and then manually split it into one or more subsets that contain a visible break in the interior, and analyze these subsets separately.

The problem with this approach is that if a break is large enough to be visible, it is virtually guaranteed to have an extremely significant p-value. In other words, the user has set up the algorithm, giving it a series that is already known to have a significant mean break. This approach can still have some practical value, as the reported  $Z(U)$  can provide an indication of the size of the break relative to 'normal' changes in the series. But computed p-values will be worthless due to being prejudiced by manual selection of the tested series.

The bottom line is this: *the 'break in mean' test is a valuable screening tool in that it allows the user to quickly identify variables that exhibit nonstationarity in the form of the mean drifting monotonically or suddenly shifting, but it should not be used for making final conclusions. Nothing can replace careful visual examination of the plotted series.*

## FREL for Feature-Weighted Classification

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 screening predictor variables in a classification application which is relatively low noise but suffers from the Curse of Dimensionality: it employs numerous predictors but has a relatively small number of cases. The implementation in *VarScreen* is derived from the algorithm in this paper. However, I devised modifications that improve on the original version. In particular, my modifications provide feature weights that are more accurate and stable than those provided by the original algorithm.

My version of this test also provides an approximate Monte-Carlo permutation test (MCPT) of the null hypothesis that all supplied predictors have equal utility. In addition, the program provides an MCPT of the null hypothesis that the group of predictors as a unit is worthless. So far, I have been unable to devise an MCPT of any null hypothesis concerning individual predictors taken singly.

### Very Basic Theory

The theory behind the FREL classification algorithm, while not terribly complex, is nonetheless beyond the scope of this text, which is primarily a user's manual for the *VarScreen* program. Please see my book "Data Mining Algorithms in C++" for theoretical details along with C++ source code.

Most readers will be familiar with nearest-neighbor classification. You define a measure of the distance between two cases. To classify a test case, you compute its distance from each training case, and let its predicted class be that of whichever case is closest to it.

This is essentially what the FREL algorithm does, though it differentially weights each predictor, giving some predictors more weight than others in the definition of the distance function. And it also uses a slightly more sophisticated relative of this simple distance function.

There are  $K$  predictors. Let a test case have predictors  $\mathbf{x} = \{x_1, \dots, x_K\}$  and let a training-set case have predictors  $\mathbf{t} = \{t_1, \dots, t_K\}$ . Then the distance between the test case and the training case is the weighted city-block distance between these cases, as shown below.

$$D(\mathbf{x}, \mathbf{t}) = \sum_k w_k |x_k - t_k|$$

In order to classify an unknown test case  $\mathbf{x}$  based on a training set, we compute the distance separating the test case from each training case. A reasonable scheme, somewhat similar to what is used by FREL, is to say that whichever training case has the minimum distance from the test case determines the class assigned to the test case. The goal of the FREL algorithm is to find the  $K$  predictor weights that are optimal in some sense. As long as the predictors have commensurate scaling, which *VarScreen* guarantees by normalizing them to unit standard deviation, one can interpret the weight assigned to each predictor as a measure of that predictor's relative importance.

There are numerous ways by which the weights could be optimized. For example, we could perform cross validation in which we find weights that minimize the number of misclassifications. That method, like nearly all simple intuitive methods, optimizes some measure of global performance. But FREL operates on a much more subtle principle. The rough intuitive explanation I am about to give is not strictly correct (see the prior reference for the correct explanation), but it is close enough for most users of *Varscreen*.

To begin, suppose that for a given test case, we can compute for each class some measure of the likelihood that this class is the correct class for the test case. Then in order to classify the test case, we pick whichever class has the

maximum likelihood of being the correct class. The exact nature of this likelihood calculation is far beyond the scope of this text; see the prior reference for details.

Now let's talk about finding optimal weights. Consider a given test case. We define the *most offending incorrect answer* for this case as the incorrect answer (class) that has the highest likelihood. This is the answer for this case which is most likely to cause an error; it is the incorrect answer that is most difficult for the model to distinguish from the correct answer. As a key part of the optimization procedure, we want the most offending incorrect answer for test cases to have likelihood as small as possible compared to the likelihood of the correct answer. The other incorrect answers are of lesser consequence because they are easier for the model to avoid. FREL focuses primarily on the most offending answers.

To be clear, what was just described is not the exact optimization criterion, but a key component. A closer definition of what we actually optimize is the average across the training set of the difference between the likelihood of the most offending incorrect answer and the likelihood of the correct answer. This produces a model that is optimal in the sense of effectively handling the most difficult cases, while largely ignoring the easy cases. Many more traditional optimization schemes, most notably regression, are wasteful in that they put a lot of effort into making easy cases even easier. FREL pays relatively little attention to the easy cases, instead focusing its efforts on the difficult cases.

## *Regularization*

There is one more issue to consider: *regularization*. Training any prediction or classification model involves walking a fine line in regard to the degree to which the model pays attention to detail. If the model glosses over individual case details, it may be too weak to perform well. At the other extreme, a model which is highly detail oriented may *overfit* the training



data, meaning that in addition to learning the useful properties of the data, it also learns noise as if that noise were an inherent property of the data, rather than just unrepeatable random noise.

This balancing act is often achieved by a process called *regularization*, in which a model is penalized for paying too much attention to detail. The specifics of FREL regularization are beyond the scope of this text. Suffice it to say that the user can specify a degree of regularization to apply, with zero meaning that the model will be trained for maximum attention to detail, and larger positive values reducing the power of the model. As the regularization constant increases, the trained weights will tend to have less spread, so that the predictors will be assigned more equal importance.

### *Bootstrapping*

The run time of FREL training is proportional to the square of the number of cases, which blows up fast if the dataset is large. We can significantly speed operation for large datasets by averaging computed weights over multiple bootstrap samples (without replacement, as exactly repeated cases in any quantity cause problems). For example, suppose we randomly split the dataset in half and train with each separately. Each half would take 1/4 the time, and there are two of them, so run time is halved. *VarScreen* allows bootstrap splitting.

A nice side effect of bootstrap training and averaging of weights is that the computed weights are more stable. When the entire dataset is processed at once, small changes in just a few training cases can have significant effects on the optimal weights due to variation in the most offending incorrect answers. But by averaging over many separate bootstraps, we dilute these changes.

Suggestions for parameter choices appear on Page 114.

## FREL Classification in VarScreen

When we select FREL Classification from the Test menu, a dialog similar to that shown below appears.

FREL feature-based classification

Predictors	Target
_SEQNUM_	_SEQNUM_
CMMMA_10	CMMMA_10
CMMMA_20	CMMMA_20
CMMMA_5	CMMMA_5
DAY_RETURN	DAY_RETURN
LIN_ATR_15	LIN_ATR_15
LIN_ATR_5	LIN_ATR_5
LIN_ATR_7	LIN_ATR_7
LINDEV_10	LINDEV_10
LINDEV_20	LINDEV_20
LINDEV_5	LINDEV_5
PVFIT_15	PVFIT_15
PVFIT_7	PVFIT_7
RSI_10	RSI_10
RSI_20	RSI_20
RSI_5	RSI_5
RTVY_12	RTVY_12
RTVY_25	RTVY_25
RTVY_6	RTVY_6

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

OK Cancel

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. Remember that only a single model will be tested, that employing *all* of the selected

variables. This test does not do any screening for predictor subsets. You will use the trained weights to compare the importance of predictors.

The *Target* column is used to specify the target variable. This variable will be partitioned into two or more classes based on its values. FREL does not permit continuous targets. It is a pure classifier, which is why continuous targets are partitioned into classes.

*Target bins* specifies the number of bins into which the target will be categorized. If *Target bins* equals the number of distinct values of the target variable, this many bins will be used. If *Target bins* is larger than the number of distinct values in the target variable, an error message will be generated and the log file will show the actual number of distinct values. If the target has few or no ties, and *Target bins* is much less than the number of distinct values, the dataset will be partitioned into this many bins, all of which have equal or very nearly equal size (depending on the number of ties).

*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. Thus, in *VarScreen*, regularization serves only to limit the weight distribution. With no regularization (0.0 specified), one or a few predictors can have a very large weight, while others have weights near zero. In some applications this is an honest reflection of reality, in which case zero regularization is appropriate. In other applications this may be a form of overfitting. Applying regularization will prevent such large spreads, and ensure that all predictors are given significant weight. It's good practice to repeat the test with several different degrees of regularization and compare results.

*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 a reasonable number of members of each target class are present in each sample. A reasonable rule of thumb for the number of iterations is to make the product of the number of iterations times the sample size several times the number of training cases. More is definitely better, so you should make this as large as possible, consistent with being able to endure the run time of the test.

If runtime allows, a *Monte-Carlo permutation test* can test some specialized null hypotheses about the predictors. It is vital to understand that these tests are radically different from the other permutation tests in *VarScreen*.

Unfortunately, I am not aware of any way of performing a perfect *individual-candidate* MCPT with FREL. In other tests, the candidate predictors are handled individually, so the p-values (at least the solo tests) are independent; each p-value applies to that particular predictor without regard for the other predictors. But FREL considers all candidates simultaneously. This dependence changes the nature of MCPT, radically changing 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 the null hypothesis for the unbiased p-values is that *all* candidates are *worthless*. Thus, those MCPT tests let us identify *individual* candidates which have predictive power when considered alone. But for FREL, the tests are relative, and much more difficult to interpret due to the strong interaction between weights in the training process. The null hypothesis is still that all candidates are worthless. But, roughly speaking, we are testing against the alternative hypothesis that a given variable has more predictive power than its competitors. Thus, with only modest misunderstanding, we can think of the null hypothesis as being that all candidates have equal predictive power, regardless of whether that power is tiny or large. Because of the joint estimation of weights, we can

think of the p-value of a variable as being a test of whether that variable is superior to its competitors. The unbiased p-values compensate for the fact that we are testing numerous candidates, and any of them may be outstanding by random luck.

Okay, many of you are rolling your eyes now, upset that I have not been more rigorous in this discussion. The sad truth is that an exact definition of the individual solo and unbiased p-values is difficult to state with any rigor. The bottom line is that we should examine the p-values for the variables at the top of the list, those with largest weights, and decide that those with tiny p-values are likely superior to the others. But keep in mind that the joint weight estimation means that the *absolute* predictive power does not affect p-values. It may be that a variable with tiny p-value is *almost* worthless, and its competitors are *completely* worthless. By the same token, it may be that this single candidate is *excellent*, while its competitors are merely *very, very good*. Please remember 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 relative ranking of competitors.

The weight interactions during training has a curious effect on the solo p-values. As we descend from the highest-weight predictors to the lowest-weight, the solo p-values may jump up and down between high and low significance seemingly randomly if there are nearly worthless predictors present. This is due to the fact that the p-values for worthless candidates in any statistical test have a uniform distribution, with all values in the range 0–1 being equally likely. For this reason we should focus on the unbiased p-values, ignoring the solo p-values except perhaps (and with great caution) for any top-ranked candidates.

All is not hopeless when it comes to p-values. *VarScreen* computes one additional p-value, called the *Grand p-value*. This is a net p-value for all predictors taken together, with their trained weights, to be effective at classification. The null hypothesis is that none of the candidates have any predictive power, and the Grand p-value is the probability that if this were

so, we would have obtained performance at least as good as that obtained. *A very small grand p-value is a necessary condition for any of the individual p-values to be meaningful.* There is no point in considering the relative power of the predictors if we cannot assert that at least one of them has predictive power. Therefore, the grand p-value should be the first performance criterion that you examine.

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; more information can be found on Page 22. Here is a briefer summary of the issue. If the user is confident that there is no serial dependency in the data, 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 permuted 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 many 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 optimization criterion 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.

## An Inappropriate but Illustrative FREL Example

I computed a handful of common indicators as well as next-day return for about 7700 days of OEX, and ran two FREL tests, one with the modest default regularization of 0.01, and one with no regularization. These results are shown on the next two pages. In both test, the consistently large unbiased p-values indicate that no single indicator stands out as clearly superior. But we see the curious outcome that with modest regularization, the grand p-value is highly significant, implying that the indicators taken as a group do have significant predictive power. However, when we remove regularization to allow tighter focus, the model learns noise patterns as if they are real, and hence the statistical significance drops. This is yet another reason to repeat FREL tests with several degrees of regularization.



```

*****
*
* Computing FREL (Feature weighting as regularized energy-based learning) *
*   17 predictor candidates                                           *
*   3 target bins                                                    *
*   Regularization = 0.0100                                          *
*   20 bootstrap iterations                                          *
* 1000 cases per bootstrap iteration                                 *
*   100 replications of complete Monte-Carlo Permutation Test      *
*   1 CUDA kernels                                                  *
*   10 CUDA granularity                                             *
*
*****

```

Target bounds...

-0.21441            0.31010

Target marginals...

0.33325            0.33338            0.33338

-----> FREL with DAY\_RETURN <-----

Variable	Weight	Solo pval	Unbiased pval
RSI_5	17.1045	0.2800	0.7400
RTVY_12	14.4541	0.9300	0.9900
CMMA_5	9.8976	0.7600	1.0000
RTVY_6	9.7870	0.7800	1.0000
RTVY_25	7.8670	0.9500	1.0000
LINDEV_10	4.8351	0.0300	1.0000
RSI_20	4.7278	0.5800	1.0000
LINDEV_20	4.5317	0.1400	1.0000
CMMA_20	4.4912	0.0600	1.0000
LIN_ATR_5	3.9567	0.0300	1.0000
LINDEV_5	3.8925	0.0700	1.0000
LIN_ATR_7	3.3670	0.4800	1.0000
PVFIT_7	2.8642	0.2300	1.0000
PVFIT_15	2.8331	0.0900	1.0000
LIN_ATR_15	2.7960	0.0300	1.0000
_RSI_10	1.6307	0.1900	1.0000
CMMA_10	0.9640	0.0400	1.0000

Grand p-value = 0.010

## 120 FREL for Feature-Weighted Classification

```
*****
*
* Computing FREL (Feature weighting as regularized energy-based learn
*   17 predictor candidates
*   3 target bins
*   Regularization = 0.0000
*   20 bootstrap iterations
* 1000 cases per bootstrap iteration
* 100 replications of complete Monte-Carlo Permutation Test
*   1 CUDA kernels
*   50 CUDA granularity
*
*****
```

```
Target bounds...
-0.21441      0.31010
```

```
Target marginals...
0.33325      0.33338      0.33338
```

```
-----> FREL with DAY_RETURN <-----
```

Variable	Weight	Solo pval	Unbiased pval
RSI_5	18.3304	0.2400	0.5500
RTVY_12	16.1206	0.7500	0.9300
RTVY_25	11.1045	0.2900	1.0000
RTVY_6	10.3342	0.6800	1.0000
CMMA_5	10.0686	0.6200	1.0000
RSI_20	6.6243	0.3000	1.0000
LINDEV_20	3.7985	0.4000	1.0000
LIN_ATR_7	3.5991	0.3100	1.0000
LINDEV_10	3.5054	0.6700	1.0000
LINDEV_5	3.1402	0.5400	1.0000
CMMA_20	3.0029	0.2400	1.0000
LIN_ATR_5	2.6766	0.6800	1.0000
PVFIT_7	2.3448	0.7100	1.0000
PVFIT_15	2.1229	0.7600	1.0000
LIN_ATR_15	2.0350	0.5600	1.0000
RSI_10	0.6177	0.8000	1.0000
CMMA_10	0.5742	0.1900	1.0000

```
Grand p-value = 0.200
```

This is more of an example of FREL not being appropriate than a useful example, the reason having been stated at the start of this chapter: “a classification application which is relatively low noise but suffers from the Curse of Dimensionality: it employs numerous predictors but has a relatively small number of cases”. This example is extremely high noise yet has about 7700 cases, the exact opposite of where we would want to use FREL. Now you see why.

## A Difficult Zero-Noise Problem, Solved

It's well known that parity problems are difficult to solve. This is because the predictors, taken individually or even in any subset that does not include *all* actual predictors, is totally worthless. Any incomplete subset of the actual predictors has zero mutual information or other relationship with the target. This is in contrast with nearly all other prediction/classification problems, in which the predictors have nonzero mutual information with the target, certainly in subsets and usually even individually. This, of course, is the basis of stepwise predictor selection, in which we cumulate more and more predictive power as we add predictors. Such is not the case with parity problems, which are strictly all-or-nothing.

I created a set of ten independent uniform random numbers in the range  $-1$  to  $1$ , and called them  $X_0$  through  $X_9$ . I then computed three binary flags using the variables  $X_3$ ,  $X_4$ , and  $X_5$ . For each of these three variables, its flag is  $1$  if the variable is greater than zero, and  $0$  otherwise. My target variable  $\text{PARITY}_3$  is the sum of these three flags, mod  $2$ . The other seven  $X$  variables are ignored. Given the values of  $X_3$ ,  $X_4$ , and  $X_5$ ,  $\text{PARITY}_3$  can be computed exactly. Therefore, this can be considered to be a zero-noise problem, though a very difficult one.

The FREL algorithm was used to test the ten  $X$  variable candidates in predicting  $\text{PARITY}_3$ . Naturally two target bins were used, since the target variable is binary  $0/1$ . In the run shown here, the default regularization of  $0.01$  was used, although runs with regularizations of  $0$  and  $0.1$  gave nearly identical results. To eliminate the impact of random learning, no bootstraps were used. There were not a large number of cases, so runtime was fast enough that  $10,000$  MCPT replications were practical. The results are shown on the next page.

## 122 FREL for Feature-Weighted Classification

---

```
*****
*
* Computing FREL (Feature weighting as regularized energy-based learning) *
*   10 predictor candidates *
*   2 target bins *
*   Regularization = 0.0100 *
*   No bootstrap (full dataset used exactly once) *
* 10000 replications of complete Monte-Carlo Permutation Test *
*   1 CUDA kernels *
*   10 CUDA granularity *
*****
```

```
Target bounds...
0.00000
```

```
Target marginals...
0.48600      0.51400
```

```
-----> FREL with PARITY3 <-----
```

Variable	Weight	Solo pval	Unbiased pval
X3	35.3597	0.0205	0.2765
X4	34.4036	0.0284	0.3028
X5	28.2576	0.0532	0.5252
X2	1.1008	0.7517	1.0000
X9	0.4625	0.9916	1.0000
X6	0.1235	1.0000	1.0000
X0	0.0867	1.0000	1.0000
X7	0.0839	1.0000	1.0000
X1	0.0672	1.0000	1.0000
X8	0.0547	1.0000	1.0000

```
Grand p-value = 0.0001
```

The first thing to notice is that the algorithm did a fabulous job of identifying X3, X4, and X5 as the predictors; their weights are far greater than any others. Also, their solo p-values are much smaller than the others. Their unbiased p-values are relatively small but not tiny. Recall that very roughly speaking, these are the p-values for the null hypothesis that the variable is not superior to its competitors. But since the three actual predictors are equally powerful, we would not expect tiny p-values. In general, individual p-values for the FREL test are of little value. But notice that the grand p-value, which tests the null hypothesis that all predictors are worthless, is its minimum possible value.

## FSCA: Forward Selection Component Analysis

The FSCA algorithm in *VarScreen* is largely based on the paper “Forward Selection Component Analysis: Algorithms and Applications” by Luca Pugcini and Sean McLoone, published in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, December 2017. I have, however, somewhat modified their algorithm to improve stability and performance.

The long-employed technique of *principal components* is based on the idea that a group of variables may have correlation that is due to them being impacted by one or more underlying common factors, which may or may not be measurable. For example, suppose we study a population of people, and we measure the height and weight of each person. These two quantities will obviously have high correlation. If we were to compute the principal components of this dataset, we would likely find that the dominant underlying factor affecting both variables is something that we might call ‘size’, a measure of how large a person is. If we were to then remove this size component from consideration and see what source of variation remains, we would find it to be something that might be called ‘body shape’. In some applications, simply naming the components of variation, such as size and shape here, is all we desire. In other applications we might want to go on and, for each case (person here), compute numerical values for their size and shape. *VarScreen* can do both, though with an algorithm that is far more sophisticated than traditional principal components or its close relative factor analysis.

Traditional principal components served the data analysis community well for decades. However, today’s data environment has presented it with a frequently insurmountable obstacle: the sheer mass of data we need to examine in many modern applications. In the early days of statistical computing, an application typically had only a handful of measured variables, or a few dozen in the largest applications. Today’s databases may contain thousands of variables. This leads to at least three major problems:

- 1) It can be impractical to measure thousands of variables in an ongoing basis. You may have the resources to collect thousands of variables for an initial experiment, but traditional principal components requires (or at least assumes) that future use of the computed component weights will employ *all* of the variables that were used to derive the principal components. It would be much more economical if the initial computation of principal components included the ability to narrow down the list of variables included in the analysis for subsequent use.
- 2) Even if your only goal in using principal component analysis is to compute new, alternative predictor variables, being able to put a name to the new variables is always good. And sometimes your *only* goal is to name the principal components in order to better understand how the original variables are related. Obviously, the fewer original variables that are involved, the easier it will be to name the linear combinations of them that define their principal components.
- 3) A more subtle problem may arise when a huge dataset contains one or more groups of highly correlated variables, all measuring nearly the same thing. For example, in a financial application we may measure market volatility several dozen ways, using different definitions and different lookbacks. Chances are that all of them will show up as facets of a 'volatility' principal component, when in fact it may be that just one or a few of them will provide nearly all of the volatility information contained in the dataset. Why measure and manipulate 40 volatility variables when a carefully selected 2 or 3 will tell nearly the same story?

## Intuitive Justification of the Algorithm

The Puggini and McLoone algorithm implemented in *VarScreen* is a clever fusion of stepwise selection with traditional principal component analysis. First let's briefly review stepwise selection of variables in a more general context.

Suppose we have a relatively large set of variables from which we want to choose a smaller subset that is optimal in some sense. This may be a prediction or classification application, or the principal components application here, or some other application; the issues are the same. The 'ideal' approach would be to test every possible subset and choose the best. But with even a moderate number of variables in the population, the combinatoric explosion would make testing every possible subset difficult, and with a large population exhaustive testing becomes impossible.

So we nearly always take an approach to subset selection that is less than optimal but still usually reasonable, and a whole lot more efficient. This more practical approach begins by finding the single variable (subset of size one) that is optimal. Then it finds a second variable, which *in conjunction with the variable already selected* maximizes the performance criterion. The key point is that the choice of this second variable is conditional on the first selected variable.

The intuitive appeal of these first two steps is obvious, but it is not perfect. For example, suppose we have three variables,  $X_1$ ,  $X_2$ , and  $X_3$ , and the pair  $X_2, X_3$  is the best possible pair. It is not inconceivable that the best *single* variable is  $X_1$ , because perhaps the  $X_2, X_3$  combination is reliant on both of them being known. In such a situation, which can be more common than we might realize, we will not find the truly best pair. We will be stuck with  $X_1$  plus either  $X_2$  or  $X_3$ , a suboptimal pair.

Once we have two variables in hand, we can take either of two approaches to advancing the variable selection process, one of which can go a long way toward resolving the problem just mentioned. The simpler and more common approach is to just continue greedily selecting the best next member of the growing subset. In particular, we find the third variable which *in conjunction with the two variables already selected* maximizes the performance criterion. Then we find the fourth, given the first three, and so forth. This is traditional forward stepwise selection; it is fast and easy to understand.

We have an alternative approach after we have two or more variables in the subset: instead of immediately attempting to add a new variable to those already in hand, we see first if we can *replace* one of our current variables, keeping the number of variables the same. Consider the problematic example just discussed, in which X2 and X3 are the best pair, while X1 is the best single variable. We will select X1 and then, say, X2 as our (suboptimal) best pair. The next step would then find that replacing X1 with X3 is the best move, thereby finding the truly best pair. If we cannot improve the optimization criterion by replacing an existing variable, then we just add one. This process continues until we have in hand the user's requested number of variables. Although this alternative algorithm will still not guaranty the best subset of variables, in nearly all cases it will produce a much more optimal subset than simple greedy forward selection.

This alternative algorithm that employs replacement does have a property that some users may consider a disadvantage. When you use ordinary (greedy) forward stepwise selection, the variables you end up with are ordered in importance, the first being most important, the second being most important conditional on having the first, and so forth. But now consider our running example with X1, X2, and X3. Because X1 is most important, it will be picked first. Then we may pick, say, X2. Finally, the replacement algorithm will replace X1 with X3, giving us the optimal subset X2, X3. But the single most important variable, X1, doesn't even make the final subset. Whether this is a problem depends on the application and the user's opinion.



Thus far we have been tossing around the term ‘optimal’ without defining it. When we are choosing an ‘optimal’ subset of variables to employ in computation of principal components, the quantity that we optimize is quite complex. Mathematical details as well as source code are in my book “Modern Data Mining Algorithms in C++ and CUDA C”. Here I will just make the vague statement that the optimized quantity is a mathematically sound measure of the fraction of the total variance in all variables that is predictable given knowledge of a trial subset of variables. This quantity is easily defined only for the simple case of a single variable in the subset. In this situation, the ‘score’ for a single variable is the mean of its squared correlation with each other variable in the population. Because this quantity may be of interest to users, *VarScreen* prints a table of these values.

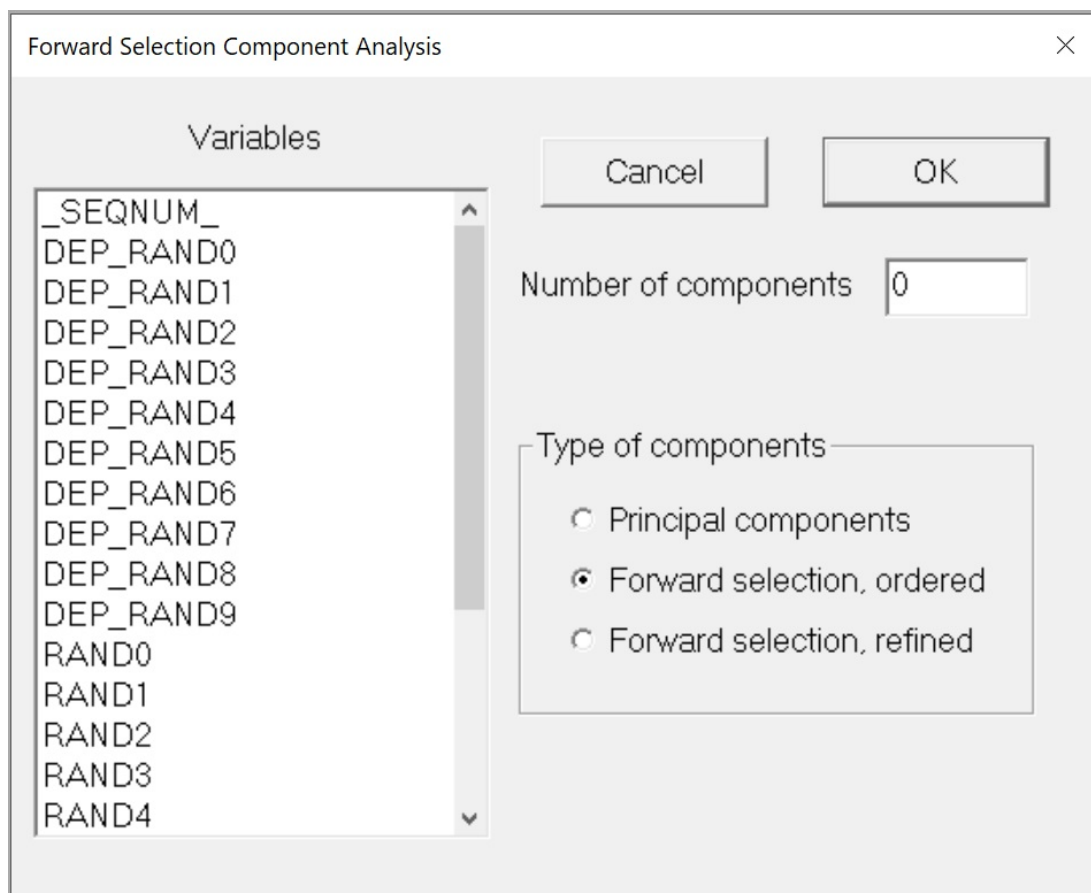
## Desirable Properties of Principal Components

When we compute principal components from a set of variables for use as inputs to a prediction or classification model, there are three desirable properties that *VarScreen* provides:

- 1) The principal components are standardized to have a mean of zero and a standard deviation of one. Nearly all training algorithms are most stable when their inputs are centered around zero rather than some number off in left field, and having all inputs be commensurate also helps stability.
- 2) The principal components are orthogonal, meaning that as far as linearity goes, they are uncorrelated. Most training algorithms prefer for their inputs to have as little correlation as possible.
- 3) It is nice if we have as many principal components as input variables, which should always be possible. If there is no collinearity among the input variables. This way the principal components capture all of the variation present in the input variables.

## Computing Principal Components

When *FSCA* is selected from the *Create* menu, a dialog box similar to that shown below will appear, from which the user makes 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.

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, which is to find a small subset of the variables which captures the majority of the variation in the complete set. 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. In this case, the number of components computed equals the number of variables selected, unless collinearity reduces the number of independent sources of variation.

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! 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 principal components computed.

For all three options, the newly computed variables will have zero mean, unit standard deviation, and they will be linearly 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.

## A Contrived Example of Ordered Forward Selection

Here is an example of the first 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 forward selection with no refinement, the traditional principal components are printed first, even though this usually is of minimal interest.

```
*****
*
* Computing Forward Selection Component Analysis
*
* Forward selected components with strict variable ordering
* 9 predictors selected
* 9 components to be computed
* This option does not use multithreading (only refinement does)
*
*****
```

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

The user selected the nine variables defined above, but *VarScreen* discovered that the nine variables include only six independent sources of variation. This is because there are three sum variables, SUM12, SUM34, and SUM1234, that are exact linear combinations of other selected variables. Those constitute three sources of collinearity with RAND1 through RAND4.

The first row is the eigenvalues of the correlation matrix, which are the variances of the principal components. The second row is the cumulative sum of the eigenvalues, expressed as a percent of their total. Here we see that the first traditional principal component alone accounts for a third of the total variation among the complete set of variables. By including a second principal component we can account for over half of the total variation.

Subsequent rows of this matrix are the correlations of the original variables with the principal components. For example, RAND1 has a correlation of 0.4835 with the first principal component. It should be no surprise that SUM1234 has nearly perfect correlation with the first principal component; this component represents the combined contributions of RAND1 through RAND4. And we see that the second principal component represents the contrast between RAND1 and RAND2 versus RAND3 and RAND4. Subsequent principal components become a bit more difficult to name.

The first eigenvector represents about a third of the total data variation. Not surprisingly, it correlates almost perfectly with SUM1234, very highly with SUM12 and SUM34, and moderately highly with RAND1-RAND4. Obviously it is a measure of grand variation.

When we remove the grand variation, the primary component we are left with is the contrast between RAND1 and RAND2 versus RAND3 and RAND4. Combined with the first component, we can account for over 55 percent of the total variation. The remaining components are other contrasts as well as RAND5 and RAND6.

*VarScreen* then prints the mean squared correlation of each selected variable with all other selected variables. This shows the user which variables carry the most and least information about the other variables. Whichever variable has the highest value of this quantity will be the first selected variable in the stepwise process. Here we see that SUM1234 carries the most communal information, SUM12 and SUM34 come in second and third, and RAND5 and RAND6 carry no information about the 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

*VarScreen* notes that due to the presence of collinearity in the variables, some slight adjustments to the correlation matrix were necessary. These adjustments are inconsequential as far as results are concerned, but they are necessary to ensure numeric stability. The program begins with the single variable which had maximum mean squared correlation, and shows the order in which new variables are added. The selection criteria are also shown, although in most cases they are of relatively little interest. The only way in which they may be interesting is if we see that adding some new variable provides only a slight increase in the criterion. In this situation we may consider reducing the number of variables to use.

NOTE: Adjusting correlations because some variables are collinear

Commencing stepwise construction with SUM1234

```
Added SUM12 for criterion=4.973085
Added RAND2 for criterion=6.011567
Added RAND6 for criterion=7.011644
Added RAND5 for criterion=8.010346
Added RAND4 for criterion=8.999844
```

The final table printed is the most important, because it shows the coefficients that would multiply each *standardized* input variable in order to compute each component. The table for this example is shown below, and pertinent observations follow.

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

Note the following points:

- 1) These coefficients assume that the input variables are standardized to have mean zero and standard deviation one.
- 2) The variables appear in the order in which they were selected, which is their order of importance, conditional on prior variables.
- 3) Each component depends on only the most recently selected variable and prior selections.
- 4) The first component is a single variable, the most important one (the variable whose value reveals the most about the other variables).
- 5) The two variables that are completely unrelated to the other variables, RAND5 and RAND6, are to within tiny random error, components unto themselves. The components have a coefficient of 1.0 for the single variable, and 0.0 for all other variables.
- 6) Some variables that are related to others are notably missing due to collinearity. SUM34 is missing because it equals SUM1234 minus SUM12, both present. RAND1 is missing because it equals SUM12 minus RAND2. And RAND3=SUM1234-SUM12-RAND4, so it is omitted. The components have no redundancy.



## A Contrived Example of Refined Selection

We use the same variables as in the prior example to demonstrate the other FSCA option, forward selection with backward refinement. As in the prior example, the traditional principal components are printed first, along with the table of mean squared correlations so we can judge the importance of each single variable. But now, in addition to seeing the variables that are added on each step, we also see some variables being removed.

```
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
```

For both FSCA algorithms, the first two variables will be the same, SUM1234 and SUM12 here. But then the algorithm is able to very slightly increase the criterion by removing SUM1234 and replacing it with SUM34. So now the two selected variables are SUM12 and SUM34, which is more intuitively appealing. It does the same ‘sum-splitting’ again, selecting RAND2, which then allows it to replace SUM12 with SUM1. We have to wait a couple steps before it does the final split of SUM34 into SUM3 and SUM4, because it chooses to select the two completely independent variables, RAND5 and RAND6, first. Note how lovely the final selection of component variables is; it’s just the basis variables, with no sum variables involved.

Eigenvalues, cumulative percent, and 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

As was pointed out earlier, the refinement process destroys any ordering by importance. In particular, note that even the single most important variable, SUM1234, which explains most of the variance of the other variables, doesn't make the final subset. But what we get in return is a subset that is superior in terms of the final performance criterion (except for rare pathological cases), and often more intuitively appealing.

As a final note on refinement, recall from Page 127 that a desirable property of principal components is that they be orthogonal. Because the selected variables are not ordered in any way, we can use any algorithm we wish to orthogonalize the principal components that are based on the refined selection of variables. The easiest and most intuitively appealing way to do this is to simply compute the traditional principal components, so this is what I do. The first line in the chart above is the eigenvalues of the correlation matrix of the selected variables, and the second line is the cumulative variance captured by the successive components. It should be no surprise here that all of the eigenvalues are about equal. This is because the selected variables are independent; any ordering that appears is just random variation in the dataset.

For this same reason (all variables are independent in this example), the correlations between the variables and the components are random. *It's pointless to try to make sense of the correlations, since they are dependent only on random variation in the dataset.* If the selected variables did have correlations, it's more likely that one could see some sense in the correlations.

Finally, note that this is a table of correlations, not weights as was the case for ordered forward selection. If you want weights, just divide each correlation by its corresponding eigenvalue. For example, the coefficient for RAND2 in the third component would be  $0.1597 / 1.002$ .

## A Practical Example With Indicators

I computed 39 straightforward indicators from several decades of OEX and performed forward selection with refinement, decreeing in advance that I would like to find 6 independent (linearly orthogonal) components. Later in this section I'll roughly define the relevant indicators, omitting some details about scaling and normalization that are discussed in my book "Statistically Sound Indicators for Financial Market Prediction". These are the results obtained:

Commencing stepwise construction with CMMA\_10

```
Added LINDEV_10 for criterion=15.561423
Added QUA_ATR_15 for criterion=18.712641
  Replaced CMMA_10 with CMMA_20 to get criterion = 19.118366
  Replaced QUA_ATR_15 with LIN_ATR_7 to get criterion = 19.201437
Added ADX15 for criterion=22.041746
Added ADX7 for criterion=24.205274
Added ADX15_A15 for criterion=25.931842
  Replaced ADX15 with ADX15_D30 to get criterion = 26.101375
```

Eigenvalues, cumulative percent, and selected principal component factor structure

Eigenvalue	1.826	1.227	1.000	0.941	0.758	0.249
Cumulative	4.681	7.826	10.391	12.803	14.746	15.385
CMMA_20	0.9141	0.0050	0.0181	0.1889	0.0647	0.3523
LINDEV_10	0.3694	-0.2088	-0.0511	-0.8639	-0.2665	0.0092
LIN_ATR_7	0.9166	-0.0207	0.0040	0.1460	0.1186	-0.3522
ADX15_D30	0.0018	0.5814	-0.6511	-0.2345	0.4276	0.0135
ADX7	0.1161	0.7809	0.0014	0.1109	-0.6033	-0.0227
ADX15_A15	-0.0005	0.4845	0.7571	-0.2654	0.3487	0.0057

The following points are noteworthy:

- 1) The single most important variable (in the sense of capturing maximum information about all other variables) is CMMA\_10. This is the current market close minus the 10-day moving average. As a side note, be aware that for my financial prediction models, this variable and its close relatives have nearly always been outstanding predictors.

- 2) Given CMMA\_10, the most important among the remaining variables is LINDEV\_10. This indicator is computed by fitting a least-squares straight line to the most recent 10 closing prices, using that line to predict what the current close would be if the trend continued, and subtracting that quantity from the actual current close. It's somewhat surprising to me that this variable would be selected second, because it would seem to be quite correlated with CMMA\_10, and hence would have little to add by its inclusion. But apparently the information obtained from a linear fit deviation and that obtained from a simple moving average deviation is different enough that LINDEV\_10 adds a lot of information above and beyond CMMA\_10. Fascinating.
- 3) The next variable added, QUA\_ATR\_15, is a measure of quadratic curvature over the prior 15 bars, normalized by average true range. In other words, it's not a trend indicator, but rather a measure of how quickly the trend is changing.
- 4) Once that variable is included, two more substitutions occur. LIN\_ATR\_7 is a simple linear trend, normalized by average true range.
- 5) ADX\_15 and ADX\_7 are the ordinary ADX indicators with specified lookback, and the last two indicators that appear are variations on ADX, its acceleration and its change over time.
- 6) Consider the six indicators chosen to, as a group, maximally capture the variation inherent in the complete set of 39 candidates. We have a deviation from the moving average, deviation from a linear fit, trend (slope of a linear fit), raw ADX, and two measures of how ADX is changing. These seem relatively independent measures of market movement, as one would expect.

- 
- 7) Interpretation of the 'Cumulative percent' row is somewhat different from how one interprets this quantity in traditional principal components. For refined forward selection, this is the percent relative to the number of non-collinear variables in the population. In the prior contrived example, we had 6 of the 9 selected variables not collinear, and 6 components were selected, so the last (rightmost) value was 100 percent. In this 'practical' example, all 39 of the selected variables are non-collinear, and we computed 6 components, so we have  $6/39=15.385$  percent of the non-collinear variables accounted for by the 6 components. So the cumulative percent row for refined forward selection is the percent of total non-collinear variable variances. Recall that the selected variables have all been standardized to unit variance, so the sum of their variances is just the number of non-collinear variables.
  - 8) Look at the eigenvalues. The first traditional principal component of the six refined variables is large. Then they drop off slowly, except for the last, which is a paltry 0.249. This tells us that we might want to rerun the procedure, requesting just five components rather than six.
  - 9) As is usually the case, putting a name to these components is difficult. However, we can probably name the first. It correlates very highly with both CMMA\_20 (0.9141) and LIN\_ATR\_7 (0.9166). It also has significant positive correlation with LINDEV\_10 (0.3694). So this component being high corresponds to the current close being much greater than the recent moving average and also for recent closes to have a strong upward trend. It also moderately corresponds to the current close being above the close suggested by projecting a least-squares straight line. So one would be inclined to call this a 'super-trend' component indicator: the market is trending upward, and the current close has even pushed above the trend.
  - 10) The second component is strictly ADX related, with a high value corresponding to ADX being high, increasing, and accelerating.

## LFS: Local Feature Selection

The feature selection algorithm described in this section is based on the fascinating algorithm presented 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). It is extremely powerful in many applications, but at the price of being quite complex. This *VarScreen* operating manual will discuss only its applicability, as well as how to invoke it and how to interpret its results. For full mathematical details as well as highly commented source code you should see my book “Modern Data Mining Algorithms in C++ and CUDA C”.

This algorithm, which is limited to classification applications (not prediction), is especially useful in a particular situation: when one or more of the candidate predictors are highly effective in a limited area of the problem domain, while being worthless or nearly worthless across most of the rest of the problem domain. As a primitive example, suppose we have two predictors,  $X_1$  and  $X_2$ , and it happens that  $X_1$  is one hundred percent correct when  $X_2$  is positive, but  $X_1$  is worthless when  $X_2$  is negative. Of course, most modern nonlinear models can handle such a situation quite well, *assuming that we supply it with good predictors*. Unfortunately, most predictor selection algorithms, including most supplied by *VarScreen*, have a strong tendency to favor candidates that perform at least moderately well over much of the problem domain. Candidates that may be fabulous performers over a limited area of the domain, and whose performance could be capitalized on by modern nonlinear algorithms, will often be missed by predictor selection algorithms that prefer candidates that perform well over a broad area of the domain. That’s where *Local Feature Selection* is valuable. It lets us find predictors that may not be broadly applicable, but that are outstanding performers in some special area(s) of the problem domain.

As a slightly more specific example of where local feature selection is useful, consider an application for which we have numerous predictor candidates, two of which are related to the class of a case in a way reminiscent of XOR logic. In particular, suppose  $X_1$  and  $X_2$  have identical symmetric distributions centered around zero, independent of the class, and we have two classes. If  $X_1$  and  $X_2$  are both positive, or they are both negative, the case is in Class 1. On the other hand, if one of these two variables is positive and the other is negative, the case is in Class 2. Other predictor candidates have weaker individual relationships with the class.

There is an interesting anomaly about the relationship between  $X_1$ ,  $X_2$ , and the class of the case.  $X_1$  has the same distribution for both classes, and the same is true of  $X_2$ . Thus, either of these variables alone has no predictive power whatsoever. Yet taken together, they can achieve perfect classification by a very simple rule that nearly any modern nonlinear model can easily learn. If any of the other competing predictors has even slight predictive power for the class of a case, it will be picked over  $X_1$  and  $X_2$  by predictor screening algorithms that focus on individual performance or broad global applicability. This is a bivariate example, so something like Bivariate Mutual Information (Page 36) would do an excellent job of finding the correct predictors here. But much more complex relationship may be present in the data, with certain candidates being excellent in only certain areas that are defined by more than two variables. Hence the need for Local Feature Selection.

My own professional experience in financial market prediction provides a more concrete example of localized performance. I have found that in a high-trend, low-volatility environment, price deviation from a short-term linear projection can be an excellent predictor of the trend deteriorating, while this same indicator is worthless in high volatility or flat markets. Other, more complex interactions are possible. Local Feature Selection can be helpful in identifying regime-dependent predictive power, as well as finding predictors that may be weak globally but powerfully locally.

## What This Algorithm Computes and Reports

The exact operation of the Local Feature Selection algorithm is far beyond the scope of this user's manual; see either of the prior references cited for details. But here is a rough overview of its operation.

First and foremost, it finds an optimal subset of predictor candidates *separately for each individual case*. Most other screening algorithms find a subset that performs exceptionally well across the entire dataset. But this algorithm looks at a *single case* and finds the subset of candidates that optimally separates this case from cases in other classes. This optimal separation is defined by looking at two quantities: how far on average is this case from the other *nearby* cases in its class, and how far on average is this case from *nearby* cases in the other class(es). We find a set of predictors that maximizes the separation of this case from nearby cases in other classes while minimizing its average separation from other nearby members of its own class. Looking at just one case at a time, and only in neighborhoods near that case, provides the ultimate locality of feature selection.

This casewise performance evaluation means that we cannot simply report a single optimal parameter set, good for the entire dataset, and we certainly cannot report a sorted list of candidate sets. But what we can do is almost as good. We just count the number of cases for which each candidate appears in an optimal subset, which tells us how often each candidate was locally optimal (in conjunction with other candidates).

For example, we might see that X5, X7, and X8 are the optimal subset for the first case in the dataset. Perhaps X3, X7, and X9 are the optimal subset for the second case, Maybe X6 and X7 are optimal for the third case, and so forth. So the count so far has X7 in the lead, with it being in the optimal subset for the first three cases. We continue counting for the entire dataset, and then print a sorted list (most popular first) of candidates.



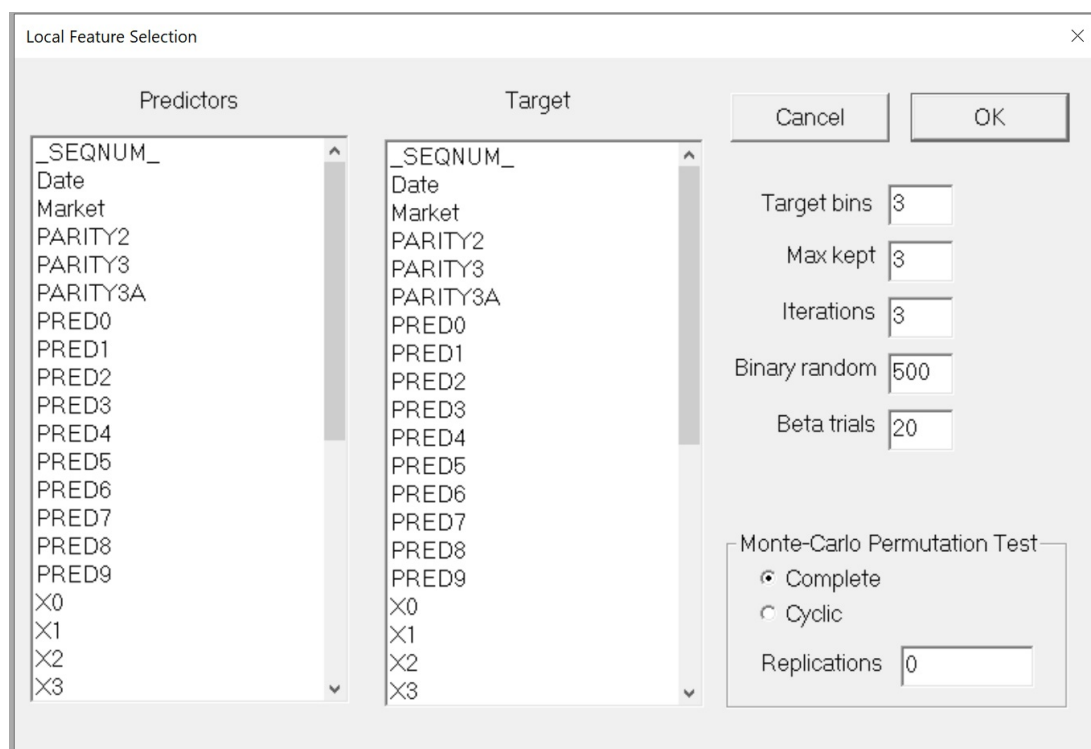
Keep in mind that appearing high on the popularity list does not mean that a candidate *used alone* is valuable; it may actually be worthless alone. As a matter of fact, in most practical problems it's best to assume that the most popular candidates are useful *only* in conjunction with other popular candidates, with its ideal partners varying across the problem domain. This is one of the strengths of this algorithm. Most common stepwise algorithms will totally disregard candidates that are worthless alone. But this algorithm tallies candidate performance in conjunction with other candidates, and even other candidates that may vary from case to case.

Superficially this may seem to be a serious problem in real-life applications. We would prefer to have a specific list of candidates which, taken as a group, are universally effective at classification. But universal applicability comes at a high price, a price that does not have to be paid when we focus on local effectiveness. So we do gain a lot with this algorithm.

And what about the cost, the notable lack of a specific broadly applicable subset of candidates? The good news is that most modern nonlinear classifiers are not hindered at all by the lack of broad-based power. They are able to sort out the complex relationships between predictors, with all their specialized local behavior patterns. All that powerful nonlinear classifiers really need is to not be overwhelmed by the presence of a huge number of predictors. So in this sense, the Local Feature Selection algorithm is a screener in the truest sense. Rather than tell the user exactly which set of candidates is ideal, it screens out the candidates with low popularity, leaving behind relatively few candidates for the classifier to digest however it chooses.

## Specifying the Test Parameters

When LFS is selected from the *Tests* menu, a dialog similar to that shown below appears, and the following items must be specified:



The dialog box titled "Local Feature Selection" contains two columns of variable lists, "Predictors" and "Target", each with a scrollable list of variables. To the right of these lists are input fields for "Target bins", "Max kept", "Iterations", "Binary random", and "Beta trials". At the bottom right is a section for the "Monte-Carlo Permutation Test" with radio buttons for "Complete" and "Cyclic", and a "Replications" input field. "Cancel" and "OK" buttons are at the top right.

Predictors	Target
_SEQNUM_	_SEQNUM_
Date	Date
Market	Market
PARITY2	PARITY2
PARITY3	PARITY3
PARITY3A	PARITY3A
PRED0	PRED0
PRED1	PRED1
PRED2	PRED2
PRED3	PRED3
PRED4	PRED4
PRED5	PRED5
PRED6	PRED6
PRED7	PRED7
PRED8	PRED8
PRED9	PRED9
X0	X0
X1	X1
X2	X2
X3	X3

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

Monte-Carlo Permutation Test  
☒ Complete  
☐ Cyclic  
 Replications: 0

The *leftmost 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. Continuous targets cannot be processed by the LFS algorithm, so the target is partitioned into bins which serve as class identifiers.

*Target bins* specifies the number of bins into which the target will be categorized. If *Target bins* equals the number of distinct values of the target variable, this many bins will be used. If *Target bins* is larger than the number of distinct values in the target variable, an error message will be generated and the log file will show the actual number of distinct values. If the target has few or no ties, and *Target bins* is much less than the number of distinct values, the dataset will be partitioned into this many bins, all of which have equal or very nearly equal size (depending on the number of ties).

*Max kept* is the maximum number of variables ever employed in a subset of candidates. In other words, this is the maximum number of variables *simultaneously* examined when computing the distances separating cases. 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, which should be perfect for most applications.

*Iterations* is the number of LFS algorithm iterations to execute in order 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 enough for all but the most critical applications. The default is 3. Even though more is better, I see no reason to ever set it to more than 4.

*Binary random* is the number of random trials employed to convert the floating-point usage flags to binary flags. (See my cited references for more details.) 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 to 1000 or more. 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 earlier. (See my cited references for more details.) More is better, but 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. This topic is also discussed on Page 22.

**Important note:** If you perform a Monte-Carlo permutation test, please see the discussion of solo and unbiased p-values that begins on Page 11 and continues onto the next page. 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.

***Runtime note:*** This predictor screening algorithm does have a potentially deal-killing problem. Its run time is proportional to the *cube* of the number of cases, so the run time blows up rapidly as the number of cases in the dataset increases. Having a CUDA-capable video card helps enormously, but that cube relationship eventually prevails. If your dataset is too large to be practical, you may need to randomly select a subset of it to test.

## A Contrived Example of Local Feature Selection

I created a dataset containing 1000 cases of 11 variables. X0 through X9 are uniformly distributed on  $[-1, 1]$ . PARITY2 is the parity of X3 and X4: PARITY2=0 if X3 and X4 are both positive or both non-positive. PARITY2=1 if X3 and X4 are on opposite sides of zero; one is positive and the other is non-positive. All other variables are ignored. As discussed earlier, this is a very difficult problem for many predictor screening algorithms. Here is the output of the LFS algorithm, with PARITY2 divided into two bins, and all other parameters at their default values. Observe how easily Local Feature Selection identified X3 and X4 as the relevant predictors, both in terms of percent of times selected as well as solo and unbiased p-values.

```
*****
*
* Computing Local Feature Selection for optimal predictor subset
* 10 predictor candidates
* 3 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
* 1000 replications of complete Monte-Carlo Permutation Test
*
*****
```

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

Variable	Pct	Solo pval	Unbiased pval
X3	91.70	0.0010	0.0010
X4	63.40	0.0010	0.0010
X6	4.50	0.8600	1.0000
X5	4.40	0.8850	1.0000
X7	3.80	0.9120	1.0000
X9	2.50	0.9610	1.0000
X8	1.80	0.9790	1.0000
X0	1.00	0.9980	1.0000
X1	0.50	1.0000	1.0000
X2	0.40	1.0000	1.0000

## A Practical Exploration of Local Feature Selection

One standard dataset often used for testing classification algorithms is the Fine Needle Aspirate Breast Cancer Wisconsin study, which characterizes the nucleus of cells in order to make a benign/malignant diagnosis. This dataset, along with a detailed description, is widely available online, including in the UCI Repository. It examines ten characteristics of the cell nucleus, and then given a sample it computes three properties of each characteristic: the mean, standard error, and worst value in the sample. Here are the results of running the LFS test on the 'mean' and 'worst' predictors in this dataset:

```
*****
*
* Computing Local Feature Selection for optimal predictor subset
*   20 predictor candidates
*   3 predictors at most will define a metric space
*   2 target bins
*   5 iterations of LFS algorithm
* 5000 random trials for real-to-binary f conversion
* 200 trial values for beta optimization
* 100 replications of complete Monte-Carlo Permutation Test
*
*****
```

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

Variable	Pct	Solo pval	Unbiased pval
M_CONCAVPTS	46.40	0.0100	0.1200
W_AREA	39.02	0.0700	0.3400
W_CONCAVPTS	30.23	0.0200	0.7400
W_PERIM	28.65	0.0100	0.8300
W_RAD	17.57	0.0200	1.0000
W_TEXTURE	7.73	0.6200	1.0000
M_CONCAVITY	3.51	0.4100	1.0000
W_COMPACT	3.34	0.5000	1.0000
M_TEXTURE	2.28	0.9900	1.0000
W_CONCAVITY	2.28	0.5600	1.0000
W_SYMMETRY	1.41	1.0000	1.0000
M_PERIM	1.23	0.4300	1.0000
W_SMOOTH	1.23	0.9700	1.0000
M_RAD	1.05	0.7400	1.0000
M_AREA	0.53	0.8700	1.0000
M_SMOOTH	0.53	1.0000	1.0000
W_FRACTAL	0.35	0.9900	1.0000
M_COMPACT	0.18	0.9700	1.0000
M_SYMMETRY	0.18	1.0000	1.0000
M_FRACTAL	0.00	1.0000	1.0000

Before discussing these results, let's run a univariate mutual information test on the same data:

```
*****
*
* Computing univariate mutual information (one predictor, one target)
* 20 predictor candidates
* 5 predictor bins
* 2 target bins
* 1000 replications of complete Monte-Carlo Permutation Test
* 0.1000 is user-specified alpha level
*
*****
```

-----> Mutual Information with DIAGNOSIS <-----

Variable	MI	Solo pval	Unbiased pval	P(<=median)
W_PERIM	0.4447	0.0010	0.0010	0.0000
M_CONCAVPTS	0.4246	0.0010	0.0010	0.0000
W_AREA	0.4220	0.0010	0.0010	0.0000
W_RAD	0.4192	0.0010	0.0010	0.0000
W_CONCAVPTS	0.4179	0.0010	0.0010	0.0000
M_PERIM	0.3681	0.0010	0.0010	0.0000
M_CONCAVITY	0.3489	0.0010	0.0010	0.0000
M_AREA	0.3481	0.0010	0.0010	0.0000
M_RAD	0.3474	0.0010	0.0010	0.0000
W_CONCAVITY	0.2947	0.0010	0.0010	0.0000
W_COMPACT	0.2172	0.0010	0.0010	1.0000
M_COMPACT	0.2073	0.0010	0.0010	1.0000
W_TEXTURE	0.1231	0.0010	0.0010	1.0000
M_TEXTURE	0.1212	0.0010	0.0010	1.0000
W_SMOOTH	0.1017	0.0010	0.0010	1.0000
W_SYMMETRY	0.0837	0.0010	0.0010	1.0000
M_SMOOTH	0.0687	0.0010	0.0010	1.0000
W_FRACTAL	0.0588	0.0010	0.0010	1.0000
M_SYMMETRY	0.0582	0.0010	0.0010	1.0000
M_FRACTAL	0.0212	0.0010	0.0010	1.0000

It's vital to understand that these two tests consider very different aspects of the relationship between predictors and the target. The univariate mutual information test examines each candidate individually, independent of the other candidates, while the local feature selection algorithm examines candidates in groups of three (as specified), making it sensitive to interactions between predictors. Also, the LFS test considers predictive power that may exist in only one or several subsets of the problem domain, while the mutual information test favors candidates that have global predictive power. Finally, and this is crucial to understanding performance probabilities, LFS results relate to frequency of selection in a competition for



selection, meaning that all results are relative to the performance of other competitors. If some competitor is selected often, then of necessity other competitors are selected less often. Such interaction between performance among competitors does not occur in the mutual information test. With these things in mind, here are some salient points for these tests:

- The apparently binary probabilities for the  $P(\leq \text{median})$  column in the univariate test results is due to the extremely consistent performance in and out of sample for all folds. These predictors behave almost identically across the entire dataset.
- As discussed on Page 23, unbiased p-values other than that for the best are upper bounds for tests whose competitor scores are independent. But due to the fact that scores for the LFS test are to a great degree relative, as mentioned in the third point above, *all* unbiased p-values for the LFS test are inflated, often to a huge degree. So small unbiased p-values are meaningful, but large values have little or no meaning, due to their being just upper bounds.
- The impact of this relative performance also shows up, though to a lesser degree, in the solo p-values for the LFS test. In the mutual information test, solo and even unbiased p-values are all hugely significant here, because every candidate has predictive power greater than a coin toss. But in the LFS test, where all performances are relative, a competitor must also stand out above the others in order to merit a tiny p-value.
- Nonetheless, the top five competitors in the LFS test, which are the only predictors having small solo p-values, are also the top five competitors in the mutual information test! Seeing this consistency tells me that these five candidates are the most promising predictors and merit further study. Not only do they have statistically significant mutual information with the diagnosis, but they most frequently are chosen in trios for case-by-case classification.

## Enhanced Stepwise Lin-Quad

In the section on Forward Selection Component Analysis, I discussed the pros and cons of traditional stepwise selection of variables in a general context. I'll repeat the key points of that discussion here as a prelude to the sophisticated stepwise selection procedure that is the topic of this section.

Suppose we have a relatively large set of variables from which we want to choose a smaller subset that is optimal in some sense. In the current section, optimality means minimizing prediction error when the subset of predictor candidates is used to predict a target variable based on a fairly simple nonlinear model. The 'ideal' approach to finding an optimal subset would be to test every possible subset and choose the best. But with even a moderate number of variables in the population, the combinatoric explosion would make testing every possible subset difficult, and with a large population of candidates, exhaustive testing becomes impossible.

So people traditionally take an approach to subset selection that is less than optimal but still usually reasonable, and a whole lot faster to compute. This more practical approach begins by finding the single variable (subset of size one) that is optimal. Then it finds a second variable, which *in conjunction with the variable already selected* maximizes the performance criterion. The key point is that the choice of this second variable is conditional on the first selected variable.

The intuitive appeal of these first two steps is obvious, but it is not perfect. For example, suppose we have three variables,  $X_1$ ,  $X_2$ , and  $X_3$ , and the pair  $X_2$ ,  $X_3$  is the best possible pair. It is not inconceivable that the best *single* variable is  $X_1$ , because perhaps the  $X_2$ ,  $X_3$  combination is reliant on *both* of them being known. In such a situation, which can be more common than we might realize, we will not find the truly best pair. We will be stuck with  $X_1$  being chosen first, plus either  $X_2$  or  $X_3$ , giving us a suboptimal pair. If we limit ourselves to two predictors, the  $X_2$ ,  $X_3$  pair will never be tested.

---

## Improving the Stepwise Process

*VarScreen's* Enhanced Stepwise Lin-Quad algorithm uses a predictor selection procedure that may not be as good as exhaustive testing of every possible subset, but that is often excellent at solving the problem just mentioned, without requiring an impractical amount of computer time. It works by keeping *several* of the best models at each step and testing new candidates with each. For example, we may specify that 3 models are kept. So the first step would be to find the best, second-best, and third-best individual candidates. Then when we go to the second step to append a second predictor, it actually tests each remaining candidate's performance in combination with each of these three original best selections. Of course this algorithm will fail if the best pair is the *fourth*-best with one of the remaining candidates, and we specified keeping only three. But this problem is not as bad as one might think, because this algorithm scales well. Its run time is roughly linear in the number of best models kept, so we can keep quite a few of the best at each step without suffering the combinatoric explosion inherent in exhaustive testing.

Just to be clear on the algorithm, suppose we have added a second predictor. To do so, we tested each of the specified number of best single variables in combination with each other remaining candidate, giving us a lot of models to compare. The specified best number of these models are retained for combining with each of the remaining candidates when we add a third predictor. So the number of best models retained at each step remains at the user-specified quantity; their number does not grow explosively. This is the secret to finding the sweet spot between exhaustive testing with impractical run time versus traditional one-at-a-time greedy selection with maximum computational speed. We get a lot more models tested but with controllable run time.

## Controlling Runaway Candidate Inclusion

A problem that has plagued stepwise predictor selection from its earliest days is that in nearly all applications, every time you add a new variable the model's performance improves. This is because the data is a mixture of authentic patterns that will continue when the model is put to use, and random noise that by definition will not continue. In general what happens is that the first few predictors that we select will focus on the authentic patterns, while additional predictors will let the model learn noise as if it were an authentic pattern. The more predictors we collect, the more accurately we model the noise, and hence the more a naive performance figure improves. If we model noise to any significant degree, the model will fail badly when it is put to use and the modeled noise does not repeat.

All kinds of techniques for controlling this problem have been developed. Some are primitive and only moderately successful. For example, limiting the sum of squared weights in a neural network may let us get away with having too many predictors without excessive modeling of noise. Other sophisticated techniques based on complexity theory are in use. My favorite method, which is used in *VarScreen*, tackles the problem directly: to evaluate the quality of a set of candidate predictors, we train the prediction model on some of the dataset cases and then test the model's performance on the remaining dataset cases. If the predictor set is doing a good job of finding authentic patterns while largely ignoring noise, the model will perform well on the data which did not go into training the model. On the other hand, if too many predictors are in use, thus learning noise as if it were authentic patterns, the model will perform poorly on the excluded data.

Actually, a single split like that just described wastes data. It is much better to use full cross validation: perform many different splits of the dataset. For example, we may split the dataset into four groups. Train the model on 1, 2, and 3, and test it on 4. Then train it on 1, 2, and 4, and test it on 3. And so forth. In this case we would do four groupings, called folds, each time leaving out one of the four groups for testing, while training with the other

three. There is an obvious tradeoff on choosing the number of folds. The more folds you employ, the larger will be the training set, hence providing a more representative model. But numerous folds will also increase computer run time. The general rule is to use as many folds as possible, consistent with having enough computer time available.

*VarScreen's* Enhanced Stepwise Lin-Quad algorithm uses cross validation with a user-specified number of folds to score each candidate predictor subset. The mean squared prediction error is cumulated across all cases in the test sets, and this is used to compute R-squared as the performance criterion. This lets us evaluate a predictor subset not on how well it performs in a training set, but on how well it performs when applied to cases that it did not see during training. It doesn't get any better than that.

## Ensuring Statistically Sound Selection

We saw in the prior section that *VarScreen* uses cross validation to select only candidates that improve out-of-sample performance. But improvement alone is not enough. Luck is always present in any test, even for out-of-sample results. What if we knew there was a significant probability that the model's apparently decent performance could actually have been achieved with nothing more than good luck? Or suppose we learned that there is a significant probability that the performance improvement obtained by including a new candidate could have been nothing more than good luck?

In order to address the issue of luck in the model-building process, every time a new variable is included, two unbiased probabilities (p-values) are printed. Both have a null hypothesis that all variables are worthless. The first is the probability that the model, taken as a whole, could have performed at least as well as it did. The second is that adding the most recent variable could have provided as much improvement as that obtained. If the former is not tiny we should reject the entire model. If the latter is not tiny we should freeze the model before adding the most recent variable.

## The Feature Evaluation Model

Until now I have been blithely talking about effective selection of predictors for a predictive model, but I've said nothing about the model. Naturally we have an enormous number of models to consider, everything from simple linear regression to large and complex neural networks. My choice for the *VarScreen* stepwise selection test lies nicely in the middle ground: it has enough nonlinear capability to handle most common situations: it can employ complete reversal of the first derivative of the target with respect to every predictor, and it also encompasses interactions of every predictor with every other predictor. On the other hand, it does not have the sort of massive nonlinearity that can engender serious overfitting. Last but not least, it does not require iterative training; it has a unique and directly computable solution that minimizes mean squared error (maximizes R-Squared).

The model I'm talking about is commonly called a *linear-quadratic* model. At its heart it is ordinary linear regression. However, the inputs to this linear regression include not only the predictors, but also the squares of the predictors and all possible pairwise products. For example, suppose we have three predictors,  $X_1$ ,  $X_2$ , and  $X_3$ . Then we create a linear regression model with inputs  $X_1$ ,  $X_2$ ,  $X_3$ ,  $X_1^2$ ,  $X_2^2$ ,  $X_3^2$ ,  $X_1 \bullet X_2$ ,  $X_1 \bullet X_3$ , and  $X_2 \bullet X_3$ . In order to ensure accurate floating-point calculations as well as easy interpretability, all predictors are standardized to zero mean and unit variance before the transforms are done. The target variable is standardized as well. The printed model coefficients refer to the standardized predictors and target.

A careful user will make sure that the dataset contains no collinearity. In other words, we should not have any predictor be an exact linear combination of other predictors. But in case this happens, the program will handle the situation well by employing singular value decomposition rather than simple matrix inversion.

## Specifying the Test Parameters

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

Enhanced stepwise selection for linear-quadratic model

Predictors	Target
_SEQNUM_	_SEQNUM_
DEP_RANDOM0	DEP_RANDOM0
DEP_RANDOM1	DEP_RANDOM1
DEP_RANDOM2	DEP_RANDOM2
DEP_RANDOM3	DEP_RANDOM3
DEP_RANDOM4	DEP_RANDOM4
DEP_RANDOM5	DEP_RANDOM5
DEP_RANDOM6	DEP_RANDOM6
DEP_RANDOM7	DEP_RANDOM7
DEP_RANDOM8	DEP_RANDOM8
DEP_RANDOM9	DEP_RANDOM9
RAND0	RAND0
RAND1	RAND1
RAND2	RAND2
RAND3	RAND3
RAND4	RAND4
RAND5	RAND5
RAND6	RAND6
RAND7	RAND7
RAND8	RAND8

Number retained: 2

Number of folds: 4

Min predictors: 1

Max predictors: 0

Monte-Carlo Permutation Test

☒ Complete

☐ Cyclic

Replications: 100

Cancel OK

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. Usually this should be set to 1, the default.

*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 starting on Page 22.



## A Contrived Example of Enhanced Stepwise Selection

Once again, this demonstration makes use of the 11 synthetic variable that I used for many other examples:

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

$$SUM1234 = RAND1 + RAND2 + RAND3 + RAND4$$

To run the test, I specified a huge number of models to be retained, which forces it to try every possible combination of predictors., and I used 10-fold cross validation, which is slow but more accurate than the default of 4 folds. The minimum number of predictors was the default of 1, which lets it begin testing for performance degradation immediately. I specified the maximum number of predictors to be the default of 0, which causes it to set the actual maximum to the number of candidates, allowing it to use every candidate if performance keeps improving. Finally, I requested 1,000 trials of the Monte-Carlo permutation test with complete permutation. Here are the results of this test:

```
*****
*
* Computing enhanced stepwise linear-quadratic model
*
*      SUM1234 is the target
*      10 predictor candidates
*      9999 candidates retained for each iteration
*      10 folds for cross validation performance
*      1 minimum predictors in final model
*      10 maximum predictors in final model
*      1000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Stepwise inclusion of variables...

R-square	MOD	pval	CHG	pval	Predictors...
0.2613	0.001	0.001	0.001		RAND3
0.5128	0.001	0.001	0.001		RAND3 RAND4
0.7486	0.001	0.001	0.001		RAND1 RAND3 RAND4
1.0000	0.001	0.001	0.001		RAND1 RAND2 RAND3 RAND4

STEPWISE terminated early because adding a new variable caused performance degradation

Final XVAL criterion = 1.00000  
In-sample mean squared error = 0.00000

Regression coefficients for standardized data:

0.504322	RAND1
0.501287	RAND2
0.505190	RAND3
0.503176	RAND4
-0.000000	RAND1 Squared
-0.000000	RAND2 Squared
0.000000	RAND3 Squared
0.000000	RAND4 Squared
-0.000000	RAND1 * RAND2
-0.000000	RAND1 * RAND3
-0.000000	RAND1 * RAND4
-0.000000	RAND2 * RAND3
-0.000000	RAND2 * RAND4
0.000000	RAND3 * RAND4
-0.000000	CONSTANT

Note the following observations:

- The R-square performance criterion increases by about 0.25 each time one of the four true predictors is included, reaching its maximum possible value of 1.0 when all four are included. The p-value for adding each of them is the minimum possible,  $1/1000 = 0.001$ .
- Even one predictor is enough to provide a model p-value of the minimum possible,  $1/1000 = 0.001$ .
- As soon as we have the four correct predictors in the subset, stepwise selection ends because adding a new variable reduces the out-of-sample performance criterion.
- The coefficients of the four correct predictors are approximately equal, and all other coefficients are zero.

## A Practical Example of Enhanced Stepwise Selection

I created a database of indicators and a target from the S&P100 index day bars, OEX. For all indicators, the number of bars to look back terminates the indicator name. These indicators and target are:

<b>CMMA_N</b>	Today's close minus moving average (N=5, 10, 20). This measures departure from recent price history.
<b>LIN_ATR_N</b>	Slope of straight line fit to log prices, normalized by ATR (N=5, 7, 15). This is a trend indicator.
<b>RSI_N</b>	Ordinary RSI (N=5, 10, 20). This is a trend indicator.
<b>DAY_RETURN</b>	Log ratio of open-to-open returns one day ahead.

Recall that all indicators, as well as the target, are automatically normalized to zero mean and unit variance. Not only does this improve numerical stability, but it makes computed coefficients easy to interpret, because they are based on commensurate values.

I used all nine indicators as predictor candidates, 10-fold cross validation, and the 5 best models retained. The defaults of 1 for the minimum number of predictors and 0 for the maximum mean that testing for stopping commences immediately, and all candidates may be included if they deserve inclusion. I specified 100 trials of the complete Monte-Carlo permutation test. The following results were obtained:

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

Stepwise inclusion of variables...

R-square	MOD	pval	CHG	pval	Predictors...
0.0057	0.010	0.010			CMMA_10
0.0058	0.010	0.210			CMMA_20 LIN_ATR_15
0.0075	0.010	0.010			CMMA_20 LIN_ATR_15 RSI_5

STEPWISE terminated early because adding a new variable caused performance degradation

Final XVAL criterion = 0.00748

In-sample mean squared error = 0.98943

Regression coefficients for standardized data:

-0.013215	CMMA_20
0.017302	LIN_ATR_15
-0.018098	RSI_5
0.451606	CMMA_20 Squared
0.079064	LIN_ATR_15 Squared
0.159801	RSI_5 Squared
-0.409961	CMMA_20 * LIN_ATR_15
-0.522759	CMMA_20 * RSI_5
0.253404	LIN_ATR_15 * RSI_5
-0.038181	CONSTANT

We see the model-retention enhancement at work. The first variable selected is CMMA\_10, but when the second variable, LIN\_ATR\_15, is added, CMMA\_10 is replaced with CMMA\_20. This could not happen with traditional forward selection; once CMMA\_10 entered in the mix, it would be there forever, even if other competitors perform better in the presence of new predictors.

The p-values for inclusion give us a slightly difficult decision to make. The first candidate has the smallest possible p-value,  $1/100=0.01$ , so no issue there. But the second candidate has a p-value of 0.21, which is not nearly as small as we would normally like. We also note that the criterion barely increases, from 0.0057 to 0.0058. So it's tempting to conclude that only one candidate makes the cut. But despite this relatively poor showing, adding this second candidate nevertheless improved the out-of-sample performance, so we decide to keep going. That's a good decision, because adding a third candidate improves the performance a lot, and the improvement has a p-value at the minimum of 0.01. A fourth inclusion degrades performance, so we stop with three predictors.

It is often difficult or impossible to confidently understand how the predictors are used by the model to make predictions; the presence of squared and interactive terms makes things a lot more complex than having only raw predictors present. But at least interpreting the model coefficients is helped by the fact that they are relative to the variables after centering at zero and scaling to unit variance. This makes the model inputs commensurate, hence the relative sizes of the coefficients are meaningful. So let's go out on a limb and try to make some sense out of this model.

LIN\_ATR and RSI are trend indicators, while CMMA measures departure from recent prices, a sort-of breakout indicator. Let's look first at two of the three largest-absolute-coefficient model inputs:  $\text{CMMA}_{20} \bullet \text{LIN\_ATR}_{15}$  and  $\text{CMMA}_{20} \bullet \text{RSI}_5$ ; impact of the third,  $\text{CMMA}_{20}$  squared, will be discussed later. Their coefficients are large negative values. Consider three possibilities for current market trend, as indicated by  $\text{LIN\_ATR}_{15}$  and  $\text{RSI}_5$ :

- In a flat market (both trend indicators near zero),  $\text{CMMA}_{20}$  has little predictive impact from these two terms.
- In a strong upward trend (both trend indicators very positive), mean reversion is in control: sudden upward departure from recent history presages a downward price move, and sudden downward departure from recent history presages an upward price move.
- In a strong downward trend (both trend indicators very negative), breakout is in control: sudden upward departure from recent history presages a continued upward price move, and sudden downward departure from recent history presages a continued downward price move.

We cannot ignore the second-largest coefficient term,  $\text{CMMA}_{20}$  squared. If the absolute value of  $\text{CMMA}_{20}$  is not large (the market has not made a sudden sharp move), the above rules are mostly in control. But if the market has just made a large move away from its recent values, up or down, an upward move is in the cards.

## RANSAC for Devaluing Noisy Cases

A common problem in modeling is the presence of excessive noise in the observations used for training or evaluating the model. Numerous specialized models and training algorithms have been proposed for dealing with this situation. Most of these focus on producing a model or training algorithm that is itself inherently robust against noisy cases.

A very different technique was first proposed in the 1981 paper “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography” by Martin Fischler and Robert Bolles. This algorithm, dubbed RANSAC, was a highly specialized technique that, while applicable to any model and training algorithm, assumed that some of the observations were pure while others were mostly or entirely noise. There was no middle ground. This was the situation for object identification in images, where part of an image was a clean depiction of the object being sought, while the rest of the image was irrelevant to the task. This breakthrough concept served as a jumping-off point for numerous variations and improvements. For a good overview of many such advancements see the 2013 paper “USAC: A Universal Framework for Random Sample Consensus” by Rahul Raguram and Ondrej Chum.

Because of my longstanding interest in prediction of financial markets, I devised a variation of RANSAC that, while also usable with any predictive model and training algorithm, assumes a continuum of noise based on probabilities: an iterative algorithm assigns to each case in the set of observations a relative probability of being excessively dominated by noise. As iteration progresses, likely noisy cases are given less and less influence on the model, while cases that have low probability of being excessively noisy are given more influence on the model. After convergence is obtained, those observations judged to have high probability of being excessively noisy are removed from consideration, and the final model is based on the remaining observations. A useful byproduct of this algorithm is a new variable: the

---

probability (expressed in percent) for each case that this is a 'reliable' case, one that is not excessively contaminated by random noise.

This technique is used for predictor screening by computing a performance criterion for each model and ranking them in order of decreasing performance. *VarScreen* gives the user the choice of two performance figures: RMS prediction error, which is useful for general prediction applications, and optimal profit factor (Page 50), which is useful for market prediction applications.

How do we use this model-based approach for indicator screening? *VarScreen* employs the linear-quadratic model described on Page 156. This model is one of my favorites because it combines significant but not excessive nonlinearity with rapid deterministic (not iterative) training. The user specifies the maximum number of simultaneous indicators that can be used by the model, from 1 to 3. Every possible such model (all combinations of predictor candidates) is evaluated with the RANSAC algorithm, and their performance criteria are presented to the user in descending order. These are in-sample estimates, so they are optimistically biased. However, the user can specify that a Monte-Carlo permutation test be used to compute p-values for each model, both solo and unbiased. This test is the new stepwise algorithm described on Page 13.

For example, if the user wants to test individual predictor candidates, he or she would specify that just 1 predictor be used by the model. For more generality, 2 predictors could be used, in which case every possible pair of candidates would be tested, in addition to individual candidates. Testing trios is also possible, although if there are many candidates the total number of models can blow up quickly and require a lot of computer time.

## The Algorithm

I have available for free download on my website (on the *VarScreen* page) complete source code for a stand-alone Windows console implementation of the RANSAC algorithm. The code is highly commented, which will clarify details of the algorithm, so I will provide just a brief overview here. Suppose there are  $N$  cases, the user has specified that the bootstrap sample size is  $M$  (typically about  $N/4$ ), and the specified noise fraction is *noise\_frac* (typically fairly small).

- 1) Initialize the case probability vector to equal values:  $1 / N$ . For each case, this is the probability that it is clean, not excessively noisy.
- 2) Select without replacement a random sample of  $M$  cases from the complete set of  $N$  cases. The probability of selecting each case is defined by that case's value in the case probability vector. This is easily done as follows:
  - A) Form the cumulative probability vector of the case probability vector, with the first value being zero, the next value being the first case probability, the next value being the sum of the first two case probabilities, and so forth. The last value in this cumulative probability vector will be less than 1. In fact, 1 minus the last cumulative value will be the last case probability value.
  - B) Generate a random number from 0 to 1. Use binary search to locate the position of this value in the cumulative probability vector, the greatest value in the vector less than or equal to the random number. This defines the case selected.
  - C) If this case has already been selected, try again. Repeat step B until we have  $M$  cases.
- 3) Train the model using these  $M$  cases. Compute predicted values for all  $N$  cases.



- 4) Compute the vector of absolute errors, the absolute value of the true minus predicted values. Compute the  $1 - \text{noise\_frac}$  fractile of these absolute errors. This can be thought of as the greatest error among the 'clean' cases, and it serves as the convergence criterion.
- 5) If this convergence criterion is less than the best criterion seen so far, reset the failure count to 0; if it failed to improve, increment the failure count. If the failure count reaches the user-specified termination count, quit because we have converged. Otherwise...
- 6) Adjust the case probability vector: for each case in the  $\text{noise\_frac}$  fraction of cases having the greatest absolute error, reduce its probability of being clean by 5 percent. (This 5 percent quantity is my own arbitrary choice; feel free to experiment). Rescale appropriately.
- 7) Go to Step 2.

After convergence, remove the  $\text{noise\_frac}$  fraction of cases having lowest cleanliness probabilities (we remove those most likely to be very noisy) and train the final model using the remaining cases, those that are most likely to be clean.

When we compute the final performance criterion for this model which has been trained on only 'clean' data, slightly different methods are used, depending on the user's optimization criterion. If the user wants to optimize RMS error, the criterion is based on only clean cases. This prevents outliers from unduly contaminating results. But if the user wants to optimize profit factor, noise will usually be somewhat less of a problem (errors are not squared!), so I take the more fair approach of computing the profit factor using all cases, noisy and clean.

Please remember that this criterion is in-sample, so it will be optimistically biased.

## 168 RANSAC for Devaluing Noisy Cases

### Running the Test

When the user clicks Create/RANSAC, the dialog shown below appears. Note that RANSAC appears under the Create menu rather than the Test menu. This is because it creates a new variable, the probability of each case being clean. Of course, the associated test results are printed in the VARSCREEN.LOG file. Brief discussions of the user-specified parameters follow.

The dialog box is titled "RANSAC screening and noise computation" and has a close button (X) in the top right corner. It contains two lists of variables, "Predictors" and "Target", each with a scroll bar. To the right of these lists are several input fields and a section for the Monte-Carlo Permutation Test.

Predictors	Target
_SEQNUM_	_SEQNUM_
ADX15	ADX15
ADX15_A15	ADX15_A15
ADX15_A30	ADX15_A30
ADX15_D15	ADX15_D15
ADX15_D30	ADX15_D30
ADX15_MAX60	ADX15_MAX60
ADX15_MIN60	ADX15_MIN60
ADX7	ADX7
ADX7_D15	ADX7_D15
ADX7_D7	ADX7_D7
ADX7_MAX30	ADX7_MAX30
ADX7_MIN30	ADX7_MIN30
ATRRAT_10	ATRRAT_10
ATRRAT_20	ATRRAT_20
ATRRAT_5	ATRRAT_5
BOLL_10	BOLL_10
BOLL_20	BOLL_20
CKURT_10	CKURT_10
CKURT_10_4	CKURT_10_4
CKURT_20	CKURT_20
CKURT_20_4	CKURT_20_4
CMMA_10	CMMA_10
CMMA_10N	CMMA_10N
CMMA_20	CMMA_20

Buttons: Cancel, OK

Min fraction kept (0-.5)  
or -1 for RMS error: 0.1

Familywise alpha: 0.1

Noise fraction: 0.05

Max predictors (1-3): 3

Sample size: 532

Termination: 20

Monte-Carlo Permutation Test

- ☒ Complete
- ☐ Cyclic

Replications: 0

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.

*Min fraction kept* would be set to a value between 0 and 0.5 to use the optimal profit factor criterion. This is the minimum fraction of cases on each side (long and short) that must be ‘traded’ in computing the optimal long and short thresholds. See the discussion of optimal profit factor on Page 50 for details. Alternatively, this parameter can be set to any negative number in order to specify that the RMS prediction error is to be used as the performance criterion. The default is 0.1, which means that the greater of the long and short profit factor is the performance criterion, and at least 10 percent of the cases must trigger a long position and at least 10 percent of the cases must trigger a short position .

*Familywise alpha* is the maximum tolerable probability of erroneously rejecting one or more null hypotheses of worthlessness. The discussion on Page 13 provides more details on this value. The default is 0.1, but a smaller value such as 0.05 or even 0.01 will promote more certainty in concluding that any models are effective.

*Noise fraction* is the maximum fraction of observations that the user considers might be dominated by noise. Results are not terribly sensitive to this value as long as it’s reasonably small. A value of zero just fits the model to the entire dataset, which is rather pointless. The default of 0.05 means that we believe that at most 5 percent of the cases are predominantly noise.

## 170 RANSAC for Devaluing Noisy Cases

---

*Max predictors* is the maximum number of predictor candidates that can be considered simultaneously by the model. This has a huge impact on run time. It may not exceed 3, and even 3 may cause excessive run time.

*Sample size* is the number of observations sampled (without replacement) from the complete set of observations during each iteration of the algorithm. It defaults to 1/4 of the cases, which is usually reasonable. I do not recommend using less than about 1/10 or more than half of the cases.

*Termination* determines convergence. Iteration ceases when this many trials in a row fail to provide improvement in the convergence criterion. Larger values give better, more reliable results but require more computer time. Values above 50 or so are legal but usually a waste of time, while values under 10 or so are often inadequate. The default of 20 is usually enough, though if you want more reliability and computer time is not a limiting factor, feel free to increase it.

Permutation test parameters are described in general terms starting on Page 11. Cyclic permutation, which must be used if there is serial correlation in the target, is discussed on Page 22. The stepwise permutation algorithm for controlling familywise error, discussed on Page 13, is always performed if more than 1 replications are specified; the user does not have the option of doing it or not.

This operation creates a new variable called RANSAC\_1 the first time it is invoked. If it is done again, the new variable will be called RANSAC\_2, and so forth. This is the percent (0-100) probability that the case is 'good' (not excessively contaminated with noise).

## An Example From Market Prediction

I generated the same indicators as those used for the Enhanced Stepwise demonstration, and used them to predict next-day log returns of OEX, the S&P 100 index. Here are the indicators and log file output, and a discussion follows.

**CMMA\_N** Today's close minus moving average (N=5, 10, 20).  
This measures departure from recent price history.

**LIN\_ATR\_N** Slope of straight line fit to log prices, normalized by  
ATR (N=5, 7, 15). This is a trend indicator.

**RSI\_N** Ordinary RSI (N=5, 10, 20). This is a trend indicator.

**DAY\_RETURN** Log ratio of open-to-open returns one day ahead.

```
*****
*
* Computing RANSAC screen for indicator candidates
*   DAY_RETURN is target
*   9 predictor candidates
*   0.1000 is minimum fraction kept
*   0.0500 is noise fraction
*   0.2000 is alpha level for familywise error
*   3 is maximum number of simultaneous predictors
*   1942 is sample size
*   20 is termination count (contiguous failures to improve)
*   1000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Criteria for all competitors

Variable(s)	Criterion	Solo pval	Unbiased pval
LIN_ATR_15	1.7370	0.0010	0.0340
LIN_ATR_5			
RSI_5			
LIN_ATR_15	1.6959	0.0010	0.0710
RSI_5			
LIN_ATR_15	1.6558	0.0020	0.1450
RSI_10			
RSI_5			
RSI_10	1.6460	0.0030	0.1730
RSI_5			

Best remaining p-value=0.2160 while alpha=0.2000, so quitting

## 172 RANSAC for Devaluing Noisy Cases

---

```
Criterion for best model = 1.73703
  0.000432  LIN_ATR_15
 -0.003567  LIN_ATR_5
 -0.004597  RSI_5
  0.000030  LIN_ATR_15 Squared
 -0.000006  LIN_ATR_5 Squared
  0.000016  RSI_5 Squared
 -0.000032  LIN_ATR_15 * LIN_ATR_5
  0.000004  LIN_ATR_15 * RSI_5
  0.000081  LIN_ATR_5 * RSI_5
  0.222571  CONSTANT
```

The output shows that I specified that at most 5 percent (0.05) of the cases are to be considered potentially contaminated with excessive noise. For financial markets this typically is unexpected news reports, unjustifiably cascading panic or enthusiasm, or some other unforeseen event that triggers unusually strong buying or selling and whose appearance was not foreseeable from the recent price history and hence not reflected in indicators.

I specified a maximum tolerable familywise error of 0.2, which in reality is much too high for comfort. But I did so to make sure that at least a few models were printed for this demonstration. Results are not spectacular, though they are pretty decent. The best model had an unbiased p-value of 0.034. Normally I like to see smaller p-values. But this does mean that even after accounting for selection bias, the optimistic bias inherent in examining multiple competing models, there is only a little over a 3 percent chance that if all competing models were completely worthless, the best among them would have done at least as well as our best model here.

As always, please note that these models all had tiny *solo* p-values. This is a clear demonstration of the importance of paying attention to only the unbiased p-value, not the solo p-value. When a lot of models are tried, some are virtually guaranteed to be unjustifiably lucky!

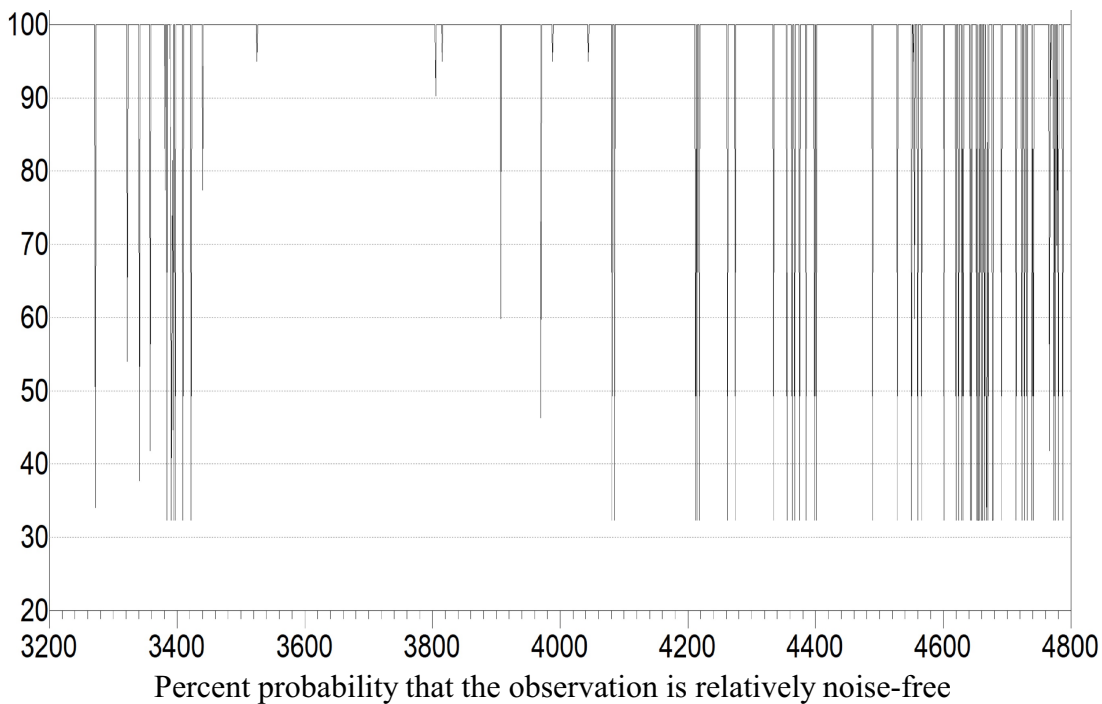
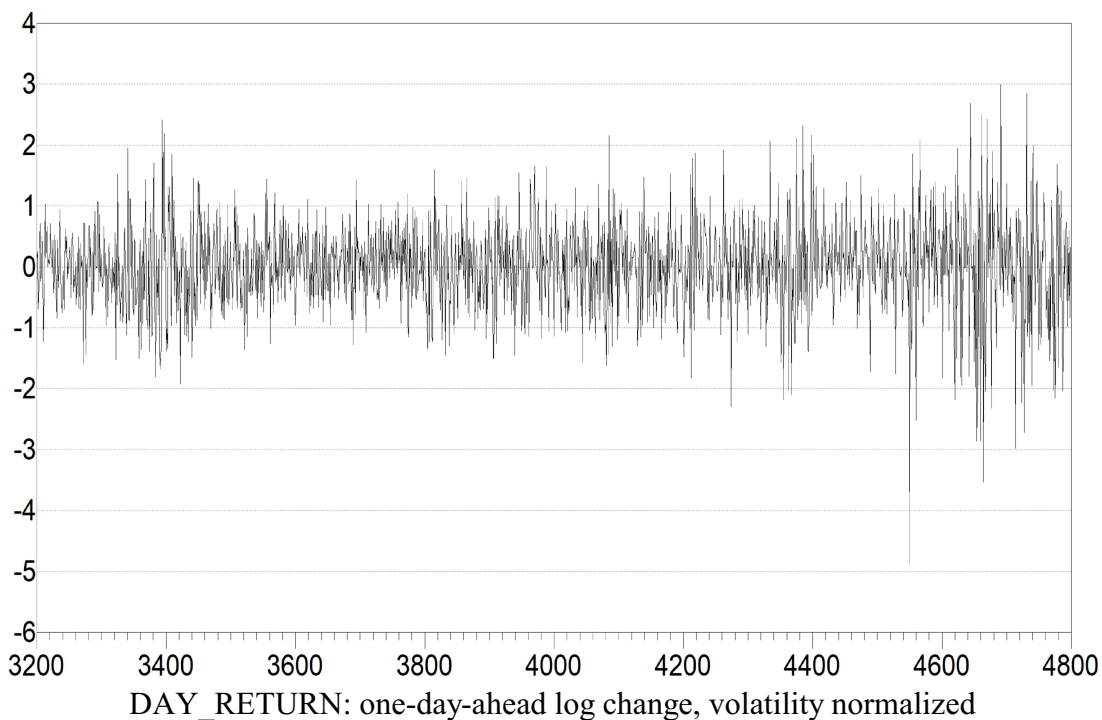
Also remember that these are in-sample performance figures. So that profit factor of 1.737 for the best model is optimistically biased, not a true estimate of future performance. Nonetheless, the smallish unbiased p-value indicates that the model probably was able to find a predictable pattern, since scrambling the returns damaged performance most of the time.

Last of all we see the coefficients of the best model. Recall that the Enhanced Stepwise test normalizes all predictor candidates and the target to have zero mean and unit variance, so that coefficients are commensurate and hence easily interpretable. No such normalizing is done for RANSAC; values as supplied are used. Thus, the coefficients here are all tiny because the predicted quantity, DAY\_RETURN, is the fractional one-day-ahead log market return, normalized for recent volatility. This is a small quantity. If I were really interested in the model coefficients I would have normalized all variables here. But RANSAC is usually just a screening run, a test to find the best predictor subsets when the impact of noise is reduced. I am also interested in the obtained profit factor, because I can think of it as an approximate upper bound on what would be obtained in real life; recall that this is an in-sample value, so it has considerable optimistic bias and hence must be considered only an upper bound.

On the next page the top plot shows a section of the DAY\_RETURN series, and the bottom plot is RANSAC\_1, the percent probability that each case is 'good' in the sense of not being excessively contaminated by noise. It is easy to see the correspondence between time periods of unusual volatility and low probability of observations being reliable. It is also interesting to see how such periods tend to clump together.

## 174 RANSAC for Devaluing Noisy Cases

---





## Nominal-to-Ordinal Conversion

Measured variables can be quantified in terms of the amount or nature of information they can inherently communicate. Four distinct levels of measurement were rigorously defined by the psychologist Stanley Smith Stevens in 1946 and continue to be useful in many areas of data analysis. Here's a definition of each, in order from least potential numeric information to most:

A *nominal* variable classifies data into distinct categories that are mutually exclusive and exhaustive. Although numbers may be used to express the values of a nominal variable, neither numbers nor any other labeling express any inherent order. Except for special circumstances such as tree-based models, it makes no sense to use numbers to specify categories and input these numbers to a model. For example, you may use the numbers 1 through 12 to express the month January through December, but it would be a rare model indeed that could make sense of this number. If you have a binary variable such as gender, you probably could get away with coding one gender as +1 and the other as -1, and using this number as an input to the model. But if there are more than two categories you almost certainly cannot use numerical class codes as inputs.

There are two common methods for using nominal variables as inputs to a model, and *VarScreen* provides a third. By far the most common is to recode the nominal value as a set of binary variables, one for each category. Set the variable that corresponds to the case's category to one, and set all others to zero. The obvious problem with this method is that if there are numerous categories, this will introduce a lot of new inputs.

The other method is useful if the categories have a special type of ordering and are cyclic. The perfect example of such a variable is the month of the year. If they are numbered in consecutive order, we can say, for example, that months 6 and 7 are, in a sense such as seasonal weather, closer than 6 and 11. The problem is that months 12 and 1 are seasonally close, yet their

numeric distance is huge. A good way to handle such a variable is to map it onto a circle and create two new variables, the sine and cosine of the circular angle. For example, we might say January is 0 degrees, February is  $360/12=30$  degrees, March is 60 degrees, and so forth. Then the sine and cosine of the angle preserve the property of months being seasonally close or far apart, even in the presence of wrap-around from December to January. And you can do this with just two new variables. Always consider this transformation if you are dealing with a suitable nominal variable.

An *ordinal* variable is one which allows the experimenter to rank values, thus allowing statements of *greater than* or *less than*, but which does not assume that the differences between ranks are uniform or quantifiable. For example, a stock rating agency may rank a stock into five categories of one star through five stars. You can then say, for example, that an agency rating of 4 stars is greater than a rating of 3 stars. But you cannot say that the quantitative difference between having 4 stars versus 3 is greater than or less than the difference between having 5 stars versus 4.

An *interval* variable has the same rank ordering properties (greater than and less than are valid), but adds one more property: differences between values are meaningful. For example, temperature in Fahrenheit or Centigrade is interval, because, for example, you can say that the difference between 50 and 60 degrees is twice the difference between 20 and 25 degrees. However, an interval variable does not have a true zero. So, for example, you cannot meaningfully say that a temperature of 60 degrees is twice a temperature of 30 degrees.

Finally, a *ratio* variable is an interval variable that has the additional property of having a true zero, which means that a value of zero indicates the complete absence of the quantity. Obviously, a temperature of zero in Fahrenheit or Centigrade does not imply a complete lack of whatever temperature is. On the other hand, temperature measured in absolute scale, in which zero implies no molecular motion, is a ratio variable.

---

## Measurement Level and Modeling

Most developers give minimal thought to the measurement level of the variables that they use as model inputs. However, there are some aspects of this issue that merit discussion.

The vast majority of applications use either interval or ratio variables as model inputs. In those exceedingly rare models in which ratios of inputs play a direct role, these variables must be ratio level, because as discussed earlier in regard to temperature, ratios of interval variables almost never make sense. But because predictive models/classifiers almost never compute ratios of inputs, the difference between interval and ratio variables is generally insignificant.

At the other extreme of measurement level, nominal variables are nearly always worthless as direct inputs to a model. (The most notable exception is tree-based models.) For example, suppose we ask a person's religion, where the choices are Christianity, Islam, Hinduism, Buddhism, Judaism, and several others. Coding them as 1, 2, 3, and so forth and using this number as a model input would, in nearly all applications, invite disaster. As discussed at the beginning of this chapter, the most common work-around would be to create a new input variable for each religion, and for each case set the corresponding input to 1.0 and all others to 0.0. This would likely produce a problematic number of new inputs.

Finally, consider ordinal variables, which supply only order relationships. Superficially, one may believe that they convey significantly less information than interval or ratio variables. However, this is not nearly as true as one might think. If one is comparing two sample means of normally distributed variables, the t-test is best. But the Mann-Whitney U-test, which replaces the original data with ranks, is asymptotically 95.5 percent as efficient as the t-test. Almost no useful information is lost, and ranking eliminates heavy tails, meaning that the U-test is superior to the t-test in many cases. Thus, *you should never disparage ordinal data in modeling.*

## Basic Elevation from Nominal to Ordinal

The essential idea of this chapter is that as part of the model training process we will find a mapping from the categories of the nominal variable to integers having ordinal level, and this mapping will satisfy some measure of optimality. For example, if the nominal variable is color, we might find that an optimal mapping is Red→4, Blue→2, Green→5, and so forth. Then, when we train or test the model, we will convert the nominal values to the ordinal numbers learned during the training period, and use these numbers as a hopefully meaningful input to the model being trained or tested.

The algorithm that learns an optimal mapping requires that we have available during the training period a variable of at least ordinal level and that is as closely related to the target as possible. In many cases we may use the target variable itself, but this is not necessary. Because we do not need to have this variable available when the model is put to use, some interesting approaches may be possible. For example, the variable may be too expensive to collect on an ongoing basis, or it may be an unknowable future value of some quantity related to the target. If we believe that it may be related to *both* the target and the nominal variable being converted to ordinal, we can compute a mapping from the nominal variable that's known at all times to this auxiliary variable that's known only at training time, and thereby be able to use the nominal variable as a proxy for the unobservable auxiliary variable.

From this point on, for convenience I will call this auxiliary variable the target variable, because this is the most common approach. But keep in mind that it need not be so. A commonly used method for finding a mapping from a nominal variable to the target variable is to compute the mean of the target separately for each category of the nominal variable, and define the mapping as these means: the mapped value for the  $i$ 'th category would be the mean of the target across all training cases whose value of the nominal variable is the  $i$ 'th category.

---

This method works well if the data is very clean (low noise) and does not suffer from a heavy tail or tails. But the mean of a poorly behaved variable is subject to serious behavior issues itself; a single wild outlier can fatally skew the mean and produce a nonsensical mapping. We could use the median instead of the mean and solve the heavy tail problem, but there is a better way that preserves a bit more useful information.

*Varscreen* computes the percentile of the target for every case, with all categories pooled. In other words, for each case the target percentile is the percent of training set cases whose target value is less than or equal to the target value of the case under consideration. Thus, we see that the case having the minimum target value will have a percentile near zero, and the case having the maximum target value will have a percentile of 100. Cases having tied target values will have tied percentiles. This effectively converts the target values to ordinal level, eliminating heavy tails (the percentile targets have a uniform distribution) but preserving order properties. Then the mapping for each category is the mean of the percentiles, giving us an ordinal level variable that captures nearly all of the information present in the target, while avoiding heavy tails and minimizing the impact of noise.

## Enhancing the Basic Algorithm With Gating

When I was actively developing automated market trading systems, I learned early on that a good system is rarely, if ever, one-size-fits-all. Markets go through what may be called *regimes*, in which it exhibits certain measurable properties based on recent historical activity and which usually change relatively slowly. For example, a market may be in a bull regime or a bear regime, or it may be in a period of high or low volatility. It often happens that the relationship between indicators and the target depends on the regime. This problem can be solved by creating a separate training set for each regime, or by putting in the training set only those cases that you believe are in a regime for which a good nominal-to-ordinal mapping exists.

Creating separate training sets is a nuisance, though. It would be easier if the mapping algorithm were able to directly handle separate regimes. As an additional benefit to this approach, by letting the algorithm internally handle regime issues, we can perform certain statistical tests regarding comparative properties of the regimes.

The twin issues of computing different mappings for different regimes, and/or ignoring certain regimes are handled by employing a single optional variable called a *gate* variable. This allows for only two different regime-defined mappings, but that should be enough to cover most common applications. The effect of the gate variable is as follows:

- 1) If the variable is positive, use this case to train Mapping 1.
- 2) If the variable is negative, use this case to train Mapping 2.
- 3) If the variable is zero (absolute magnitude less than  $1.e-15$ ), ignore this case. The tiny fudge factor allows for minor floating-point errors.

Thus, for example, if all cases are either positive or zero, or if all cases are either negative or zero, only one mapping is learned. If all cases are either positive or negative, none zero, then no cases are ignored. If all cases are positive, or if all cases are negative, you should simply not use a gate variable, as all cases will be used and only one mapping will be learned, which is just what happens when no gate variable is specified.

---

## Testing the Authenticity of the Mapping

Random variation is the bane of the model developer's existence, and nominal-to-ordinal conversion does not have a magical exemption. The algorithm described in this chapter and implemented in *VarScreen* will always produce a mapping function, whether such a function inherently exists in the data or not. Even if the variable values in the dataset have been produced by a research assistant in the back room rolling dice, a mapping will be found, and it will probably look decent. So what we need is a test of the mapping's authenticity.

In order to design such a test, we need a test statistic, a measurement of the degree to which the nominal variable and the target variable (as well as the gate variable if used) are related. Suppose that there is no relationship between the nominal variable and the target. Then, by definition, the expected value of the target for each case is independent of the value of the nominal variable. As a result we will likely find that the mapping values, the quantity assigned to each nominal category, are all about the same, differing only by random variation. Conversely, if there is a relationship between the nominal variable and the target, we will find that some categories have a large mapping value (mean rank for that category), while others have a small value. Thus, we are inspired to use the range of the mapping values, the largest minus the smallest, as our test statistic. The magnitude of this difference is directly related to the degree to which the nominal variable and the target are related.

I am not aware of any direct statistical test for the significance of any particular range value. However, we can easily design a decently accurate approximate test of significance. The null hypothesis is that there is no relationship between the nominal variable and the target, which we will test against the alternative that there is a relationship (with gating taken into account as discussed later). We can easily simulate this null hypothesis by shuffling the target, thus eliminating any relationship. If we repeat the shuffling hundreds or thousands of time, we can get a good estimate of the

distribution of the test statistic under the null hypothesis of there being no relationship. Then all we need to do is count the number of times the shuffled test statistic is less than or equal to the original, unshuffled statistic. The fraction of such comparisons gives us a p-value for the test, the probability that our test statistic could have been as large as it is even if there is no relationship between the nominal variable and the target. If we find that this p-value is very small, 0.05 or even 0.01 or less, we can be reasonably certain that there is a true relationship. Such Monte-Carlo permutation tests are incredibly valuable in applications like this.

If there is no gate variable, only one test is performed, and it is exactly as just described: the test statistic is the maximum mean rank across all categories of the nominal variable minus the minimum. We expect that the more a relationship exists, the greater will be this difference. But if there is a gate variable, we perform multiple tests. It is vital to keep in mind that when we compute multiple separate p-values, selection bias will come into play; there is an inflated probability that an apparently significant but undeserved p-value will be found just by random chance. Thus, if there is a gate, we should require very small p-values in order to have confidence in a relationship.

There is one more consideration in regard to a gate variable. If there is a gate variable, each case falls into one of three categories: contribute to the positive gate mapping, contribute to the negative gate mapping, and ignore this case. All of these significance tests act as if all three possibilities occur in the dataset. But if the gate variable takes only zero and positive values, or only zero and negative values, only one mapping is computed. In order to test as if two mappings are computed, the unused 'mapping' maps all values of the nominal variable to the median rank of the target, which will be very close to 50 except in pathological cases of extreme ties.

Here are the alternative hypotheses for the tests performed when there is a gate variable; the null hypothesis for all tests is that there is no relationship between the nominal variable and the target:



*Difference between gate values for each category* - Separately for each category of the nominal variable, we test whether there is a relationship between the gate (positive versus negative) and the target. The test statistic is the absolute difference for this category's mean target rank for cases having a positive gate versus those having a negative gate. This provides an indication of whether for some categories the gate is particularly important or unimportant to creating meaningful separate mappings. In particular, if none of the p-values are small we should consider not using the gate.

*The maximum of the differences computed above* is tested. Because the test above computes a p-value separately for each category of the nominal variable, there will be substantial selection bias; even if there is no relationship between the gate and the target for any category, there is substantial probability that random luck will cause one or more of the computed p-values to be significant. But the maximum difference among the categories is a single value and thus is free of selection bias. The upshot is that even if you get one or more highly significant p-values in the test above, if you do not also get a significant p-value on this maximum test you should probably ignore the significant result in the test above. It could easily be just selection bias at work.

*Maximum difference between categories for each gate value* - Separately for the positive and negative gate values, we test whether there is a relationship between the nominal variable and the target. This gives us two tests. First we consider only cases having a positive gate value, and we compute the difference between the maximum across categories of the mean target rank and the minimum. A greater difference implies a greater relationship. We also do the same for only negative gate cases. This pair of tests lets us know if the nominal variable / target relationship exists significantly for only one gate value, or for both, or for neither.

*The maximum of the two differences computed above* is tested. Because the test above computes a p-value separately for each of the two gate possibilities (positive and negative), there will be small but important selection bias; even if there is no relationship between the nominal variable and the target for either gate value, there is non-trivial probability that random luck will cause one of the two computed p-values to be significant. But the maximum difference for positive and negative gates is a single value and thus is free of selection bias. The upshot is that even if one or both p-values in the test above are highly significant, if you do not also get a significant p-value on this maximum test you should probably ignore the significant result in the test above. It could easily be just selection bias at work.

Of course, despite the 'maximum' unbiased tests we will still be looking at multiple p-values, so selection bias is unavoidable. But in order to detect the many ways in which a nominal variable / target relationship is present, we need to perform several tests. Thus, some selection bias will always be present.

## Creation of a New Variable

Nominal-to-Ordinal Mapping appears under the *Create* menu rather than the *Test* menu, because it creates a new variable as part of the testing process. If this is the first time that nominal-to-ordinal conversion is done in this run, this variable will be named **NomOrd\_1**. The second time, the new variable will be **NomOrd\_2**, and so forth. This new variable can be used in subsequent tests, and it can also be written to a text file using the 'File / Write Data' menu command.

## Invoking Nominal-to-Ordinal Conversion

When *Nominal-to-ordinal* is selected from the *Create* menu, a dialog similar to that shown below appears, and the following items must be specified:

Predictors	Gate	Target
_SEQNUM_	_No gate	_SEQNUM_
Date	_SEQNUM_	Date
Market	Date	Market
MONTH_M1	Market	MONTH_M1
NOT_VOLATILE	MONTH_M1	NOT_VOLATILE
ORDER_CLASS	NOT_VOLATILE	ORDER_CLASS
PRICE_TREND	ORDER_CLASS	PRICE_TREND
VOLATILE	PRICE_TREND	VOLATILE
Z_DAY_RET	VOLATILE	Z_DAY_RET
	Z_DAY_RET	

Monte-Carlo Permutation Test

☒ Complete  
☐ Cyclic

Replications: 100

The *leftmost column* specifies one or more variables that determine the nominal variable classes. Multiple variables 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.

There are two ways to input the nominal variable. The most straightforward is to specify one predictor, which for clearest operation should take on integer values from zero through one less than the number of nominal categories. If the values are not integers, the values are rounded to the nearest non-negative integer. The other method is to specify two or more predictors, one predictor for each category. The category of a case is whichever variable has the greatest value.

The *Gate* column is used to select an optional gate variable. If there is no gate variable, select *\_No gate*.

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

*Replications* is the number of Monte-Carlo Permutation Test iterations to be performed. It is usually best to set this to at least 100, and ideally as much as 10,000 or more so that accurate p-values will be computed. Note that the minimum possible p-value is the reciprocal of the number of permutation iterations. 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 extremely fast, so performing a large number of iterations is feasible.

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. This topic is also discussed on Page 22.

## Market Prediction Example 1: No Gate

Using over 8800 days of the index OEX, I created a few variables:

**PRICE\_TREND** is a nominal variable with integer values ranging from 0 through 7. The one bit is set if and only if the linear trend with a 10-day lookback is positive. The two bit is set if and only if the linear trend with a 50-day lookback is positive. The four bit is set if and only if the linear trend with a 250-day lookback is positive. Thus, for example, if the 10-day and 250-day trends are positive while the 50-day trend is not positive, this variable will have the value  $4+1=5$ .

**DAY\_RET** is the volatility-normalized one-day-ahead log return.

I ran nominal-to-ordinal conversions without using any gate control, and obtained the following results:

```
*****
*
* Computing nominal-to-ordinal conversion
*
* PRICE_TREND is the sole predictor
* There is no gate
* DAY_RET is the target
* 10000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Class bin counts...

Class	Count
0	508
1	523
2	395
3	612
4	899
5	1062
6	1771
7	3032

Class bin mean percentiles...

Class	Mean pctile
0	48.97
1	47.87
2	50.74
3	47.25
4	53.39
5	49.48
6	51.72
7	49.18

p-value for max mean percentile minus min mean percentile = 0.006  
NomOrd\_1 mean=50.00568 standard deviation=1.70703

The bin counts are not surprising for a strongly up-trending market like OEX; bins that have few or no positive trends are relatively few, while bins with upward trends are more common, with all three trends positive being the most prevalent.

The mean percentiles for each class show an interesting effect. Class 4 has the highest mean percentile. This is the class in which the long-term trend (250 days) is positive while the shorter trends are not positive. What we are apparently seeing is a mean-reversion to an upward trend. When the market has been trending upward for a long lookback period, but is not trending upward over shorter lookbacks, we are in a situation in which it is especially likely that the market will move upward the next day.

Class 3 shows the same effect in bear and flat markets. This class represents situations that are opposite Class 4: the long-term trend is flat or down, but the short and medium-term trends are up. This class exhibits the minimum or most negative movement the next day, a reversion to the bear trend.

Could this effect be just random luck? The p-value of 0.006 says that there is only a 6 in 1000 chance, 0.6 percent, that a between-class difference this extreme could have arisen by random chance. Thus, I am comfortable concluding that this nominal-to-ordinal conversion is legitimate, at least as far as Class 3 versus Class 4 is concerned.

The mean and standard deviation of the ordinal variable is shown.

## Market Prediction Example 2: Ignoring Cases

I now posit a theory that market patterns related to the relationship between trends at three different time extents might exist mostly for times in which the market is significantly moving, times of relatively high volatility. I kept the **PRICE\_TREND** and **DAY\_RET** variables of the prior example, and added a new variable: **IS\_VOLATILE**. This variable is positive for roughly half of the cases, those whose volatility is relatively high, and zero in lower volatility times. As a result, this mapping will ignore cases having low recent volatility. The mapping will be defined by only those cases having relatively high volatility. Cases having relatively low volatility will generate a constant ordinal value of approximately 50, the center of the possible range. The following results are obtained:

```
*****
*
* Computing nominal-to-ordinal conversion
*
* PRICE_TREND is the sole predictor
* IS_VOLATILE is the gate
* DAY_RET is the target
* 10000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Class bin counts...

Class	Gate-	Gate0	Gate+
0	0	218	290
1	0	175	348
2	0	233	162
3	0	377	235
4	0	388	511
5	0	463	599
6	0	990	781
7	0	1814	1218

Class bin mean percentiles...

Class	Gate-	Gate0	Gate+
0	50.01	46.01	51.20
1	50.01	49.05	47.28
2	50.01	50.74	50.75
3	50.01	48.05	45.96
4	50.01	52.58	54.00
5	50.01	49.67	49.33
6	50.01	51.47	52.04
7	50.01	49.00	49.46

For each class individually, p-value for positive gate vs negative gate...

Class	p-value
0	0.479
1	0.073
2	0.747
3	0.028
4	0.002
5	0.553
6	0.038
7	0.476

p-value for max across classes of the gate +/- difference = 0.125

p-value for max class mean percentile minus min, for negative gate = 1.000

p-value for max class mean percentile minus min, for positive gate = 0.019

p-value for max of the above two = 0.019

NomOrd\_1 mean=50.13352 standard deviation=1.46188

Obviously, the bin counts for a negative gate are all zero, because there are no cases with a negative gate. The mean percentiles for every class with a negative gate are reported to be a constant 50.01, although this is largely irrelevant, due to the fact that in this application no cases will have a negative gate. In practice, all this means is that if in the future you happen to have a case with a negative gate (should not happen!) you should set its ordinal value to this quantity.

For both the ignored cases (Gate0) and the used cases (Gate+) we see the same bull trend reversion that we saw in the first example, with Class 4 having the highest mean percentile. And for the positive gate cases we also see the same bear trend reversion that appeared in the first example, though this does not appear quite so strongly among the ignored cases.

The table of individual class p-values for the gate is interesting. Because there are no negative-gate cases, this test amounts to comparing, for each class separately, mean rank for Gate+ cases versus all cases. We see the tiny, impressive p-value of 0.002 for Class 4, but then we must curb our enthusiasm when we see a p-value of 0.125 for the greatest gate difference among the 8 classes. For the positive gate cases we see a fairly significant inter-class p-value of 0.019, but in the first example, which ignored the gate, we got the much more impressive p-value of 0.006. Thus, it appears that my theory of volatility involvement is probably worthless.



## Market Prediction Example 3: Full Gating

I'm not ready to give up on volatility gating yet, so I created the variable **VOLATILE** that partitions the cases into three groups. This variable is +1 for cases that lie in the approximately 40 percent most volatile cases, -1 for those in the approximately 40 percent least volatile, and 0 for the middle approximately 20 percent. This should provide a clear distinction between high volatility nominal/ordinal correlation and low volatility nominal / ordinal correlation. The following results were obtained:

```
*****
*
* Computing nominal-to-ordinal conversion
*
* PRICE_TREND is the sole predictor
* VOLATILE is the gate
* DAY_RET is the target
* 10000 replications of complete Monte-Carlo Permutation Test
*
*****
```

Class bin counts...

Class	Gate-	Gate0	Gate+
0	159	75	274
1	103	104	316
2	166	71	158
3	285	105	222
4	302	111	486
5	319	184	559
6	760	289	722
7	1396	548	1088

Class bin mean percentiles...

Class	Gate-	Gate0	Gate+
0	47.92	43.85	50.98
1	45.99	54.01	46.46
2	50.69	51.99	50.24
3	47.64	49.27	45.80
4	51.34	58.51	53.50
5	48.22	50.98	49.71
6	51.63	50.65	52.25
7	48.93	48.82	49.70

For each class individually, p-value for positive gate vs negative gate...

Class	p-value
0	0.285
1	0.886
2	0.889
3	0.477
4	0.314
5	0.463
6	0.675
7	0.503

p-value for max across classes of the gate +/- difference = 0.838

p-value for max class mean percentile minus min, for negative gate = 0.412

p-value for max class mean percentile minus min, for positive gate = 0.034

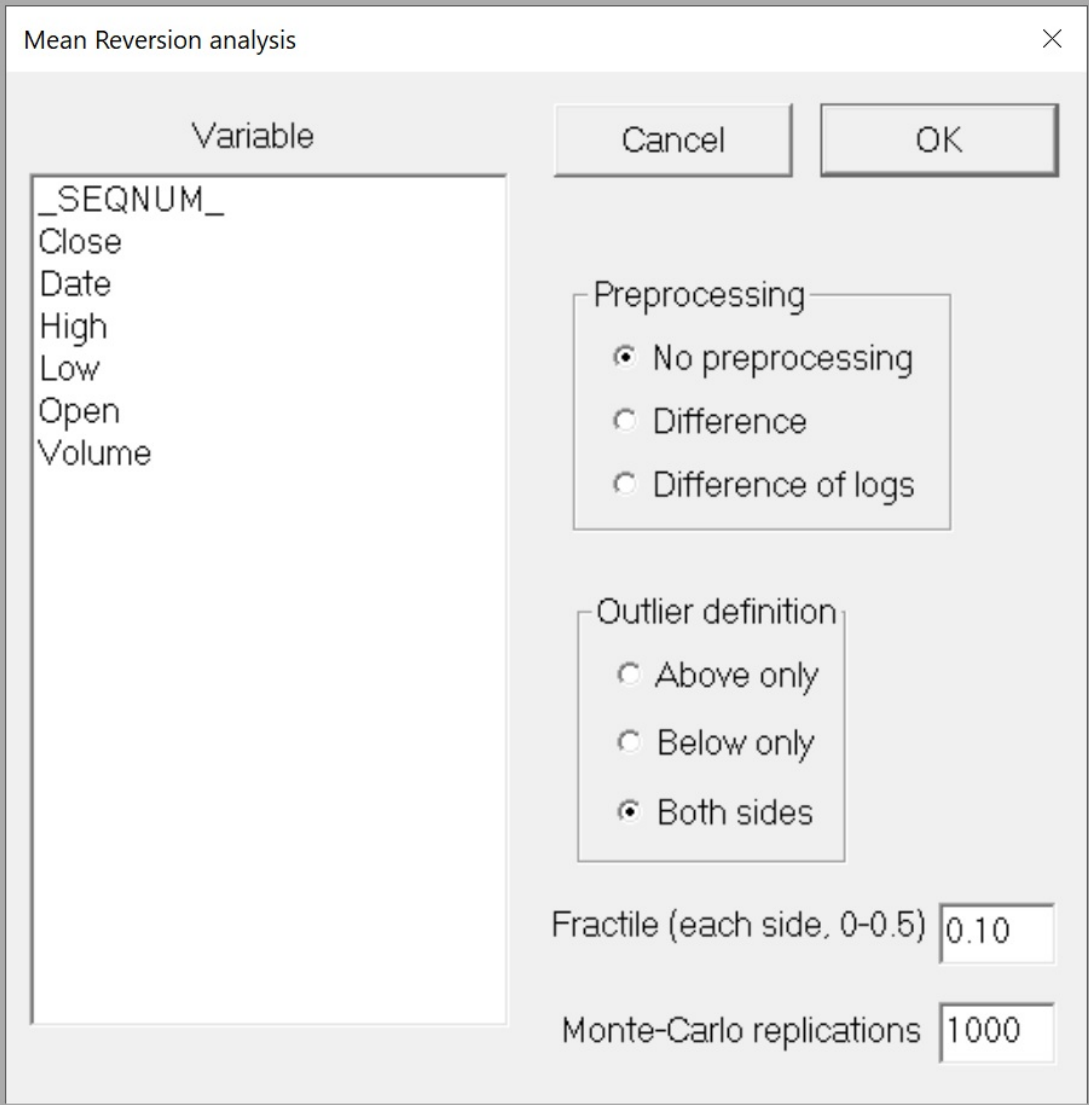
p-value for max of the above two = 0.154

NomOrd\_1 mean=49.92870 standard deviation=1.75329

One look at the table of mean percentiles for each class and gate tells me immediately that I'm way off base in partitioning the cases into unusually high and unusually low volatility. The mean percentiles have the largest spread for the Gate=0 group, the very group I'm ignoring! It ranges from a high of 58.51, greater than the high in either gate group, to a low of 43.85, lower than the low in either gate group. So in this test I'm telling the nominal-to-ordinal converter to ignore the group that shows the greatest relationship between the nominal class and the target! In response, I might create a new gate variable that flags when we are in intermediate volatility, but probably the best result would be obtained by skipping gating entirely, as shown in the first example.

## Mean Reversion

This test measures the degree to which a time series reverts toward the mean after an extreme value. The following dialog appears when this test is selected:



The dialog box is titled "Mean Reversion analysis" and has a close button (X) in the top right corner. It contains several sections for configuring the analysis:

- Variable:** A list box containing the following variables: `_SEQNUM_`, Close, Date, High, Low, Open, and Volume.
- Preprocessing:** A group box containing three radio buttons:
  - ☒ No preprocessing
  - ☐ Difference
  - ☐ Difference of logs
- Outlier definition:** A group box containing three radio buttons:
  - ☐ Above only
  - ☐ Below only
  - ☒ Both sides
- Fractile (each side, 0-0.5):** A text input field with the value `0.10`.
- Monte-Carlo replications:** A text input field with the value `1000`.

At the top right of the dialog, there are two buttons: "Cancel" and "OK".

The user selects a single *variable* to analyze.

Two types of optional *Preprocessing* are available. If *no preprocessing* is selected, the variable values are used to define extremes. If *Difference* is selected, extremes are defined by the change in value from one observation to the next. If *Difference of logs* is selected, extremes are defined by the change in the log of the value from one observation to the next. In this last case, all values must be positive.

The *Outlier definition* specifies whether the user is interested in only positive extremes (*Above* the mean), only negative extremes (*Below* the mean), or extremes on *Both sides*.

The *Fractile* is the fraction of observations on each side of the mean that are to be considered extreme. For example, if we specify a fractile of 0.1, then the ten percent most positive values (or differences if selected for preprocessing) would define positive extremes, and the ten percent most negative values (or differences if selected for preprocessing) would define negative extremes.

Finally, we must specify the number of *Monte-Carlo replications* to be performed to compute a p-value for the null hypothesis that there is no mean reversion beyond what would be expected for a random series.

## Preprocessing

Note that *Differencing* as performed here is different from the concept of *no preprocessing of difference data*. In other words, it is tempting to think that if we use an outside program to difference a series, and we test that differenced series specifying *No preprocessing*, we would get the same result as if we tested the original series with *Differencing* specified. This is not the case. With *No preprocessing* here, reversion is defined as moving toward the mean after an extreme. The raw value defines extremes as well as reversion. But

with *Differencing* specified here, extremes are defined by the differences (a sudden large change in the series), while reversion is defined as the raw value moving toward the mean. Thus, if you difference externally and specify *No preprocessing* here, reversion will be defined as the externally computed difference series moving toward the mean of the differences, which is almost certainly not what you want to do. If you want to define extremes in terms of case-to-case changes rather than individual values, you should difference here, not externally, unless you fully understand the effect of external differencing and want it done that way. It's hard to imagine any application in which this is what is desired.

Consider an example of this distinction for market price moves. Case 1 uses the raw market prices and tests with *Preprocessing by Difference of log*. Case 2 precomputes returns as difference of logs and tests these returns with *No preprocessing*. In both cases, extremes are defined the same way, using differences of log prices, so both cases will have the same extreme dates. But reversions are defined very differently. In Case 1, a reversion is defined as an opposite direction move; the raw price has retreated from its extreme value. For *ABOVE* the price has fallen, and for *BELOW* the price has risen. But in Case 2 a reversion is defined as the return retreating from its extreme. For example, with *ABOVE*, a return of 2.5 followed by a return of 2.4 would be a reversion in Case 2, while Case 1 would require a negative return in order to count as a reversion.

Thus, for testing price reversion in a market you would almost certainly want to use the raw market prices and *Preprocess as Difference of logs*. This would define extremes based on the change in log price, and define a reversion as a retreat from the price which gave the extreme change. On the other hand, suppose you have a time series of quarterly returns in a bond market, and you want to see if a quarter having unusually large return is likely to be followed by a lesser return the next quarter. Then you would use *No preprocessing* so that the return would define both the extreme and the reversion.

## The Algorithm

Just to be perfectly clear on how extremes and reversions are defined, here is the algorithm for *ABOVE* extremes. That for *BELOW* is a straightforward inversion of tests. Here,  $X_i$  is the  $i$ 'th value of the series or the differenced series, and Threshold is the fractile that defines a positive extreme.

Extremes = 0

*Counts extremes*

Reversions = 0

*Counts reversions*

For all  $i$

  If ( $X_i \geq \text{Threshold}$ )

*We have an extreme?*

    Extremes = Extremes + 1

    If no preprocessing

*Raw values, so look for decrease*

      If ( $X_{i+1} < X_i$ )

*Definition of reversion if using raw values*

        Reversions = Reversions + 1

    Else

*Differenced series, so look for negative change*

      If ( $X_{i+1} < 0.0$ )

*Definition of reversion if using differences*

        Reversions = Reversions + 1

Reversion percent =  $100 * \text{Reversions} / \text{Extremes}$

---

## Expected Reversions For a Random Series

*VarScreen* will use the permutation test to compute not only the null hypothesis distribution of the reversion rate, but also an estimate of the mean reversion rate under the null hypothesis which should be reasonably accurate under all distribution assumptions. So we do not need a theoretical computation. Nonetheless, some users may wonder about the theoretical reversion rate in a special but common situation, a time series in which each value is independent and identically distributed. Consider the case of *no preprocessing*, which is what we would almost certainly use in this case. Suppose we define extremes to be the  $x$  percent most extreme values. Then we would expect a reversion rate of  $100 - x/2$  percent. To see why, consider the set of observations one past the set of extremes. By definition, we expect  $x$  percent of them to also be extreme. This is because our null-hypothesis assumption is that the observations are independent and identically distributed. So automatically we will have  $100-x$  percent reversions, because  $100-x$  percent of the successors are not extreme and hence must be a reversion. Of the  $x$  percent extreme successors, on average half will be above and half below the extreme that comes before, again due to independence and identical distributions, so that gives us an additional  $x/2$  percent recursions.

In the case of differencing, reversion is defined by the difference being negative for **ABOVE** and positive for **BELOW**. In other words, reversion is when the raw value, before differencing, drops for **ABOVE** and rises for **BELOW**. Thus, for the special case of a random walk in which the changes are independently and identically distributed with a mean of zero, we expect reversion to be about 50 percent. Cases other than this special case have no theoretical solution that I know of, so we rely on the mean null-hypothesis reversion provided by the permutation test.

## Mean Reversion in the Bond Market?

There is, in some quarters, an opinion that a year of unusually high or low Treasury Bond returns is likely to be followed the next year by a rebound to more 'normal' returns. This theory is easy to test in *VarScreen*. Using over a century of bond returns, I ran three separate tests, ABOVE only, BELOW only, and BOTH. The following results were obtained:

*Computing mean reversion analysis*

*No preprocessing*  
*Above threshold only*  
*0.100 is fractile threshold*  
*100000 Monte-Carlo replications*  
*bonds is the tested variable*

**Cases above 14.90056 are considered extreme**  
**91.304 percent reversions in data**  
**95.181 percent reversions average for null hypothesis**  
**p-value = 0.891**

*Computing mean reversion analysis*

*No preprocessing*  
*Below threshold only*  
*0.100 is fractile threshold*  
*100000 Monte-Carlo replications*  
*bonds is the tested variable*

**Cases below -2.08394 are considered extreme**  
**100.000 percent reversions in data**  
**95.159 percent reversions average for null hypothesis**  
**p-value = 0.306**

*Computing mean reversion analysis*

*No preprocessing*  
*Above and below threshold*  
*0.100 is fractile threshold*  
*100000 Monte-Carlo replications*  
*bonds is the tested variable*

**Cases below -2.08394 or above 14.90056 are considered extreme**  
**95.556 percent reversions in data**  
**95.170 percent reversions average for null hypothesis**  
**p-value = 0.617**

We see that for positive extremes, there is actually less reversion than one would expect. For negative extremes, we get a 100 percent reversion rate, but even that is not statistically significant. And not surprisingly, looking at both sides gives almost exactly the expected random rate.



---

## Mean Reversion in the Equity Market?

Although the equity markets are well known to have long-term trends, many traders believe that short, sharp shocks quickly revert to more normal returns. I used about 9300 days of the index OEX and *Difference of Logs* preprocessing to compute daily returns. The following results appeared:

```
Computing mean reversion analysis
  Difference of logs is used
  Above threshold only
  0.100 is fractile threshold
  10000 Monte-Carlo replications
    CLOSE is the tested variable
```

```
Cases above 0.01197 are considered extreme
48.602 percent reversions in data
46.408 percent reversions average for null hypothesis
p-value = 0.082
```

```
Computing mean reversion analysis
  Difference of logs is used
  Below threshold only
  0.100 is fractile threshold
  10000 Monte-Carlo replications
    CLOSE is the tested variable
```

```
Cases below -0.01168 are considered extreme
57.097 percent reversions in data
53.364 percent reversions average for null hypothesis
p-value = 0.008
```

```
Computing mean reversion analysis
  Difference of logs is used
  Above and below threshold
  0.100 is fractile threshold
  10000 Monte-Carlo replications
    CLOSE is the tested variable
```

```
Cases below -0.01168 or above 0.01197 are considered extreme
52.849 percent reversions in data
49.886 percent reversions average for null hypothesis
p-value = 0.005
```

These results confirm that statistically significant reversion is happening for both upward and downward shocks, with recovery from a downward spike more likely than retreat from an upward spike. However, despite the statistical significance, the actual difference in reversion rate compared to the rate with random changes is quite small.



## Entropy Tests for Structure

Most of the entropy tests described in this section compute a measure of the degree to which an ordered time series has structure (serial dependence) of any sort, versus being random. The following dialog appears when *Entropy* is selected:

Entropy tests

Variables

- \_SEQNUM\_
- DEP\_RANDOM
- DEP\_RANDOM1
- DEP\_RANDOM2
- DEP\_RANDOM3
- DEP\_RANDOM4
- DEP\_RANDOM5
- DEP\_RANDOM6
- DEP\_RANDOM7
- DEP\_RANDOM8
- DEP\_RANDOM9
- RANDOM0
- RANDOM1
- RANDOM2
- RANDOM3
- RANDOM4
- RANDOM5
- RANDOM6
- RANDOM7
- RANDOM8
- RANDOM9
- SUM12
- SUM1234
- SUM34

Preprocessing

- ☒ No preprocessing
- ☐ Difference
- ☐ Difference of logs

Entropy type

- ☒ Shannon
  - Number of bins: 10
  - Delay: 1
  - Order: 3
- ☐ SVD
  - Delay: 1
  - Order: 3
- ☐ K2
  - Delay: 1
  - Order: 3
  - R Fractile: 0.0500
- ☐ AP
  - Delay: 1
  - Order: 3
  - R Fractile: 0.0500

MCPT replications: 100

Cancel OK

The user selects a single *variable* to analyze. Several types of optional *Preprocessing* are available. If *no preprocessing* is selected, the variable values are used to compute the entropy. If *Difference* is selected, the entropy is computed using the change in value from one observation to the next. If *Difference of logs* is selected, it uses the change in the log of the value from one observation to the next.

If *MCPT replications* is set greater than 1 (typically at least 100, and at least 1000 if possible), a Monte-Carlo permutation test is performed, testing the null hypothesis that the time series is random and hence unpredictable.

Four common types of entropy can be computed:

- *Shannon Entropy* is a generalized form of the most common, traditional entropy measure, which is based on partitioning the signal points into bins which are equally likely under the null hypothesis.
- *SVD Entropy* measures the degree to which sequences of length  $n$  in the time series can be described by fewer than  $n$  independent values. Under the null hypothesis,  $n$  independent values would be needed.
- *K2 Entropy* measures the rate at which information is lost over time in a dynamical system. It is closely related to the Lyapunov exponent, which quantifies how nearby trajectories in phase space diverge. Under the null hypothesis, information is lost instantly.
- *AP Entropy* (*Approximate entropy*) is based on the likelihood that relatively short-duration patterns in the time series are repeated. It is especially sensitive to periodic components in the time series. Under the null hypothesis, the presence of any pattern has no impact on the likelihood of this same pattern appearing later.

In order to compute each of these entropies, we examine observations in groups of *Order* historical values, each separated in time by *Delay*.

## Shannon Entropy

When most people think of entropy, they are thinking of the classic *Shannon Entropy*. Very roughly, one can think of the traditional Shannon entropy of a collection of values of a variable as an upper bound on the amount of information provided by that collection. Whether that information is useful to any particular application is another issue which does not concern us here; we are considering traditional Shannon entropy information in a generic sense. Also note that unlike its generalized form presented later, as well as other entropy measures presented in this chapter, traditional Shannon entropy says nothing about whether a time series is predictable versus random. The primary reason we would be interested in traditional Shannon entropy is to assess the potential utility of a predictor variable. In most situations (though there are some exceptions) we want to transform a predictor variable in a way that maximizes its traditional Shannon entropy, thus maximizing the upper bound on its information carrying capacity.

Shannon entropy is most often computed for discrete variables, variables which can take on only a finite number of possible values. There are ways to estimate Shannon entropy for continuous random variables, but they are fraught with numerous problems. Thus, it is useful to convert a continuous variable into a discrete variable by partitioning cases based on the value of the variable. In *Varscreen*, Shannon entropy is computed by partitioning values of the variable into a user-specified number of bins, the ***Number of bins*** parameter, which of course must be at least 2. Two different methods of partitioning are used. If the user chooses to compute traditional Shannon entropy by specifying ***Order=1***, then the bin boundaries are defined by dividing the range of the variable into equally spaced intervals, and ***Delay*** is ignored. If generalized Shannon entropy is chosen by specifying ***Order>1***, the bin boundaries are defined by partitioning the entire dataset into bins that contain equal numbers of cases, or at least as equal as possible.

### *Traditional Shannon Entropy*

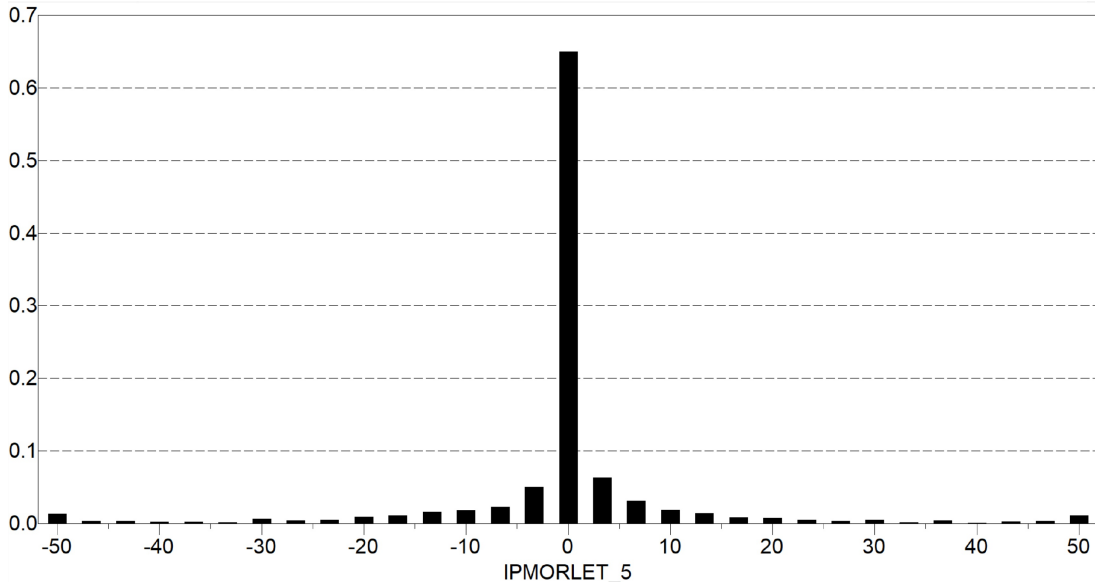
Traditional Shannon entropy is computed by specifying *Order*=1, in which case *Delay* is ignored. This entropy measure says nothing about whether a time series itself is predictable; rather it is an upper limit on the amount of potentially useful predictive information contained in the variable. This makes it useful in choosing a form of a predictor that maximizes its entropy.

Suppose the cases are partitioned into  $M$  bins. This is done for traditional Shannon entropy by dividing the range of the variable into equally spaced intervals. Let  $p_i$  be the fraction of cases in bin  $i$ . Then the Normalized Traditional Shannon Entropy of the set of observations is given by the following equation:

$$\text{Normalized Shannon Entropy} = -\frac{1}{\log(M)} \sum_{i=1}^M p_i \log(p_i)$$

This is the value computed by *VarScreen*. It ranges from 0 through 1. Note that  $x \log(x)$  is defined as 0 when  $x=0$ . Also note that traditional Shannon entropy is unaffected by permutation, because no historical values play a role in its computation. Therefore, this statistic cannot test whether a time series has structure in the sense of serial dependence, and Monte-Carlo permutation tests are pointless; all permutations will have the same entropy.

The main use for computing Traditional Shannon Entropy is for gauging the potential quality of predictor variables. Although there are exceptions, in general the entropy of model inputs should be as large as possible to maximize their information-carrying ability. It is very often to our advantage to apply a nonlinear monotonic transform to a predictor variable so as to increase its Traditional Shannon Entropy.



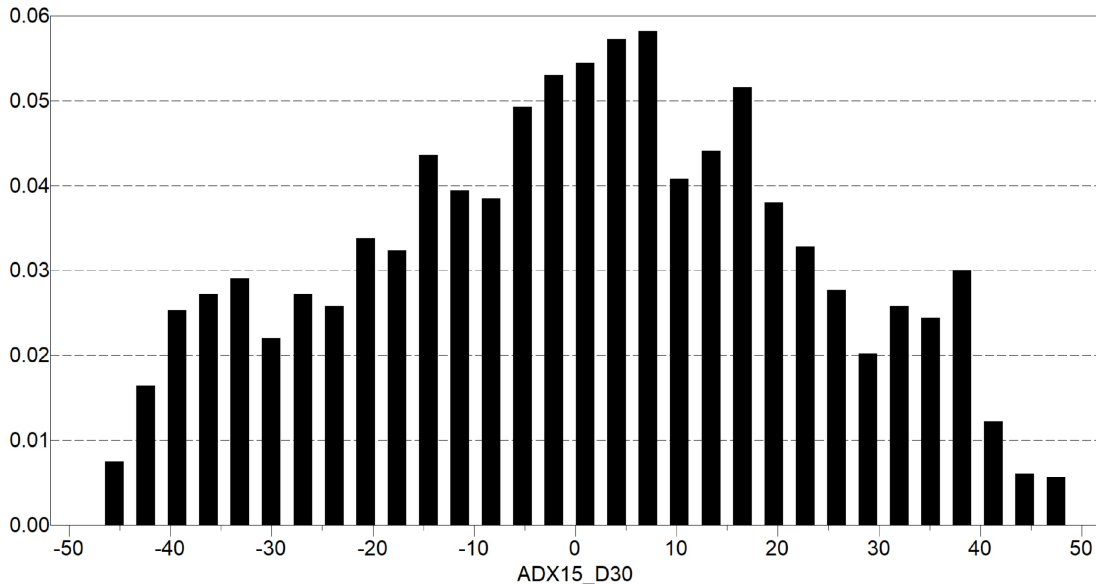
I computed an in-phase Morlet wavelet indicator using many decades of OEX, and then ran a traditional Shannon entropy test on this indicator. Its histogram is shown above, and this histogram would make most people suspicious that it has limited information-carrying capacity due to the fact that the vast majority of cases are tightly clustered around zero. A traditional Shannon entropy test confirms that this indicator can contain only a little over half of the theoretical maximum information.

```

*****
*                                     *
*   Performing ENTROPY test           *
*   No preprocessing                 *
*   Performing Shannon entropy test  *
*   10 is number of bins             *
*   1 is time delay                  *
*   1 is order                       *
*   Order 1 means mcpt test is useless, so reps removed *
*   No Monte-Carlo Permutation Test *
*   IPMORLET_5 is the tested variable *
*                                     *
*****

```

Normalized 0-1 Shannon entropy = 0.56033



Using the same historical market data, I computed an ADX-based indicator, whose histogram is shown above. Based on the wide, well diversified spread of the values, one would expect that this indicator could contain a large amount of information. An entropy test shows this to be the case, with this indicator having the capability of containing nearly the theoretical maximum possible information.

```

*****
*                                     *
*   Performing ENTROPY test          *
*   No preprocessing                 *
*   Performing Shannon entropy test  *
*   10 is number of bins             *
*   1 is time delay                  *
*   1 is order                       *
*   Order 1 means mcpt test is useless, so reps removed *
*   No Monte-Carlo Permutation Test  *
*   ADX15_D30 is the tested variable *
*                                     *
*****

```

Normalized 0-1 Shannon entropy = 0.96638

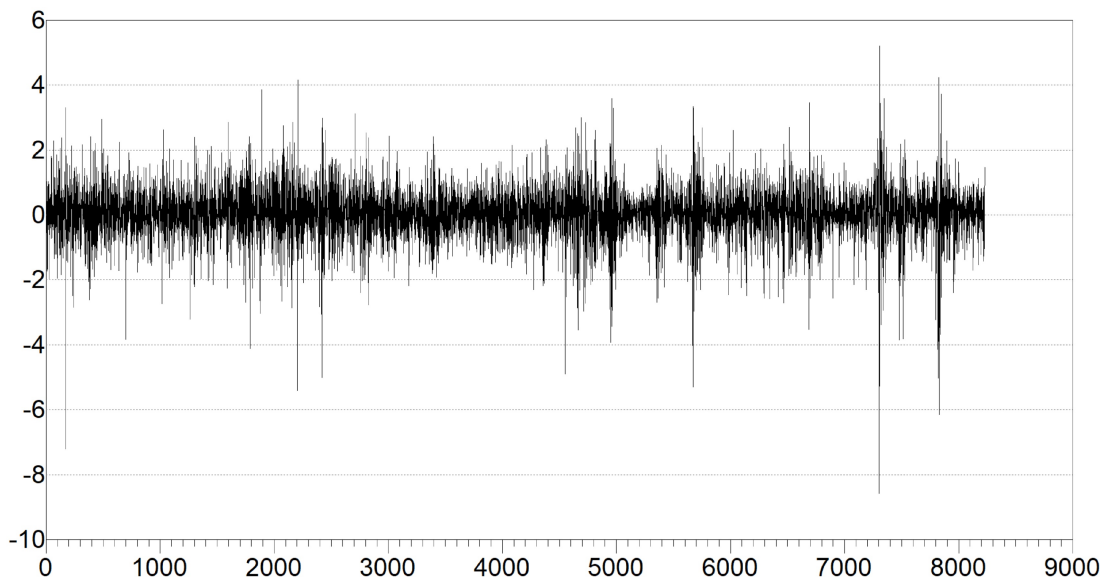


### *Generalized Shannon Entropy*

We can extend the definition of Shannon entropy to include historical values in its computation. This lets us use generalized Shannon entropy to assess the degree to which a time series has structure (serial dependence) and hence is predictable. In this case, bin boundaries are computed from the entire dataset and defined in such a way that each of the *Number of bins* bins contains the same number of cases, or at least as close as possible given the possible presence of ties.

We do this generalization by increasing the number of bins and including lagged values of the variable in the bin assignment. The number of what may be called hyper-bins can rise explosively, because it is the user-specified *Number of bins* raised to the power of the user-specified *Order*. It is easy to visualize this by considering a hyper-grid of hyper-bins. For example, suppose *Number of bins* is 3 and *Order* is 2. Then we will have 9 hyper-bins arranged in a 3 by 3 square. Each case will be assigned to one of these 9 hyper-bins. Its row will be defined by the bin assignment (one of three) of the current time series value, and its column will be defined by the bin assignment (one of three) of the lagged time series value. If *Order* is 3 (current value plus two lagged values) we will have a 3 by 3 by 3 cube of 27 bins, and so forth. It should be apparent that under the null hypothesis of no serial dependence, we would expect on average for each hyper-bin to contain the same number of cases, which gives rise to the maximum possible entropy.

The formula for computing the generalized Shannon entropy in this case is the same formula that appeared on the prior page for traditional Shannon entropy, except now each of the hyper-bins takes part in the calculation, so  $M = \text{Number of bins}$  raised to the power of *Order*. As in the traditional case, this entropy ranges from 0 to 1, with the maximum being obtained when every hyper-bin contains exactly the same number of cases.



I computed ATR-normalized daily returns for over 8000 days of OEX, shown above. This is the log ratio of successive daily closes, divided by average true range over the prior year in order to scale according to volatility. For the generalized Shannon entropy test, I partitioned the returns into just three bins: an especially positive return, an especially negative return, and a middle-of-the-road return. I examined only the current return and the prior day's return. As we see in the results below, the generalized entropy is very high, nearly the maximum of 1. Nonetheless, we could reject the p-value of no structure at the astonishing level of 0.0001, the minimum possible.

```
*****
*                                                                 *
*      Performing ENTROPY test                                     *
*      No preprocessing                                           *
*      Performing Shannon entropy test                             *
*      3 is number of bins                                         *
*      1 is time delay                                             *
*      2 is order                                                  *
* 10000 replications of complete Monte-Carlo Permutation Test    *
*      DAY_RET is the tested variable                             *
*                                                                 *
*****
```

```
Normalized 0-1 Shannon entropy = 0.99854
Mean null hypothesis entropy = 0.99989
For null hypothesis of no structure (entropy=1), pval = 0.00010
```

## Singular Value Decomposition Entropy

*Singular Value Decomposition Entropy* is one of the most powerful and effective methods for quantifying structure (unspecified serial dependence) in a time series. The intuition behind this entropy measure is straightforward. Use the univariate time series being tested to create a new multivariate dataset. The first case (row) in this new dataset would be the first observation along with one or more lagged values (columns). The second case in the new dataset would be the second observation along with one or more similarly lagged values, and so forth. The new multivariate dataset would have almost as many observations as the tested dataset; a few at the oldest end would be unavailable due to the lack of lagged values.

Then the task, roughly stated, involves measuring the degree to which the cases in the new dataset can be described by fewer values than the number of columns. For example, suppose that each new multivariate case consists of the current value of the original series, along with two lagged values; each case has three adjacent values. Also suppose there is no serial dependence in the original time series; it is completely random. Then we would need three independent values to describe the three variables for each case. On the other hand, suppose there is some serial dependency for short lags in the original time series. Then we would not need information from three fully independent variables in order to describe the new cases. The serial dependence would create some redundancy among the three new variables, which reduces the amount of information needed to fully describe the new cases.

How can we quantify the reduction of information needed to describe the cases in the new dataset? The *singular values* of the new matrix are closely related to the eigenvalues that describe dimension sizes in principal components. At one extreme, the case of all columns of the new matrix being independent, the singular values will be equal, signifying that we will need fully independent information to describe each case. This is analogous to the situation of the principal components of a dataset all having equal

eigenvalues. But as dependency between the current and lagged values of the original time series increases, the singular values will spread out, with some being larger and others smaller. This is analogous to the situation of some principal components being more prominent (larger eigenvalues) than others. This shows that information in some dimension(s) accounts for redundancy, while information in lesser dimensions clarifies departures from redundancy. So in order to quantify redundancy, all we need to do is quantify the degree to which the singular values depart from equality.

We can now make the process more rigorous. Let  $x_i$  be the  $i$ th value of the tested time series. Let  $\tau$  be the number of observations skipped for each lag, *Delay* in the dialog box. This is typically 1, but can be larger. Let *order* be the number of columns in the new data matrix, the current value and all lagged values. Let  $N$  be the number of observations in the tested time series. Then the new matrix, called the *embedded matrix*, will have  $N-(\text{order}-1)*\tau$  rows. We lose some cases at the end due to the fact that lagged values are unavailable. The  $i$ th row of this matrix is  $\{x_i, x_{i+\tau}, x_{i+2\tau}, \dots, x_{i+(\text{order}-1)\tau}\}$ .

Compute the singular value decomposition of the embedded matrix, and let  $w_i$  for  $i$  from 1 to *order* be the singular values. Normalize them to sum to 1 as shown below:

$$\sigma_i = \frac{w_i}{\sum_{j=1}^{\text{order}} w_j}$$

Then compute the normalized SVD entropy:

$$\text{Normalized SVD Entropy} = -\frac{1}{\log(\text{order})} \sum_{i=1}^{\text{order}} \sigma_i \log(\sigma_i)$$

In this equation,  $x \log(x)$  is defined as 0 for  $x=0$ . The normalized SVD entropy ranges from 0 (completely deterministic) to 1 (completely random).

Because permutation destroys any actual structure in a time series, replacing it with random ordering, a Monte-Carlo permutation test is easy to perform. The null hypothesis for this test would generally be that the time series has no structure. Thus, to compute a p-value for this null hypothesis one would count the number of times the entropy for a permuted series is less than or equal to that of the original series. Add 1 to this count and divide by the number of permutations plus 1. This fraction is the approximate probability that if there is no structure (serial dependence, predictability) in the time series, we would have found its entropy to be as low as what we obtained.

I used the same ATR-normalized daily returns of OEX as were used in the prior test, and performed an SVD entropy test with the same delay (1) and order (2). The following results were obtained. Note that this SVD test has essentially no power to detect the nature of the structure in these returns.

```
*****
*
*      Performing ENTROPY test
*      No preprocessing
*      Performing Singular Value Decomposition entropy test
*      1 is time delay
*      2 is order
* 10000 replications of complete Monte-Carlo Permutation Test
*      DAY_RET is the tested variable
*
*****

Normalized 0-1 SVD entropy = 0.99997
Mean null hypothesis entropy = 0.99998
For null hypothesis of no structure (entropy=1), pval = 0.22630
```

## Kolmogorov K2 Entropy

*Kolmogorov K2 Entropy* is an intensely theoretical entropy, with relatively little obvious intuition supporting it. Vaguely stated, K2 entropy measures the rate at which information provided by data up to the current time deteriorates as time marches into the future. If the time series has no serial dependence, all information is lost instantly.

K2 entropy has several practical disadvantages as well. Its runtime is proportional to the square of the number of observations, times the order, meaning that for large datasets its runtime may be a problem. There does not appear to be any way to normalize it to a consistent range such as 0-1, so the measured entropy is useful only for relative comparisons; *K2 entropy* has no absolute meaning in isolation. And possibly most annoying of all, it relies on a user parameter whose ‘ideal’ setting is not clear, but which can have a significant impact on results, at least in absolute terms. Nonetheless, when used with a Monte-Carlo permutation test it can be extremely powerful for identifying structure in a time series.

I won’t even try to delve into the extremely complex theory behind K2 entropy, focusing instead on the algorithm. Users who want the theory should consult the definitive paper, “Estimation of the Kolmogorov Entropy From a Chaotic Signal” by Peter Grassberger and Itamar Procaccia, readily available from online sources.

Let  $x_i$  be the  $i$ th value of the tested time series. Let  $\tau$  be the number of observations skipped for each lag, **Delay** in the dialog box. This is typically 1, but can be larger. Let **Order** be the number of lagged values (including the current value) considered in the calculation; it must be at least 2. **R Fractile** must be in the range 0-1 and will typically be quite small, perhaps 0.01 if there are thousands of cases, or somewhat larger if there are fewer cases. It would almost never be larger than 0.1. This parameter will be discussed later when the algorithm is described.

Let  $d$  be an integer much smaller than the number of observations. This is the **Order** parameter specified by the user. Although the observation with index  $i$  is a scalar, we will treat observation  $i$  as a  $d$ -vector consisting of  $\{x_i, x_{i+\tau}, x_{i+2\tau}, \dots, x_{i+(d-1)\tau}\}$ . This will slightly reduce the number of valid cases, because we cannot consider cases whose lag pushes them beyond the end of the series. In particular, suppose there are  $N$  original cases. Then there will be  $N_{\text{valid}} = N - (d-1)\tau$  valid cases.

Let  $n$  and  $m$  be two different indices into the time series. Define the norm of the difference between observations  $n$  and  $m$  in the usual way:

$$Norm_{n,m,d} = \left[ (x_n - x_m)^2 + (x_{n+\tau} - x_{m+\tau})^2 + \dots + (x_{n+(d-1)\tau} - x_{m+(d-1)\tau})^2 \right]^{1/2}$$

Let  $\epsilon$  be a small number. We count the number of pairs of observations whose difference norm is less than  $\epsilon$ . Because the norm is symmetric, we need to consider only those pairs for which  $m > n$ , and there will be  $N_{\text{valid}} * (N_{\text{valid}} - 1) / 2$  such pairs. In the following equation, define the function  $1(\cdot)$  to have the value 1 if the expression it operates on is true, and 0 if false. Then the *Correlation integral* is defined as:

$$C_d(\epsilon) = \frac{2}{N_{\text{valid}}(N_{\text{valid}} - 1)} \sum_{m>n} 1(Norm_{n,m,d} < \epsilon)$$

The paper cited earlier defines the *K2 entropy* as:

$$K_{2,d}(\epsilon) = \frac{1}{\tau} \log \left( \frac{C_d(\epsilon)}{C_{d+1}(\epsilon)} \right)$$

Notice that we need to specify 2 parameters in order to compute the *K2 entropy*. The  $d$  parameter in the definition is just the user-specified **Order**, the number of lags that will be examined, and this is usually fairly easy to specify for an application, though of course it may also just be a hunch on the part of the user. The real problem is the search radius  $\epsilon$ . If you make it too small, the numerator or denominator may be zero. If you make it too large,

the accuracy of the result may be poor due to excessive ‘blurring’ of the comparison of adjacent  $d$  values.

Here is how I choose to compute a reasonable value for  $\varepsilon$ , though other methods are possible. First, note that the numerator in the equation above will never be less than the denominator, because fewer entries are going into the norm calculation. Second, note that the denominator must never be zero, or equivalently, there must always be at least one pair of observations within the search radius. And by the first note, if the denominator is positive, the numerator is guaranteed to also be positive. So  $C_{d+1}(\varepsilon)$  is the critical consideration.

To ensure that  $C_{d+1}(\varepsilon)$  is positive, I compute and store all  $N_{\text{valid}} * (N_{\text{valid}} - 1) / 2$  norms for  $d+1$ , and sort them in ascending order. Then I locate the user-specified ***R Fractile*** fractile of these norms and use this as  $\varepsilon$ . By specifying ***R Fractile*** to be a very small positive number, we ensure (except for very rare pathological cases) that the numerator and denominator are both positive, while at the same time ensuring that the correlation integrals will not be based on a search radius so large that accuracy is lost. According to the cited paper,  $\varepsilon$  should be ‘small’. My algorithm guarantees that it is small, but not so small that the denominator becomes zero.



I used the same ATR-normalized daily returns of OEX as were used in the prior tests, and performed a K2 entropy test with the same delay (1) and order (2). The following results were obtained.

```
*****
*
*      Performing ENTROPY test
*      No preprocessing
*      Performing Kolmogorov-2 entropy test
*      1 is time delay
*      2 is order
* 0.05000 is fractile of squared norms for radius
* 10000 replications of complete Monte-Carlo Permutation Test
*      DAY_RET is the tested variable
*
*****
```

K2 entropy (no normalization) = 1.11626  
 Radius = 0.55661 Last Corr = 0.050 Prior Corr = 0.153  
 Mean null hypothesis entropy = 1.16538  
 For null hypothesis of no structure, pval = 0.00010

The *K2 entropy* is printed first, although it has little value in isolation. In the next line we see the  $\varepsilon$  search radius, the denominator  $C_{d+1}(\varepsilon)$ , and the numerator  $C_d(\varepsilon)$ . These correlation integrals may be of some interest to users. The third line shows the mean *K2 entropy* for the null hypothesis cases, the permuted series values. Observe that the null hypothesis entropy is significantly larger than the unpermuted entropy. So it should come as no surprise that the p-value for the null hypothesis of the series having no structure, against the alternative of the series possessing some sort of serial dependence, is tiny, the minimum possible value. It appears that of the three entropy tests for structure conducted so far, K2 entropy is most powerful for detecting the presence of structure in daily market returns.

## AP (Approximate) Entropy

*Approximate Entropy* is, like K2 entropy, an intensely theoretical entropy, with relatively little obvious intuition supporting it. It also shares with K2 several practical disadvantages as well. Its runtime is proportional to the square of the number of observations, times the order, meaning that for large datasets its runtime may be a problem. There does not appear to be any way to normalize it to a consistent range such as 0-1, so the measured entropy is useful only for relative comparisons; *AP entropy* has no absolute meaning in isolation. And possibly most annoying of all, it relies on a user parameter whose ‘ideal’ setting is not clear, but which can have a significant impact on results, at least in absolute terms. Nonetheless, when used with a Monte-Carlo permutation test it can be extremely powerful for identifying structure in a time series.

I won’t even try to delve into the theory behind AP entropy, focusing instead on the algorithm. Users who want the theory should consult the definitive paper, “Approximate Entropy as a Measure of System Complexity” by Steven M. Pincus, readily available from online sources.

Let  $x_i$  be the  $i$ th value of the tested time series. Let  $\tau$  be the number of observations skipped for each lag, **Delay** in the dialog box. This is typically 1, but can be larger. Let **order** be the number of lagged values considered in the calculation; it must be at least 2. **R Fractile** must be in the range 0-1 and will typically be quite small, perhaps 0.01 if there are thousands of cases, or somewhat larger if there are fewer cases. It would almost never be larger than 0.1. This parameter will be discussed later when the algorithm is described.

Let  $d$  (**Order**) be an integer much smaller than the number of observations. Although the observation with index  $i$  is a scalar, we will treat observation  $i$  as a  $d$ -vector consisting of  $\{x_i, x_{i+\tau}, x_{i+2\tau}, \dots, x_{i+(d-1)\tau}\}$ . This will slightly reduce the number of valid cases, because we cannot consider cases whose lag pushes

them beyond the end of the series. In particular, suppose there are  $N$  original cases. Then there will be  $N_{\text{valid}} = N - (d-1)\tau$  valid cases.

Let  $n$  and  $m$  be two different indices into the time series. Define the norm of the difference between observations  $n$  and  $m$  as the maximum of the individual absolute differences:

$$Norm_{n,m,d} = \text{Max} [|x_n - x_m|, |x_{n+\tau} - x_{m+\tau}|, \dots, |x_{n+(d-1)\tau} - x_{m+(d-1)\tau}|]$$

Let  $\varepsilon$  be a small number. For an observation  $i$  we count the number of other observations whose difference norm from observation  $i$  is less than  $\varepsilon$ . Because we don't count the difference between an observation and itself, there will be  $N_{\text{valid}} - 1$  tested pairs. In the following equation, define the function  $1(\cdot)$  to have the value 1 if the expression it operates on is true, and 0 if false.

$$C_i^d(\varepsilon) = \frac{1}{N_{\text{valid}} - 1} \sum_{j \neq i} 1(Norm_{i,j,d} < \varepsilon)$$

Then we define Phi as follows:

$$\Phi^d(\varepsilon) = \frac{1}{N_{\text{valid}} - 1} \sum_i \log C_i^d(\varepsilon)$$

The paper cited earlier defines the *Approximate entropy* as:

$$ApEn(d, \varepsilon) = \Phi^d(\varepsilon) - \Phi^{d+1}(\varepsilon)$$

Notice that we need to specify 2 parameters in order to compute the *Approximate entropy*. The  $d$  parameter in the definition is just the user-specified **Order**, the number of lags that will be examined, and this is usually fairly easy to specify for an application, though of course it may also just be a hunch on the part of the user. The real problem is the search radius  $\varepsilon$ . If you make it too small, one or more of the  $C_i^d(\varepsilon)$  may be zero, making its log

Here is how I choose to compute a reasonable value for  $\varepsilon$ , though other methods are possible. Note that  $C_i^d(\varepsilon)$  in the equation above will never be less than  $C_i^{d+1}(\varepsilon)$ , because fewer entries are going into the former's norm calculation. So  $C_i^{d+1}(\varepsilon)$  is the critical consideration.

I used the same ATR-normalized daily returns of OEX as were used in the prior tests, and performed an approximate entropy test with the same delay (1) and order (2). The following results were obtained.

```
*****
*
*   Performing ENTROPY test
*   No preprocessing
*   Performing Approximate entropy test
*   1 is time delay
*   2 is order
* 0.05000 is fractile of norms for radius
* 10000 replications of complete Monte-Carlo Permutation Test
*   DAY_RET is the tested variable
*
*****
```

---

AP entropy (no normalization) = 0.00063  
Radius = 7.33758 Last Phi = -0.002 Prior Phi = -0.001  
Mean null hypothesis entropy = 0.00065  
For null hypothesis of no structure, pval = 0.00190

The *AP entropy* is printed first, although it has little value in isolation. In the next line we see the  $\epsilon$  search radius, the final  $(d+1)$  Phi, and the prior  $(d)$  Phi. These values may be of some interest to users. The third line shows the mean *AP entropy* for the null hypothesis cases, the permuted series values. Finally we get the p-value for the null hypothesis of the series having no structure, against the alternative of the series possessing some sort of serial dependence. We see that the actual entropy is only slightly less than the null hypothesis mean entropy, but this difference is still quite significant, with less than a 0.2 percent chance of having been obtained if the series has no structure.

Be aware of yet another potential weakness of AP entropy, which we see clearly in this example. In the results just shown, observe that the computed radius is 7.33758, which is huge compared to the standard deviation of the DAY\_RET variable, 0.81661. Compare this to the radius of 0.55661 computed for K2 entropy for the same dataset. How can this happen? The problem is that for AP entropy, the final radius is the *maximum* fractile radius over all  $i$ . If the dataset contains any wild outliers, which is certainly the case for stock market returns, an outlier will almost certainly have a large individual radius, which will greatly skew the final radius upward. So in general, AP entropy should be avoided for data with a heavy tail.



---

## Plotting

Several plot functions are available from the *Plot* menu. These are as follows:

### Series

This selection causes a time series to be displayed. The user must specify a variable and optionally check or uncheck the *Connect* box. This option controls whether the points are connected, or displayed as vertical lines with a baseline at zero. The display can be magnified by positioning the mouse at the left side of the desired area of magnification, pressing the left mouse button, dragging to the right as desired, and releasing the button. Once magnified, the enhanced area can be shifted left or right by dragging the horizontal scroll at the bottom of the screen, or clicking on the scroll bar in the usual Windows fashion. At this time, the program does not allow expanding the image back to its original size, although the same effect can be had by plotting the series again.

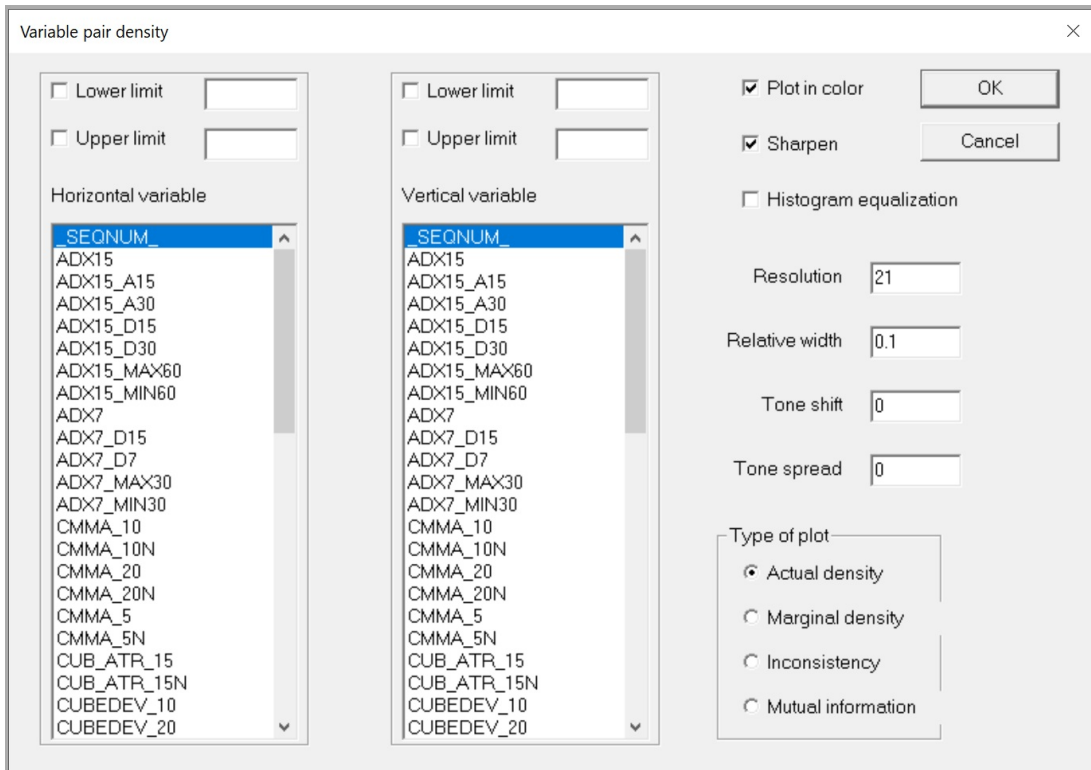
### Histogram

A histogram of a selected variable is displayed. By default, the entire range of the variable is displayed. The user can impose a lower and/or upper limit for display by clicking the corresponding checkbox and entering the desired limit(s). This is handy for variables that have one or a few extreme values that cause the graph to become unnaturally compressed. Cases outside the specified limits accumulate in a bar at the lower (and/or upper) edge of the display.

The user also specifies the number of bins to use. Making this an odd number causes the graph to be centered at zero if the upper and lower limits are equal, which is good in many cases.

## Density, Inconsistency, and Mutual Information Plots

This family of plots is quite complex, so we begin with the dialog box that appears when this family is selected. First we will discuss the four types of plots available, selected in the lower-right corner of the dialog.



A density plot is a sophisticated version of a scatterplot for showing the relationship between two variables. A scatterplot displays individual cases on a map that uses the horizontal axis for the value of one variable and the vertical axis for the value of the other variable. Each case has a unique position on this map and this position is typically marked with a small dot.

The problem with this approach is that if there are numerous cases, so many that limited display or visual resolution results in multiple cases being



overlaid on the same spot, it is impossible to see the quantity of cases at that spot. Whether one case lies there, or a thousand, it's still a single dot.

The *Actual Density* plot included in *VarScreen* overcomes this difficulty by fitting a bivariate Parzen window to the data and plotting the smoothed density as defined by the Parzen-window density approximation. Much information about Parzen windows is available online, so I won't pursue it in detail here. Roughly speaking, it is a kernel-based exponential smoother whose value at any location is proportional to the number of cases near that location. This plot uses the grayscale tone or the color of a location in the plot to represent the number of cases in the vicinity of this location.

A *Marginal Density* plot will not commonly be used, but it can sometimes be of interest. While the *Actual Density* plot uses grayscale or color to display the density of cases that are actually observed in every region of the domain of the plotted variables, the *Marginal Density* plot uses grayscale or color in the same way to display the *theoretical* density that one would expect for *independent* variables, given the marginal distributions of the variables. Remember that if two variables are unrelated (independent), their bivariate density is the product of their marginal densities. As a rough example, suppose that we are plotting the variables  $X$  and  $Y$ . Suppose there is a 0.1 probability that a case is near  $X=3.2$  and there is a 0.06 probability that a case is near  $Y=2.7$ . Then if these two variables are independent, there is a  $0.1 \cdot 0.06 = 0.006$  probability that a case is simultaneously near  $X=3.2$  and  $Y=2.7$ . A *Marginal Density* plot displays this theoretical relationship.

By visually examining an *Actual Density* plot and a *Marginal Density* plot simultaneously, we can assess the degree and nature of the relationship between the two variables. If the two plots are nearly identical, then there is little or no relationship between the variables. But if the plots differ, we can see the nature of any relationships.

An *Inconsistency* plot can make this comparison and identification of relationships more clear than simply examining two plots simultaneously. A

problem with scatterplots as well as the two density plots just described is that their marginal densities in the display can lead to confusion. If one of the variables clusters at one extreme, most of the cases will be plotted there. In this situation, even if the two variables are unrelated, the map will show a potentially misleading gradient, when what most people are most interested in is departures (inconsistencies) in the actual bivariate density *after taking marginal distributions into account*. As described earlier, we can often see any inconsistencies by visually examining the actual and marginal densities side by side. But there is a better way.

An *Inconsistency* plot overcomes these problems by using the gray level or color of the display to portray the log ratio of the actual density to the theoretical density (product of the smoothed marginals) of cases in every area of the map. It does this by fitting a Parzen smoothing window to the data in order to produce an estimated bivariate density as well as both marginal densities. If the two variables are completely unrelated, the bivariate density will equal the product of the marginal densities everywhere, and so the log of their ratio will be identically zero. Areas in which cases are unnaturally concentrated will have positive values, and unnaturally sparse areas will have negative values. This lets the user see areas of the  $(X, Y)$  domain where there are more or fewer cases than one would expect if the variables were unrelated.

A fourth display possibility involves *mutual information*, the information that is shared between two variables. Mutual information is widely discussed online, so I won't delve into fine details here. Roughly, it is the sum (or integral) over the bivariate domain of the product of the joint density times the log of the ratio of the joint density to the product of the marginal densities. When seen as such a sum, it is clear that if the two variables are related, the contribution to the mutual information contained in the pair of variables varies across the domain, with some areas contributing more than other areas. We can plot this contribution, with the result that areas of the bivariate domain that contribute most greatly to the mutual information are highlighted.

---

The following parameters may be set by the user:

***Horizontal variable*** - Selects the variable plotted along the horizontal axis. By checking the Upper Limit and/or Lower Limit box above the list of variables and filling in a numeric value, the user can limit the range plotted. Cases outside this range still take part in computations, but the plot does not extend to include them.

***Vertical variable*** - Selects the variable plotted along the vertical axis. The same plot limit options as above apply.

***Resolution*** - This is the number of grid locations along both axes that are used for computation. The plot interpolates between the grid points. Larger values result in more detail but longer run time.

***Relative width*** - This is the width of the Parzen smoothing window. The user should set this in accord with the quantity of noise in the data. Smaller values produce very precise density estimates, but if the data contains a large amount of noise, these noise points will be considered valid data and thus make an undue contribution to the display. Larger width values smooth out noise, at the price of less precision in the density estimates.

***Tone shift*** - This controls how the density estimates map to grey levels or color on the display. Positive values will push the display in the positive direction, and negative values in the negative direction. Use this option to highlight desired features.

***Tone spread*** - Like Tone shift, this has no effect on computation, but it controls how the density estimates map to grey levels or colors on the display. Negative values are legal but rarely useful. Positive values, generally less than 10, act to sharpen contrast, emphasizing boundaries near the middle of the range of displayed densities at the expense of blurring detail at the extremes. Use this option to highlight desired features.

***Plot in color*** - By default the display is black and white, with black used for areas of high values of the plotted quantity, and white for low values.. Checking this box causes the display to be shades of blue (low values of the plotted quantity) and yellow (high values).

***Sharpen*** - Increases contrast in the vicinity of extreme values at the expense of discrimination for moderate values.

***Histogram equalization*** - Modifies the display by making all tones/colors appear in equal quantity. This usually emphasizes differences in the mid-range while de-emphasizing extremes.

The user can select from one of four plot types:

***Actual density*** - This plots the actual bivariate density, the most basic plot. This is the smoothed equivalent of a conventional scatterplot.

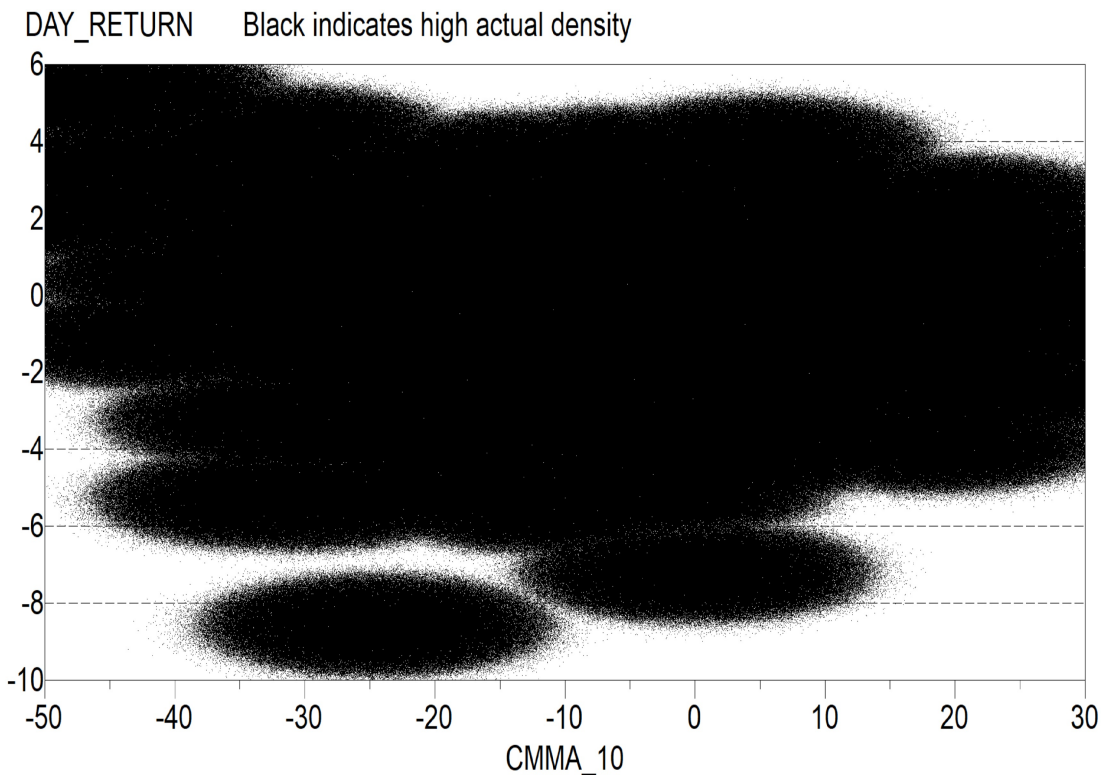
***Marginal density*** - This plots the product of the two marginal densities. This plot is only rarely useful, and then only as a possible comparison baseline to contrast with an actual density.

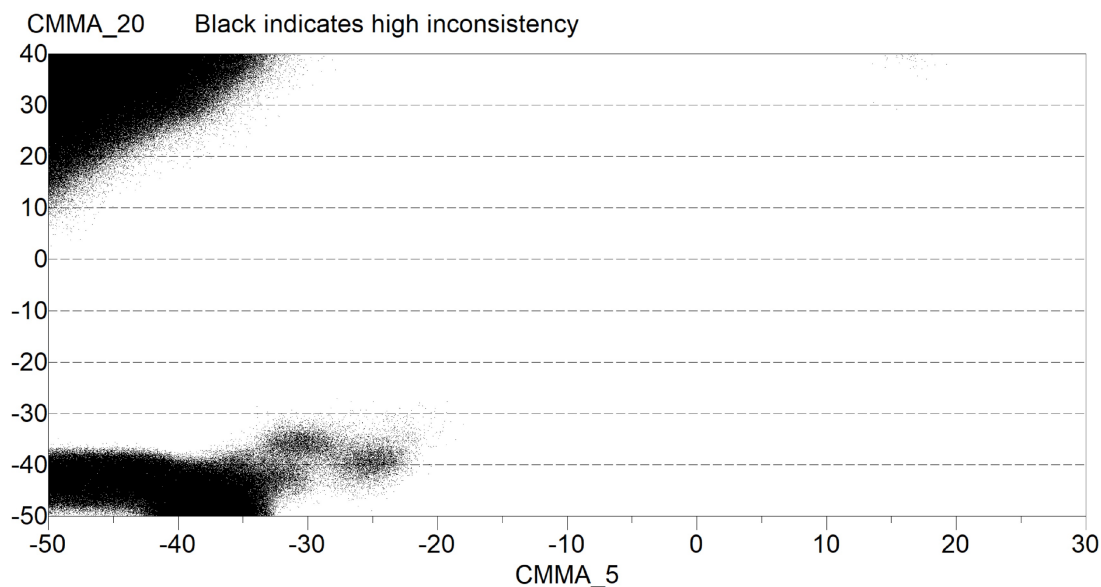
***Inconsistency*** - This plots the inconsistency between the actual bivariate density and the theoretical bivariate density. It is the log ratio of the actual density to the product of the marginal densities. Areas of unusually large actual density are highlighted by being black or yellow versus white or blue.

***Mutual information*** - This shows the contribution of each region to the mutual information between the variables. Regions of the bivariate domain that contribute the most to the mutual information are black or yellow.

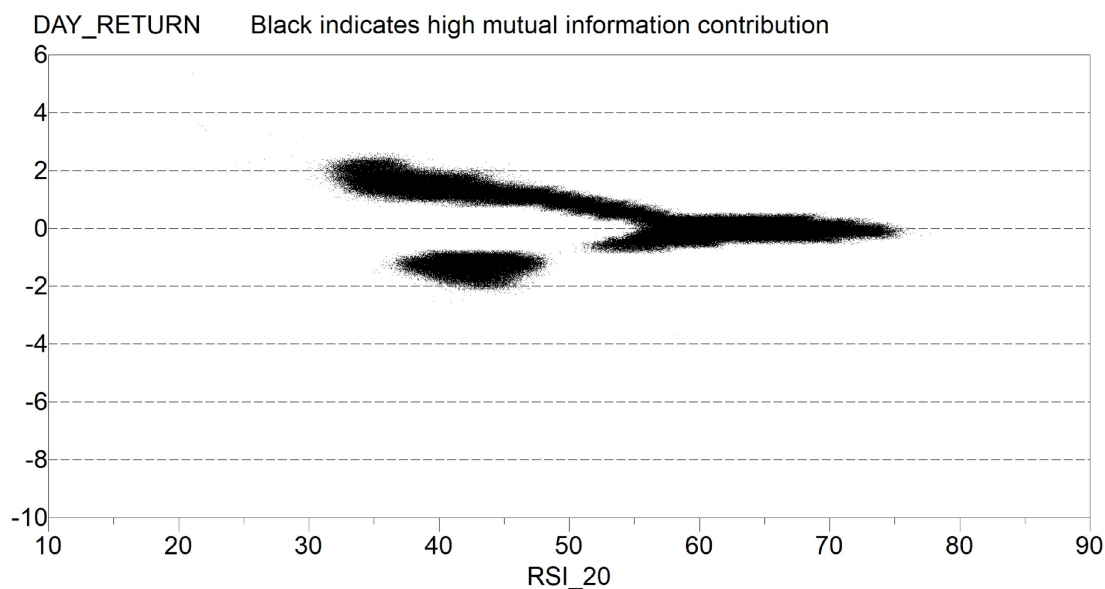
Note that the algorithms for computing the plotted values of these four displays are described in detail in my book "Data Mining Algorithms in C++" published by the Apress division of Springer.

One of my favorite indicators for many years has been the current close minus the recent moving average, normalized for recent volatility. A commonly useful target is the log price return for the next day, normalized for recent volatility. The 'actual density' plot below shows, not surprisingly, that there is not a huge relationship between these variables; if there were, we could get rich quickly by using just this one indicator. But we do see the interesting effect that if this indicator is very small (we just closed far below recent prices), a significant loss the next day is nearly impossible, and the largest gains lie in this situation. This of course is well-known mean reversion after a sharp plunge. But for modest price drops (CMMA\_10 around -25) most of the time we have a wide range of next-day returns. However, a small fraction of the time the price drop continues at least into the next day, with the return less than -8. My guess is that this represents bear markets, where we regularly have strings of down days.





For the CMMA indicator family, we may be interested in where most of their differences in information occur. The plot above shows that their inconsistencies are greatest for small values of CMMA\_5 when CMMA\_20 has an extreme value, positive or negative. Finally, the plot below shows the regions where an ordinary RSI indicator with a 20-day lookback interacts most highly with the next-day return.



---

## Appendix: Version Updates

- 1.0    Univariate mutual information between predictor candidates and a single target  
  
        Bivariate mutual information between a pair of predictor candidates and one or more target candidates
- 1.1    Added the option of uncertainty reduction instead of mutual information for bivariate mutual information, in order to accommodate targets with widely differing entropies
- 1.2    Peng, Long and Ding (2005) “Feature Selection Based on Mutual Information: Criteria of Max-Dependency, Max-Relevance, and Min Redundancy” algorithm implemented to select an optimal subset of predictors based on maximum relevance at predicting the target while simultaneously minimizing redundancy within the predictor set.
- 1.3    Hidden Markov models are defined using up to three predictors, without regard to a target. Then these models are sorted according to the multivariate correlation of their state probability vectors with a user-supplied target variable.
- 1.4    The univariate mutual information test now prints a new column: the probability that a selected candidate will have XVAL out-of-sample mutual information less than or equal to the median out-of-sample mutual information for all candidates.
- 1.5    One or more time series are examined for a break in their mean using the Mann-Whitney U-test. The user specifies how far in recent history to look back for a break. The test includes compensation for examining more than one series simultaneously, as well as compensation for repeating the test as time passes and new values for the series become available.

- 1.6 Feature Weighting as Regularized Energy-Based Learning (FREL): A recent development for feature ranking and weighting that is excellent for low-noise, high-dimension, small-sample-size applications.
- 1.7 Forward selection, as well as optional backward refinement, is used to find a relatively small subset of a very large set of variables such that the principal components of this subset capture the most variance possible from a subset of that size. This is valuable when faced with an extremely large set of predictors.
- 1.8 LFS (Local Feature Selection) for identifying predictors that are optimal in localized areas of the feature space but may not be globally optimal. Such predictors can be effectively used by nonlinear models but are neglected by many other feature selection algorithms.
- 1.81 CUDA computation added to the LFS algorithm, resulting in huge speed increase for problems with a large number of cases.
- 1.82 Fixed a serious non-thread-safe bug in a random number generator. Under certain unusual but possible conditions this could compromise Monte-Carlo permutation test results, especially for cyclic permutation.
- 1.9 Enhanced stepwise selection  
Nominal-to-ordinal conversion  
Assessing memory in a time series by fitting a hidden Markov model
- 2.0 Compute and print table of autocorrelation and partial autocorrelation
- 2.1 Improved mean break test



- 
- 2.2 Threshold optimization allows optimal profit factor to be the indicator selection criterion.
  - 2.3 Bug fix for Optimal Profit Factor introduced in Version 2.2  
Plot time series, histogram, and bivariate densities (and relatives).
  - 2.4 Added to threshold-optimized indicator selection the option of ordered stepwise selection with strongly controlled familywise error.
  - 2.5 Significant acceleration of the stepwise threshold-optimized indicator selection.
  - 2.6 Small bug fix in stepwise option in threshold-optimized indicator selection (monotonicity)
  - 2.7 Implemented stepwise MCPT in univariate mutual information.
  - 2.8 Implemented stepwise MCPT in bivariate mutual information  
Small bug fix for FSCA  
Small bug fix for Nominal-Ordinal conversion
  - 3.0 Fixed inconsistent behavior: Normally, VarScreen.log is placed in the directory that contains the data file read. But if the user's computer had no CUDA device or an outdated video driver, VarScreen.log was placed in the directory that contains the executable. This latter behavior has been eliminated.  
Beginning with this version, the CUDA runtime DLL is no longer needed.
  - 3.1 The RANSAC algorithm trains a linear-quadratic prediction model by iteratively estimating the probability that each case is excessively dominated by noise. The final model is based on only non-noise cases, and performance of predictor candidates is used to score predictors. The stepwise permutation test with fixed familywise error

computes p-values corrected for selection bias. A new variable is created which is the percent probability that each case is not dominated by noise.

- 3.1a Tiny modification to RANSAC algorithm
- 3.3 Mean Reversion testing and direct reading of a market price history file are implemented.
- 4.0 Entropy tests: Shannon, SVD, K2
- 4.1 Add AP (Approximate) entropy test
- 4.2 Bug fix in which Optimal Profit Factor ignored Stepwise option  
Bug fix: removed unused Stepwise option from Optimal Transform  
R-square option in Optimal Transform
- 4.3 Minor bug fix in Mean Reversion
- 4.4 Expand Shannon Entropy for order>1