Target Variables

All of the variables presented to this point have been *indicators*, which in the context of predictive modeling means that they look only at market information that is known up to and including the current (most recently available) bar. Indicators cannot see into the future. *Target variables*, on the other hand, deliberately look into the future as their reason for existing. These are the variables that a predictive model is trained to predict. This section presents the target variables that are available in *TSSB*.

NEXT DAY LOG RATIO

This target variable measures the one-bar change of the market in the immediate future, *roughly* expressed as an annualized percent if the bars are days. The implied trading scenario is that after today's close we make an entry decision for the next day. We enter at the next day's open, and we close the position at the following day's open. Let O_i be the open price i days in the future, where i=0 is today (whose open has passed, since we are at the end of the day when variables for the day are computed). NEXT DAY LOG RATIO is defined as follows:

$$V = 25000 * \ln\left(\frac{O_{i+2}}{O_{i+1}}\right) \tag{7}$$

Note that there is no requirement to be working with day bars in order to use this target. It's just that the normalization by 250 * 100 = 25000 produces an approximate annualized percent return for day bars, which makes for easy interpretability. Also note that the near equivalence of log ratios and percent returns holds only for small price changes. For large price changes the approximation becomes poor, although this is still an excellent target variable, regardless of the degree of price change or the duration of a bar. For example:

```
DAYRET: NEXT DAY LOG RATIO
```

The preceding definition provides an excellent target variable for capturing the price movement from the open of the next bar to the open of the bar after it.

NEXT DAY ATR RETURN Distance

This target variable measures the one-bar change of the market in the immediate future, relative to recent *Average True Range* (*ATR*) taken over a specified

distance. The implied trading scenario is that after the current bar's close we make an entry decision for the next bar. We enter at the next bar's open, and close the position at the following bar's open. Let O_i be the open i bars in the future, where i=0 is the current bar (whose open has passed, since we are at the end of the bar, when variables for the bar are computed). This variable is defined as follows:

$$V = \frac{\left(O_{i+2} - O_{i+1}\right)}{ATR\left(Distance\right)} \quad \textbf{(8)}$$

If *Distance* is specified as zero, the denominator is one so that the value is the actual point return, not normalized in anyway. By normalizing with ATR, we imply that we take a position that is inversely proportional to recent volatility, an action that is common among professional traders. ATR normalization is especially useful in multiple-market applications, because it does an excellent job of ensuring conformity across markets. Without such normalizations, high volatility markets would dominate training of predictive models, with low volatility markets playing little role. For example:

```
DAYRET: NEXT DAY ATR RETURN 250
```

The preceding definition computes the price movement from the open of the next bar to the open of the bar after that, normalized by average true range during the prior year (250 trading days).

SUBSEQUENT DAY ATR RETURN Lead Distance

This is identical to *NEXT DAY ATR RETURN* except that instead of looking one bar ahead, it looks *Lead* bars ahead. For example:

```
DAYRET5: SUBSEQUENT DAY ATR RETURN 5 250
```

The preceding definition computes the price movement from the open of the next bar to the open five bars past that, normalized by average true range during the prior year (250 trading day bars).

NEXT MONTH ATR RETURN Distance

This is identical to *NEXT DAY ATR RETURN* except that the return is computed starting from the first day of the first month following the current day, and ending the first day of the next month after that. If *Distance* is set to zero, the return is the actual point return, not normalized in anyway. This is a highly specialized target

variable, probably not appropriate for many applications.

HIT OR MISS Up Down Cutoff ATRdist

The ATR (average true range) is computed for a history of *ATRdist* bars, and then the future price move is examined up to *Cutoff* bars ahead, beginning at the open of the bar after the current bar. If the price goes up at least *Up* times ATR before going down at least *Down* times ATR, the value of the target variable is *Up*. If the price goes down at least *Down* times ATR before going up at least *Up* times ATR, the value of the variable is minus *Down*. If the price hits neither of these thresholds by the time *Cutoff* bars have passed, the value of the variable is the price change divided by ATR.

If *ATRdist* is set to zero, no normalization is done. The price movement is defined as the actual point return.

This variable has two properties that make it especially attractive for use as a target in predictive modeling:
1) It mimics real-life trading using limit and stop orders.
2) Its distribution cannot have outliers. Such good behavior helps the training process.

For example:

```
HITMISS_2_5: HIT OR MISS 2 5 40 250
```

The preceding definition computes ATR for the most recent year of data (assuming 250 day bars). If during the next 40 bars the price moves up at least twice the ATR before it moves down five times ATR, the value of this target is 2. Conversely, if during the next 40 bars the price moves down at least five times ATR before it moves up twice ATR, the value of this target is -5. If neither bound is hit during those 40 bars, the value of this target is the price change divided by ATR.

FUTURE SLOPE Ahead ATRdist

This is the slope (price change per bar) of the least-squares line looking forward in time *Ahead* bars, divided by ATR (average true range) looking back *ATRdist* bars. For example:

```
FSLOPE: FUTURE SLOPE 20 250
```

The preceding definition fits a least-squares straight line to the next 20 bars after the current bar. It divides the slope of this line by the average true range over the most recent 250 bars to compute the value of this target variable. Note that this target may be useful for a model that is a component of a committee, as it does capture information that is quite different from the information contained in other target variables available in the TSSB library. However, it does not correlate well with real-life trading, so it is not recommended as the sole target in a modeling scheme.

RSQ FUTURE SLOPE Ahead ATRdist

This is the slope (price change per bar) of the least-squares line *Ahead* bars, divided by ATR looking back *ATRdist* bars, multiplied by the R-square of the fit. In other words, RSQ FUTURE SLOPE is identical to the FUTURE SLOPE described above, except that the normalized slope 'FUTURE SLOPE' is then multiplied by the R-square of the linear fit. The effect is to de-emphasize price movements that are noisy and emphasize price changes that are consistent across the *Ahead* time interval. For a discussion of the applicability of this target, see the preceding discussion of FUTURE SLOPE.

Screening Variables

TSSB contains so many indicators in its built-in library that the user can easily feel like a kid in a candy shop, hungrily perusing dozens or even hundreds of enticing candidates. Blithely throwing so many indicators at a model-training procedure is just asking for trouble in the form of overfitting, excessive training time, and needlessly wasted development time.

In order to help solve this problem, *TSSB* contains two different screening procedures. They are both based on contingency tables, the simultaneous partitioning of an indicator and a target into a small number of categories, and computing a measure of the degree to which the category membership of the indicator is related to the category membership of the target. However, they differ in that one method examines each candidate indicator in isolation from the other candidates and is based on individual indicator-target relationships only. The other uses a stepwise procedure to build a set of candidates, and it also considers the relationship between a candidate indicator and the set of indicators already in the set.

The ***indicator-target only*** method (discussed on the next page and called *chi-square tests*) is usually the better choice. It produces a ranked ordering of the indicator candidates which can often allow the developer to partition the candidates into three groups:

- The largest group is typically those candidates which clearly have no useful relationship with the target. This allows fast and easy pruning of the candidate set.

- The smallest group is usually those candidates which have a significant relationship with the target. These indicators should be included in the development procedure with high priority.

- Those candidates which do not fall into either of the above categories are the 'fallback' choices, ignored at first and included later only if the most significant indicators prove insufficient.

The ***relationship-plus-redundancy*** method (discussed here) has much more intuitive appeal, though generally less practical value. It builds a set of indicators which have maximum relationship with the target but which also have minimal relationship with one another. In other words, this procedure identifies a minimal set of indicators which have maximal relationship with the target. The problem with this method is that the set ultimately produced is usually small, denying the

subsequent model a diversity of choices. In the (common) event that the screening algorithm discovers indicator-target relationships that the model is incapable of using, an indicator set that looks wonderful in this procedure may fail miserably when put to use by a model.

Chi-Square Tests

It can be interesting to test a set of individual predictor candidates to discover the degree to which they are related to a target. The following command does this in the most basic version. Several variations are available and will be described later. A mouse-based GUI interface is also available and will be discussed later.

```
CHI SQUARE [Pred1 Pred2 ...] (Nbins) WITH Target (Nbins);
```

The user specifies a list of predictor candidates, the number of bins for predictors (at least 2), a single target, and the number of bins for this target (at least 2).

The program will print to AUDIT.LOG a list of the predictor candidates, sorted from maximum relationship (Cramer's V) tominimum. In addition to printing the chi-square value, it will also print the contingency coefficient, Cramer's V, and a p-value.

The contingency coefficient is a nominal form of correlation coefficient. Note, however, that unlike ordinary correlation, its maximum value is less than one. If the table is square with k bins in each dimension, the maximum value of the contingency coefficient is sqrt ((k-1)/k).

Cramer's V is a slightly better nominal correlation coefficient. It ranges from zero (no relationship) to one (perfect relationship).

The p-value does not take into account the fact that we are doing multiple tests. It is the p-value associated with a single test of the given predictor with the target. An option for taking selection bias (choosing the best indicators from a set of many candidates) into account will be discussed later.

The advantage of a chi-square test over ordinary correlation is that it is sensitive to nonlinear relationships. An interaction that results in certain regions of the predictor being associated with unexpected values of the target can be detected.

Larger numbers of bins increase the sensitivity of the test because more subtle relationships can be detected. However, p-values become less reliable when bin counts are small. As a general rule of thumb, most cells should have an expected frequency of at least five, and no cell should have an expected frequency less than one. But keep in mind that because we are doing multiple comparisons, individual p-values don't mean much anyway. The primary value of this test is the ordering of predictor candidates from maximum target relationship to minimum.

Options for the Chi-Square Test

The prior section discussed the most basic version of the chi-square test. Any or all of several options may be added to the command to extend the test. Suppose we have the following basic chi-square test:

```
CHI SQUARE [ X1 X2 X3 ] ( 3 ) WITH RET ( 3 ) ;
```

The test shown above splits each indicator candidate (X1, X2, X3) into three equal bins, and it does the same for the target, RET. We can replace the number of bins for the indicators with a fraction ranging from 0.0 to 0.5 (typically 0.05 or 0.1) and the word TAILS to specify that the test employ just two bins for the indicator, the most extreme values. The following command says that cases with indicator values in the highest 0.1 (ten percent) of all cases go into one bin, cases whose indicator is in the lowest 0.1 go into the other bin, and the 80 percent middle values of the indicator will be ignored.

```
CHI SQUARE [ X1 X2 X3 ] 0.1 TAILS WITH RET ;
```

Another option is to specify that instead of using equal-count bins for the target, the sign of the target (win/loss) is used. This implies that there are only two target bins: win and lose. (Note that using three bins is often better than splitting at zero, because this splits the target into big win, big loss, and near zero.) We split at zero by replacing the bin count with the word SIGN. The following command does this:

```
CHI SQUARE [ X1 X2 X3 ] ( 3 ) WITH RET SIGN ;
```

Finally, it was already mentioned that the p-values printed with the basic test are individual, appropriate only if you test exactly one indicator. When you test many indicators and look for the best, lucky indicators will be favored. This is called selection bias, and it can be severe, causing huge underestimation of the correct p-value. A Monte-Carlo Permutation Test (MCPT) can be employed to approximately control for selection bias. This is done by appending MCPT = Nreps at the end of the command, as is shown in the following example. Note that if the target has significant serial correlation, as would be for look-aheads more than one bar, the computed p-value will be biased downward.

```
CHI SQUARE [ X1 X2 X3 ] ( 3 )  WITH RET ( 3 ) MCPT=100 ;
```

Finally, we can combine any or all of these options as shown below:

```
CHI SQUARE [ X1 X2 X3 ] 0.1 TAILS WITH RET SIGN MCPT=100;
```

See *Running Chi-Square Tests from the Menu* here, here for more discussion of

these options.

Output of the Chi-Square Test

The following table is a typical output from the chi-square test:

Chi-square test between predictors (2 bins) and DAY_RETURN_1 (3 bins (Sorted by Cramer's V) Tails only, each tail fraction = 0.050

Variable	Chi-Square	Contingency	Cramer's V	Solo pval	Unbiased p
INT_5	30.02	0.225	0.231	0.0000	0.0010
DPPV_10	13.43	0.152	0.154	0.0012	0.2370
DINT_10	12.70	0.148	0.150	0.0017	0.3270
PENT_2	10.59	0.104	0.105	0.0050	1.0000

The heading for this table shows that two bins were used for the predictor (indicator here, though it can be interesting to use targets as predictors!) and three for the target. Only the tails were used for the predictors, the upper and lower five percent.

The indicators are listed in descending order of Cramer's V. This is a better quantity to sort than either chi-square or the contingency coefficient, because it is the best at measuring the practical relationship between the indicator and the target. Chi-square and the contingency coefficient are especially problematic for sorting when the indicator contains numerous ties. Also, the real-life meaning of chi-square and the contingency coefficient is vague, while Cramer's V is easily interpretable because it lies in the range of zero (no relationship) to one (perfect relationship).

The solo pval column is the probability associated with the indicator's chi-square, *when that indicator is tested in isolation*. This would be a valid p-value to examine if the user picked a single indicator in advance of the test. But because we will typically be examining a (frequently large) collection of candidate indicators, and focusing on only the best from among them, there is a high probability that indicators that were simply lucky hold our attention. Their luck will produce a highly significant (very small) p-value and we will be fooled. Thus, it is important that we compensate for the impact of luck. Also note that the solo p-value, like nearly all test statistics, assumes that the cases are independent. This assumption will be violated if the target looks ahead more than one bar.

A Monte-Carlo Permutation Test with 1000 replications was performed, and its results are shown in the last column labeled 'unbiased pval'. The smallest p-value possible is the reciprocal of the number of replications, so the minimum here is

0.001. This unbiased p-value will be correct (within the limitations of a MCPT) for only the indicator having largest Cramer's V. All others will somewhat over-estimate the correct p-value. But that's fine, because we at least know that the true p-value does not exceed that shown. There does not appear to be any way to compute MCPT p-values that are correct for all indicators. Also, the MCPT p-values will be biased downward if the target looks ahead more than one day.

The most important thing to note is that for targets that have little or no serial correlation, the unbiased p-value, which takes selection bias into account, can (and usually does) greatly exceed the solo p-value.

It the user specified that the predictors are to be tails only, the table shown above is followed by a smaller table showing the value of the lower and upper tail thresholds. This table looks like the following:

Variable	Lower thresh	Upper thresh
CMMA_20	-31.6299	35.5345
CMMA_5	-35.1684	37.1704

Running Chi-Square Tests from the Menu

As an alternative to issuing commands in a script file, chi-square tests can be run from the menu by clicking Describe / Chi-Square. The following dialog box will appear:

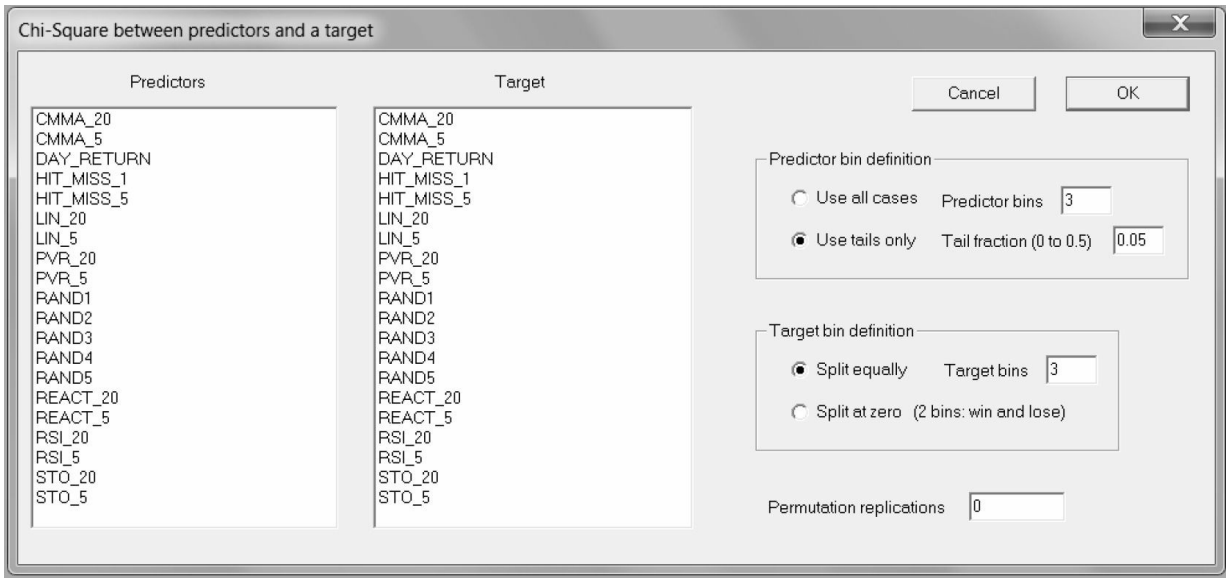

Figure 13: Dialog box for chi-square tests

The user selects one or more predictors (usually indicators, but target variabless can be selected as well) from the left-hand list. Selection can be done in any of

three ways:

1) Drag across a range of predictors to select all of them.

2) Select a predictor, hold down the Shift key, and select another predictor to select all predictors that lie between them.

3) Hold the *Ctrl* key while clicking any predictor to toggle it between selected and not selected, without impacting existing selections.

A single target must be selected.

The user must specify how the predictor bins are defined. There are two choices:

1) Select 'Use all cases' and enter the number of equal-count bins to employ.

2) Select 'Use tails only' and enter the tail fraction on each side (greater than zero and less than 0.5) to employ two bins, one for each tail, with interior values ignored. Values of 0.05 or 0.1 are typical. Keeping more than ten percent of each tail usually results in significant loss of predictive power. The majority of predictive power in most indicators lies in the most extreme values.

The user must specify how the target bins are defined. There are two choices:

1) Select 'Split equally' and specify the number of equal-count bins to employ. In most cases this is the best choice, with three bins used. Three equal-count bins split the target into 'big win', 'big loss', and 'fairly inconsequential' trade outcomes. (This labeling assumes that the target distribution is fairly symmetric, the usual situation.)

2) Select 'Split at Zero', in which case there are two bins, with bin membership defined by the sign of the target. A target value of zero is considered to be a win.

Note that if the user selects 'Use tails only' for the predictor and 'Split equally' for the target, the target bin thresholds are defined by the entire target distribution, not the distribution in the predictor tails only. In most cases this results in a more sensitive test than determining the split points using predictor tails only.

Finally, the user can specify 'Permutation replications' to be a number greater than zero in order to perform a Monte-Carlo Permutation Test of the null hypothesis that all predictors are worthless for predicting the target (at least in terms of the chi-square test performed). If this is done, at least 100 replications should be used, and 1000 is not unreasonable. The smallest computed p-value possible is the reciprocal of the number of replications. The MCPT will produce a column labeled 'Unbiased p' which contains an upper bound (exact for the first predictor listed, increasingly

conservative for subsequent predictors) on the probability that a set of entirely worthless predictors could have produced a best predictor as effective as the one observed. It would be nice to have exact unbiased p-values for predictors below the first (best), but there does not appear to be a practical way of computing this figure. Still, having the p-values be conservative (too large) is better than having them anti-conservative (too small). At least you know that the true p-value is no worse than that shown for each predictor. But note that if the target has significant serial correlation, as would be for look-aheads more than one bar, the computed p-value will be biased downward.

Nonredundant Predictor Screening

The chi-square test described in the prior section is an effective way to rapidly screen a set of candidate indicators and rank them according to their power to predict a target variable. However, it has one disadvantage: indicators can be seriously redundant. This is especially true in regard to the information in the indicators which is related to the target. A statistical study of indicators alone (ignoring a target) may show that they have little correlation. However, if one devises a test which considers only that information component of the indicators that is related to a target variable, the degree of redundancy of this predictive information can be high. Thus we are inspired to consider an alternative test.

The method employed in *TSSB* operates by first selecting from a list of candidates the indicator that has the most predictive power. Then the candidate list is searched to find the indicator which adds the most predictive power to that obtained from the first selection. When a second indicator has been found, a third is chosen such that it adds the most predictive power to that already obtained from the first two. This repeats as desired.

The algorithm just described is in essence ordinary stepwise selection, similar to that obtainable from building predictive models. However, it is different in several ways:

- Because the algorithm is based on segregating the data into bins, it can detect nonlinear relationships without imposing the restrictions of a formally defined model.

- Unlike the models in *TSSB* (and those available in any other statistics packages known to the author), this algorithm allows the option of examining only extreme values of the candidate indicators. It is well known that in most situations, the majority of useful predictive information is found in the tails of indicators.

- The relative simplicity and speed of this algorithm makes it possible to perform Monte-Carlo Permutation Tests of the indicator sets to help decide which predictors and sets have legitimate predictive power and which were just lucky.

The following command does this in the most basic version. Note that it is identical to the syntax of the chi-square test of the prior section, except for the beginning of the command. Several variations are available and will be described later. A mouse-based GUI interface is also available and will be discussed later.

```
NONREDUNDANT PREDICTOR SCREENING
     [Pred1 Pred2 ...] (Nbins) WITH Target (Nbins) ;
```

The user specifies a list of predictor candidates, the number of bins for predictors (at least 2), a single target, and the number of bins for this target (at least 2).

The program will print to AUDIT.LOG a list of the predictor candidates, sorted into the order in which the predictors were selected. This order can be interpreted as saying that the first predictor listed is the single most important, the second predictor is the one that contributed the most *additional* predictive information, and so forth.

Several columns of useful data are printed. These are:

Mean Count is the mean number of cases expected in each cell. This is the total number of cases going into the test, divided by the number of cells. The number of cells for the first (best) predictor is the number of predictor bins times the number of target bins. As each additional predictor is added to the set, the number of cells is multiplied by the number of predictor bins. Obviously, if numerous predictor bins are specified, the number of cells will increase rapidly, and hence the mean per-cell count will drop rapidly. Larger counts produce a better test. If the count drops below five, a warning message will be printed.

100*V is a nominal (category-based) correlation coefficient. It ranges from zero (no relationship) to 100 (perfect relationship). The primary disadvantage of Cramer's V is that it is symmetric: the ability of the target to predict the predictor is given the same weight as the ability of the predictor to predict the target. This is, of course, counterproductive in financial prediction.

100*Lambda is another nominal measure of predictive power which ranges from zero (no relationship) to 100 (perfect relationship). Unlike Cramer's V, it is one-sided: it measures only the ability of the predictor to predict the target. Also unlike Cramer's V, it is proportional in the sense that a value which is twice another value can be interpreted as implying twice the predictive power. However, it has the property that it is based on only the most heavily populated cells, those that occur most frequently. If more thinly populated cells are just noise, this can be good. But in most cases it is best to consider all cells in computing predictive power.

100*UReduc is an excellent measure of predictive power which is based on information theory. The label *UReduc* employed in the table is an abbreviation of *Uncertainty Reduction*. If nothing is known about any predictors, there is a certain amount of uncertainty about the likely values of the target. But if a predictor has predictive power, knowledge of the value of this predictor will reduce our uncertainty about the target. The *Uncertainty Reduction* is the amount by which our

uncertainty about the target is reduced by gaining knowledge of the value of the predictor. This measure is excellent. Like Lambda, it is one-sided, as well as proportional. But like Cramer's V and unlike Lambda, it is based on all cells, not just those most heavily populated. So it has all of the advantages and none of the disadvantages of the other two measures.

Inc pval is printed if the user specified that a Monte-Carlo Permutation Test be performed. For each predictor, this p-value is the probability that, if the predictor truly has no predictive power *beyond that already contained in the predictors selected so far*, we could observe an improvement at least as great as that obtained by including this new predictor. A very small p-value indicates that the predictor is effective at adding predictive power to that already available. Note that if the target has significant serial correlation, as would be for look-aheads more than one bar, the computed p-value will be biased downward.

Grp pval is printed if the user specified that a Monte-Carlo Permutation Test be performed. For each predictor, this p-value is the probability that, if the set of predictors found so far (including this one) truly has no predictive power, we could observe a performance at least as great as that obtained by including this new predictor. A very small p-value indicates that the predictor set to this point is effective. Note that if the target has significant serial correlation, as would be for look-aheads more than one bar, the computed p-value will be biased downward.

The essential difference between these two p-values is that the inclusion p-value refers to predictive power gained by *adding the predictor to the set so far*, while the group p-value refers to the predictive power of the *entire set of predictors found so far*.

The pattern of these p-values can reveal useful information. Suppose a very few predictors have great predictive power and the remaining candidates have none. The inclusion p-values for the powerful predictors will be tiny, and the inclusion p-values for the other candidates will tend to be larger, thus distinguishing the quality. But the group p-values will be tiny for all candidates, even the worthless ones, because the power of the effective few carries the group.

Now suppose that a few candidates have *slight* power, and the others are worthless. As in the prior example, the inclusion p-values for the effective candidates will be tiny, and the inclusion p-values for the other will tend to be larger. But the group p-value will behave differently. As the slightly effective predictors are added to the set, the group-p-value will tend to drop, indicating the increasing power of the group. But then as worthless candidates are added, the group p-value will tend to rise, indicating that overfitting is overcoming the weak predictors.

Remember, though, that when the mean case count per cell drops too low(indicated by a dashed line and warning), the behavior just discussed can change in random and meaningless ways.

By default, *Uncertainty reduction* is used for sorting the selected predictors and for the optional p-value computation. Also by default, the maximum number of predictors is fixed at eight, which is nearly always more than enough. Neither of these defaults can be changed when this test is executed from within a script file, although this should not be a problem because they are excellent choices. They can be changed by the user if this test is performed via the menu system, as discussed later.

Options for Nonredundant Predictor Screening

The options for nonredundant predictor screening are identical to those for the chi-square test described in a prior section. Nevertheless, they will be repeated here for the reader's convenience, and expanded upon later. Suppose we have the following basic test:

```
NONREDUNDANT PREDICTOR SCREENING
      [ X1 X2 X3 ] ( 3 ) WITH RET ( 3 ) ;
```

The test shown above splits each indicator candidate (X1, X2, X3) into three equal bins, and it does the same for the target, RET. We can replace the number of bins for the indicators with a fraction ranging from 0.0 to 0.5 (typically 0.05 or 0.1) and the word TAILS to specify that the test employ just two bins for the predictor, the most extreme values. The following command says that cases with indicator values in the highest 0.1 (ten percent) of all cases go into one bin, cases whose indicator is in the lowest 0.1 go into the other bin, and the 80 percent middle values of the indicator will be ignored.

```
NONREDUNDANT PREDICTOR SCREENING
      [ X1 X2 X3 ] 0.1 TAILS WITH RET ;
```

Another option is to specify that instead of using equal-count bins for the target, the sign of the target (win/loss) is used by splitting the bins at zero. This implies that there are only two target bins: win and lose. (Note that using three bins is often better than splitting at zero, because this splits the target into big win, big loss, and near zero.) We split at zero by replacing the bin count with the word SIGN. The following command does this:

```
NONREDUNDANT PREDICTOR SCREENING
      [ X1 X2 X3 ] ( 3 ) WITH RET SIGN ;
```

It was already mentioned that p-values can be computed. When you test many predictors and repeatedly look for the best to add, lucky predictors will be favored. This is called selection bias, and it can be severe, causing worthless predictors to be selected. A Monte-Carlo Permutation Test (MCPT) can be employed to approximately control for selection bias. This is done by appending MCPT = Nreps at the end of the command, as is shown in the following example:

```
NONREDUNDANT PREDICTOR SCREENING
     [ X1 X2 X3 ] ( 3 )  WITH RET ( 3 ) MCPT=100 ;
```

Finally, we can combine any or all of these options as shown below:

```
NONREDUNDANT PREDICTOR SCREENING
     [ X1 X2 X3 ] 0.1 TAILS  WITH RET SIGN MCPT=100 ;
```

Running Nonredundant Predictor Screening from the Menu

As an alternative to issuing commands in a script file, nonredundant predictor screening can be run from the menu by clicking Describe / Nonredundant predictor screening. The following dialog box will appear:

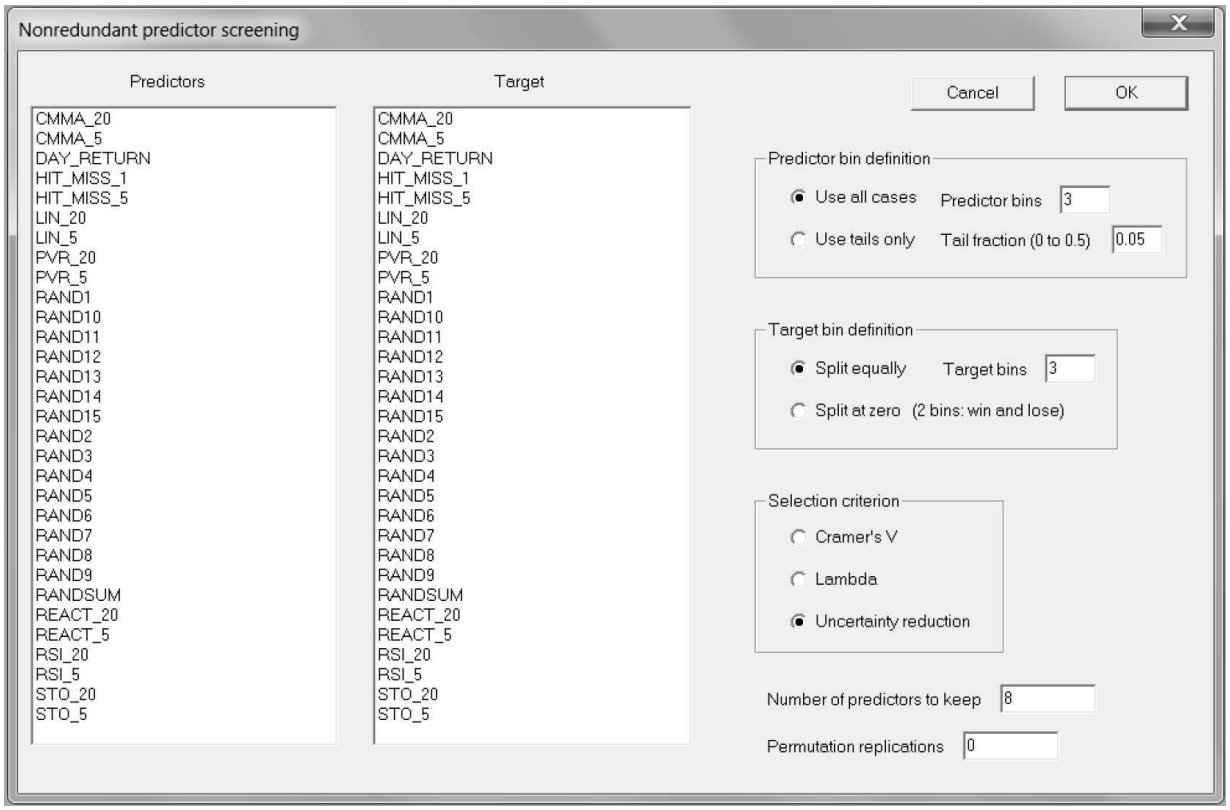

Figure 14: Dialog box for nonredundant predictor screening

The user selects one or more predictors (usually indicators, but target variabless can be selected as well) from the left-hand list. Selection can be done in any of three ways:

1) Drag across a range of predictors to select all of them.

2) Select a predictor, hold down the Shift key, and select another predictor to select all predictors that lie between them.

3) Hold the *Ctrl* key while clicking any predictor to toggle it between selected and not selected, without impacting existing selections.

A single target must be selected.

The user must specify how the predictor bins are defined. There are two choices:

1) Select 'Use all cases' and enter the number of equal-count bins to employ.

2) Select 'Use tails only' and enter the tail fraction on each side (greater than zero and less than 0.5) to employ two bins, one for each tail, with interior values ignored. Values of 0.05 or 0.1 are typical. Keeping too little of each tail usually results in such rapid reduction in the mean per-cell count that very few predictors can be safely selected.

The user must specify how the target bins are defined. There are two choices:

1) Select 'Split equally' and specify the number of equal-count bins to employ. In most cases this is the best choice, with three bins used. Three equal-count bins split the target into 'big win', 'big loss', and 'fairly inconsequential' trade outcomes. (This labeling assumes that the target distribution is fairly symmetric, the usual situation.)

2) Select 'Split at Zero', in which case there are two bins, with bin membership defined by the sign (win/lose) of the target. A target value of zero is considered to be a win.

Note that if the user selects 'Use tails only' for the predictor and 'Split equally' for the target, the target bin thresholds are defined by the entire target distribution, not the distribution in the predictor tails only. In most cases this results in a more sensitive test than determining the split points using predictor tails only.

The user may select which of the three available measures of predictive power is to be used for predictor selection and optional Monte-Carlo Permutation Tests. The default, Uncertaintyreduction, is almost always far superior to the two alternatives, so it should be chosen unless the user wishes to experiment.

The number of predictors to keep defaults to 8. In nearly all cases this is more than enough, as cell count reduction will render the tests meaningless before even 8 are reached, unless there happens to be a huge number of history cases available. The user may set this to any desired quantity. For example, if extensive history of many markets is in the database, increasing this above 8 may be appropriate. The only advantage to specifying a smaller number to keep is that smaller numbers kept will speed a Monte-Carlo Permutation Test.

Finally, the user can specify 'Permutation replications' to be a number greater than one in order to perform a Monte-Carlo Permutation Test. If this is done, at least 100 replications should be used, and 1000 is not unreasonable. The smallest computed p-value possible is the reciprocal of the number of replications. The MCPT will produce a column labeled 'Inclusion p-value' which contains the probability that, if the predictor truly has no predictive power beyond that already contained in the predictors selected so far, we could observe an improvement at least as great as that obtained by including this new predictor. A very small p-value indicates that the predictor is effective. Note that if the target has significant serial correlation, as would be for look-aheads more than one bar, the computed p-value will be biased downward.

Examples of Nonredundant Predictor Screening

We end this discussion with a few progressive examples of nonredundant predictor screening. First, we'll explore how the algorithm behaves in a simple contrived application. We generate a set of random numbers by means of the following commands:

```
TRANSFORM RAND1 IS RANDOM ;
TRANSFORM RAND2 IS RANDOM ;
TRANSFORM RAND3 IS RANDOM ;
TRANSFORM RAND4 IS RANDOM ;
TRANSFORM RAND5 IS RANDOM ;
TRANSFORM RAND6 IS RANDOM ;
TRANSFORM RAND7 IS RANDOM ;
TRANSFORM RAND8 IS RANDOM ;

TRANSFORM RAND_SUM IS EXPRESSION (0) [
   RAND_SUM = RAND1 + RAND2 + RAND3 + RAND4 + RAND5
   ] ;

TRAIN ;
```

The commands just shown generate eight different random series, and then it creates a new series *RAND_SUM* by summing the first five of these eight random

series. Note that the TRAIN command is required because transforms are no computed until they are needed, such as for training, walkforward testing, or cross validation.

Now suppose the following nonredundant predictor screening command appears:

```
NONREDUNDANT PREDICTOR SCREENING
   [ RAND1 RAND2 RAND3 RAND4 RAND5 RAND6 RAND7 RAND8 ](2)
   WITH RAND_SUM (2) MCPT=1000 ;
```

This command says that we will test all eight random series as predictors of a target which consists of the sum of the first five of the predictors. Two bins will be used for the predictors, and two for the target. A Monte-Carlo Permutation Test having 1000 replications will be performed. Because we have contrived this test, we know that we can expect RAND1 through RAND5 to be selected, and the other rejected.

The output produced by this command appears below:

Nonredundant predictive screening between predictors (2 bins) and RAND_SUM (2 bins) with n=5715
Selection criterion is Uncertainty reduction

Variable	Mean count	100*V	100*Lambda	100*UReduc	Inc pval	Grp pval
RAND2	1428.8	32.74	32.73	7.88	0.0010	0.0010
RAND4	714.4	45.72	33.11	16.36	0.0010	0.0010
RAND3	357.2	55.38	50.72	25.33	0.0010	0.0010
RAND5	178.6	63.86	51.17	35.30	0.0010	0.0010
RAND1	89.3	72.74	68.95	47.75	0.0010	0.0010
RAND7	44.6	72.97	68.95	48.23	0.3360	0.0010
RAND6	22.3	73.31	68.95	48.97	0.6130	0.0010
RAND8	11.2	73.90	68.95	50.09	0.9140	0.0010

We see that as expected, RAND1 through RAND5 are selected first, ordered from most to least predictive. Each time one of them is added to the predictor set, the uncertainty about the target is reduced by a considerable amount. Moreover, the inclusion p-value for each of them is 0.001, the minimum possible with 1000 replications. Then, adding the remaining three candidates produces only a minor improvement in the predictability measures. Moreover, the p-values for these last three candidates are insignificant. The group p-values are all tiny because the five effective predictors carry the power of the larger group.

Two comments about these inclusion p-values must be made. First, it is largely coincidental that they increase in this example (0.3360, 0.6130, 0.9140). In the chi-square test discussed in the prior section, the p-values always increase because by

definition the candidates are less predictive as we move down the table. But in this test, the inclusion p-values do not refer to the candidates in isolation. Rather, they refer to the *additional* predictability gained by including each successive candidate. Naturally, there is a tendency for additional power to decrease as the candidate set shrinks. We will be choosing from a less and less promising pool of candidates as the best are taken up. But it is always possible that after some only slightly advantageous predictor is added, its presence suddenly makes some previously worthless candidate a lot more powerful due to synergy. In this situation its p-value will probably be better (smaller) than the prior p-value.

The other thing to remember about p-values, not just in this test but in *any* hypothesis test, is that if the null hypothesis is true (here, the candidate predictor is worthless), the p-values will have a uniform distribution in the range zero to one. It is a common misconception among statistical neophytes that if a null hypothesis is true, this will always be signaled by the p-value being large. Not so. If the null hypothesis is true, one will still obtain a p-value less than 0.1 ten percent of the time, and a p-value less than 0.01 one percent of the time. This gets to the very core of a hypothesis test: if you choose to reject the null hypothesis at a given p-value, and the null hypothesis is true, you will be wrong the p-value fraction of the time.

Now let's modify this test by increasing the resolution. The target will have three bins instead of two, and the predictors will be increased to ten bins. This is accomplished with the following command:

```
NONREDUNDANT PREDICTIVE SCREENING
[ RAND1 RAND2 RAND3 RAND4 RAND5 RAND6 RAND7 RAND8 ](10)
WITH RAND_SUM (3) MCPT=100 ;
```

The output, shown below, is rendered considerably different by this increase in resolution.

Nonredundant predictive screening between predictors (10 bins) and RAND_SUM
(3 bins) with n=5715
Selection criterion is Uncertainty reduction

Variable	Mean count	100*V	100*Lambda	100*UReduc	Inc pval	Grp pval
RAND5	190.5	29.71	22.34	8.31	0.0100	0.0100
RAND3	19.1	44.53	34.57	19.40	0.0100	0.0100
----------> Results below this line are suspect due to small mean cell count <----------						
RAND4	1.9	64.50	53.67	44.49	0.0100	0.0100
RAND1	0.2	93.32	88.08	88.88	1.0000	0.0100
RAND2	0.0	99.78	99.55	99.62	1.0000	0.0100
RAND8	0.0	100.00	100.00	100.00	1.0000	0.0100
RAND6	0.0	100.00	100.00	100.00	1.0000	0.5500
RAND7	0.0	100.00	100.00	100.00	1.0000	1.0000

We see here that the mean count per cell plummets by a factor of ten as each new candidate is added, because the predictors have ten bins. As a result, by the time we get to the third candidate, the count is below the rule-of-thumb threshold of five, so a warning is printed. The basic results are still reasonable: RAND1 through RAND5 are selected before the others, and the addition of each considerably reduces the uncertainty. Adding the final three does almost nothing. However, we do note that although the inclusion p-values for the first three predictors are nicely small, they jump to 1.0 for RAND1 and RAND2, when we would have liked to see them also be small. Unusual behavior also happens with the group p-values. This is a direct result of the diminished mean count per cell, and it is a strong argument for using as few bins as possible.

We now advance to a more practical example involving actual market indicators and a forward-looking target. Here is the command:

```
NONREDUNDANT PREDICTIVE SCREENING
     [ CMMA_5 CMMA_20 LIN_5 LIN_20 RSI_5 RSI_20 STO_5
       STO_20 REACT_5 REACT_20 PVR_5 PVR_20 ] ( 2 )
     WITH DAY_RETURN ( 3 ) MCPT=1000 ;
```

This configuration is one of the three most commonly used. Splitting the target into three bins is generally a perfect choice because the three bins have practical meaning: the trade is a big winner, a big loser, or has an intermediate profit/loss that is relatively inconsequential. The predictor is a little more complex. If an important goal is to capture as many predictors as possible, then using two equal bins as is done here is the best choice. The mean count per cell decreases most slowly with this choice. Another option is to use three equal bins. This captures more information from the predictors, with the price being more rapid decrease in mean count per cell. Finally, the user may opt to use only the tails of the predictors. Since for most indicators, the majority of the predictive information is in the tails, this usually provides more predictive power and produces results that correlate relatively well with model-based trading systems. But if the tail fraction is small, the dropoff in mean count per cell is so rapid that few predictors will be retained.

The command just shown produces the following output:

```
Nonredundant predictive screening between predictors (2 bins) and
      DAY_RETURN (3 bins) with n=5715
      Selection criterion is Uncertainty reduction
```

Variable	Mean count	100*V	100*Lambda	100*UReduc	Inc pval	Grp pval
CMMA_20	952.5	10.14	6.17	0.47	0.0010	0.0010
PVR_20	476.3	8.41	6.17	0.66	0.0010	0.0010
CMMA_5	238.1	9.65	6.96	0.86	0.0140	0.0010
LIN_20	119.1	10.43	7.03	1.02	0.4310	0.0010
STO_20	59.5	11.98	7.74	1.38	0.2530	0.0010
REACT_5	29.8	14.59	9.48	2.09	0.1310	0.0010
RSI_20	14.9	17.56	11.21	3.06	0.4750	0.0010
REACT_20	7.4	20.77	12.65	4.36	0.7230	0.0010
LIN_5	---					
RSI_5	---					
STO_5	---					
PVR_5	---					

By default (changeable only from the menu interface), the best eight predictors are kept. Those not selected are still listed for the user's convenience, but the names are followed by dashes (---). The first two predictors are extremely significant, and the third is also quite significant. Then significance plummets with the fourth predictor. But the first three are so powerful that the group p-value remains highly significant, not suffering from overfitting.

There is a vital lesson in this example: later predictors improved the uncertainty much more than the early predictors, yet their significance is inferior. For example, adding REACT_20 improved the uncertainty reduction from 3.06 to 4.36, yet its p-value was 0.723. This is a common outcome; when numerous predictors are present, adding even a random, worthless predictor which is optimally selected from the remaining candidates can refine the predictive power of the kept predictors to a considerable degree. The inclusion p-value provides a valuable indication of whether the improvement is due to true predictive power or just due to selection bias.

Finally, we'll slightly modify the example above by examining only the tails of the predictors:

```
NONREDUNDANT PREDICTIVE SCREENING
      [ CMMA_5 CMMA_20 LIN_5 LIN_20 RSI_5 RSI_20 STO_5
        STO_20 REACT_5 REACT_20 PVR_5 PVR_20 ]   0.1 TAILS
      WITH DAY_RETURN ( 3 ) MCPT=1000 ;
```

Nonredundant predictive screening between predictors (2 bins) and DAY_RETURN (3 bins) with n=5715
Selection criterion is Uncertainty reduction
Tails only, each tail fraction = 0.100

Variable	Mean count	100*V	100*Lambda	100*UReduc	Inc pval	Grp pval
RSI_20	190.3	21.76	12.38	2.18	0.0010	0.0010
PVR_5	26.2	19.85	10.26	3.82	0.4230	0.0190
REACT_5	6.5	26.15	13.40	6.43	0.6740	0.3160
----------> Results below this line are suspect due to small mean cell count <----------						
PVR_20	1.4	37.37	21.05	12.33	0.0720	0.0300
RSI_5	0.6	37.96	26.67	13.21	0.3670	0.0400
CMMA_20	0.3	37.96	26.67	13.21	0.4470	0.0350
CMMA_5	0.1	37.22	26.92	12.65	0.5740	0.0620
LIN_5	0.1	35.19	24.00	11.09	0.5300	0.0800
LIN_20	---					
STO_5	---					
STO_20	---					
REACT_20	---					

There are several items to note concerning these results:

- Performance measures in this example, which employed only the tails (most extreme values) of the predictors showed predictability very much greater than in the prior example, which used the entire distribution.

- Restricting data to tails caused the mean count per cell to drop much more rapidly than when the entire distribution was used. As a result we can trust this report for only the first three predictors.

- The order of selection is different, indicating that the information in just the tails can be different from the information in the entire distribution.

- Only one predictor had a significant inclusion p-value. Remember that this procedure is different from the chi-square procedure of the prior section. The chi-square procedure tested each indicator separately. On the other hand, the nonredundant predictor procedure here tests them sequentially, evaluating the successive inclusion of more predictors. So what we see here is that the information in the tails of RSI_20 contains all of the predictive power available in the set of candidates. No other candidate's tails can significantly add to the predictive information available in RSI_20's tails.

- The uncertainty reduction obtained using the tails of just RSI_20, which was 2.18 percent, was almost three times as great as that obtained using the entire distribution of all three significant predictors (0.86 percent) in the prior example! So although we pay a high price in mean per-cell count by using tails only, the increase in predictive power with fewer predictors is nearly always a worthy tradeoff.

- Note that it is possible for these predictive power measures to decrease as we add a predictor. This is extremely rare for Uncertainty reduction and Lambda,

essentially a pathological situation. However, it is fairly common for Cramer's V and is a natural part of its definition. Of course, once the cell count becomes small, all calculations become unreliable.

Models 1: Fundamentals

TSSB contains a wide variety of predictive models that can be used to develop automated trading systems and signal filtering systems. Here is a list of the models that are currently available, along with brief descriptions. Each individual model will have its own detailed section later.

Linear regression - Ordinarymultiple linear regression. Many experts consider this to be the best all-around model. It is fast to train, powerful if given well designed indicators, and less likely to overfit than nonlinear models.

Quadratic regression - Fits a quadratic function to the indicators. Linear and squared terms are used, as well as all pairwise cross products. Because this can involve numerous terms in the complete quadratic expression, internal cross validation is used to choose the optimal terms to include. This model is somewhat slower to train than linear regression, but it embodies a nice compromise between nonlinearity and robustness against overfitting. The quadratic terms allow one 'reversing curve' of nonlinearity, which is enough to accommodate most common types of nonlinearity, but not enough to seriously encourage overfitting.

GRNN (General Regression Neural Network)- An extremely powerful nonlinear prediction model. It is capable of handling practically any nonlinearity likely to be encountered in a financial application. It also includes internal cross validation to reduce the possibility of overfitting. Still, overfitting is more likely with this model than with most others. Also, training time is terribly slow, meaning that it is practical only for situations in which there are relatively few training cases. (However, the CUDA option described in the *TSSB* manual allows users with CUDA-enabled Nvidia video cards to use the processing power of the card to speed GRNN training by orders of magnitude.)

MLFN (Multiple-Layer Feedforward Network)- This is the original and still most popular neural network in most circles. One major advantage of this model is that the degree of nonlinearity, which controls the tradeoff between power and likelihood of overfitting, can be easily specified by the user. The principal disadvantage of the MLFN model is that training time can be prohibitively slow.

Tree - This is a very simple model that makes its predictions by means of a

(usually) short series of binary decisions (splits or partitions of the predictor space) based on whether the indicators are above or below trained thresholds. The main advantages of a tree model are that it is extremely fast to train and its resultant decision logic is easy to understand. Sometimes it can be valuable for a user to be able to clearly understand the steps taken by a model to make a prediction. A tree is the ideal model when this is the case. On the other hand, trees are the weakest model in the *TSSB* repertoire, which severely limits their utility.

Forest - A (usually large) collection of independently trained trees whose predictions are pooled to form a group consensus. This is an extremely popular model among some communities of researchers. Its utility for financial modeling is open to question. Also, the process for training a forest employs random numbers, which means that the final model is dependent on the sequence of random numbers spit out by the generator. Thus, results cannot be replicated in subsequent runs, and you have no idea whether the particular sequence employed in a given training operation is 'best' in a reasonable sense. Many people find this distasteful.

Boosted Tree - A collection of trees that are trained using the classic boosting algorithm. Like the forest, the boosted tree has an enthusiastic following among some groups of researchers. Also like the forest, the applicability of the boosted tree for financial applications is open to debate. Still, the basic idea behind the boosted tree is sound: a sequence of trees is found, with each tree's training adjusted in such a way that it focuses on the errors incurred by prior trees. Unfortunately, training a boosted tree can be a very slow process.

Operation String - Combines indicators using a (usually short) string of basic mathematical and logical operations. The biggest advantage of the operation string model is that it is very easy to interpret. For example, an operation string model may say to multiply X_1 by three, subtract seven, and compare the result to X_2 in order to make a trade decision. Its applicability to financial applications is debatable, as it seems to lack the ability to discover subtle patterns that are buried under massive noise.

Split Linear - Consists of two or three separate linear models that may include different indicators. It also includes a gate variable which determines which of the sub-models is employed. The split linear model comes in two different forms. One always makes a prediction using one or the

other of the sub-models according to the value of the gate variable. The other form uses the value of the gate variable to decide whether a trial case is legitimate data or noise.

Overview and Basic Syntax

When the user defines a model in a script file, certain information is required. This includes the following:

Model name - The user must specify a name for a model. The same rules apply as for naming a variable: only letters, numbers, and the underscore character (_) are legal. The name may contain at most 15 characters. The name may not duplicate a variable name or the name of any other model, committee, oracle, or transform. The model name will be used in the audit log when model specifications and performance figures are given. Also, the model name may be used later in the input list for a committee.

Model type - The user must specify the type of model, such as linear regression, forest, et cetera. The following keywords are used for the various model types:

LINREG - Linear regression
QUADRATIC - Quadratic regression
GRNN - General regression neural network
MLFN - Multiple-layer feedforward network
TREE - Tree
FOREST - Forest
BOOSTED TREE - Boosted tree
OPSTRING - Operation string
SPLIT LINEAR - Split linear

Specifications - The user specifies various aspects of the model and its training procedure. Many of these specifications are common to all model types. For example, every model needs to know its input indicators and it output target. These common specifications are presented in the next section. Some models require unique specifications. For example, a multiple-layer feedforward network needs for its hidden layer size to be specified. Unique specifications are covered in individual model sections.

The syntax for specifying a model is as follows:

MODEL *ModelName* **IS** *ModelType* [Specifications];

It is legal to continue a command onto multiple lines. The semicolon terminates the command. Thus, for the sake of clarity, it is common practice (though not required) to place the individual specifications on separate lines. Here is a typical example

of a model definition. The specifications are discussed in the next section.

```
MODEL MOD_A IS LINREG [
   INPUT = [ Var1-Var85 ]
   OUTPUT = TargVar
   MAX STEPWISE = 3
   CRITERION = LONG PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;
```

The example just shown defines a linear regression (LINREG) model that the use names MOD_A. The specifications include the inputs (indicators), output (target) and several items that control training. Clarity is enhanced by placing the closing bracket and terminating semicolon on a separate line after all specifications have been listed.

Mandatory Specifications Common to All Models

Many specifications are applicable to all models. Some of these are mandatory; others are optional. This section discusses model specifications that are mandatory for all models.

The INPUT list

The user must name the indicator(s) that serve as input candidates for the model. This is done with the following syntax:

INPUT = [Variables]

The variables may be listed individually, as in the following example:

```
INPUT = [ X1 X2 X3 ]
```

It is also legal (and often handy) to specify a range of indicators separated by a dash. In this case, the two named indicators, as well as all indicators between them in the variable definition file or database file, are included. It is crucial to understand that it is this order that defines inclusion, not some alphabetic or numeric scheme that may make intuitive sense to the user but which cannot be understood by the literal-minded program. For example, suppose we have the following input statement:

```
INPUT = [ X1 - X3 ]
```

But suppose that the variable definition file (here) or the header for a READ DATABASE file (here) has variables in this order:

X1 X1A X1B X3 X3A X3B X2 X2A X2B X4 X4A X4B

Then the INPUT statement above is equivalent to this one, which may not be intended:

```
INPUT = [ X1 X1A X1B X3 ]
```

It is legal to mix both formats in a single INPUT statement. For example:

```
INPUT = [ X1 X5 X10-X15 X20 ]
```

It is also legal to use family specifiers in an INPUT list. This advanced topic is discussed in the *TSSB* manual.

The OUTPUT Specifier

The user must specify the variable that the model is trained to predict. In most cases, this is a target variable such as one of those discussed starting here. The user names the target variable after the OUTPUT= keyword. For example:

```
OUTPUT = HitMiss_1_1
```

Note that because there can be only one target, there is no need to enclose the name in square brackets, as was done for the INPUT list.

There is one more possibility for an OUTPUT specifier.here we will discuss sequential prediction. In this application, instead of a model predicting a target variable, it predicts the residual error of a prior model. This can be a powerful technique, because one model can do gross predictions, while a second model can be used to tweak the predictions. In this situation, the syntax is as follows:

OUTPUT = **MODEL** *ModelName* **RESIDUAL**

The user names a model that has already appeared in the script file. Then the current model predicts the difference between the true value of the prior model's target and its prediction. See here for more details on this sophisticated and powerful technique.

Number of Inputs Chosen by Stepwise Selection

In most cases, the user's INPUT statement will specify more (often*many* more) indicators than are desired for the final prediction model. The final set of 'optimal' indicators is chosen via stepwise selection, either simple forward selection, or one of the more advanced methods described later. The user specifies the maximum number of indicators to employ via the MAX STEPWISE command. For example the following command tells the program that at most three indicators are to be used in the model:

```
MAX STEPWISE = 3
```

In most cases, the specified maximum number of indicators will be used. However, in some circumstances it can happen that training performance actually deteriorates with the addition of a new indicator. When this happens, addition of indicators will cease and the model will employ fewer than the maximum specified.

If MAX STEPWISE is set to 0, all indicators in the INPUT will be used.

The Criterion to be Optimized in Indicator Selection

It was noted in the discussion of the MAX STEPWISE specification that a subset o the indicators in the INPUT list is chosen for use in the prediction model. The indicators are chosen on the basis of a performance criterion in the training set. The user specifies the performance criterion used for choosing indicators via the CRITERION command. For example, the following command specifies that th indicators be chosen so as to maximize the R-square of the model:

```
CRITERION = RSQUARE
```

The following criteria are available:

RSQUARE - The quantity used to select indicators is the fraction of the predicted variable's variance that is explained by the model. Note that R-square will be negative in the unusual situation that the model's predictions are, on average, worse than guessing the output's mean. If R-square is optimized, the threshold used for computing threshold-based performance criteria (see here and here) is arbitrarily set at zero.

LONG PROFIT FACTOR- The quantity maximized and then used to select indicators is an analog of the common profit factor, under the assumption that only long (or neutral) positions are taken. A decision threshold is simultaneously optimized. Only cases whose predicted value equals or exceeds the threshold enter into the calculation of this criterion. The program sums all such cases (bars) for which the true value of the predicted variable is positive, and also sums all such cases for which the true value of the predicted variable is negative. It divides the former by the latter and flips its sign to make it positive. This is the LONG PROFIT FACTOR criterion. If, in the user application, the predicted variable is actual wins and losses, this criterion will be the traditional profit factor for a system that takes a long position whenever the predicted value exceeds the optimized threshold. It bears repeating that the wins and losses that go into computing the profit factor are those of individual records (bars), with the implied trade duration determined by the definition of the target. This method of computing profit factor is more conservative and accurate than pooling net profits across bars that have the same position. One bar represents one complete trade.

SHORT PROFIT FACTOR - This is identical to LONG PROFIT FACTOR abov except that only short (and neutral) positions are taken. A threshold is optimized, and only cases whose predicted values are less than or equal to the threshold enter into the calculation. Since this criterion

assumes short trades only, a positive value of the predicted variable implies a loss, and conversely.

PROFIT FACTOR- This combines the LONG and SHORT profit factor criteri above. The two (long and short trade) thresholds are simultaneously optimized. If a case's predicted value equals or exceeds the upper threshold, it is assumed that a long position is taken, meaning that a positive value in the target variable implies a win. If a case's predicted value is less than or equal to the lower threshold, it is assumed that a short position is taken, meaning that a positive value in the target variable implies a loss.

ROC AREA - The criterion used to select indicators is the area under the profit/loss ROC (Receiver Operating Characteristic) curve. Thi criterion considers the entire distribution of actual profits and losses relative to predictions made by the model. A random model will have a value of about 0.5, a perfect model will have a ROC AREA of 1.0 and a model that is exactly incorrect (the opposite of perfect) will have a value of 0.0. Note a crucial difference between the ROC AREA criterion and the various PROFIT FACTOR criteria. The ROC ARE looks at the entire range of predicted values, while the PROFIT FACTOR criteria look at only the tail area(s), cases beyond ar optimized threshold. In most financial applications, the most useful information is in the tails, meaning that ROC AREA is usually not very effective.

MEAN_ABOVE_10
MEAN_ABOVE_25
MEAN_ABOVE_50
MEAN_ABOVE_75
MEAN_ABOVE_90 - These criteria are the mean target value for those cases whose predicted values are in the top specified percentage of all cases. The various PROFIT FACTOR criteria previously discusse find an optimal threshold that maximizes the corresponding profit factor, and then use this for selecting the indicators. In contrast, the MEAN_ABOVE criteria do not perform any optimization of th threshold. Instead, the threshold is explicitly computed based on the distribution of predicted values. For example, the MEAN_ABOVE_1 criterion would compute the threshold as the 90'th percentile (100 minus 10) of the predictions in the training set. The indicator having the largest sum of targets whose predictions are at or above this threshold is selected by the stepwise algorithm. Note that the MEAN_ABOVE criteria are currently not compatible with XVA

STEPWISE training (here).

This family is most useful for situations in which the user wants to produce (for a standalone trading system) or keep (for filtering) a large number of trades. For example, when used for a long system, the MEAN_ABOVE_90 criterion would choose indicators based on th mean profit obtained from keeping 90 percent of trading opportunities. Similarly, when used for a short system, the MEAN_ABOVE_1 criterion would choose indicators based on their ability to eliminate the worst 10 percent of trading opportunities. By eliminating the 10 percent of trades that are predicted to produce the largest upward moves (losses in a short system), we likely would improve the performance of the system while still accepting most trades.

MEAN_BELOW_10
MEAN_BELOW_25
MEAN_BELOW_50
MEAN_BELOW_75
MEAN_BELOW_90 - The MEAN BELOW criteria are the *negative* of the mean target value for those cases whose predicted values are in the *bottom* specified percentage of all cases. They are analogous to the MEAN_ABOVE criteria just discussed, except that by flipping the sign of the target mean they make sense for their usual applications. For example, MEAN_BELOW_90 would be useful for a short system i which one wishes to keep 90 percent of all potential trades. MEAN_BELOW_10 would be appropriate for a long system that kee 90 percent of its trades, because this criterion will have a large value when the 10 percent lowest predictions have very negative targets. In other words, MEAN_BELOW_10 would select indicators that ar good at eliminating the worst trades in a long system. Note that each of these MEAN_BELOW criteria has an analog in the MEAN_ABOV family, and the user is free to choose whichever interpretation is more comfortable. For example, suppose you are developing a long trading system and you want to keep most potential trades. You could use MEAN_ABOVE_90, which would base the selection on keeping th 90 percent most likely large winning trades. Or your could use MEAN_BELOW_10, which would focus on eliminating the 10 percer most likely large losing trades. Except for rare pathological situations, these two options would provide the same indicator sets and trading thresholds. But be aware that this same analogous behavior does *not* apply to the PF families discussed next.

PF_ABOVE_10

PF_ABOVE_25
PF_ABOVE_50
PF_ABOVE_75
PF_ABOVE_90 - These criteria are similar to the MEAN_ABOVE criteria. Se that section on the prior page for details. The only difference between the two is that the PF_ABOVE criteria are based on the profit factor o the selected cases, while the MEAN_ABOVE criteria are based o their target mean.

PF_BELOW_10
PF_BELOW_25
PF_BELOW_50
PF_BELOW_75
PF_BELOW_90 - These criteria are similar to the MEAN_BELOWcriteria. Se that section on the prior page for details. The only difference between the two is that the PF_BELOW criteria are based on the profit factor o the selected cases, while the MEAN_BELOW criteria are based o their mean. Note that the PF families do not exhibit the analogous behavior seen with the MEAN families. For example MEAN_ABOVE_90 and MEAN_BELOW_10 are equivalent. E PF_ABOVE_90 and PF_BELOW_10 are different due to profit facto involving ratios, not sums.

BALANCED_01
BALANCED_05
BALANCED_10
BALANCED_25
BALANCED_50 - The BALANCED family of criteria are most useful for th development of market-neutral trading systems. They are valid onlyif the FRACTILE THRESHOLD optionhere) is also used. They are similar to the PROFIT FACTOR criterion here) in that the quantity used to select indicators is the profit factor obtained by taking a long position for cases at or above the upper threshold *and* a short position for cases at or below the lower threshold. The difference is that the PROFIT FACTOR computes optimal upper and lower threshold *separately*, meaning that positions will usually be unbalanced (not market neutral). At some particular date/time, it may be that most or all markets will be long, or most/all short. Such an unbalanced position leaves the trader vulnerable to sudden mass market moves. The BALANCED criteria reduce this risk by guaranteeing marke neutrality. They do this by setting upper and lower thresholds that are balanced. BALANCED_01 decrees that the highest one percent o

market predictions in each time slice will take a long position, and the lowest one percent of market predictions will take a short position. The other variations cover 5, 10, 25, and 50 percent thresholds. For all thresholds except 50, the position is inclusive. In other words, BALANCED_10 will take a long position for any market whose prediction is at or above the upper ten percent threshold, and it will take a short position for any market whose prediction is at or below the lower ten percent threshold. However, this is obviously not possible for the 50 percent threshold, as inclusion would result in markets at exactly the 50 percent fractile being both long and short! Thus, a market that happens to land at exactly 50 percent will remain neutral.

There is one more criterion that can be specified. In the next chapter, here, we will see that the LINREG model (linear regression) allows the user to optionally specify that the indicator coefficients be computed so as to optimize various quantities other than the default R-square. For example, we will see that the coefficients can be chosen so as to maximize the profit factor, or minimize the Ulcer Index. We will often want to make sure that the optimization criterion for stepwise selection of the indicators is the same as that for the indicator coefficients. The user could simply specify them identically for those that are available. However, the following criterion forces this to be the case, thus saving the user the responsibility of doing it explicitly, as well as allowing criteria that are not normally available to stepwise selection:

```
CRITERION = MODEL CRITERION
```

A Lower Limit on the Number or Fraction of Trades

Regardless of the CRITERION specified for selecting indicators, optimal upper (long) and lower (short) decision thresholds are computed for actual trading. These thresholds are always computed separately, and in such a way as to maximize the profit factor. If the user were not to demand that at least a reasonable minimum number of trading opportunities were to be taken, anomalous results would often be obtained. For example, suppose the largest predicted value corresponds to a positive target. The program would be inclined to set the upper threshold so high that only that one trade were taken, resulting in an infinite profit factor. This is obviously a problem.

The solution is for the user to decree that the threshold be set liberally enough that a reasonable quantity of trades are taken in each direction (long and short). There are two ways to do this. Either may be used, but the user must employ one or the other.

They are:

```
MIN CRITERION CASES = Integer

MIN CRITERION FRACTION = RealNumber
```

The first option shown above allows the user to specify the minimum number of cases (trading opportunities) that must be taken *on each side* (long and short). The second option lets the user specify the fraction (0-1) of training set cases that must result in a trade being taken on each side. For example, the following command says that at least ten percent of all training set cases must result in a long trade, and at least ten percent must result in a short trade. A reasonable setting for MIN CRITERION FRACTION is in the range of 0.05 to 0.20. This is in keeping with t notion that most of the information is found in the tails of the prediction distribution.

```
MIN CRITERION FRACTION = 0.1
```

Summary of Mandatory Specifications for All Models

The section that began here and ends here described the specifications that apply to all models and that are required. Each of these items must be specified when a model is defined:

INPUT = ...
The list of indicator candidates

OUTPUT = ...
The output (target) variable that is to be predicted

MAX STEPWISE = ...
The maximum number of indicators that will be used by the model

CRITERION = ...
The criterion that will be used to select indicators

MIN CRITERION CASES/FRACTION = ...
The minimum number or fraction of trading opportunities that must be taken

Optional Specifications Common to All Models

The prior section covered specifications that apply to all models and that are required. This section covers those that apply to all models but that are optional.

Mitigating Outliers

Most models are sensitive to the presence of outliers. If one or a few cases have extreme values for the target (predicted) variable, the model will expend great effort in learning to predict these wild cases, to the detriment of the majority of 'usual' cases. (Outliers in the indicator set are equally harmful, but because indicators are under user control, while targets often are not, the focus is on handling extremes in targets.) It is nearly always in our best interest to tame outlying targets. This can be done in *TSSB* by including the following option in a model definition:

```
RESTRAIN PREDICTED
```

The restraint is done by applying a monotonic compressing function to the tails of the target distribution. Because the transformation is monotonic, order relationships are preserved. In other words, if one were to sort the cases in the order of their target variable, the order would be the same both before and after the compressing transform is applied. Cases with usually large (or small) target values before the compression would still have unusually large (or small) values after the compression, just not so extreme.

This compression does lead to one quandary in regard to interpreting results. Inclusion of the original value of a wild trade will distort performance measures. For example, suppose one happened to have a position open on Black Monday of October 1987. Is it legitimate to include this profit or loss in overall performance figures? In a way it is, because it truly happened. Real money (at least as far as a simulation is concerned) changed hands. Then again, it is inconceivable that a model could have predicted the magnitude of this move. So in this sense, the profit/loss is not legitimate, at least not in its true magnitude. We should probably not simply eliminate the trade, but it would seem reasonable to reduce its impact on total performance figures.

With these thoughts in mind, *TSSB* prints two sets of performance figures when the RESTRAIN PREDICTED option is used. The first set is based on the compressed targets, and the second set is based on the original targets. The user is then free to consider either or both sets of results.

Testing Multiple Stepwise Indicator Sets

By default, ordinary forward stepwise selection is used for indicator selection. First, each individual indicator candidate is tested for its solo performance in the complete training set. The best single performer is chosen. Then, each of the remaining candidates is tested in conjunction with the single indicator already selected. The best candidate for pairing with the first indicator is selected. At this point we have two indicators. Then each of the remaining candidates is tried as a 'third' indicator in conjunction with the two already selected. This process of adding candidates to the indicator set is continued until either the user's MAX STEPWISE criterion is reached or until improved performance in the complet training set is not obtained.

One serious weakness of the basic forward stepwise selection algorithm just described is its assumption that the best indicator set at any step will also be the best indicator set after inclusion of a new variable. Suppose, for example, that we have three indicator candidates and the user wants to use two of them. Suppose X1 and X2 together do an excellent job of predicting the target, but either alone is worthless. If X3 is even slightly good at predicting the target, it will be selected first, and the excellent pairing of X1 and X2 will never be tested. This is not just a theoretical problem; it happens in real life frequently.

In order to alleviate this primacy problem, the user can employ the following option:

```
STEPWISE RETENTION = Integer
```

This option specifies that instead of retaining just the single best candidate set, the program will keep the specified number of best sets. For example:

```
STEPWISE RETENTION = 3
```

The above option will cause stepwise selection to begin by testing all indicator candidates and retaining the best three performers. Then, each of these three best indicators will be paired with each of the remaining candidates (including the other two in the best set of three). The best three pairs of indicators will be retained. Each of these three pairs will be tested with a 'third' indicator, and so forth. Obviously, this will result in much slower operation, because the program is testing many more possible candidate sets. However, because it will be less likely to miss a good combination, performance will likely be improved.

In practice this is almost always an extremely valuable option, well worth the extra training time involved. If the number of indicator candidates is fairly small, one can even set this option to a gigantic number, such as 99999999. By making the number

large enough (it is legal to exceed the maximum theoretical limit on combinations), the user can ensure that every possible combination of indicators is tested. You can't get more thorough than that!

Stepwise Indicator Selection With Cross Validation

It is well known that a model's performance in the training set is optimistically biased relative to what can be expected when the model is presented with new cases. This is because the model will tend to learn idiosyncracies of the training data that are unlikely to be repeated in the future. The real test of a model's ability to predict a target variable comes not from its training set performance, but when the model is shown cases that it has not seen during training. Thus, one may legitimately criticize the process of selecting indicators based on performance in the dataset on which the model was trained. It may be that this performance figure, which has the vital task of indicator selection, is actually measuring the model's ability to predict aspects of the dataset that are noise as opposed to authentic patterns. What we really want to measure when we assess the quality of an indicator set is how well the information conveyed by the indicator set is able to capture authentic patterns, and hence be able to generalize to unseen cases.

This need inspires us to evaluate the performance of an indicator set using cross validation. *TSSB* employs ten-fold cross validation to select indicators when the following option is used in a model definition:

```
XVAL STEPWISE
```

The process works as follows: The first ten percent of training cases are temporarily removed from the training set and the model is trained on the remaining 90 percent of cases. Then this trained model is used to predict the ten percent of cases that were withheld. The predictions are saved. Next, the second ten percent of cases are withheld, the model is trained on the remaining 90 percent of cases, and it is used to predict these newly withheld cases. These predictions are also saved. This process is repeated eight more times, so that every case in the training set has been withheld once, for a total of ten training/prediction cycles. Notice that every case's prediction has been made using a model whose training has not involved that predicted case. As a result, these predictions provide a good indication of how well an indicator set can capture authentic patterns in the data and hence generalize to data it has not seen during training.

It should be obvious that the XVAL STEPWISE option slows training b approximately a factor of ten. Also, this option often fails to improve performance as much as one might hope. If faced with the choice of using STEPWISI

RETENTION or XVAL STEPWISE to improve on default indicator selection, t former option is almost always more useful. Of course, it is legal to use both, although training time might be prohibitive.

When the Target Does Not Measure Profit

We saw here a variety of target variables that are built into *TSSB*. It is also legal to read a target variable from an external database. In most situations, the target can be readily interpreted as a measure of profit. For example, NEXT DAY LOG RATIO and NEXT DAY ATR RETURN have obvious interpretations as the prof from holding a position for the day after a trade decision is made. The HIT OR MISS target variable can also be seen as a measure of profit, with the 'hit' aspect corresponding to the market reaching a limit order and the 'miss' aspect corresponding to the trade being stopped out. But what about FUTURE SLOPE an RSQ FUTURE SLOPE? These measure the future trend direction of the market, b they surely cannot be connected with a numeric profit in any reasonable way.

In this 'non-profit' situation we run into a problem. Several of the performance criteria used to select indicators rely on profit factors. Trading thresholds are computed so as to maximize profit factors. Cross-sectionally normalized returns surely are not profits. And last but not least, many aspects of a model's performance report involve profit-based statistics. Computing profit factors using targets that have little to do with actual profits does not make sense. At the same time, it would be terribly restrictive to insist that a model's target always be a measure of profit. Many useful targets exist that are not profit measures.

This quandary is solved by means of the following option inside a model definition:

```
PROFIT = VariableName
```

The above command is used in addition to the mandatory OUTPUT specification which names the target. The PROFIT command names a variable whose value is assumed to be the profit associated with a trade. For example:

```
OUTPUT = FutSlope
PROFIT = DayAhead
```

If the preceding pair of specifications appears inside a model definition, the model will be trained using the variable FutSlope as the target. MSE (mean squared error) and R-square will be computed using FutSlope, but all other performance measures, because they are profit-based, will be based on the variable DayAhead.

Multiple-Market Trades Based on Ranked Predictions

By default, trade decisions are based on the numerical value of predictions. If the predicted value of a target is at or above the upper threshold, a long position is taken. If the prediction is at or below the lower threshold, a short position is taken. This is usually in accord with the wishes of the trader because it makes fundamental sense: if the model predicts a large value of the target, taking a long position is reasonable. Similarly, if the prediction is strongly negative, taking a short position is sensible.

One arguable problem with this default 'sensible' approach to trading is that slow variations in market behavior can result in clusters of trades. For example, suppose the model is behaving like a long-only trend follower. Then, in a strongly up-trending market the model will trigger numerous long trades. But if the market turns down for an extended period, trades will cease.

Obviously, one could argue that this is exactly the behavior that we want! A long-only system should open numerous positions when the market is moving up, and shut off when the market is retreating. However, some people who are trading multiple markets would prefer to distribute their trading activity more evenly over time. Even in a down market, the user may wish to take long positions in the markets that are predicted to drop less than the other markets. This can be a valuable way to prevent whipsawing: be in the market at the moment of reversal. It may also be part of a market-neutral strategy, in which the trader has long positions open in the markets most likely to rise (or at least not fall as far as the others), and simultaneously have short positions open in the markets likely to fall the most (or at least not rise as much as the others).

It is easy to make trade decisions based on the relative predictions for a set of markets. This is done by ranking the predictions for each market, taking long positions for the highest ranked predictions, and taking a short position for the lowest ranked markets. The model option for doing this in *TSSB* is:

`FRACTILE THRESHOLD`

This option makes sense only if there are multiple markets. When this option appears, the numerical predictions for each market at a given date/time are replaced by a measure of their relative ranking. The market having the maximum predicted target on that date/time will be given a new 'predicted value' of 1.0, and the market having the minimum predicted target will be given a new 'predicted value' of -1.0. Intermediate predictions will be assigned intermediate values according to the formula $prediction = 2 * fractile - 1$. All thresholds, both for internal use and those reported in the log file, are based on these transformed

predictions. The original predictions effectively vanish.

It bears repeating that this option can produce counterintuitive results. For example, on some date/time it may be that all markets are predicted to decline substantially, yet long trades will nonetheless be entered for those markets that are predicted to drop the least.

Restricting Models to Long or Short Trades

By default, all Models, Committees, and Oracles execute both long and shor trades. This allows reporting results three ways: long trades considered alone, short trades considered alone, and net performance of all trades (long and short combined). Reporting all three situations provides the user maximum information, which is obviously good. However, executing both types of trade can be a disadvantage in one situation, because when a Portfolio is used to combine trades from several sources, all trades will be combined in the Portfolio. Why can executing both long and short trades be a problem for a Portfolio? Because sometimes we want to develop one Model/Committee/Oracle that specializes in long trades, and another that specializes in short trades, and then execute only those trades that come from the associated specialist. These two independent trading systems can be combined into a Portfolio. The following options, which can appear in the definition of a Model, Committee, or Oracle, force it to take only trades of one type:

```
LONG ONLY
SHORT ONLY
```

Prescreening For Specialist Models

It is well established that asking a single model to predict the future under a wide variety of conditions is asking a lot of it. Most of the time, better performance can be obtained by splitting the problem domain into two or more subsets and training separate models for each subset. For example, we might want to train one prediction model to handle high volatility epochs, and a different model to handle low volatility. Or we might want to train three models, one for up-trending markets, one for down-trending markets, and one for flat markets. *TSSB* has two ways of handling such specialization. Later, here, we will see how a *Split Linear* model can be used for differentiation among regimes. Here we will discuss a different approach.

If a PRESCREEN specification appears in a model definition, the model will b

trained using only those cases that satisfy the specification. This takes the following form:

```
PRESCREEN VariableName Relationship Number
```

The *Relationship* can be:

<	*Less than*
<=	*Less than or equal*
>	*Greater than*
>=	*Greater than or equal*

For example, the following command would cause the model to be trained on only those cases for which the variable VOLATIL_25 is less than 12:

```
PRESCREEN VOLATIL_25 < 12
```

It is legal to use multiple PRESCREEN commands in a model definition. Fo example, the following pair of specifications would cause the model to be trained on only those cases for which the variable TREND_50 is between -20 and 20 inclusive:

```
PRESCREEN TREND_50 >= -20
PRESCREEN TREND_50 <= 20
```

It is often useful to know how well a set of specialist models performs overall as well as individually. For example, we might have two models based on volatility, one for high volatility and one for low. We can devise a sophisticated type of committee called an Oracle that will call upon these two specialists according to the volatility of each case. This advanced topic is discussed in detail here.

When a model definition includes one or more PRESCREEN specifications, the lo file will report the effects on the data of prescreening separately for every fold of cross validation or walkforward testing. This information may look something like the following:

```
PRESCREEN:
   VAR15 > 0.00000
   8376 of 20643 (40.58 %) of database cases pass
   8023 of 19012 (42.20 %) of training set cases pass
prescreen
```

It echoes the prescreen specification to remind the user. Then it prints two lines. The first line concerns the entire database. It will, of course, be the same for every

fold. The second line concerns the training set used for the current fold.

Building a Committee with Exclusion Groups

The predictions of two or more models may be combined into a single, usually superior prediction by a committee (sometimes called an ensemble of models). Committees are discussed in detail starting here, but the motivation is that combining the predictions of models based on a diversity of information is good. Much as a stock portfolio benefits from including stocks with return streams that are somewhat independent, committees benefit from including models with whose prediction errors have some degree of independence. The model options discussed in this and the next sections encourage the development of a diverse set of models. Many of these component-creation techniques are invoked by means of model options that are common to all model types. One of them, the EXCLUSION GROU option, will be discussed in this section.

It is often the case that the user will have a large number of indicator candidates, dozens or perhaps even hundreds of them. It is likely that quite a few of them carry useful predictive information. However, overfitting (making the model so powerful that it learns patterns of noise in addition to authentic patterns) is always a danger in financial applications, which have a monstrous noise component. Unless the training set is huge and varied, using more than two or three indicators in a model practically guarantees overfitting.

A good way to take advantage of the information contained in many indicators, while still discouraging overfitting, is to train several models, each of which employs just two or three indicators. Then combine the predictions of these models with a committee.

But how should we assign indicators to models? There are an infinite number of possibilities, but one simple method stands out as intuitively justified: use stepwise selection to find the best two or three indicators, and assign them to a model. Then exclude these chosen indicators from the pool and repeat the selection process for a second model. Repeat as desired.

This exclusion process is implemented in *TSSB* with the following command:

```
EXCLUSION GROUP = Integer
```

The specified *Integer* can be any positive integer. The value of the integer itself is irrelevant. Models that have the same EXCLUSION GROUP number will b guaranteed to not share any indicators. For example, consider the following two

models. MOD1 will have the best pair of indicators, and MOD2 will have the nex best pair:

```
MODEL MOD1 IS LINREG [
   INPUT = [ X1 - X48 ]
   OUTPUT = DAY_RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   EXCLUSION GROUP = 1
   ] ;

MODEL MOD2 IS LINREG [
   INPUT = [ X1 - X48 ]
   OUTPUT = DAY_RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   EXCLUSION GROUP = 1
   ] ;
```

You may be wondering why we need to specify an integer. It would seem that a simple keyword such as just EXCLUSION GROUP would suffice to accomplis the task of excluding indicators from successive models. The answer is that this is fine if all we want to do is generate one set of component models that are otherwise identical. But when generating committee components, it can be useful to vary not only the indicators but also the fundamental model type. If we have two different model types that share the same indicators, they will still most likely provide complementary information. For example, suppose we have defined the two LINREG models just shown. We may also generate two GRNN models as show below:

```
MODEL MOD3 IS GRNN [
   INPUT = [ X1 - X48 ]
   OUTPUT = DAY_RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   EXCLUSION GROUP = 2
   ] ;

MODEL MOD4 IS GRNN [
   INPUT = [ X1 - X48 ]
   OUTPUT = DAY_RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   EXCLUSION GROUP = 2
   ] ;
```

A GRNN model is so unlike a LINREG model that even if they have exactly th same indicators, they will still make quite different predictions and hence be useful to a committee. By specifying a different EXCLUSION GROUP for the tw GRNNs (2 for the GRNN; 1 for the LINREG) we allow re-use of the indicator We still forbid the two GRNNs from sharing indicators with each other, but they can share indicators with the LINREGs.

Building a Committee with Resampling and Subsampling

Two other techniques for generating a diversity of models to serve as committee members (Committees are discussed here) are random resampling and random subsampling. As with the EXCLUSION GROUPS specification described in tl prior section, the idea is that we are trying to take advantage of the information provided by numerous indicators while avoiding destructive overfitting.

Resampling and subsampling are similar in that they randomly select cases from the entire training set and use this random set to train a model. This random selection has two beneficial effects:

1) Because the training algorithm for different models is acting on different cases, there is a good chance that different indicators will be chosen, which expands indicator representation in the final committee.

2) Even if some degree of overfitting happens in component models, because the different models are being trained on different training sets they will be exposed to different noise patterns. At the same time, authentic patterns will tend to be represented in all randomly sampled

training sets. So when the predictions of the component models are merged by means of a committee, the predictions due to learned noise will tend to cancel while the predictions due to learned authentic patterns will tend to reinforce.

The difference between resampling and subsampling is that a resampled training set is the same size as the original training set, and repetition of cases is allowed. Subsampling, on the other hand, reduces the size of the training set, and repetition of cases is not allowed. Thus, in a resampled training set, it is likely that some cases in the original training set will not appear, while other cases will appear multiple times. Note that some models, in particular the GRNN, may produce anomalous results if duplicate cases appear in the training set. For this reason, resampling should not be used with a GRNN.

Resampling is invoked with the following specification in the model definition:

```
RESAMPLE
```

Subsampling requires that the user specify the percentage of the training set that is to be kept. This option is invoked with the following specification:

```
SUBSAMPLE Pct PERCENT
```

For example, the following specification would train the model using a randomly selected 66 percent of the original training set:

```
SUBSAMPLE 66 PERCENT
```

Avoiding Overlap Bias

The acid test of a model is not how well it performs in the training set, but how well it performs with data it has not seen during training. To accomplish this, we usually perform cross validation (Pages here and here) or walkforward (here) testing. Both of these testing methods involve training and testing blocks that are adjacent in time. In other words, we will have a group of contiguous cases that are used for training the model, and another group of contiguous cases that are used for testing the model, and these two groups have a common boundary. The case just prior to the boundary line belongs to one group (training or testing) and the case just after the boundary belongs to the other group.

The fact that the training set and the test set adjoin each other is no problem when we are predicting one bar (typically one day) ahead. But if the target variable extends two or more bars into the future, test results will be given an optimistic

bias due to boundary effects.

This bias happens because of the interaction of two phenomena. First, in virtually all applications, the indicators have large serial correlation. This is because indicators are based on recent historical market information, and each successive step forward in time removes the oldest bar from the lookback window and replaces it with the most recent bar. For example, suppose an indicator is a 40-bar trend. When we move ahead in time by one bar, the new indicator value and the prior value will have 39 bars of data in common. The value of this trend indicator will not change much from one bar to the next.

The second phenomenon happens in the future direction. Suppose the target variable is the market change from the open of the next bar to that two or more bars beyond it. Just as was the case for indicators, this target will usually not change much from bar to bar because successive values of the target have one or more market moves in common.

Now consider what happens when the test set follows immediately after the training set and we are testing the first case in the test set. The indicators for this test case will be practically the same as the indicators for the last case in the training set. Moreover, the value of this test case's target will be correlated with the target for the last training case. In fact, if the target looks many bars into the future, the two target values will be practically identical.

The implication is that this test case will have excellent representation in the training set. We would be fine if either the indicators or the targets were unrelated. But the indicators and targets are both similar, so the training process will bias the model toward doing well on test cases near the boundary that have overlap in the future and past.

TSSB allows the user to solve this problem by means of the following option:

```
OVERLAP = Integer
```

When this specification appears in a model definition, the training set is shrunk away from boundaries with the test set by the specified amount. The *Integer* should normally be set to one less than the lookahead distance of the target variable.

For example, suppose the target is SUBSEQUENT DAY ATR RETURN 3 25 This target is the ATR250-normalized market change from the open one bar in the future to the open four bars in the future (a three-bar lookahead). In order to prevent overlap bias, we should use the following specification in the model definition:

```
OVERLAP = 2
```

It's worth analyzing exactly what happens in this example. Suppose we designate time 0 as the first case in the walkforward or cross validation test set. Its target is the market change from the open at time 1 to the open at time 4. The last case in the 'normal' training set prior to this test case is at time -1. But since we have set OVERLAP=2, the actual last training case will be at time -1-2=-3. Its target is th market change from the open at time -2 to the open at time 1. Recall that the target for the test case begins at the open at time 1. Thus, there is no overlap, nor is there any waste. The target computation for the last training case ends at the bar where the target for the test case begins.

The explanation above applies to walkforward testing and to the end boundary of the lower (earlier in time) training set in cross validation. But what about the beginning boundary of the upper (later in time) training set in cross validation? (If this layout is not clear, see here) In this case, let time 0 be the first case in the upper training set as defined by the cross validation algorithm. The last case in the test set is at time -1. Its target is the market change from the open at time 0 to the open at time 3. If we had not used the OVERLAP=2 option, the target for the first trainin case would be the market change from time 1 to that at time 4. But the OVERLAI option pushes the training set beginning two bars into the future, so the first training case is now at time 0+2=2. Its target is the market change from time 3 to time 6. So once again, there is no overlap, and there is no waste. The target for the test case ends at time 3, which is where the target for the first training case begins.

A Popularity Contest for Indicators

Most of the time, the user will specify a relatively large set of indicators as candidates for use by a model. The stepwise selection algorithm will ultimately choose just a few of them as the best performers. If the STEPWISE RETENTIO option (here) is used in the model definition, some indicators may be selected for the 'best set' at some points in the selection process but not make it all the way to inclusion in the final indicator set. Even though these indicators do not make the final cut, their inclusion in intermediate cuts is noteworthy. Also, if cross validation or walkforward testing is done, the training process is repeated multiple times.

TSSB allows the user to request a table of the number of times each candidate indicator was selected during simple training, walkforward, or cross validation. If STEPWISE RETENTION is used, the selection count includes those times indicator was selected for an intermediate 'best set' during training. In order to request this table, include the following specification in the model definition:

```
SHOW SELECTION COUNT
```

This command produces a table similar to that shown below, which lists every candidate predictor and the percent of times it was selected. The table is sorted from most to least popular.

Name	Percent
QUA_ATR_15	11.48
LIN_ATR_7N	11.48
QUA_ATR_15N	10.56
LINDEV_5	7.41
LINDEV_10	7.04
CMMA_5	6.67
CMMA_5N	5.19
LIN_ATR_7	5.00
MOM_5_100	4.63
LINDEV_20	4.26
LIN_ATR_15N	3.89
CMMA_20N	3.15
CMMA_10	2.96
CMMA_20	2.78
CMMA_10N	2.59
MOM_20_100	2.59
MOM_10_100	2.59
LIN_ATR_15	2.04
CUB_ATR_15	1.85
CUB_ATR_15N	1.85

Bootstrap Statistical Significance Tests for Performance

Obtaining decent profit on a walkforward or cross validation test is only half the battle in confirming the effectiveness of a trading algorithm. It's a necessary but not sufficient condition for having confidence in the quality of a trading system. The other requirement is to be confident that the observed results are not due to just good luck.

TSSB makes available several traditional hypothesis test approaches to investigating the issue of good luck. The idea of a hypothesis test is to compute the probability (called the *p-value*) that, if the trading algorithm were truly worthless, results as good as or better than those obtained could be due to luck. If this probability is small, we can be reasonably sure that the profits we saw are legitimate. But if this probability is not small, we should be wary of putting too much faith in the system.

Financial market returns are almost always have highly non-normal distributions, so this disqualifies most common statistical tests which assume a normal distribution. Luckily, there is a common method for approximating this probability in terms of the mean return of the system's trades, while not requiring a normal distribution for the data. A *bootstrap* test assumes that the mean return of the trading system is not better than the mean return of the same number of random trade decisions. This is one definition of 'uselessness' of the system. The bootstrap test then computes the probability, under this assumption, that the superior returns we observed could have been obtained by pure luck.

In order to compute an ordinary bootstrap for the pooled out-of-sample cases, the following specification should be included in the model definition:

```
ORDINARY BOOTSTRAP = Ntrials
```

The user must specify the number of trial samples used to compute the statistic. This should certainly be at least several hundred, and a thousand is more reasonable. Several thousand would not be excessive.

When the above specification appears, several lines of bootstrap results will be printed in the audit log and report log immediately after the usual walkforward or cross validation summary. The first of this information looks something like this:

```
OOS n=8266 Mean=-0.2801 Buy/Hold PF=0.79
Long n=1601 Mean=-0.1563 PF=0.90
Short n=1634 Mean=-0.1294 PF=1.16
```

This information has nothing to do with a bootstrap test. It is just basic statistics on

the data and prediction model. The OOS information concerns the entire out-of-sample dataset, pooled across all folds. The mean of the target and the buy/hold profit factor are printed. The sell/hold profit factor is not printed, but it is the reciprocal of this quantity.
The long section concerns those cases in the walkforward period whose predictions equaled or exceeded the upper trade threshold, the long trades. It was 1601 in this example. The short section concerns those cases whose predictions were less than or equal to the lower threshold, the short trades. It was 1634 here. The bootstrap report then appears as two lines:

```
Ordinary bootstrap Long: Basic p=0.082 Percentile p=0.079
Ordinary bootstrap Short:Basic p=0.991 Percentile p=0.994
```

Results are computed separately for long and short trades. The p-values printed are the estimated probability that the mean target value of the trades executed could be as good as or better than that obtained if the model were truly worthless. There are several algorithms for computing this p-value with a bootstrap. The two most common are called the *basic* method and the *percentile* method. In most but all cases the percentile method is more accurate. Ideally they should be nearly the same. If they are terribly different, the data probably has a strongly skewed distribution, and the validity of any bootstrap test should be questioned. Bootstraps are notoriously vulnerable to highly skewed distributions.

Like most traditional hypothesis tests, a critical assumption of the ordinary bootstrap is that the samples are independent. Many users are concerned about the small but possibly significant serial correlation in financial data. In order to address this, two other bootstrap tests are available in *TSSB*, the stationary bootstrap and the tapered block bootstrap. Many experts consider the latter to be superior to the former, but both are in common use. In order to invoke either or both of these, any or all of the following specifications should appear in the model definition:

```
STATIONARY BOOTSTRAP = Ntrials
STATIONARY BOOTSTRAP ( BlockSize ) = Ntrials
TAPERED BLOCK BOOTSTRAP = Ntrials
TAPERED BLOCK BOOTSTRAP ( BlockSize ) = Ntrials
```

As with the ordinary bootstrap, the user must specify the number of trials, at least several hundred, and ideally a thousand or more. The user may also optionally specify the block size, enclosed in parentheses. This should be done if the user has a good estimate of the number of bars for which serial correlation is significant. The block size is this number of bars. For example, you could use the following specification if you believe that serial correlation becomes insignificant after ten bars:

```
STATIONARY BOOTSTRAP ( 10 ) = 1000
```

In many cases, however, it is best to let *TSSB* analyze the correlation and automatically compute the optimal block size. This is because a size that is too small will fail to compensate for serial correlation, and a size that is too large will weaken the power of the test. The program will print the automatically computed block size as part of the bootstrap report.

Monte-Carlo Permutation Tests

The bootstrap tests described in the prior section have two significant disadvantages: they address only a single statistic, the mean return per trade, and they are more vulnerable to skewed distributions than their proponents like to admit. Admittedly, bootstrap tests can be adapted to test statistics other than the mean, but such adaptations are often suspect.

This section describes an alternative to the bootstrap: the Monte-Carlo Permutation test. This test works equally well for *any* performance statistic, and it is not impacted by any characteristic of the data distribution, including skewness. Unfortunately, it has two disadvantages of its own: no version of this test yet exists for handling serial correlation in the data (bootstraps have stationary and tapered-block versions to accomplish this), and the validity of Monte-Carlo permutation tests has not yet been confirmed with rigorous mathematical proofs. This family of tests is relatively new and not well studied.

In order to invoke the suite of Monte-Carlo permutation tests for the pooled out-of-sample cases, include the following specification in the model definition:

```
MCP TEST = Ntrials
```

As with a bootstrap, the user specifies the number of trial replications, which should be at least several hundred and ideally should be a thousand or more.

A table of p-values resembling the following will be printed after walkforward or cross validation is completed. As with the bootstrap, this table concerns the pooled out-of-sample cases. If the user has also specified the RESTRAIN PREDICTE option (here), this table will be printed for both the restrained and unrestrained targets.

```
Monte-Carlo Permutation Test p-vals...
   R-square: 0.6700
   ROC area: 0.8400
   Profit factor: 0.0000
      Long only: 0.0400
      Short only: 0.0000
```

In many applications, the p-values associated with R-square and ROC area are o little or no interest. It is the profit factor that is the best performance measure, so users should focus on these three values. The *Profit Factor* line is for the net trading result, taking both long and short trades into account. This is then broken down into separate long and short components. The values printed are the hypothesis test p-values, the probability that a truly worthless model would have performed as well as or better than that obtained.

An Example Using Most Model Specifications

The prior two sections discussed the mandatory and the optional specifications that are applicable to all models. We now present a model definition that includes most of these options, all that are possible without conflict. Most model definitions will not be anywhere near this long and complex, but this is shown so that the user can have a concrete example of how to specify options.

```
MODEL ManySpecs IS LINREG [                          here
    INPUT = [ Var1 - Var10 ]                         here
    OUTPUT = TARGET                                  here
    RESTRAIN PREDICTED                               here
    FRACTILE THRESHOLD                               here
    PRESCREEN VAR15 > 0.0                            here
    MAX STEPWISE = 2                                 here
    STEPWISE RETENTION = 10                          here
    XVAL STEPWISE                                    here
    CRITERION = LONG PROFIT FACTOR                   here
    MIN CRITERION FRACTION = 0.1                     here
    EXCLUSION GROUP = 1                              here
    SUBSAMPLE 80 PERCENT                             here
    OVERLAP = 10                                     here
    SHOW SELECTION COUNT                             here
    ORDINARY BOOTSTRAP = 1000                        here
    STATIONARY BOOTSTRAP = 1000                      here
    TAPERED BLOCK BOOTSTRAP = 1000                   here
    MCP TEST = 1000                                  here
    ];
```

Sequential Prediction

It can be unrealistic to ask a single model to handle the complexities of a complicated prediction. One major problem is that if the model is made powerful enough to handle a complex pattern, the model will also be vulnerable to overfitting; it will be likely to 'learn' random noise along with legitimate patterns in the data. By definition, noise in the data will not be repeated when the model is put to use, so its performance will suffer. Another problem is that powerful models almost always take longer to train than simple models, often so much longer that they become impractical.

One approach to solving this problem is to use several models to predict the same target, and then combine these predictions using a committee. This is discussed here.

Another approach, sequential prediction, is based on the idea that in many applications, the overall predictable pattern in the data is made up of two or more sub-patterns. We may have a relatively simple pattern that dominates the relationship between the indicator(s) and the target. Beneath this relationship we may have a more subtle relationship that involves different indicators and a different form of the model. If we were to try to find one grand model that incorporates the dominant pattern and its indicators as well as the more subtle pattern and its indicators, wewould probably end up with a model so complicated that it would take a long time to train, and it would overfit the data.

A much better approach would be to use one simple model to predict the dominant pattern. Then we find the errors that this model makes, the differences between its predictions and the target. These errors will be made up of the more subtle pattern along with noise. We use a simple model to learn these differences. In other words, the target for this second model is not the original target. Rather, it is the *difference* between the original target and the predictions of the first model, the model that handles the dominant pattern. Then, when we need to make a prediction of the original target, we invoke both models and add their predictions. One component of this sum is the predicted value of the dominant pattern, and the other component is the model's prediction of how much the dominant prediction deviates from the original target. This latter prediction is the 'touch-up' value, the more subtle pattern that lies beneath the dominant pattern.

TSSB allows a practically unlimited number of such sequential predictions. One can continue predicting successive differences many times. However, in most cases using more than two or three models sequentially is pointless. Even though each model is (ideally) simple and unlikely to overfit on its own, the point soon comes that the error term being predicted by the next model is just noise, not authentic

patterns. Thus, although sequential prediction is more robust against overfitting than a single comprehensive model, it is not immune to the problem.

We now present a simple demonstration of three-stage sequential prediction. Here are the three model definitions, using just the minimum model specifications. The RESIDUAL output specification was discussedhere. The first model, SEQ 1, directly predicts the target variable, DAY_RETURN_1. The second model, SEQ2, predicts the residual of the first model, the difference between the first model's predictions and DAY_RETURN_1. The third model, SEQ3, predicts the residual between the sum of the predictions of the first two models and the original target, DAY_RETURN_1. When the program makes a prediction of the original target DAY_RETURN_1, it adds the predictions of these three models. This sum incorporates the gross prediction made by model SEQ1, a tweak to that prediction made by SEQ2, and a tweak to the tweak, made by SEQ3.

```
MODEL SEQ1 IS LINREG [
   INPUT = [ CMMA_5 - VMUTINF_3 ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;

MODEL SEQ2 IS LINREG [
   INPUT = [ CMMA_5 - VMUTINF_3 ]
   OUTPUT = MODEL SEQ1 RESIDUAL
   MAX STEPWISE = 2
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;

MODEL SEQ3 IS LINREG [
   INPUT = [ CMMA_5 - VMUTINF_3 ]
   OUTPUT = MODEL SEQ2 RESIDUAL
   MAX STEPWISE = 2
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;
```

Here is an extract from the log file produced by running a script file in which these three models are trained:

```
LINREG Model SEQ1 predicting DAY_RETURN_1
Regression coefficients:
         -0.002581  QUA_ATR_15
         -0.003445  DAU_MIN_32_2
          0.036100  CONSTANT
MSE = 0.64428   R-squared = 0.00565   ROC area = 0.55308
Buy-and-hold profit factor = 1.142   Sell-short-and-hold =
0.876
Dual-thresholded outer PF = 1.342
   Outer long-only PF = 1.586   Improvement Ratio = 1.389
   Outer short-only PF = 1.159  Improvement Ratio = 1.323


LINREG Model SEQ2 predicting DAY_RETURN_1
This model predicts the residual of model SEQ1
Regression coefficients:
          0.001510  PSKEW_10
          0.001711  IDMORLET_10
          0.006449  CONSTANT
MSE = 0.64311   R-squared = 0.00745   ROC area = 0.56407
Buy-and-hold profit factor = 1.142   Sell-short-and-hold =
0.876
Dual-thresholded outer PF = 1.533
   Outer long-only PF = 1.608   Improvement Ratio = 1.408
   Outer short-only PF = 1.396  Improvement Ratio = 1.594


LINREG Model SEQ3 predicting DAY_RETURN_1
This model predicts the residual of model SEQ2
Regression coefficients:
         -0.002006  DCKURT_20_4
          0.001901  VMUTINF_2
         -0.026775  CONSTANT
MSE = 0.64212   R-squared = 0.00897   ROC area = 0.57058
Buy-and-hold profit factor = 1.142   Sell-short-and-hold =
0.876
Dual-thresholded outer PF = 1.620
   Outer long-only PF = 1.616   Improvement Ratio = 1.416
   Outer short-only PF = 1.627  Improvement Ratio = 1.857
```

It's important to note that all performance figures given in the log file, including basic statistics such as MSE and R-square, are based on predicting the original target, not the residual. For example, model SEQ2 predicts the quantity DAY_RETURN_1 minus the prediction of model SEQ1. But the user never sees the MSE or R-square of this actual model, based on predicting the residual of SEQ1. Rather, model SEQ2 makes its predictions and each prediction is added to the corresponding prediction of SEQ1 to get a 'tweaked' prediction of DAY_RETURN_1. This sum is used to compute MSE, R-square, and all other performance figures for SEQ2. This makes the report more meaningful than it would be if based on the residuals. In fact, profit factors would be meaningless without incorporation of the gross prediction from SEQ1, as it would be silly to

apply a threshold to a residual!

It's interesting to examine the performance figures for these three models. Note that they all improve for successive models. The R-squares for SEQ1, SEQ2, and SEQ3 are 0.00565, 0.00745, and 0.00897, respectively. This should not be surprising, because each model is 'improving' on the prior model. MSE similarly decreases. Although there is no mathematical guarantee that profit factors will also improve, we see that in this case they do, which is the nearly universal result. Of course, these are all in-sample figures. If we did a cross validation or walkforward test, the out-of-sample results might tell a different story!

Models 2: The Models

TSSB contains an effective variety of predictive models that can be used as the basis of trading and filtering systems. The prior chapter provided brief summaries of the available models as well as detailed descriptions of the options that apply to all models. This chapter discusses each model in greater detail, and it presents those options that are unique to each model.

Linear Regression

Many experts consider linear regression to be the best all-around model for financial applications. There is a popular idea that predictable market relationships are so highly nonlinear that effective prediction requires a nonlinear model. However, much practical experience indicates that there may be a common flaw in this idea: most nonlinearities occur near the extremes of indicators, and even then they are monotonic. Thus, thresholds applied to simple linear combinations of such indicators are sufficient to capture most predictive information. Of course, this is not universal. There is a place for nonlinear models in the prediction of financial markets. However, the advantages of linear regression are so overwhelming that it should be strongly considered. It is fast to train, it is much less likely to overfit than other practical models, and because of its simplicity its actions are easier to interpret than most nonlinear models.

The keyword used to define a linear regression model is LINREG. Thus, we might define a linear regression model by beginning with a line like the following:

```
MODEL MyFirstModel IS LINREG [
```

The line above must be followed with all of the mandatory specifications described in the prior chapter here. Optional specifications may also appear. One optional specification that is unique to LINREG models is described in the next section.

The MODEL CRITERION Specification for LINREG Models

By default, the indicator coefficients are computed in the traditional linear regression manner, by a least-squares fit. In other words, the coefficients are computed such that the mean squared error (MSE) between the target and the prediction is minimized. However, it is well known that this criterion is often not the best for automated trading of financial markets. One obvious example of why

minimizing MSE is problematic concerns extreme values of the target. The operation of squaring emphasizes large errors. Thus, a training operation that minimizes MSE will expend enormous effort to accommodate any cases that exhibit an unusually large win or loss, to the detriment of the 'average' cases that make up the bulk of trading opportunities. The trading system may not even be able to take advantage of the entire large market move, in which case the ability to detect extreme moves is of little value. Even if the trading system could take advantage of the full move, such unusual moves are nearly always unpredictable random events, so the model will have little chance of being able to generalize this predictive ability outside the training set. The bottom line is that computing 'optimal' indicator weights by minimizing MSE may not be the best approach when the model will be used for trading financial markets.

Because of this fact, *TSSB* contains a variety of LINREG optimization criteria that are often more appropriate for market-trading applications. Unfortunately, optimizing these criteria is tremendously slower than minimizing MSE. These options may be impractical when the training set contains a huge number of cases and stepwise selection must investigate a large number of indicator candidates. Also, the use of the STEPWISE RETENTION or XVAL STEPWISE options may be inadvisable when any of these special optimization criteria are employed, as they, too, substantially increase training time.

The MODEL CRITERION option should not be confused with the mandatory CRITERION specification described in the prior chapter here. They have entirely different purposes. The CRITERION specification controls selection of indicators in the stepwise selection process. It is mandatory and applies to all models. The MODEL CRITERION option applies only to linear regression (LINREG) models. It is optional, and it controls computation of the weights associated with the chosen indicators. One rough way of looking at the situation is that CRITERION controls which indicators are chosen, and MODEL CRITERION controls how the chosen indicators are used (weighted) in the model.

The following MODEL CRITERION specifications are available:

MODEL CRITERION = LONG PROFIT FACTOR - The long-trades-only profit factor is maximized. See the description of the CRITERION = LONG PROFIT FACTOR specification here for details on how the computation is done.

MODEL CRITERION = SHORT PROFIT FACTOR - The short-trades-only profit factor is maximized. See the description of the CRITERION = SHORT PROFIT FACTOR specification here for details on how the computation is done.

MODEL CRITERION = PROFIT FACTOR The net (long plus short) profit factor is maximized. See the description of the CRITERION = PROFIT FACTOR specification here for details on how the computation is done.

MODEL CRITERION = LONG ULCER INDEX (Equity) The long-trades-only Ulcer Index is minimized. The user must specify an initial equity in parenthese, which greatly impacts the result. It is important that the initial equity be large enough so that a series of losing trades never drops the running equity to zero. It is commonly accepted that very large values of the initial equity provide the most stable performance. The Ulcer Index is the square root of the mean squared drawdown. As of this writing, a detailed explanation, including formulas and rationale, can be found on Wikipedia as well as the original source: http://www.tangotools.com/ui/ui.htm.

MODEL CRITERION = SHORT ULCER INDEX (Equity) This is identical to the LONG ULCER INDEX except that only short trades are considered.

MODEL CRITERION = ULCER INDEX (Equity) This is identical to the LONG ULCER INDEX except that all trades (long and short) are considered.

MODEL CRITERION = LONG MARTIN RATIO (Equity) TheMartin Ratio is the net change in equity (ending equity minus initial equity) divided by the Ulcer Index (discussed above). This quantity is maximized for long trades only. Short trades are ignored.

MODEL CRITERION = SHORT MARTIN RATIO (Equity) This is identical to the LONG MARTIN RATIO except that only short trades are considered.

MODEL CRITERION = MARTIN RATIO (Equity) This is identical to the LONG MARTIN RATIO except that all trades (long and short) are considered.

The Identity Model

Sometimes the user needs to treat an indicator as if it itself is the output of a model. In other words, the user wants *TSSB* to base trade decisions on the value of this indicator, with no transformations of any sort. Optimal trade thresholds are to be

computed automatically. Stepwise selection of the single best such indicator may even be desired. This is accomplished by defining a LINREG model and including the following option:

```
FORCE IDENTITY
```

When this option appears in the LINREG model specification list (it may not be used with any other type of model) no training is done. Instead, the indicator's weight is set to 1.0 and the constant offset (Y intercept) is set to 0.0. The result is a linear model that does nothing but pass through its input, the value of the indicator, to its output.

Here is a simple example of this technique. Suppose we have five indicators, X1 through X5, and we want to choose one of them for making trade decisions for a long-only system. Our plan is to find an optimal threshold and place a trade when the observed value of the chosen indicator is on the appropriate side of the threshold. For this example, suppose we do not know in advance whether it is unusually large or unusually small (perhaps large negative) values of this indicator that should trigger a trade. In this situation we need to also have five more indicators, the negatives of the original indicators. This is because *TSSB* opens a long trade when the 'predicted value' of the target is at or above the long threshold. Since under the FORCED IDENTITY option, the value of the indicator is treated as the predicted target, if we are to test both sides of the threshold we need both signs. Otherwise, long trades would be entered only for unusually large values of the indicator. We'll call these negative indicators NEG_X1 through NEG_X5. Here is a reasonable way to handle this situation:

```
MODEL IND_MOD IS LINREG [
   FORCE IDENTITY
   INPUT = [ X1 - X5  NEG_X1 - NEG_X5 ]
   OUTPUT = ScaledReturn
   MAX STEPWISE = 1
   CRITERION = LONG PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;
```

Note that we specified MAX STEPWISE=1. This is what we almost always want to do. However, it is legal to include more than one indicator. In this case, all indicator coefficients are set to 1.0, resulting in the 'prediction' being their sum. This would be reasonable only if all indicators have similar natural scaling. For example, RSI has a natural scaling of 0 to 100 while CLOSE MINUS MOVING AVERAGE has a natural scaling of -50 to +50. This is perfect compatibility because both have a range of 100. On the other hand, if one indicator has a range of 1000 and the other has a range of 0.01, their sum would be meaningless; the value of the first would dominate the value of the second, and so the second would be

practically ignored.

Quadratic Regression

In all modeling tasks, there is an inherent tradeoff regarding the degree of nonlinearity. By definition, linear models can capture only linear relationships between indicators and targets. This is a great advantage because it discourages overfitting, which is usually due to the excessive nonlinear modeling of noise. On the other hand, sometimes the relationship between the indicators and the target is significantly nonlinear. In this unfortunate situation, we must use a nonlinear model if we are to capture authentic predictive patterns. Nevertheless, we should ideally use a model that has as little nonlinearity as possible.

Quadratic regression is often an excellent compromise between strict linearity (no bends in the prediction function) and massive nonlinearity. Loosely speaking, quadratic models allow only one reversal of direction in the predictive function in each dimension, which helps avoid serious overfitting. At the same time, much experience indicates that a single bend is enough to handle the majority of nonlinear relationships encountered in financial modeling. Furthermore, although quadratic regression is a lot slower to train than ordinary linear regression, it is still much faster than most other nonlinear models. For this reason, if LINREG linea regression fails to perform well, quadratic regression should be the next choice.

The keyword used to define a quadratic regression model is QUADRATIC. Thus we might define a quadratic regression model by beginning with a line like the following:

```
MODEL MySecondModel IS QUADRATIC [
```

A quadratic model is in essence a linear model with additional inputs that consist of all squares and cross products of the original indicators. Here are three simple examples showing the original input(s) to the left of the arrow and the terms that actually are available to the linear model. A constant term is also included in all cases, but not shown here. These examples are for one, two, and three inputs, with the indicators called A, B, and C.

$A \rightarrow A\ A^2$
$A\ B \rightarrow A\ B\ A^2\ B^2\ AB$
$A\ B\ C \rightarrow A\ B\ C\ A^2\ B^2\ C^2\ AB\ AC\ BC$

It should be obvious that the number of terms in the ‘quadratically expanded linear model’ blows up as the number of indicators rises. With just three inputs, we already have nine terms plus the constant. This could result in massive overfitting if

something were not done about the situation. Remember, a quadratic model is just an ordinary linear model in which the original inputs are supplemented by squares and cross products of the original inputs. Thus, even though a linear model is relatively unlikely to overfit a dataset, the sheer number of indicators presented to a linear model by quadratic expansion can itself result in overfitting. *TSSB* uses internal (invisible to the user) forward stepwise selection and ten-fold cross validation to find the optimal subset of expanded predictors.

Here is an example QUADRATIC model, followed by its trained coefficients as set forth in the audit log:

```
MODEL VAL1 IS QUADRATIC [
   INPUT = [ LIN_ATR_5 - CUB_ATR_15 ]
   OUTPUT = RETURN
   MAX STEPWISE = 3
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;


QUADRATIC Model VAL1 predicting RETURN
Stepwise based on R-Squared  (Final best crit = 0.0023)
Standardized regression coefficients:
       -0.026413  QUA_ATR_15
        0.022498  CUB_ATR_5 * QUA_ATR_15
        0.023273  LIN_ATR_5 * QUA_ATR_15
       -0.012373  CUB_ATR_5 Squared
        0.024532  CONSTANT
```

This model chose three indicators: QUA_ATR_15, CUB_ATR_5, and LIN_ATR_5. As we saw a moment ago, this makes a total of nine terms plus the constant available to the quadratically-expanded linear model. However, internal stepwise selection and cross validation decided that only four of these nine possible terms are worthy of inclusion. One of them is the ordinary linear term, one is a squared term, and two are cross products.

Notice that these coefficients are labeled *Standardized* regression coefficients. This is because the numbers printed are not the actual coefficients used in the model. Rather, they are the actual coefficients times the standard deviation of the associated term. This makes them comparable in terms of relative importance. For example, suppose one indicator has a standard deviation that is 100 times greater than that of another indicator in the model. If they are equally 'important' to the prediction, the one with the larger standard deviation would have a coefficient that is 100 times smaller than that of the other. Without knowing about this disparity in variation, one might mistakenly conclude that the one with the larger coefficient is 100 times more important than the other! Multiplying by the standard deviation

eliminates this problem.

Also note that the ordinary linear regression model LINREG prints actua coefficients, not standardized. The reason for this disparity is that in a QUADRATIC model we have squared terms, which amplifies the scaling problem If one indicator has 100 times the standard deviation of another, their squared terms will be unequal by a factor of 10,000! So standardization is much more important for quadratic models than for linear models.

The General Regression Neural Network

The general Regression Neural Network (GRNN) is arguably the most nonlinear o all common models. In theory, except for pathological situations, the GRNN is capable of fitting every case in a training set to arbitrary accuracy. Nonlinearity does not get better than that! Thus, if you have reason to believe that the relationship between the indicator(s) and the target is extremely nonlinear, the GRNN may be an excellent choice for a predictive model.

This extreme nonlinearity comes at a high cost. The developer faces a painful quandary with a GRNN. On the one hand, the the tremendous fitting power of the GRNN means that overfitting is likely unless the training set is large. There must be enough cases that the authentic predictable patterns are detectable against the backdrop of noise. On the other hand, training time for a GRNN is proportional to something between the square and the cube of the number of cases. The CUDA option (See the *TSSB* manual) can reduce this by a substantial factor, but the power relationship still holds for the run time. The implication is that for very large training sets, the GRNN may require an impractical amount of training time.

Despite this quandary, there is a place for the GRNN in financial market prediction. The most important key is to limit the number of indicators used in the model. Two is often the practical limit, and three is almost certainly the most that one should ever use. This will reduce the likelihood of overfitting, even if the training set is not huge.

Here is a GRNN model definition, followed by its trained output from the audit log file:

```
MODEL GMOD IS GRNN [
   INPUT = [ CMMA_5 CMMA_10 CMMA_20 CMMA_5N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = LONG PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;

GRNN Model GMOD predicting DAY_RETURN_1
Stepwise based on long pf with min fraction=0.100
Sigma weights:
         20.640718  CMMA_5
          3.611279  CMMA_5N
```

A full discussion of the theory of a GRNN is beyond the scope of this document The most important fact to consider for interpretation of sigma weights is that their values are *inversely* proportional to the importance of an indicator. Thus, smaller

sigma weights imply greater importance.

The Multiple-Layer Feedforward Network

The multiple-layer feedforward network (MLFN) was the first practical neural network to be developed, and it is still a standard workhorse for nonlinear modeling. A major advantage of this predictive model is that the user can easily control its degree of nonlinearity by adjusting the number of hidden neurons. We've already seen that a LINREG linear regression model is strictly linear. At the other extreme, a GRNN has profound nonlinearity. Quadratic regression is a reasonable compromise, with moderate nonlinearity. But the MLFN is an adjustable compromise, ranging from slight nonlinearity to an extreme that approaches that of the GRNN. This adjustability can be a useful property, because the user can often find a sweet spot that captures any nonlinearity inherent in the data without being so excessive that overfitting occurs.

MLFNs have two serious disadvantages. First, they can be terribly slow to train with the training time being dependent on subtle aspects of the patterns in the data. Second, random number generators play a key role in the training of an MLFN. This means that the exact model produced by the training operation can be at the whim of the state of the random number generator at the start of training. In most situations, the models produced by different random sequences will be similar, so in practice this is almost never a problem. On the other hand, this dependence on randomness is psychologically offensive. Also, the lack of exact repeatability can lower some people's confidence in results. The only answer is to train for as long a time as possible, because in most cases this will reduce the dependence on the random number generator. Eventually, the weights will converge on a common solution, irrespective of the initial random seed used to initiate the training process.

The MLFN model allows several specifications that are unique to this model type. Some of these are mandatory, and others are optional. The next few sections discuss some of the things that can/must be specified.

The Number of Neurons in the First Hidden Layer

The user must specify the number of hidden neurons in the first (and usually only) hidden layer. This is done with the following mandatory specification:

```
FIRST HIDDEN = Number
```

Setting FIRST HIDDEN equal to one will result in linear operation. The MLFN model will be, for all practical purposes, ordinary linear regression, though implemented in an inefficient manner. (To be technically correct, the prior statement is true only if the OUTPUT LINEAR option, described later, i

employed. Since this is nearly always the case, we can accept this statement as correct.)

In most situations, it is appropriate to set FIRST HIDDEN equal to 2 (for modes nonlinearity) or 3 (for considerable nonlinearity). Larger values will allow degrees of nonlinearity that, for most financial applications, will produce serious overfitting. Hence, values in excess of 3 should be used for only those situations in which the indicators have a pronounced nonlinear relationship with the target, and the training set is huge. This does not happen very often.

The Number of Neurons in the Second Hidden Layer

In some rare cases it can happen that using two hidden layers of neurons is appropriate. Or at least rumors to this effect are circulating. By default, no second layer is used, as this is nearly universally the best choice. But if desired, a second layer can be specified as follows:

```
SECOND HIDDEN = Number
```

Those experimenters who use two hidden layers generally recommend that the number of neurons in the second hidden layer be smaller than the number in the first hidden layer. Thus, one might want to use the following pair of specifications if one wants to take the bold step of using two hidden layers:

```
FIRST HIDDEN = 3
SECOND HIDDEN = 2
```

As with the first hidden layer, using just one neuron is pointless. You need at least two hidden neurons in a layer in order to achieve nonlinearity in that layer.

Functional Form of the Output Neuron

The earliest version of the MLFN used a sigmoid function for the output neuron The result was an output that was bounded in the range 0 to 1 or -1 to 1, depending on the exact function used. This can be useful for binary decisions and some forms of encoded classification. However, it is obviously useless when the goal is general prediction. Also, training a model with a sigmoid output is slow. For this reason, most modern developers use a linear output neuron. The user is required to specify one or the other. Thus, one of the following two specifications is mandatory:

```
OUTPUT SQUASHED
OUTPUT LINEAR
```

The second option (OUTPUT LINEAR) is highly recommended for virtually a applications.

The Domain of the Neurons

The vast majority of the time, we want the entire MLFN to operate in the rea domain. However, *TSSB* allows some or all of the neurons to operate in the complex domain (real and imaginary components), which is occasionally useful.

The user is required to specify the domain. Exactly one of the following three domain options must appear in the MLFN model specification:

DOMAIN REAL - This is by far the most common type of MLFN. Processing is entirely in the real domain, beginning to end. If you are unsure of what domain you require, specify this one. It is always valid. Most indicators used for financial prediction are strictly real numbers.

DOMAIN COMPLEX INPUT - Only the inputs are complex-valued. The first input is treated as real, the second imaginary, the third (if any) real, the fourth imaginary, et cetera. The user must specify an even number of inputs. Stepwise selection is not performed; all inputs are used. This option should be used only if the inputs truly have a complex interpretation, such as real and imaginary Morlet wavelets. There is no reason to believe that truly real inputs, which includes the vast majority of common indicators, would ever be appropriate in a complex MLFN.

DOMAIN COMPLEX HIDDEN - This option is identical to that above, except that the hidden neurons also operate in the complex domain. This is almost always more useful than a complex input only. Thus, if you have complex inputs, you are almost always best off using the DOMAIN COMPLEX HIDDEN option so that calculations in the hidden laye continue in the complex domain.

A Basic MLFN Suitable for Most Applications

The vast majority of MLFN applications are well served by a real-domain model with one hidden layer containing two or three neurons. Here is an example definition of such a model, followed by the trained weights as they appear in the

audit log file:

```
MODEL MOD_R_3 IS MLFN [
   INPUT = [ CMMA_5 - CMMA_20 ]
   OUTPUT = DAY_RETURN_1
   OUTPUT LINEAR
   DOMAIN REAL
   FIRST HIDDEN = 3
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;

MLFN Model MOD_R_3 predicting DAY_RETURN_1
Stepwise based on pf with min fraction=0.100  (Final best
crit = 0.3795)
  Input-to-Hid1 neuron 1 weights:
    CMMA_5: -0.014280
    CMMA_10: -0.057529
    Bias: -2.582003
  Input-to-Hid1 neuron 2 weights:
    CMMA_5: -14.734776
    CMMA_10: -3.221220
    Bias: -3.710321
  Input-to-Hid1 neuron 3 weights:
    CMMA_5: -7.644637
    CMMA_10: -1.242368
    Bias: -1.043874
  Hid1-to-Output weights, bias last:
     1: 0.144360
     2: 0.157811
     3: -0.191516
     4: 0.162412
```

A Complex-Domain MLFN

Considerable experience indicates that when the inputs to an MLFN are inherently complex numbers (they contain a real and imaginary component), performance is enhanced by doing as many computations as possible in the complex domain. *TSSB* contains a family of indicators called Morlet Wavelets (here) which are complex-valued. It makes sense to use a complex-domain neural network to process such indicators. Here is an example of such a model, followed by the trained weights as they appear in the audit log file. Note that the MAX STEPWISE command does not appear because in a complex-domain MLFN all inputs are used.

```
MODEL MOD_CH_2 IS MLFN [
   INPUT = [ RMORLET_5 IMORLET_5 RMORLET_10 IMORLET_10 ]
   OUTPUT = DAY_RETURN_1
   OUTPUT LINEAR
   DOMAIN COMPLEX HIDDEN
   FIRST HIDDEN = 2
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;

MLFN Model MOD_CH_2 predicting DAY_RETURN_1
Stepwise not used; all predictors available to model
  Input-to-Hid1 neuron 1 weights:
    RMORLET_5 ; IMORLET_5: (-1.136622 0.190527)
    RMORLET_10 ; IMORLET_10: (1.213799 0.949340)
    Bias: (-0.024983 -0.191141)
  Input-to-Hid1 neuron 2 weights:
    RMORLET_5 ; IMORLET_5: (-1.020735 -0.699088)
    RMORLET_10 ; IMORLET_10: (-0.211185 1.160136)
    Bias: (0.711997 -0.161859)
  Hid1-to-Output weights, bias last:
     1: (0.038938 0.015374)
     2: (-0.029881 0.030525)
     3: (0.031820 0.178404)
```

Note that all of the weights occur in pairs. Following convention, the first number in each pair is the real component of the weight, and the second number is the imaginary component.

The Basic Tree Model

The tree is the simplest model available in *TSSB*. Its primary advantage is that it is extremely easy to understand how the model transforms the values of the indicators into a prediction. A tree is just a series of binary decisions. An indicator is compared to a threshold. If the indicator's value is on one side of the threshold, one decision is made. If it's on the other side, a different decision is made. These decisions may be a final prediction for the target, or they may lead to another binary decision. An example of this process will appear soon in this section.

The main problem with a tree model is that it is weak. Its predictions come from a usually small number of discrete values. Moreover, the prediction decisions are based on a small number of binary decisions. The tree is unable to take advantage of subtleties in the data.

On the other hand, trees form the basis of two more powerful models available in *TSSB*: the *Forest* and the *Boosted Tree*. Also, some developers love the simplicity of a tree. For these reasons, this model is made available to users, and it is discussed here.

There are two special tree options, one mandatory and one optional. These are:

TREE MAX DEPTH = Integer

This mandatory specification limits the maximum depth of the tree. Note that pruning may result in shallower trees. A depth of one means that a single decision is made, splitting the root into two nodes based on the value of a single indicator. A depth of two means that these two nodes may themselves be split, and so forth.

MINIMUM NODE SIZE = Integer

This option specifies that a node will not be split if it contains this many cases or fewer. Note that some nodes may be smaller than this if a split occurs near a boundary, so this is not really the minimum size of a node. It just prevents small nodes from being split. This optional specification defaults to one if omitted. The default value of one means that the tree-building algorithm is given full freedom to act as it sees best. This is almost always a good thing.

Here is a sample tree model specification, followed by the trained tree as it appears in the audit log file. Note that when we define a tree model, we almost always set MAX STEPWISE = 0 so that all indicators are made available to th model. This is because the tree-building algorithm uses internal (invisible to the user) cross validation and selection to choose the optimal indicators. This is almost

always better than setting MAX STEPWISE to a positive number and henc invoking the usual stepwise selection algorithm that applies to all models.

```
MODEL TREE1 IS TREE [
   INPUT = [ CMMA_5 - VMUTINF_3 ]
   OUTPUT = DAY_RETURN_1
   TREE MAX DEPTH = 3
   MAX STEPWISE = 0
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;

TREE Model TREE1 predicting DAY_RETURN_1
Stepwise not used; all predictors available to model

 Node          Split...

    0          LIN_ATR_7 <= -14.27825928  ---> 1
               LIN_ATR_7 >  -14.27825928  ---> 2

    1          DAU_NLRG_32_3 <= 15.60567141  = 0.01773210
               DAU_NLRG_32_3 >  15.60567141  = 0.17067198

    2          CMMA_5 <= -2.03893173  = -0.04221814
               CMMA_5 >  -2.03893173  =  0.04450752
```

Examine the log file output shown above. Node zero is always the root node, the first decision made. This tree considers the value of the indicator LIN_ATR_7 anc compares it to a threshold of –14.278. If the indicator's value is less than or equal to this threshold, it proceeds to Node 1 for the next decision. If, on the other hand, the indicator's value exceeds this threshold, it proceeds to Node 2 for the next decision. Nodes 1 and 2 are called terminal nodes because they produce the final prediction for the target. For example, Node 2 looks at the indicator CMMA_5. I this indicator's value is less than or equal to –2.0389, then the tree predicts that the target's value will be –0.0422.

Note that the script file specified TREE MAX DEPTH = 3, but the tree produce has a depth of two, not three. (The depth is two because Node 0 is one level, and Nodes 1 and 2 are at a second level.) It is common for the final depth to be less than the maximum specified. This is because the tree-building algorithm uses internal cross validation to determine the optimal depth. It often happens that a deep tree, despite its many weaknesses, still manages to overfit. This overfitting is detected by the cross validation, and the depth of the tree is reduced accordingly.

A Forest of Trees

One way to overcome the serious weakness of a tree model is to train a large number of trees and combine their individual predictions into a mean prediction for the ensemble. Each tree is trained on a different, randomly selected subset of the complete training set. In general, this will result in numerous different indicators being used, because different subsets of the training data will frequently favor different indicators.

Also, although the possible predicted values will still be a finite discrete set, the process of combining many trees will greatly increase the number of possible predicted values. In fact, if hundreds of trees are used in the forest, the predicted value will be almost continuous.

Finally, because each tree is based on a different subset of the training data, overfitting is discouraged. Noise will show up as different false patterns in different trees, and hence tend to cancel when the predictions are combined. At the same time, authentic patterns will tend to appear in most trees and hence reinforce one another.

The TREE MAX DEPTH and MINIMUM NODE SIZE tree options may be used in a forest declaration. See here for a description of these options. There is also a specification that is mandatory for a forest. The user must specify the number of trees in the forest. This will typically be several hundred or so.

```
TREES = Integer
```

It should be noted that forests do not seem to do well with extremely noisy data, such as market data. The bagging algorithm, which is a key part of forest generation, produces component trees that favor very extreme binary decision thresholds. The result is that the majority of predictions are constant. For this reason, use of forest models in *TSSB* is discouraged. Still, for those who wish to do so, here is a sample forest model declaration, followed by the output produced in the audit log file. Note that as was the case with trees, we almost always set MAX STEPWISE = 0 so that all indicators are made available to the forest model. The model training algorithm will efficiently find its own optimal indicator set.

```
MODEL FOREST1 IS FOREST [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   TREE MAX DEPTH = 3
   TREES = 100
   MAX STEPWISE = 0
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

```
FOREST Model FOREST1 predicting DAY_RETURN_1
Stepwise not used; all predictors available to model

Importance of the predictors

     CMMA_5   13.75 %                CMMA_10   14.61 %
    CMMA_10   14.61 %                 CMMA_5   13.75 %
    CMMA_20    9.17 %               QUA_ATR_15   13.47 %
    CMMA_5N    2.87 %                LIN_ATR_7    9.74 %
   CMMA_10N    1.72 %                  CMMA_20    9.17 %
   CMMA_20N    5.73 %               CUB_ATR_15    9.17 %
  LIN_ATR_7    9.74 %              LIN_ATR_15N    6.88 %
 LIN_ATR_15    5.44 %                 CMMA_20N    5.73 %
 QUA_ATR_15   13.47 %                LIN_ATR_15    5.44 %
 CUB_ATR_15    9.17 %                LIN_ATR_7N    4.58 %
 LIN_ATR_7N    4.58 %                   CMMA_5N    2.87 %
LIN_ATR_15N    6.88 %              CUB_ATR_15N    2.29 %
QUA_ATR_15N    0.57 %                 CMMA_10N    1.72 %
CUB_ATR_15N    2.29 %              QUA_ATR_15N    0.57 %
```

All of the indicators available to the model are listed in two columns, along with a measure of importance. The left column lists the indicators in the order in which they appear in the variable definition file or database. The right column lists them in descending order of importance. Thus, indicators near the top of the right column are the ones most relied on by the forest model when it makes predictions. One potential value of the forest model is the ranked list of indicators. If a more compute-intensive model is being contemplated, such as GRNN, the ranked list can indicate which indicators might best be considered as candidates, although this may be risky. Trees and GRNNs use indicators very differently, so an indicator that is important in one model may be worthless in the other.

Once again, it should be emphasized that the original forest algorithm, which is implemented in *TSSB*, does not usually perform well with financial data. Some day the program may incorporate a modified version that does better in the presence of extreme noise, but until this modification is implemented, forests should be avoided.

Boosted Trees

In the prior section we saw that creating a forest of independently trained trees and averaging their predictions is one way to pool predictions of many models. Unfortunately, forests do not perform well in extreme-noise environments. A method for combining multiple trees that does do well with noisy data is to use the traditional boosting algorithm to train a sequence of trees. Each new tree in the sequence focuses its efforts on handling cases that give trouble to prior trees in the sequence.

The TREE MAX DEPTH and MINIMUM NODE SIZE tree options may be used a boosted tree declaration. See here for a description of these options. There are also several mandatory and optional specifications for a boosted tree. These are:

MIN TREES = Integer

This mandatory specification is the minimum number of trees in the boosting set. The optimum number will be determined by internal (invisible to the user) cross validation. MIN TREES should be at least two. Larger values require more training time but may produce a more effective model.

MAX TREES = Integer

This mandatory specification is the maximum number of trees in the boosting set. The optimum number will be determined by internal (invisible to the user) cross validation. MAX TREES should be greater than or equal to the MIN TREES specification. Larger values require more training time but may produce a more effective model.

ALPHA = Real

This mandatory specification is the Huber Loss value. Its maximum value of one results in no Huber loss, and is best for clean data. Smaller values (zero is the minimum) cause extreme values of the target to be ignored. Values less than 0.9 or so are only rarely appropriate. Values between 0.9 (for targets with extreme values, such as unrestrained daily returns) and 0.99 (for well behaved targets such as hit-or-miss) are usually appropriate.

FRACTION OF CASES = Real

This option specifies the fraction of cases, zero to one, that are used to train each component tree. If omitted it defaults to 0.5, which is reasonable.

SHRINKAGE = Real

This option specifies the learning factor for each component tree. In general, smaller values produce better generalization but require more trees to learn the data pattern. If omitted, this defaults to 0.01.

Here is an example declaration of a boosted tree model, followed by the results of training as they appear in the audit log file. Note that as was the case with trees, we almost always set MAX STEPWISE = 0 so that all indicators are made availabl to the model. The model training algorithm will efficiently find its own optimal indicator set.

```
MODEL BOOSTREE1 IS BOOSTED TREE [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   TREE MAX DEPTH = 3
   MIN TREES = 2
   MAX TREES = 20
   ALPHA = 1.0
   MAX STEPWISE = 0
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

```
BOOSTED TREE Model BOOSTREE1 predicting DAY_RETURN_1
Stepwise not used; all predictors available to model
BOOSTREE  M=13  in-sample MSE=0.28801  RMS=0.53666
Importance of the predictors
     CMMA_5   26.67 %              CMMA_5   26.67 %
    CMMA_10    3.33 %          QUA_ATR_15   16.67 %
    CMMA_20    3.33 %           LIN_ATR_7   13.33 %
    CMMA_5N    6.67 %          CUB_ATR_15   10.00 %
   CMMA_10N    0.00 %          LIN_ATR_15    6.67 %
   CMMA_20N    3.33 %             CMMA_5N    6.67 %
  LIN_ATR_7   13.33 %          LIN_ATR_7N    6.67 %
 LIN_ATR_15    6.67 %             CMMA_20    3.33 %
 QUA_ATR_15   16.67 %         CUB_ATR_15N    3.33 %
 CUB_ATR_15   10.00 %             CMMA_10    3.33 %
 LIN_ATR_7N    6.67 %            CMMA_20N    3.33 %
LIN_ATR_15N    0.00 %         LIN_ATR_15N    0.00 %
QUA_ATR_15N    0.00 %         QUA_ATR_15N    0.00 %
CUB_ATR_15N    3.33 %            CMMA_10N    0.00 %
```

The ‘M’ parameter printed is the number of trees used, determined by internal cross validation. As with a forest, all of the indicators available to the boosted tree model are listed in two columns, along with a measure of importance. The left column lists the indicators in the order in which they appear in the variable definition file or database. The right column lists them in descending order of importance. Thus, indicators near the top of the right column are the ones most relied on by the boosted tree model when it makes predictions.

Operation String Models

Most predictive models have a preordained structure, with only certain parameters being determined by training. For example, a tree is a series of binary decisions which compare the values of indicators to thresholds. Linear regression is the weighted sum of indicators. But an operation string model is more general in that it can employ a variety of arithmetic and logical operations on a set of indicators. For example, an operation string model might say to multiply X1 by 3.7, subtractX2, and compare the result to 4.5 in order to produce a decision as to whether the market is predicted to move strongly upward.

Because an operation string model has no fixed structure, it cannot be trained by ordinary methods. Instead, this model is evolved using a genetic algorithm. Readers not familiar with genetic algorithms and their associated terms should consult any of the widely available references. The implication of using a genetic algorithm is that the randomness inherent in evolution makes it unlikely that the same model will emerge if training is performed with different sequences of random numbers. This is annoying, but as long as training is performed with a large population size and for a considerable number of generations, the resulting models will usually have similar performance.

The version of operation string models used in *TSSB* takes one additional step beyond what is done by most traditional operation string models: as a part of training, it fits a least-squares straight line that maps the operation string result to the target. The advantage of this approach is that the 'fit' between the prediction and the target will be optimal in the least-squares sense, which means that traditional performance measures such as R-square will be meaningful. Thus, selecting indicators based on the R-square criterion is meaningful. Also, if the user happens to compare the predictions to the targets, the comparisons will make sense. This linear fit is not strictly necessary, because trade decisions are based on thresholds applied to the output of the model. But it's a nice touch that costs very little in terms of computing time and avoids much potential user confusion.

Let A and B be numbers, perhaps indicators or intermediate results such as sums or differences. The following arithmetic and logical operations are used in *TSSB*'s implementation of operation strings:

```
ADD:  A + B
SUBTRACT:  A - B
MULTIPLY:  A * B
DIVIDE:  A / B
GREATER:  1.0 if A > B, else 0.0
LESS:  1.0 if A < B, else 0.0
AND:  1.0 if A > 0.0 and B > 0.0, else 0.0
OR:  1.0 if A > 0.0 or B > 0.0, else 0.0
NAND:  0.0 if A > 0.0 and B > 0.0, else 1.0
NOR:  0.0 if A > 0.0 or B > 0.0, else 1.0
XOR:  1.0 if (A > 0.0 and B <= 0.0) or (A <= 0.0 and B > 0.0), else 0.0
```

The operation string model definition does not have any mandatory specifications. However, numerous options are available. These are:

MAX LENGTH = Integer

The maximum length of the operation string, which includes indicators, constants, and operations. Smaller values force simpler operations, which reduces power but also reduces the likelihood of overfitting. The actual string length may be less than this quantity if the training algorithm sees an advantage to shortening the string. This quantity must be odd. It will be decremented to an odd number if specified even. If omitted, this defaults to 7, which is reasonable. The meaning of the string length will become clear later in this section when an example of an operation string appears.

POPULATION = Integer

This is the number of individuals in the population. Generally, this should be many times the number of candidate predictors in order to provide sufficient genetic diversity. The default is 100, although values as high as 1000 or even 10,000 are recommended if training time permits.

CROSSOVER = RealNumber

This is the probability (0-1) of crossover. The default is 0.3, which should be reasonable for most applications.

MUTATION = RealNumber

This is the probability (0-1) of mutation. It should be small, because mutation is destructive far more often than it is beneficial. The default is 0.02, which should be appropriate for most applications.

REPEATS = Integer

If this many generations in a row fail to improve the best child, training terminates. The default is 10, although values as small as 2 or 3 may be reasonable if the population is huge (thousands). Of course, training can

be manually terminated at any time by pressing the ESCape key.

CONSTANTS (Probability) = ListOfConstants

By default, no constants are allowed in an operation string. Operations are performed on indicators and intermediate results only. Except for very simple operation strings (length of 3 or 5) this is a severe limitation. The CONSTANTS command lets the user request tha specific constants or random constants within a range may be used. A range is specified by surrounding a pair of numbers with angle brackets, as in <-40 40>. The Probability (0-1) specifies the probability that a constant will be used rather than an indicator. This should usually be large, close to 1.0. Constants will be discussed in detail in the next section.

CONSTANTS IN COMPARE

If the user specifies a CONSTANTS list, then by default constants ma be used in any operations. The CONSTANTS IN COMPARE optic restricts constants to being used in compare operations (greater/less than) only. Use of constants in arithmetic is forbidden.

CONSTANTS IN COMPARE TO VARIABLE

This is even more restrictive than the CONSTANTS IN COMPAR command. It restricts constants to use in comparisons to variables only. Comparisons between constants and intermediate results is forbidden, whereas this is allowed with the CONSTANTS IN COMPARE option.

Use of Constants in Operation Strings

Only the simplest models can get away without using constants. For example, a simple prediction might be based on the sum or difference of two indicators. Or a model might compare the values of two indicators and base its prediction of a market move on which of the two indicators is the greater. But any rule even slightly more complex needs constants. Constants can be employed in any of three ways. These are listed here in order of increasing generality:

1) The most restrictive use of constants in is comparing the value of an indicator with the constant. For example, at some point in an operation string the model may inquire whether or not X1 > 4.734. As discussed earlier, when the legal operations were listed, this logical expression will take on a value of 1.0 if it is true, and 0.0 if it is false. If the user wishes to allow only this highly restrictive use of constants, the CONSTANTS IN COMPARE TO VARIABLE option should be specified the model definition.

2) A less restrictive use of constants also allows comparisons with intermediate results. For example, an operation string might inquire whether (X1 + X2) <= -5.0. In order to allow this level of usage, specify the CONSTANTS IN COMPAR option in the model definition.

3) By default, if no restriction appears in the model definition, constants may also play a role in arithmetic. For example, the quantity (X1 - 3.7) may play a role in an operation string.

The question then arises as to exactly what constants the evolutionary algorithm may employ. It certainly can't just pull wild numbers out of a hat; a constant of 10,000,000 probably would make no sense in an operation string! Sometimes the user has certain specific constants in mind, such as 0.0 and 1.0. Other times the user wants to give the evolutionary algorithm the freedom to make random choices, but within a sensible range. *TSSB* allows both options.

Legal constants are specified with the CONSTANTS option discussed earlier. This command lists all legal constants. Also, ranges for random selection can be specified by enclosing the range in angle brackets <>.

The user can also control how frequently constants are allowed to appear in an operation string. This is done by specifying a probability in the CONSTANTS command. This is that probability that, if a constant could legally be used at some point in the computation, it will be used (as opposed to an indicator being used). Because numerous rules disallow use of a constant in certain circumstances, it is usually best to set this probability to a value very close to one, such as 0.99. This will usually result in a reasonable number of constants appearing. If too many constants appear, reduce this probability.

Here is an example of the specification of constants that can appear in an operation string. Suppose that one wants to allow constants equal to exactly 0.0, or exactly 1.0, or a random number in the range 10 to 40, or a random number in the range 100 to 500. The following option would accomplish this:

```
CONSTANTS (0.99) = 0.0 1.0 <10 40> <100 500>
```

Here is an example definition of a simple operation string model. As was the case with trees, we almost always set MAX STEPWISE = 0 so that all indicators are made available to the model. The evolutionary training algorithm will efficiently find its own optimal indicator set.

```
MODEL RET IS OPSTRING [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 0
   MIN CRITERION FRACTION = 0.1
   MAX LENGTH = 7
   POPULATION = 10000
   CROSSOVER = 0.3
   MUTATION = 0.05
   REPEATS = 2
   CONSTANTS (0.9999999999) = <0.0 50.0>
   CONSTANTS IN COMPARE TO VARIABLE
   ] ;
```

This model definition says that the maximum length of an operation string can be seven. This count includes indicators, constants, and operations. The size of the population being evolved is 10,000, which is large but doable because the dataset is not gigantic. Crossover and mutation rates are set to reasonable values. If two generations in a row (REPEATS=2) produce no superior offspring, training terminates. No specific constants are made available, but the evolutionary algorithm is allowed to choose constants randomly in the range 0 to 50. Finally, use of constants is limited to comparisons with indicators. This is probably overly strict. The CONSTANTS IN COMPARE option would be more flexible, and son users might want to avoid even that restriction and allow the algorithm its default operation of using constants as it sees fit.

Here is the output produced after training. An explanation follows.

```
OPSTRING Model RET predicting DAY_RETURN_1
Stepwise not used; all predictors available to model
Operations in Reverse Polish:
  LOAD CMMA_20
  LOAD 20.52028
  LESS
  LOAD CMMA_5N
  MULTIPLY
  LOAD LIN_ATR_7N
  GREATER

Operations in infix notation:
((CMMA_20 < 20.52028) * CMMA_5N) > LIN_ATR_7N

Slope = 0.08831  Intercept = -0.01015
```

The evolutionary training algorithm operates internally in a highly efficient notation scheme called Reverse Polish, so the actual operation string is printed first in this version. For readers who used and loved the early Hewlett-Packard calculators, this notation will be familiar. For everyone else, the mathematical operations

corresponding to the operation string are printed in standard infix (sometimes called algebraic) notation.

We now examine this set of mathematical and arithmetic operations. The first action is to compare the indicator CMMA_20 to the constant 20.52028. If this comparison is true(CMMA_20 is less than the constant) this logical expression will evaluate to one, and if the comparison is false it will evaluate to zero. This intermediate result is then multiplied by the indicator CMMA_5N. Thus, according to the truth of the comparison just mentioned, we will end up with either the value of CMMA_5N or zero. This quantity, whichever it happens to be, is compared to the indicator LIN_ATR_7N. This gives us the (almost) final value of the operation string, which in this case is binary. It will be either one or zero.

There is one last step to the training. It's not theoretically necessary, especially when the result of the operation string is binary as opposed to a continuous value. But it often helps with visual interpretation of plots of results, as well as ensuring the validity of R-square as a performance measure. This final step is to find the slope and intercept of the least-squares straight line that maps the operation string output to the target. These quantities are printed for the user's edification.

Split Linear Models for Regime Regression

It is well known that asking a single model to handle a large variety of conditions is often unrealistic. When the market is in a period of high volatility, one model may perform well. An entirely different model may be best when the market is trending upward, and a still different model might be a star performer in down-trending markets.

TSSB contains a 'model' that provides a solution to this problem. The user treats a *SPLIT LINEAR* model exactly the same way any other model is treated. However, its internal structure is unusual in that it really consists of two or three independent linear models. The training algorithm handles separate training of the component sub-models.

These sub-models may employ identical indicators but use them differently (have different regression weights). Or the sub-models may employ different indicators. For this reason, selection of the indicators in a SPLIT LINEAR model takes plac in two steps. The general stepwise process used for every model in *TSSB* produces a pool of variables from which indicators for all sub-models will be chosen. At each step, the indicators in the current common pool will be optimally assigned to the individual sub-models.

In addition to selection of indicators, a gate variable will be chosen from a list supplied by the user. An optimal split point (or two points) based on the gate variable will be computed in order to define the regimes.

A simple example may make this more clear. Suppose we wish to employ two different linear models, each of which specializes in a certain volatility regime. Perhaps one will be used for predictions when recent volatility is low, and the other will be used for high volatility. We may supply the split linear model with four candidate indicators: X1, X2, X3, and X4. We may also give it two volatility indicators: V1 and V2. Finally we may tell the training algorithm that in order tc reduce the possibility of overfitting, each sub-model can use at most two indicators.

After training, an optimal split linear model may be found that has the following characteristics:

- One sub-model uses X2 and X3 as indicators.
- The other sub-model uses X3 and X4 as indicators
- V2 is chosen as the best volatility indicator for choosing the sub-model
- The first sub-model is used when V2 is less than or equal to 2.7 (a threshold that is automatically chosen as optimal).
- The second sub-model is used when V2 is greater than 2.7.

The above characteristics are, of course, a hypothetical example. The point is that the SPLIT LINEAR training algorithm automatically finds the best indicators fc each sub-model, automatically finds the best gate variable for selecting the sub-model, and automatically computes the optimal threshold for choosing which sub-model to use.

There are many mandatory and optional specifications that apply to SPLIT LINEAR models. These are:

SPLIT VARIABLE = [Var1 Var2 ...]

This mandatory specification lists one or more candidates for the variable that will define regimes. As in an INPUT list, a range may be specified with the dash (-). Exactly one of these candidates will be chosen as the gate variable, the indicator whose value determines which of the sub-models will be selected to make a prediction for a case.

SPLIT LINEAR SPLITS = Integer

This option specifies the number of split points, and it must be one or two. The number of regimes (sub-models) is one more than the number of split points. If omitted, the default is one, which gives two sub-models.

SPLIT LINEAR SPLITS = NOISE

This option specifies that the domain is split into two regimes, but only one of the two regimes is modeled. The other regime is considered to be noise. Any prediction made within the noise regime will be set equal to the mean of the target variable within the training set. Note that it will not always be possible to meet the specified SPLIT LINEAI MINIMUM FRACTION (described soon) specification when tl NOISE option is invoked.

SPLIT LINEAR MAXIMUM INDICATORS = Integer

This specifies the maximum number of indicators that will be used in each sub-model. These indicators will be optimally chosen from the pool of candidates provided by the model's stepwise selection process. If you want to guarantee that the regimes can have completely disjoint predictor sets, the model's MAX STEPWISE must be equal to SPLI LINEAR MAXIMUM INDICATORS times the number of regimes. makes no sense to make it larger. If omitted, the default is 2.

SPLIT LINEAR MINIMUM FRACTION = RealNumber

This option specifies the minimum fraction of the training cases that

must fall into each regime for the ordinary version of the model, and the minimum for the non-noise regime in the NOISE version. If the SPLI LINEAR SPLITS = NOISE option is used, it may not always l possible to satisfy this minimum fraction, especially if the value is 0.5 or greater. In this case, the program will come as close as it can. Also, note that if the NOISE option is not used, the maximum value for this MINIMUM FRACTION that the user may specify is 0.4. This ensur that degenerate conditions are avoided. Realize that in the ordinary version, a minimum exceeding 0.5 would be illegal because it is not possible for all regimes to contain more than half of the cases! If omitted, the default is 0.1.

SPLIT LINEAR RESOLUTION = Integer

This option is the number of split points that will be tested in order to define the regimes. Larger values require more run time but produce higher accuracy. If omitted, the default is 10. If there is one split point (two sub-models), this quantity will determine how many splits are tested to find an approximate split, and then the exact optimum will be found by further iterations. If there are two split points (three sub-models), no further refinement is done.

An Ordinary SPLIT LINEAR Model

Here is an example model definition for a SPLIT LINEAR model that operates i the ordinary fashion, as two independent models, each of which handles a different regime. It declares SPLIT LINEAR SPLITS = 1 which means that there will be tv sub-models. The maximum number of indicators for each is 2, so if we want to allow each sub-model to have its own different set of indicators, we need to set MAX STEPWISE to 2*2=4. We supply a range of variance ratio (PVARRAT indicators to be tried as candidates for determining the regime of each case. We also declare that at least 0.4 of the training set be assigned to each of the two regimes (sub-models). This is pretty close to the maximum legal value of 0.5, and hence may be a touch restrictive for many applications. Finally, we could have eliminated the SPLIT LINEAR RESOLUTION = 10 option because 10 is t default. Still, sometimes it is nice to specify options, just to be clear.

```
MODEL MOD_SPLIT IS SPLIT LINEAR [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 4
   SPLIT VARIABLE = [ PVARRAT_5 - DPVARRAT_20 ]
   SPLIT LINEAR SPLITS = 1
   SPLIT LINEAR MAXIMUM INDICATORS = 2
   SPLIT LINEAR MINIMUM FRACTION = 0.4
   SPLIT LINEAR RESOLUTION = 10
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;
```

After training, the following output appears in the audit log:

```
SPLIT LINEAR Model MOD_SPLIT predicting DAY_RETURN_1
Regression coefficients for PVARRAT_20 <= 9.495   n = 5594
of 11187  mse = 0.4183
          0.008293  CMMA_5N
         -0.007935  LIN_ATR_7N
          0.000000  LIN_ATR_15N
          0.025171  CONSTANT

Regression coefficients for PVARRAT_20 > 9.495   n = 5593
of 11187  mse = 0.5147
          0.000000  CMMA_5N
         -0.001288  LIN_ATR_7N
          0.001726  LIN_ATR_15N
          0.030695  CONSTANT
```

Each of the two sub-models is described. PVARRAT_20 was chosen as the gate variable. When its value is less than or equal to 9.495 the case is determined to be in the first regime, and of the 11187 cases in the training set, 5594 of them landed in this regime. The mean squared error of the training cases in this regime is 0.4183. The other sub-model is similarly described.

Recall that the model definition specified MAX STEPWISE = 4 and SPLI LINEAR MAXIMUM INDICATORS = 2. As it happens, the stepwise procedu chose only three variables to use for the SPLIT LINEAR model: CMMA_5N LIN_ATR_7N, and LIN_ATR_15N. It found no benefit to using a fourth variable even though it had the right to do so. Because we limited the number of indicators in each sub-model to 2, in each sub-model we see that one of the coefficients is exactly zero, meaning that it is ignored.

The NOISE Version of the SPLIT LINEAR Model

If one uses the SPLIT LINEAR SPLITS = NOISE option, a slightly differ

version of the SPLIT LINEAR model will be employed. In the usual version, one o the two or three sub-models is always used to make a prediction. But in the NOISE version, there are always two models, and only one of them is used to make predictions. When the value of the gate variable for a case indicates that the case belongs to the other regime, the case is considered to be unpredictable noise. In this situation, the 'prediction' is the mean value of the target within the entire training set.

This can be useful when the developer has reason to believe that certain regimes are inherently unpredictable. For example, it is believed by many that high volatility markets are dominated by so much noise that there is no point in even trying to predict market moves in such conditions. Thus, the developer may choose to use volatility as a gate variable and employ the NOISE version of the model Because predictions equal to the target mean are unlikely to be extreme enough to pass the optimal trading threshold, trading will most likely cease for cases in the NOISE regime.

Sometimes the developer will receive a surprise with the NOISE version. Consider the prior example, in which the developer believes that high-volatility regimes are unpredictable noise. It may turn out that the program thinks otherwise. In other words, the program may inform the developer that it is the high-volatility regime that is most predictable, and hence classify the low-volatility regime as unpredictable noise. When this happens, the developer needs to reconsider his or her preconceptions about the data!

Here is an example script file that defines the NOISE version of the SPLIT LINEAR model. Note that it is identical to the prior example except that this definition requires that the training process attempt to place at least 0.7 (70 percent) of the training cases in the non-noise set.

```
MODEL MOD_NOISE IS SPLIT LINEAR [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 4
   SPLIT VARIABLE = [ PVARRAT_5 - DPVARRAT_20 ]
   SPLIT LINEAR SPLITS = NOISE
   SPLIT LINEAR MAXIMUM INDICATORS = 2
   SPLIT LINEAR MINIMUM FRACTION = 0.7
   SPLIT LINEAR RESOLUTION = 10
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;
```

The following trained output is written to the audit log file:

```
SPLIT LINEAR Model MOD_NOISE predicting DAY_RETURN_1
Stepwise based on pf with min fraction=0.100  (Final best
crit = 0.4899)

Regression coefficients for PVARRAT_20 <= 29.981   n=8499
of 11187  mse = 0.4299
          0.006939  CMMA_5N
         -0.006041  LIN_ATR_7N
          0.018619  CONSTANT

Regression coefficients for PVARRAT_20 > 29.981  (Noise)
n=2688 of 11187  mse = 0.5818
         -0.000830  CMMA_5N
          0.000000  LIN_ATR_7N
          0.057355  CONSTANT
```

The format of this output is almost identical to the format for the normal version of the SPLIT LINEAR model. The gate variable and its threshold are shown. The number of training cases that fall into each regime are given, as well as the regression coefficients for each sub-model. The only difference is that in this case, the noise regime is identified as such. Also note that the regression coefficients for the noise regime are, in a sense, pointless. This is because they are not used for actual predictions. Cases that fall into the noise regime are assigned a prediction equal to the mean of the target variable in the entire training set.

Committees

One of the most important modeling discoveries of the last few decades is that much is to be gained by training several different models and then combining their predictions to produce a committee decision. Committees are based on the same idea as Modern Portfolio Theory - combining investments whose return streams are independent to some degree results in a portfolio whose risk (return variance) is reduced. In predictive modeling, prediction error is the analogue of risk. Whatever can be done to reduce it is useful. Committees are a way to do this.

These models do not need to be completely independent. It is common for the predictions of the component models to be highly correlated, although the less correlation among models (more diversity), the better the committee will operate.

It should be noted that committees and models, as implemented in *TSSB*, have much in common. They share many specifications and options. Also, several *TSSB* models can also be used as committees.

A committee must be given a name, just as is the case with models. The name may contain up to 15 characters, and no special characters other than the underscore (_) may be used. This name must be unique, not a duplicate of the name of any model or variable. Several examples of committee declarations will appear later in this chapter.

There are many ways to create the component models that will be used as inputs to a committee. For example, they may use different indicators, or be different fundamental forms. Many examples of methods for generating component models will be presented at the end of this chapter. But the main focus of this chapter is a detailed description of every committee available in *TSSB*. The available committees differ in how they combine models' predictions.

Model Specifications Used by Committees

There are many options and specifications already discussed in the context of models that are also usable for committees. Rather than repeating the details of each, they will be listed here in brief summary. When appropriate, a reference is given to the page in the model section where more details can be found.

INPUT = [ModelNames]

Thismandatoryspecification names the models whose outputs are to be used as inputs to the committee. Note that when the INPUT command is used for model, it lists indicators. But when the INPUT command is used for a committee, it lists models. Each model must be named individually. The dash (-) option (here) that is usable for model inputs is not legal for committee inputs.

OUTPUT = VariableName

This mandatory specification names the target variable. It is legal for this to be different than the targets of the component models, but it is hard to imagine a situation in which this would be appropriate. In virtually all cases of interest, all component models as well as the committee share the same target.

PROFIT = VariableName

This option must be used when the OUTPUT variable is not a measure of profit. The PROFIT option names a variable that measures profit and is used for profit-based performance criteria. See here for details on this important option.

LONG ONLY
SHORT ONLY

One of these options specifies that the committee will execute only long (or short) trades. Probably the only use of this option is in creating separate trading systems that specialize in only long or short trades and then combining them into a Portfolio.

MAX STEPWISE = Integer

This mandatory specification is the maximum number of models that will be permitted to serve as inputs to the committee. As with models, setting this equal to zero forces all inputs to be used in the committee.

STEPWISE RETENTION = Integer

This option causes the stepwise selection algorithm to operate in a slower but much more thorough manner of searching for the optimal set

of component models to employ. See here for details on this very useful option.

XVAL STEPWISE

This option causes internal (invisible to the user) cross validation to be used to evaluate the selection criterion for stepwise selection. See here for details on this occasionally useful but slow option.

CRITERION = Criterion

This mandatory specification is the criterion that is used to select models in the stepwise selection procedure. See here for the extensive list of legal options.

MIN CRITERION CASES = Integer
MIN CRITERION FRACTION = RealNumber

It is required that one or the other of these two specifications be used. This limits computation of the trading threshold in such a way that a meaningful number of trades are executed. See here for details on this pair of specifications.

RESTRAIN PREDICTED

This option causes extreme values of the target to be truncated. If the target variable contains extreme values, this option will almost always improve performance by stabilizing the training process. See here for details.

OVERLAP = Integer

This option should be used if the target looks ahead more than one bar. It prevents optimistic bias in cross validation and walkforward testing due to overlapping edges. See here for details.

FRACTILE THRESHOLD

This option, legal only ifmultiple markets are present, causes trade decisions to be based on relative predictions among the markets, as opposed to the absolute prediction for each market. See here for details.

MCP TEST = Integer

This option causes a Monte-Carlo Permutation Test to be performed after cross validation or walkforward testing. See here for details.

We will now discuss the various committee types. The types differ with respect to the method used to combine the committee's inputs.

The AVERAGE Committee

The simplest, most intuitive method for combining the predictions of several models is to average them. This is what the AVERAGE committee does. It adds the predictions of all of the component models and divides by the number of models. Here is a sample declaration of an AVERAGE committee. It is given six models from which to choose up to three for averaging. By setting STEPWISE RETENTION to a huge number, we guarantee that the committee will try every possible trio of models, and then select the trio having maximum profit factor.

```
COMMITTEE COMM_AVG IS AVERAGE [
   INPUT = [ MOD_EXCL1 MOD_EXCL2 MOD_DIFFVARS
             MOD_DIFFCRIT MOD_SUBSAMP MOD_RESAMP ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 3
   STEPWISE RETENTION = 999999
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

The main advantage of the AVERAGE committee is that, other than selection of the models to include, it does no optimization. As a result, it is the least likely of all committees to overfit the data. This strength is also its weakness; it has no choice but to weigh all models as equally important, even though in fact some models may be better than others. Here is a sample of the output produced by an AVERAGE committee as it appears in the audit log file:

```
AVERAGE Committee COMM_AVG predicting DAY_RETURN_1
Stepwise based on pf with min fraction=0.100  (Final best
crit = 0.4128)
          0.333333  MOD_EXCL2
          0.333333  MOD_DIFFCRIT
          0.333333  MOD_RESAMP
```

The LINREG (Linear Regression) Committee

We saw that the main disadvantage of the AVERAGE committee is that it gives equal weight to all models, even though some models may be better than others. The obvious solution to this problem is to use ordinary linear regression to combine the predictions of the component models into a single pooled prediction. Unfortunately, we then lose a major advantage of the AVERAGE committee: resistance to overfitting. Computing optimal weights for the models is another stage of fitting the data, which provides the training process with another opportunity to model noise in addition to authentic patterns. Still, if the dataset is very large, linear regression can be an effective way of forming a committee. Here is a sample LINREG committee definition, followed by the trained output as it appears in the audit log file. As is common in committees with relatively few component models, it is good to set STEPWISE RETENTION to a large value to force testing of a possible combinations of models.

```
COMMITTEE COMM_LIN IS LINREG [
   INPUT = [ MOD_EXCL1 MOD_EXCL2 MOD_DIFFVARS
             MOD_DIFFCRIT MOD_SUBSAMP MOD_RESAMP ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 3
   STEPWISE RETENTION = 999999
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

```
LINREG Committee COMM_LIN predicting DAY_RETURN_1
Stepwise based on pf with min fraction=0.100  (Final best
crit = 0.4101)
Regression coefficients:
          4.243192  MOD_EXCL2
          0.668108  MOD_DIFFCRIT
          1.110925  MOD_RESAMP
         -0.160828  CONSTANT
```

Note that in some unusual situations, one or more model weights can be negative. It can even happen that some weight(s) may be enormous, while some may be large negative numbers. This generally happens when two or more of the models are so similar that their outputs are highly correlated. If this is observed, the developer should search for such correlation and remove redundant models. Such a situation is dangerous and should always be avoided.

Constrained Linear Regression Committee

There is a way to obtain an excellent compromise between averaging model predictions and using linear regression to compute optimized weights. The technique is to compute weights that are optimal in a sense that is related to the relative importance or quality of the models, but apply some effective constraints to the values of these weights. There are two constraints that are used:

1) No weight is allowed to be negative. This makes sense, because it is silly to think that the prediction of a model should be inverted in the pooled prediction. This would be a poor reflection on the quality of the model!

2) The weights must sum to one. This prevents highly correlated models from driving linear regression weights to ridiculous values. It also makes sense that the total contribution of all models should be one.

Much experience indicates that this is an excellent committee, one which has the ability to weight models according to their quality while not allowing as much overfitting as linear regression. Here is a sample CONSTRAINED committee definition, followed by the trained output as it appears in the audit log file. As is common in committees with relatively few component models, it is good to set STEPWISE RETENTION to a large value to force testing of all possible combinations of models.

```
COMMITTEE COMM_CONSTR IS CONSTRAINED [
   INPUT = [ MOD_EXCL1 MOD_EXCL2 MOD_DIFFVARS
             MOD_DIFFCRIT MOD_SUBSAMP MOD_RESAMP ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 3
   STEPWISE RETENTION = 999999
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;

CONSTRAINED Committee COMM_CONSTR predicting DAY_RETURN_1
Stepwise based on pf with min fraction=0.100  (Final best
crit = 0.3928)
Regression coefficients:
          0.002639  MOD_EXCL1
          0.295632  MOD_EXCL2
          0.701729  MOD_RESAMP
```

Models as Committees

Most of the models available in *TSSB* can be used as committees. In other words the model becomes the combining mechanism. In addition to the LINREG already shown, these include the GRNN, MLFN, TREE, FOREST, and BOOSTED TRI In order to use any of them as a committee, just follow the pattern of the prior three examples. Here are examples of the first line of declarations:

```
COMMITTEE COMM_GRNN IS GRNN [

COMMITTEE COMM_MLFN IS MLFN [

COMMITTEE COMM_TREE IS TREE [

COMMITTEE COMM_FOREST IS FOREST [

COMMITTEE COMM_BOOSTREE IS BOOSTED TREE [
```

However, experience indicates that none of these is particularly useful as a committee, compared to the AVERAGE and CONSTRAINED committees alread described. Models other than AVERAGE and CONSTRAINED are either to powerful, and hence prone to overfitting, or they are too weak and hence worthless. As a general rule, if the training set is small, the AVERAGE committee, with its considerable immunity to overfitting, is best. If the dataset is large, the CONSTRAINED committee is an excellent way to weight the models according t their abilities. These two workhorse committees are all that most people will ever need.

Creating Component Models for Committees

There are many ways to create a diversity of models whose predictions will be combined by means of a committee. This section will give examples of some of them. These are the models referenced as committee inputs in the prior sections. But first, there are a few general principles to keep in mind:

- It is a common myth that the models must be independent, or nearly so. This is a nice ideal, because the more independent sources of information the committee has, the better will be its predictions. However, this ideal is rarely, if ever reached. Even highly correlated models are usable, because to whatever degree we have independent information, even if very little, a good committee can use it.

- The two most popular and generally effective committees for use with numeric prediction (as opposed to classification) are simple averaging (the AVERAGE committee) and constrained linear combination (the CONSTRAINED committee). Rarely, ordinary linear regression (th LINREG committee) is appropriate, but it must*never* be used if there is substantial correlation among the models.

- The AVERAGE committee is safer to use (less prone to overfitting) than the CONSTRAINED committee, but if your training set is very large and yo have reason to believe that there might be considerable differences of predictive ability among the models, the CONSTRAINED committee wi probably be the better choice.

- The two most common methods for generating component models are by varying the indicators chosen as predictors and by varying the cases selected from the training set. These are both effective, and they complement each other. Neither should be considered superior to the other.

- There are two common methods for varying the indicators selected as predictors. The most reliable but not necessarily most effective method is direct intervention: forbid indicators used in some models from being used in other models. *TSSB* implements this by means of the EXCLUSION GROU option described here. The other method is to employ different selection criteria (the CRITERION specification describedhere). The latter method has the advantage of favoring 'good' indicators, but it is not guaranteed to produce different indicators.

- There are two common methods for varying the cases selected from the training set. Subsampling (the SUBSAMPLE option describedhere) takes a random subset from the training set. Thus, the new training set is smaller than

the original training set, and no cases are duplicated. This lack of duplication is crucial for some models, such as the GRNN. Resampling (the RESAMPL option described here.) randomly selects cases from the original training set, building a set of as many cases as in the original set, and allowing duplication. The fact that resampling preserves the size of the training set is often good, as long as the model can handle duplicated cases.

- Varying the indicators produces models that approach the problem from different directions. Models that use different indicators may find different patterns that enable the same quantity to be predicted, but by means of different information. In contrast to this, varying the cases selected from the training set may, as a side effect, also result in different indicators being selected. But more importantly, varying the cases results in the component models seeing similar legitimate patterns but different noise patterns. Thus, when the models are combined via a committee, the legitimate patterns reinforce while the noise patterns cancel.

Exclusion Groups

The most straightforward method of generating a set of models for committee use is with the EXCLUSION GROUP option described here. Models that have the same exclusion group number are guaranteed to use different indicators. This guarantee is the principle advantage of this method. On the other hand, it may happen that all of the useful indicators are 'used up' by the first model(s) and subsequent models are worthless because their indicators are worthless. For this reason, the EXCLUSION GROUP method should not be used to generate a large number of models. Indicator quality may be depleted quickly.

Here is an example of two models that are guaranteed to use different indicators by virtue of the EXCLUSION GROUP option:

```
MODEL MOD_EXCL1 IS LINREG [
   INPUT = [ CMMA_5 - CMMA_20N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   EXCLUSION GROUP = 1
   ] ;
```

```
MODEL MOD_EXCL2 IS LINREG [
   INPUT = [ CMMA_5 - CMMA_20N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   EXCLUSION GROUP = 1
   ] ;
```

Explicit Specification of Different Indicators

The user may have different families of indicators available. For example, there may be several different lookback periods, or several different ways of measuring trend or volatility. Perhaps the developer could put all indicators having a 10-day lookback into one model, and those having a lookback of 15 days into another model. Combining the predictions of those two models might provide better results than using just one or the other.

Here is a model that uses indicators that are explicitly different from the prior models. The input list for this model does not contain any indicators that are in the input list for the prior two models.

```
MODEL MOD_DIFFVARS IS LINREG [
   INPUT = [ LIN_ATR_7 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

Using Different Selection Criteria

Yet another way of (hopefully) finding models with different indicator sets is to employ different stepwise selection criteria. The following model is identical to the models from the EXCLUSION GROUP example except for the fact that thos models selected their indicators based on profit factor, while the model in this example selects based on R-square. These different criteria can provide useful alternative sets of information to the committee. Of course, we run the risk that different criteria may end up selecting the same indicator sets, in which case we have gained nothing.

```
MODEL MOD_DIFFCRIT IS LINREG [
   INPUT = [ CMMA_5 - CMMA_20N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

Varying the Training Set by Subsampling

The SUBSAMPLE option describedhere can be used to select a random subset of the original training set. If this is done for several (often a great many) models, the predictions of these models can be combined via a committee. Because the different models will generally be based on different training data, the patterns of noise will be different in the training sets. As a result, if the models are overly powerful and thereby learn to predict noise in addition to authentic patterns, they will generally make different predictions of the noise component. When these predictions are combined via a committee, the noise predictions will tend to cancel, while the predictions based on authentic patterns will tend to reinforce.

The principle disadvantage of subsampling is that the size of the training set is reduced. The implication is that there are fewer exemplars of authentic patterns, making it more difficult for the models to learn these authentic patterns. The models may be confused by the presence of noise. This leads to an uncomfortable tradeoff: keeping a smaller subsample of the original training set produces more variety in training data, which is what we need for good committee performance. But the smaller subsample also interferes with a model's ability to learn what it needs to make good predictions. On the other hand, if we keep a high percentage of the training set, we reduce that problem but we also reduce the variety of predictions. It becomes more likely that the component models will respond similarly to noise, because there is a lot of overlap in the training data across models. Choosing an appropriate subsample size can be an agonizing decision if training data is limited.

Here is the subsampled model that is used in the prior committee examples:

```
MODEL MOD_SUBSAMP IS LINREG [
   INPUT = [ CMMA_5 - CMMA_20N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   SUBSAMPLE 70 PERCENT
   RESTRAIN PREDICTED
   ] ;
```

Varying the Training Set by Resampling

We saw in the prior section that subsampling forces the developer into an unpleasant decision, trading precious sample size for equally precious variation in the training set. There is often a way around this dilemma: resampling. This can be achieved with the RESAMPLE option described here.

In resampling, we do not keep a subset of the original training set. Rather, we repeatedly select (with replacement) randomly chosen cases from the original training set, doing this as many times as there are cases in the original training set. Thus, the size of the training set is not reduced. On the other hand, this is not necessarily the boon that it may seem at first. For one thing, some models (notably the GRNN) cannot deal well with repeated training cases. But a more subtle problem is that even though we have a larger training set than we have with subsampling, we do not really have more information. The 'additional' cases that resampling has relative to subsampling are just duplicates of existing cases. Patterns of noise may actually be emphasized with resampling if particularly noisy cases are duplicated. For this reason, resampling should not necessarily be considered superior to subsampling, even for models that tolerate duplicated training cases.

Here is the resampled model that is used in the prior committee demonstrations:

```
MODEL MOD_RESAMP IS LINREG [
   INPUT = [ CMMA_5 - CMMA_20N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESAMPLE
   RESTRAIN PREDICTED
   ] ;
```

Oracles

The committees discussed in the prior chapter are a valuable way of combining the predictions of several models. However, ordinary committees have one property that can be a limitation in some applications: the models are combined with the same 'formula' for every case. In other words, the relative importance of the models is determined by examining the entire training set, and the models' predictions are weighted according to this fixed importance forever after.

However, it may be that some extraneous factor shifts the relative importance of the component models. For example, Model *A* may be twice as 'accurate' as Model *B* when the market is trending, but Model *B* may be better than Model *A* in a flat market. An oracle is an advanced form of a committee that allows the relative importance of the component models to vary according to the values of one or more *gate variables*. The gate variable(s) provide the oracle with information about relevant extraneous market conditions. For example the gate variable might tell us the degree to which the market was trending, or the current volatility of the market.

Oracles do have two disadvantages relative to most committees. First, the use of a gate variable adds power to the prediction system, meaning that overfitting is more likely with an oracle than with an otherwise similar committee. (Of course, increasing the size of the training set can mitigate overfitting, but this is not always possible.) Second is that training time for an oracle can be horrendous, roughly on the order of almost the cube of the number of training cases. Therefore, we have a discouraging conflict: because of the danger of overfitting, we need a large training set. But training time can be impractically long with a large training set. The key to success is to use only one gate variable (multiple gates are legal, but training time and risk of overfitting explode with more than one gate), and use as few component models as possible. In many practical applications, the oracle will combine the predictions of just two component models. By keeping the problem small, we can usually manage the situation.

There are two quite different philosophies for using an oracle. Neither is inherently superior to the other. Different applications may favor different alternatives. In what might be called the 'traditional' approach, every model is trained on every case in the training set. The gate variable then controls the differential weighting of the predictions of the component models. The other approach is to train specialist models via the PRESCREEN option (here). With this method, the training of each component model is done with only a subset of the entire training set, according to a market condition defined by the value of the prescreen variable(s). Then the oracle is used to choose the appropriate model(s) according to the prescreen condition(s). These two approaches will be discussed separately in this chapter,

and we will conclude with some hybrid alternatives.

Model Specifications Used by Oracles

There are many options and specifications already discussed in the context of models that are also usable for oracles. Rather than repeating the details of each, they will be listed here in brief summary. When appropriate, a reference is given to the page in the model section where more details can be found.

Note that unlike the case for committees, stepwise selection of the models is not allowed for oracles. All models named in the INPUT statement are employed by the oracle in making its prediction. For this reason, no option related to stepwise selection is allowed.

INPUT = [ModelNames]

Thismandatory specification names the models whose outputs are to be used as inputs to the oracle. Note that when the INPUT command is used for model, it lists indicators. But when the INPUT command is used for a committee or oracle, it lists models. Each model must be named individually. The dash (-) option that is usable for model inputs is not legal for oracle inputs.

OUTPUT = VariableName

This mandatory specification names the target variable. It is legal for this to be different from the targets of the component models, but it is hard to imagine a situation in which this would be appropriate. In virtually all cases of interest, all component models as well as the oracle share the same target.

PROFIT = VariableName

This option must be used when the OUTPUT variable is not a measure of profit. The PROFIT option names a variable that measures profit and is used for profit-based performance criteria such as profit factor. See here for details on this important option.

LONG ONLY
SHORT ONLY

One of these options specifies that the oracle will execute only long (or short) trades. Probably the only use of this option is in creating separate trading systems that specialize in only long or short trades and then combining them into a Portfolio.

MIN CRITERION CASES = Integer
MIN CRITERION FRACTION = RealNumber

It is required that one or the other of these two specifications be used.

This limits computation of the trading threshold in such a way that a meaningful number of trades are executed. See here for details on this pair of specifications.

RESTRAIN PREDICTED

This option causes extreme values of the target to be truncated. If the target variable contains extreme values, this option will almost always improve performance by stabilizing the training process. See here for details.

OVERLAP = Integer

This option should be used if the target looks ahead more than one bar. It prevents optimistic bias in cross validation and walkforward testing due to overlapping boundaries. See here for details.

FRACTILE THRESHOLD

This option, legal only if multiple markets are present, causes trade decisions to be based on relative predictions among the markets, as opposed to the absolute prediction for each market. See here for details.

MCP TEST = Integer

This option causes a Monte-Carlo Permutation Test to be performed after cross validation or walkforward testing. See here for details.

Traditional Operation of the Oracle

Regardless of whether the oracle will operate in the traditional mode, as described in this section, or the prescreen mode, as described in the next section, the user must name one or more gate variables using the following specification:

```
GATE = [ GateNames ]
```

Because it is legal (though almost never advisable) to specify more than one gate, the list of gate variables must be enclosed in square brackets, even if there is only one gate variable.

In the traditional operation of an oracle, no PRESCREEN option is used for an model. Thus, every model is trained with every case, and the prediction of every model will play a role in the output of the oracle. The relative weight of each model's prediction varies in a smooth manner as a function of the value of the gate variable(s).

An example of traditional operation is shown on the here. First, we have the

definition of two models. This example uses the EXCLUSION GROUP optic (here) to guarantee that the models are different. This is quick, simple, and legitimate. However, in many real-life applications, the diverse component models will be more cleverly designed, such as by using different lookback periods or different methods for measuring trend.

```
MODEL MOD_TRAD1 IS LINREG [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   EXCLUSION GROUP = 1
   ] ;

MODEL MOD_TRAD2 IS LINREG [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   EXCLUSION GROUP = 1
   ] ;
```

Here is the definition of a traditional oracle that combines the predictions of the two models. The gate variable is PVARRAT_10, which is a measure of volatility.

```
ORACLE ORACLE_TRAD [
   INPUT = [ MOD_TRAD1 MOD_TRAD2 ]
   GATE = [ PVARRAT_10 ]
   OUTPUT = DAY_RETURN_1
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

Finally, here is the trained oracle as it appears in the audit log file:

```
ORACLE  ORACLE_TRAD predicting DAY_RETURN_1
Stepwise not used; all predictors available to model
Sigma weights:
          0.012962  PVARRAT_10
```

When only one gate variable is used, which should nearly always be the case to avoid overfitting and slow training, printing the sigma weight for the single gate variable is of marginal value. Sigma weights are usually of interest only for

comparing several gates. In this situation, smaller values of the sigma weight equate to more importance, assuming that the gate variables have similar variances. Still, the sigma weight is printed even if there is only one gate. An unusually large value (which indicates that the gate is worthless) or an unusually small value (which indicates that the gate is of suspiciously high importance) might be of interest to some developers.

Prescreen Operation of the Oracle

The prior section gave an example of traditional operation of an oracle, in which every case in the training set takes part in training every model, and every model participates in each of the oracle's predictions. Only the input model's relative contributions to the oracle's prediction vary in accord with the gate variable. This section demonstrates an alternative approach: the component models use the PRESCREEN option (here) so that the models are specialists, trained on different subsets of the complete training set. Then, the oracle is used to choose the appropriate model or models each time it is invoked. A prescreen-based oracle can make its prediction at a given point in time based on only a subset of its input models.

For prescreen operation of the oracle, the declaration of the oracle contains one or two specifications in addition to the traditional version's specifications. These are the following. The first is mandatory for prescreen operation, and the second speeds training greatly when it can be employed.

```
HONOR PRESCREEN
PRESCREEN ONLY
```

The HONOR PRESCREEN Option

The first of these two commands, HONORPRESCREEN, tells the oracle disregard the predictions of models whose PRESCREEN option(s) disqualify th case under consideration. For example, suppose a component model has the following option in its declaration:

```
PRESCREEN  X > 0
```

Now suppose a case is presented to all of the component models, including the one just sited, and the oracle is asked to combine the models' predictions into a pooled prediction. If the case has a value of a gate variable X that is greater than zero, this model's prediction will go into the pooled prediction. But if X is less than or equal to zero for this case, the prediction of this model will be ignored when the pooled prediction is computed.

The PRESCREEN ONLY Option

The PRESCREEN ONLY option is extremely useful, but one must be careful t employ it only when it is appropriate. When this option appears in an oracle

declaration, all training of the oracle is skipped. This, of course, changes the training time of the oracle from potentially huge to zero! The resulting untrained oracle will thus act as a simple switch, choosing one and only one model's prediction for each case. This option is appropriate in only one situation: there is one gate, the prescreen variable, and the prescreen regions for the component models are mutually exclusive and exhaustive. In other words, any possible case will fall into the prescreen region of exactly one model. In this specific situation there is nothing to train.

Seen yet another way:

1) There must be no overlap; no case must satisfy the prescreen condition for more than one model.

2) Every possible case must be covered by some model; it must never happen that a case fails to satisfy the prescreen condition of any model.

Here is an example of a pair of prescreen conditions which, if used in separate models, would justify use of the PRESCREEN ONLY option in the oracle. Th prescreen regions are mutually exclusive and exhaustive.

```
PRESCREEN  X <= 0.0
PRESCREEN  X > 0.0
```

The following example shows a situation in which the PRESCREEN ONLY optio cannot be used because the prescreen regions are not exhaustive; the value 0.0 is omitted, and so a case for which X=0.0 would be rejected by both models.

```
PRESCREEN  X < 0.0
PRESCREEN  X > 0.0
```

Finally, we have an example that shows a situation in which the PRESCREEN ONLY option cannot be used because the prescreen regions are not mutually exclusive; they overlap. Values greater than zero but less than one are covered by both models.

```
PRESCREEN  X < 1.0
PRESCREEN  X > 0.0
```

Note that the conditions in which the PRESCREEN ONLY option is inappropriat are not automatically detected by the program; *it is the user's responsibility to make sure that use of this option is appropriate*. Failure to do so will result in incorrect results with no warning given.

An Example of Prescreen Operation

Here are the definitions of a pair of prescreened models, followed by the definition of an oracle that operates in prescreen mode. Because the prescreen regions of the models are mutually exclusive and exhaustive, we can use the PRESCREEN ONLY option in the oracle declaration in order to skip the time-consuming training for the oracle.

```
MODEL MOD_PS1 IS LINREG [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   PRESCREEN PVARRAT_10 <= 0.0
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;

MODEL MOD_PS2 IS LINREG [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   PRESCREEN PVARRAT_10 > 0.0
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;

ORACLE ORACLE_PS_ONLY [
   INPUT = [ MOD_PS1 MOD_PS2 ]
   GATE = [ PVARRAT_10 ]
   OUTPUT = DAY_RETURN_1
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   HONOR PRESCREEN
   PRESCREEN ONLY
   ] ;
```

More Complex Oracles

So far we have discussed two methods for using oracles. The 'traditional' method trains all models using all cases in the training set. The 'prescreen' method trains specialist models that typically employ prescreen regions that are mutually exclusive and exhaustive, and the prescreen variable is the only gate. These two methodologies should suffice for the majority of applications. However, the oracle implemented in *TSSB* allows much more generality. Here are a few thoughts on alternative ways to use an oracle.

In order to understand this advanced section, the reader must be clear on the distinction between the PRESCREEN statement and the GATE statement:

A prescreen variable is part of a model definition. It is used to segment the training cases such that the model will be trained on only certain cases. A gate variable is part of an oracle definition. It determines which of the models in an oracle will participate in the prediction made for a given case.

Consider the prescreen operation discussed in the prior section. Remember that when a prescreened model is used in an oracle, one should make the prescreen variable a gate and specify the HONOR PRESCREEN option in the oracle declaration. Otherwise the model will be trained as a specialist, but the specialization will be ignored when the model is used by the oracle. This is silly. However, in a prescreened situation, it is perfectly legal to also use one or more gate variables that are not prescreen variables. Of course, when more than one gate is used, training time can explode, along with the likelihood of overfitting. Still, it's worth considering this possibility. Here is the 'prescreen' oracle we saw in the previous example, but we have added a second gate. The indicator PVARRAT_10 was used to prescreen the two component models, but PVARRAT_20 was not used for prescreening. Because of this additional gate, we can no longer use the PRESCREEN ONLY option.

```
ORACLE ORACLE_PS_EXTRA [
   INPUT = [ MOD_PS1 MOD_PS2 ]
   GATE = [ PVARRAT_10 PVARRAT_20 ]
   OUTPUT = DAY_RETURN_1
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   HONOR PRESCREEN
   ] ;
```

Here is another variation on oracle use. In the example on the prior page, we employed two prescreened models, MOD_PS1 and MOD_PS2. They used thes prescreen regions:

```
PRESCREEN PVARRAT_10 <= 0.0
PRESCREEN PVARRAT_10 > 0.0
```

These two models have prescreen regions that are totally separate; there is no overlap at all. Now suppose we introduce a third model that provides extra coverage in the area around zero. This might be nice for those cases that are not very distant from zero. Cases near zero might not be handled well by either MOD_PS1 or MOD_PS2 because those models might focus on values o PVARRAT_10 that are distant from zero. Here is this 'middle-of-the-road' specialist model:

```
MODEL MOD_PS3 IS LINREG [
   INPUT = [ CMMA_5 - CUB_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   PRESCREEN PVARRAT_10 > -10.0
   PRESCREEN PVARRAT_10 < 10.0
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   ] ;
```

We can include this model in the oracle. Here is the definition of a suitable oracle. We can no longer use the PRESCREEN ONLY option, because the prescree regions of the three component models are not mutually exclusive. They overlap in the region around zero.

```
ORACLE ORACLE_PS_MORE [
   INPUT = [ MOD_PS1 MOD_PS2 MOD_PS3 ]
   GATE = [ PVARRAT_10 ]
   OUTPUT = DAY_RETURN_1
   MIN CRITERION FRACTION = 0.1
   RESTRAIN PREDICTED
   HONOR PRESCREEN
   ] ;
```

Testing Methods

TSSB contains a useful variety of methods for testing the performance of predictive-model-based trading systems. These testing methods feature the following:

- In-sample performance of the entire dataset
- Walkforward testing by years, months, or days
- Cross validation by years, months, or days
- Cross validation based on values of a flag variable
- Cross validation by random blocks
- The ability to preserve out-of-sample predictions for export and graphical display
- Performance of trades triggered when the market is in a defined state
- Performance of trades triggered when the market transitions to a defined state

This chapter describes in detail the various testing options available in *TSSB*. They should be invoked late in the script file, at the bottom, ***after all transforms, models, committees, and oracles are define***d. The tests are performed in the order in which they appear in the script file.

Performance for the Entire Dataset

Sometimes we want to create a predictive-model-based trading system that is optimized for the entire dataset we have available. This is done with the following simple command:

```
TRAIN
```

The performance of this system, often called the in-sample performance, will be optimistically biased. Still, there are several reasons why we might want to do this anyway:

- The true expected future performance will, on average, be inferior to that observed by optimizing for the entire dataset. Thus, if performance from the TRAIN command is inadequate, we might as well go back to the drawing board.
- Stepwise selection of indicators will tell us which indicators are most valuable when the entire dataset is taken into account.
- Although trading frequency is highly variable, we can get at least a rough idea of how often the system would trade if put into use.
- We may find that our trading system is much more effective for long trades than for short trades, or vice versa. This can be interesting information.
- The nature of the predictive model(s), such as the coefficients of a linear model or the relative importance of models being pooled by a committee are always interesting.

Walkforward Testing

By far the most widely used and accepted method of evaluating the performance of an automated trading system is *walkforward testing*. In this testing methodology, the trading system is trained with data up to a certain date. The trained system is then executed with a batch of data that begins on the next bar after the training period and that ends a fixed time interval later. The cutoff date for training is then moved up to the end of the period just tested and the process is repeated. This procedure is explained in detail here. That discussion should be reviewed if necessary.

The key parameter for walkforward testing is the length of time that the system is tested after the training period ends. This parameter involves an important tradeoff. Markets change to some degree over time, so we can get the most accurate estimate of performance in walkforward testing if we test for only a short time interval after training ceases. However, because the training period is moved forward by the amount of time tested in the prior round, the number of training cycles required is inversely proportional to the testing period. For example, suppose we are working with daily data and we set the end of the first training period to a date three years prior to the current date, so that we have three years of out-of-sample data to work with. If we test for one year after training, then three training cycles (three folds of training/testing) are needed, one for each OOS year. But if we test for just one day after training, and if we assume 250 trading days in a year, then we will need 3*250=750 training cycles. If training requires a significant amount of computer time, we may be limited in our choices for the testing period length.

TSSB allows three test period lengths: a year, a month, or a day. These are invoked by the following commands. In each case, the user specifies two parameters: the length of the training period and the starting date. The date is as indicated.

```
WALK FORWARD BY YEAR  TrainingYears  YYYY ;
WALK FORWARD BY MONTH  TrainingMonths  YYYYMM ;
WALK FORWARD BY DAY  TrainingDays  YYYYMMDD ;
```

Here are two examples: Walk forward by testing a year at a time, beginning testing at the start of 2005 and using 6 years of training data each time:

```
WALK FORWARD BY YEAR  6  2005 ;
```

Walk forward by testing one day at a time, beginning testing on June 15, 2008 and training 500 days of history each time:

```
WALK FORWARD BY DAY  500  20080615 ;
```

Cross Validation by Time Period

If you want to make more efficient use of the data than is obtained by walkforward testing, and you are willing to assume that the market indicators and targets are reasonably stationary, and if you are willing to accept the possibility of some small bias, then cross validation can be used instead of walkforward testing. The mechanics of this technique are described in detail here. That material should be reviewed if necessary.

As with walkforward testing, the fundamental parameter for cross validation is the length of the time period withheld for testing during each fold. This involves a tradeoff: by keeping the quantity of data withheld for testing small, we maximize the size of each training set, which generally results in more accurate performance estimates. However, this also results in a large number of training cycles being necessary, which may be impractical if the time required for training is significant.

TSSB allows three withholding-period lengths: a year, a month, or a day. These are invoked by the following commands:

```
CROSS VALIDATE BY YEAR ;
CROSS VALIDATE BY MONTH ;
CROSS VALIDATE BY DAY ;
```

It should be noted for completeness that strictly speaking, cross validation is not totally unbiased. The OVERLAP option (here) eliminates the most serious source of bias. However, there are several other sources of bias in cross validation, all of which are intensely theoretical and far beyond the scope of this text. On the other hand, much practical experience indicates that the bias is not only small, but much more likely to be pessimistic than optimistic. For these reasons, cross validation is widely used and accepted as an effectively unbiased method for estimating performance.

Cross Validation using a Control Variable

Some users may want to perform specialized types of cross validation. This can be done by including a control variable in the database. This control variable would need to be computed externally and read in with a READ DATABASE o APPEND DATABASE command. Here are a few forms of specialized cros validation that might provide useful information:

- Define folds by a time period other than the three (year, month, day) provided by the program. For example, the user might want to use a quarter (three months) as a fold. There would be as many folds as there are quarters in the dataset. The first quarter in the dataset might be coded with the value ‘1' for the first quarter, ‘2' for the second quarter, and so forth. If coded this way, the largest value of the control variable would be the number of quarters in the dataset.

- Determine whether the month is relevant to performance, or if the month is of no consequence. Code January as ‘1' and so forth, through December as ‘12' for all records. This will result in 12 folds. Because *TSSB* provides detailed results for every fold in cross validation, it is easy to see if some month over or under-performs relative to the rest of the year. Of course, unless the dataset is extensive, these results will have large error variance due to the small sample size after dividing the data into twelve months, so caution is advisable.

- Investigate whether some measurable market state impacts performance or is unrelated to performance. For example, one could code the control variable as ‘1' in periods of low recent volatility, ‘2' for moderate volatility, and ‘3' for high volatility. This will result in three folds. If volatility has negligible impact on the performance of the trading system, we would expect to see similar results in all three folds. If the results vary greatly, we should pursue the reason.

One must be careful to understand exactly what the second and third ideas just shown actually do. Consider the third option, and suppose we see that in the test fold with low volatility, performance is exceptionally good. It is tempting to believe that we have just shown that low-volatility regimes are especially predictable. In a sense this is a legitimate conclusion. However, recall that because of the nature of cross validation, the trading system was trained entirely on data that is *not* low volatility. The training set for this low-volatility test fold was cases with moderate and high volatility. Thus, what we have really demonstrated with this result is that volatility does impact performance. If volatility were unrelated to the performance of out trading system, all folds would have about the same

performance, despite the fact that we are training under some volatility condition and testing under another. In order to get a better estimate of how well we can do in a low-volatility environment, we need to use the PRESCREEN option (here) or TRIGGER techniques (here).

There are two ways to implement control-variable cross validation in *TSSB*. The most straightforward is generally used when the control variable has a fixed value for each fold. This would be the case in all three examples mentioned on the prior page. The command is:

```
CROSS VALIDATE BY VarName ;
```

When this command appears in the control script file, cross validation is performed using a separate fold for each unique value of the variable VarName. The actual values are irrelevant. For example, suppose you want three folds. The values of the control variable could be 0, 1, and 2. Or they could be 1, 2, and 3. Or they could be 21, -5, and 17. Each unique value produces a fold that contains all cases having this value.

Sometimes it is more convenient to use a continuous variable for fold control, perhaps because such a variable already exists and the user does not want to have to go to the trouble of recoding it into a few discrete values. For example, recall the third example given earlier, in which we divide the dataset into low, moderate, and high volatility regimes. Rather than having to recode volatility into values of 1, 2, and 3 as was done in that example, it may be more convenient to work directly with a volatility variable. This can be done with the following command:

```
CROSS VALIDATE BY VarName ( BoundaryList ) ;
```

In the command, *BoundaryList* is a list of one or more numbers that are the boundaries of the folds. There will be one more fold than there are boundaries. For example, consider this command, which contains two boundaries and hence produces three folds:

```
CROSS VALIDATE BY VarName ( -20.0 25.0 ) ;
```

Values less than -20 go into one fold. Values greater than or equal to -20 but less than 25 go into a second fold. Values of 25 or more go into the third fold.

This command should not be used if the target looks ahead more than one bar. Otherwise, boundary effects will cause potentially serious optimistic bias in results. The OVERLAP option (here) is ignored, because fold boundaries may in general be distributed throughout the dataset.

Cross Validation by Random Blocks

Sometimes the user may not want to associate folds with any identifiable quantity. Instead, the user may want to randomly assign cases to folds. The following command will do so:

```
CROSS VALIDATE Nfolds RANDOM BLOCKS ;
```

In this command, *Nfolds* is the number of folds to use. It must be at least 2 and may be at most equal to the number of unique dates in the database.

In a multiple-market situation, records for a date are never separated into different folds by this command. In other words, fold membership is determined by randomly assigning *dates* to folds, not by randomly assigning *records* to folds. This is to ensure that the FRACTILE THRESHOLD option (here) works correctly.

This command should not be used if the target looks ahead more than one bar. Otherwise, boundary effects will cause potentially serious optimistic bias in results. The OVERLAP option (here) is ignored, because fold boundaries are distributed throughout the dataset.

Preserving Predictions for Trade Simulation

NOTE... The command discussed in this section is deprecated except for one uncommon use: data preparation for the TRADE SIMULATOR. This command has been preserved in the latest version of *TSSB* for the sake of backward compatibility with existing script files and invocation of the TRADE SIMULATOR. However all of its former actions are now accomplished automatically, without the necessity of using this command. In fact, using it in the latest version of *TTSB* may occasionally introduce anomalous behavior and should be avoided except when it is used in conjunction with the TRADE SIMULATOR.

If the TRADE SIMULATOR(see the manual) is invoked, it requires a special form of the predictions made by models, committees, and oracles. This is accomplished with the following command:

```
PRESERVE PREDICTIONS
```

This command may appear only once in the script file, and it makes sense only if at least one TRAIN, CROSS VALIDATE, or WALK FORWARD command appears before it. The most recent of these three commands affects the nature of the preserved predictions, as shown here:

TRAIN

The predictions cover the entire time period of the database. They are all in-sample.

CROSS VALIDATE

The predictions cover the entire time period of the database. They are all out-of-sample, being derived from the hold-out periods of the folds.

WALK FORWARD

The predictions cover only the beginning of the test period through the end of the database. These predictions are all out-of-sample, being derived from the walk-forward test periods. Values prior to the beginning of the test period are set to 0.0.

The TRADE SIMULATOR may be invoked any time after a PRESERVE PREDICTIONS command has appeared.

Market States as Trade Triggers

We already saw here how the PRESCREEN option can be used to create a mode that specializes in a particular market state as defined by a variable, and an ORACLE (here) can optionally be used to combine the predictions of several such models. But presceening as a method of specialization generally makes sense only for models, not committees or oracles. It would be absurd to use a committee to combine the predictions of models that specialize in different conditions. Even if the user wanted all models and committees to specialize in the same condition, duplicating PRESCREEN commands across a set of models as well as a committe would be a burden for the user. And oracles, by definition, are intended to merge all specialist models, so prescreening an oracle would usually be pointless. When the goal is to create an *entire trading system* that specializes in a particular market state, as opposed to just component models that specialize, *TSSB* offers an alternative called triggering that simplifies the process of creating such trading systems and evaluating their performance. here we will compare and contrast prescreening and triggering. For now, we will simply explore the concept of triggering.

The basic idea behind triggering is that a trigger variable defines a subset of the complete dataset. This subset is used for all training and out-of-sample testing. Thus, we can develop and evaluate the performance of an *entire trading system*, including all transforms, models, committees, and oracles, under a defined market condition. When under the influence of a trigger, *TSSB* behaves as if the database contains only cases that satisfy the trigger condition. All other cases are effectively removed from the database.

There are two types of trigger available in *TSSB*. The simplest version is based on the current value of the trigger variable. Typically, this variable would be binary, with a value of 1.0 indicating a triggered or *true* state (the case is retained) and a value of 0.0 indicating a non-triggered or *false* state (the case is ignored). However, the trigger variable does not have to be binary, because the actual threshold is 0.5 (the midpoint between *true* and *false*). Values greater than 0.5 are triggered, and values less than or equal to 0.5 are non-triggered. The examples presented soon will show how to apply a threshold to an indicator to generate a binary flag ideal for triggering.

A more complex and rarely used type of trigger responds not to the current state of the trigger variable, but rather to a change of state. This trigger happens when the variable transitions from being less than or equal to 0.5 (*false*) to being greater than 0.5 (*true*). This version will be discussed in detail later.

In order to implement the simple (current state) version of the trigger option, place

the following command in the control script file:

```
TRIGGER ALL  VarName ;
```

When this command appears, all cases that fail to satisfy the trigger condition (in other words, all cases whose value of the named variable is less than or equal to 0.5) are removed from the database. Thus, all subsequent TRAIN, CROS VALIDATE, and WALK FORWARD commands will operate on this reduced se those cases whose value of *VarName* exceeds 0.5.

It is legal to use more than one TRIGGER command. They do not cumulate. Rathe when a new TRIGGER command appears, all cases removed by the prio TRIGGER are first restored to the database, so the new TRIGGER command starting with a clean slate. Prior TRIGGER commands do not affect subsequer TRIGGER commands. They are totally independent.

It is also possible to 'undo' the effect of a TRIGGER command. The followin command will restore the entire original database:

```
REMOVE TRIGGER ;
```

So, for example, we might implement one trigger, do a cross validation, implement a different trigger, do another cross validation, and finally restore the original database for a cross validation with no triggering at all. Here is an example of this:

```
TRIGGER ALL Trig1 ;
CROSS VALIDATE BY YEAR ;
TRIGGER ALL Trig2 ;
CROSS VALIDATE BY YEAR ;
REMOVE TRIGGER ;
CROSS VALIDATE BY YEAR ;
```

An Example of Simple Triggering

On the here we see a script control file that demonstrates the relationship between using a model with the PRESCREEN option and using an ordinary model wit triggering. This example is naive in that both methods are equivalent here, as we will see in the explanation that follows the example script. In fact, their equivalence is the main point of this demonstration. However, if the trading system also included trainable transforms, committees or oracles, there would be no way to use PRESCREEN options to accomplish what we can do with the TRIGGE command. It is crucial that the user understand that PRESCREEN is a *property of a model* that causes that particular model to specialize in a certain market state. It has

no broader impact. On the other hand, a TRIGGER command literally *changes the database*, removing cases that fail to satisfy the trigger, thus impacting all subsequent aspects of operation, including training, testing, and even subsequent operation.

```
TRANSFORM HIFLAG IS EXPRESSION ( 0 ) [
   HIFLAG = ADX > 35.0
   ] ;

TRANSFORM LOFLAG IS EXPRESSION ( 0 ) [
   LOFLAG = ADX <= 35.0
   ] ;

MODEL MOD_FIRST IS LINREG [
   INPUT = [ LIN_ATR_7 LIN_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 0
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;

MODEL MOD_HI IS LINREG [
   INPUT = [ LIN_ATR_7 LIN_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   PRESCREEN HIFLAG > 0.5
   MAX STEPWISE = 0
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;

MODEL MOD_LO IS LINREG [
   INPUT = [ LIN_ATR_7 LIN_ATR_15N ]
   OUTPUT = DAY_RETURN_1
   PRESCREEN LOFLAG > 0.5
   MAX STEPWISE = 0
   CRITERION = RSQUARE
   MIN CRITERION FRACTION = 0.1
   ] ;

TRAIN ;
TRIGGER ALL HIFLAG ;
TRAIN ;
TRIGGER ALL LOFLAG ;
TRAIN ;
```

The first thing done in this script file example is to define two binary flags, HIFLAG and LOFLAG, based on the value of ADX. Expression transforms a discussed here. For now, take it as given that these two flags have the value 1.0

when *true* and 0.0 when *false*.

Model MOD_FIRST is a simple linear regression model with two indicators MOD_HI is the same model, but using the PRESCREEN option to cause it specialize in cases for which HIFLAG is true (> 0.5, the result of ADX bein greater than 35). Similarly, modelMOD_LO specializes in LOFLAG being true.

Here is the output for MOD_HI after the first TRAIN command. That comman processes the entire database, but the PRESCREEN option in MOD_HI causes tl model to use only a subset of the cases (those for which ADX>35).

```
LINREG Model MOD_HI predicting DAY_RETURN_1
Stepwise not used; all predictors available to model
PRESCREEN:
   HIFLAG > 0.50000
   6390 of 11187 (57.12 %) of database cases pass
   6390 of 11187 (57.12 %) of training set cases pass
Regression coefficients:
         -0.001408  LIN_ATR_7
         -0.000196  LIN_ATR_15N
          0.056058  CONSTANT
```

Now look at the output for MOD_FIRST produced by the TRAIN command th follows the TRIGGER ALL HIFLAG command:

```
Processing Expression HIFLAG in training fold
TRIGGER at 6390 of 11187 cases (57.12 percent)
LINREG Model MOD_FIRST predicting DAY_RETURN_1
Regression coefficients:
         -0.001408  LIN_ATR_7
         -0.000196  LIN_ATR_15N
          0.056058  CONSTANT
```

We will now see how, in simple situations, prescreening and triggering can produce identical results. Notice in the first set of output lines, the PRESCREEI option let 6390 of 11187 cases pass. The coefficients of the model are printed. In the second set, produced by the TRIGGER command, the program detected trigger at 6390 of 11187 cases, and we (naturally!) ended up with exactly the same model. Thus, if we have only models in the application, no trainable transforms, committees, or oracles, we can use the PRESCREEN option in the model, or w can use a TRIGGER command before training, cross validating, or walkin forward. They are equivalent. But if we have a trainable transform, committee, or oracle that must itself specialize, rather than just a model, we can only use triggering. This will be discussed in more detail here.

Triggering Based on State Change

A less commonly used but interesting method of triggering is to trigger on a change of state from *false* to *true* rather than triggering on the state itself. In other words, a case is consider to be 'triggered' when the trigger variable is *true* (greater than 0.5) for the current bar AND the trigger variable is *false* (less than or equal to 0.5) for the prior bar in that market. The bar prior to the first bar in the database is assumed to be *false*. In order to implement this trigger, use the following command:

```
TRIGGER FIRST VarName ;
```

Here is an application in which this would be useful. The indicator CMMA_1(which will be used in this example is the close of the current bar, minus the 10-bar moving average (with some scaling and compression). Like most indicators in the *TSSB* library, its natural range is -50 to 50.

Suppose we hypothesize that when a market breaks out to a much higher price than its recent history, this is often the beginning of a new bull market. But what if it's a false alarm? We want to use a predictive model to distinguish between those breakouts that signal a new bull market from those that do not. The variable CMMA_10 is useful as a breakout trigger, because it will have a large value wher the current bar closes far above recent history. We arbitrarily choose a threshold of 30 as the trigger value. Here is the script file code for this application:

```
TRANSFORM BREAKOUT IS EXPRESSION ( 0 ) [
   BREAKOUT = CMMA_10 > 30.0
   ] ;

MODEL MOD_NEW IS LINREG [
   INPUT = [ CMMA_5 - VMUTINF_3 ]
   OUTPUT = DAY_RETURN_1
   MAX STEPWISE = 2
   CRITERION = LONG PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;

TRIGGER FIRST BREAKOUT ;
WALK FORWARD BY YEAR 20 1983 ;
```

As in the prior example, we use an expression transform (see here) to create a binary flag that we call BREAKOUT. This flag will be *true* (1.0) when CMMA_1(exceeds 30, our chosen breakout condition. Otherwise the flag will be *false* (0.0). The model is simple linear regression with stepwise selection from a wide variety of indicators. We use the TRIGGER FIRST command to filter the database to retai only those cases in which the market first breaks out above its recent history, as indicated by CMMA_10 first exceeding 30. Then we walk this forward starting a

1983, using 20-year training sets.

The results of triggering for the entire dataset are as shown below. We see that the basic condition of BREAKOUT being *true* is satisfied for 1514 of the 11187 database cases. This is the number of cases we would have if we used the TRIGGER ALL command. But only 492 of these cases are the *first* occurrence in a block of contiguous *true* values, so this is the total number of cases we will have for the entire walkforward test. This is a disadvantage of the TRIGGER FIRS command; the size of the dataset is greatly reduced in most practical applications.

```
TRIGGER at 1514 of 11187 cases (13.53 percent)
FIRST = 492 of 11187 cases (4.40 percent)
```

This application performs surprisingly well, given the small size of the dataset after TRIGGER FIRST filtering. Here is the walkforward summary from the audit lc file:

```
Pooled out-of-sample...
   Target grand mean = 0.02097
      42 of 288 cases (14.58%) at or above outer high
           threshold  (Mean = 0.15545 versus -0.00199)
     100 of 288 cases (34.72%) at or below outer low
           threshold  (Mean = -0.02494 versus 0.04539)
MSE = 0.43529   R-squared = -0.04081   ROC area = 0.54712
Buy-and-hold profit factor = 1.098   Sell-short = 0.911
Dual-thresholded outer PF = 1.349
   Outer long-only PF = 1.995   Improvement Ratio = 1.817
   Outer short-only PF = 1.129  Improvement Ratio = 1.240
```

There were only 288 cases (first-occurrence breakouts) in the entire out-of-sample walkforward period, and only 42 of them passed the threshold for trading. However, these 42 cases had a profit factor of 1.995, which is greater than the buy-and-hold profit factor of 1.098 by a factor of 1.817.

Triggering Versus Prescreening

Superficially, prescreening models and triggering in the database seem to accomplish the same thing. And in a narrow range of circumstances, they are equivalent. However, they actually take profoundly different approaches to the task of regime specialization, so it is worth devoting an entire section to comparing and contrasting the two approaches.

If one were to seek a short, general distinction between the two techniques, it would be this:

- A system based on *prescreening* trades the market under *all conditions*, basing its decisions on one or more models, each of which were trained to handle specific market states (regimes). Only the *models* are trained to specialize. Transforms, committees, and oracles are trained using *all* market states, with no specialization. Summary results printed in the audit log concern the entire duration of the market, all regimes.

- A system based on *triggering* trades the market *only* when the market is in a specified state. All aspects of the trading system, including transforms, models, committees, and oracles, are specialists in the specified state. Summary results printed in the audit log reflect only trades executed when the market is in the specified state.

This section will present four different script files in order to clarify these distinctions. All four script files will include a linear regression transform (here) to illustrate how trained transforms relate to these two techniques. They will also include a few simple linear models that employ a limited number of simple trend indicators, as well as one volatility indicator. Here is a brief description of each of the four examples:

- The first example is a trading system based on prescreening of models. There are two models, one specializing in high volatility regimes, and the other specializing in low volatility. An oracle examines volatility and chooses the model(s) to use at each bar.

- The second example is similar to the first in that it uses two models and an oracle to combine them. However, it does not use prescreening to force specialization. Rather, it relies on the user's intuition and the power of an oracle to make intelligent choices.

- The third example uses triggering to develop and test an entire trading system that executes only in a high-volatility regime.

- The fourth example uses triggering to develop and test an entire trading system that executes only in a low-volatility regime, the complement to the high-volatility regime of the prior example. It also includes a PORTFOLIO command to combine the equity curves of the low and high volatility systems in order to provide information on the net performance across both regimes.

Commands Common to All Four Examples

There are some initial commands that begin each of the four examples. To avoid repeating them with each example, we'll look at them just once, now. Most of them are the straightforward initializations seen many times before: reading the market histories and computing the indicators from the built-in library. Here, we also use the RETAIN YEARS command to retain just three recent years, because the walkforward tests done later will walk forward only one year: 2011. Most practical applications will walk several years, but doing just one year of walkforward simplifies presentation of the example. Here are the initial commands that are common to all four examples:

```
RETAIN YEARS 2009 THROUGH 2011 ;
READ MARKET LIST "D:\BOOSTER\TEST\SYM_OEX.TXT" ;
READ MARKET HISTORIES "E:\SP100\OEX.TXT" ;
CLEAN RAW DATA 0.6 ;
READ VARIABLE LIST "D:\BOOSTER\TEST\EQUITY.TXT" ;

TRANSFORM LIN_TREND IS LINEAR REGRESSION [
   INPUT = [ LIN_ATR_5 LIN_ATR_15 ]
   TARGETVAR = RETURN
   ] ;
```

The commands above do contain one slightly unusual item, a linear regression transform. This transform will be discussed in detail here in the next chapter. Here, we note only that this transform, which is automatically integrated into the training/testing cycle, computes a least-squares-optimal linear equation for predicting the target from the two named linear trend indicators. This reduces the information in two indicators (LIN_ATR_5 and LIN_ATR_15 here) into just one new indicator, LIN_TREND. This operation is likely to increase the signal-to-noise ratio of the information. In most applications, the user would include more than two indicators in the linear regression transform, but we are keeping this example simple.

One other item to note is that the variable definition file EQUITY.TXT, used in several examples in other chapters of the tutorial, contains a volatility indicator called, reasonably enough, *P_VOLATILITY*. As is the case with nearly all

indicators in the built-in library, this indicator is defined in such a way that it is centered near zero and has a natural range of approximately -50 to 50. A histogram of this volatility indicator is shown in Figure 15 on the next page. In the forthcoming examples, we will use zero as the threshold for distinguishing between high volatility and low volatility because this is the natural center of this variable's range.

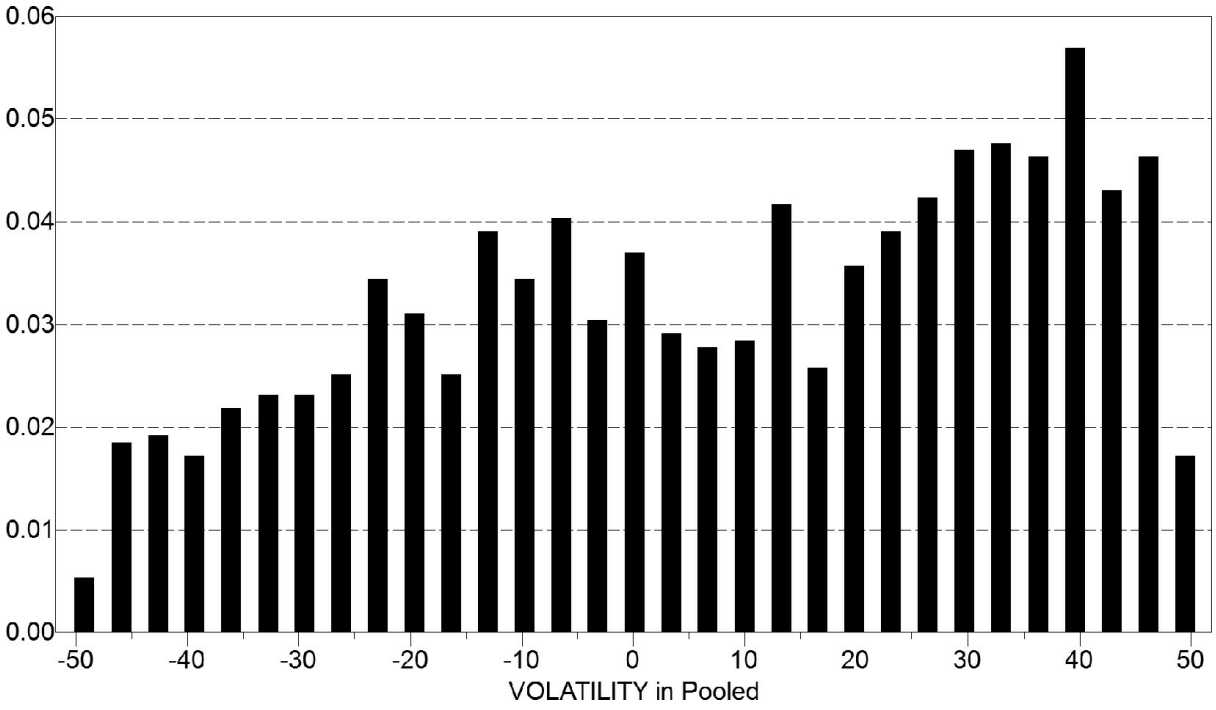

Figure 15: Histogram of the P_VOLATILITY indicator

Finally, for those who may be interested in the exact definitions of the indicators and target used in these four examples, here is the variable definition file EQUITY.TXT:

```
LIN_ATR_5: LINEAR PER ATR 5 100
QUA_ATR_5: QUADRATIC PER ATR 5 100
CUB_ATR_5: CUBIC PER ATR 5 100
LIN_ATR_15: LINEAR PER ATR 15 100
QUA_ATR_15: QUADRATIC PER ATR 15 100
CUB_ATR_15: CUBIC PER ATR 15 100
P_VOLATILITY: PRICE VARIANCE RATIO 5 4

RETURN:    NEXT DAY ATR RETURN 250
```

Example 1: Model Specialization via PRESCREEN

The first example uses two models, one specializing in high volatility regimes, and the other specializing in low volatility. An oracle examines current volatility and chooses the model to use. Here are the relevant commands:

```
TRANSFORM HI_VOLATILITY IS EXPRESSION ( 0 ) [
   HI_VOLATILITY = P_VOLATILITY > 0.0
   ] ;

MODEL MOD_HI IS LINREG [
   INPUT = [ LIN_TREND QUA_ATR_5 QUA_ATR_15 CUB_ATR_5
             CUB_ATR_15 ]
   OUTPUT = RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   PRESCREEN HI_VOLATILITY > 0.5
   ] ;

MODEL MOD_LO IS LINREG [
   INPUT = [ LIN_TREND QUA_ATR_5 QUA_ATR_15 CUB_ATR_5
             CUB_ATR_15 ]
   OUTPUT = RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   PRESCREEN HI_VOLATILITY < 0.5
   ] ;

ORACLE ORAC_HONOR [
   INPUT = [ MOD_HI MOD_LO ]
   GATE = [ HI_VOLATILITY ]
   HONOR PRESCREEN
   OUTPUT = RETURN
   MIN CRITERION FRACTION = 0.1
   ] ;

WALK FORWARD BY YEAR 1 2011 ;
```

The first operation employs an expression transform (here) to create a binary flag called HI_VOLATILITY that will have the value 1.0 (*true*) when the P_VOLATILITY indicator is greater than zero and 0.0 (*false*) when P_VOLATILITY is less than or equal to zero. (The threshold of zero was chose because it is the center of the indicator's natural range.)

Two linear models follow. (These are not to be confused with the linear regression transform mentioned earlier.) They are almost identical, in that they employ the

same predictor candidate list and target, and they have the same optimization parameters. The indicator candidates are output of the LIN_TREND linea regression transform along with an assortment of other indicators. The only difference is that MOD_HI uses the PRESCREEN option to cause it to specialize in high-volatility regimes, while MOD_LO specializes in low volatility. Note tha there is no requirement that the PRESCREEN variable be binary. These model could just as well reference the P_VOLATILITY indicator directly instead of goin through a transform that creates a binary flag. The effect would be the same. However, the TRIGGER command, which will be used in Examples 3 and 4, doe require a binary flag. So, for the sake of clarity, the same binary flag was used in the PRESCREEN option as well. This creates consistency across the examples.

The oracle (see here) ORAC_HONOR combines the predictions of these tw models. Because the HONOR PRESCREEN option is used, the choice of model based on the PRESCREEN variable: when a model's PRESCREEN is fals indicating that we are processing a market case outside the model's specialization, that model's prediction is ignored. Since we have deliberately designed these models' specializations to be mutually exclusive and exhaustive, a good habit to follow, one and only one model will be used by the oracle to make the final prediction. In other words, in this situation the oracle acts as nothing more than a simple gate, choosing one and only one model to make the prediction. We will see a different approach in the next example.

Finally, we walk this trading system forward one year, using one year of training data. By using just one walkforward year, we have only one testing fold, so presentation of results is more compact. In a real-life application, walkforward should probably cover several years of out-of-sample testing.

Here is the slightly edited output produced by this example. It is long, so we'll split it up and go through it one section at a time. After identifying the walkforward fold, the first step is to compute the optimal weights for the LIN_TREND linea regression transform. Naturally, this computation is based only on training data, not the out-of-sample data.

```
-------------------------------------------------------
Walkforward test date 2011 training 252 cases, testing 250
-------------------------------------------------------

Training LINEAR REGRESSION TRANSFORM LIN_TREND
LIN_TREND read 252 cases for training
Transform LIN_TREND regression coefficients:
          0.001045  LIN_ATR_5
          0.000744  LIN_ATR_15
          0.021420  CONSTANT
```

The model MOD_HI, which specializes in high volatility, is then trained. The log shows the number and percent of cases in the entire database that satisfy the prescreening condition, as well as the number and percent of training set cases. The coefficients for the model are printed, followed by the in-sample performance results.

```
LINREG Model MOD_HI predicting RETURN
PRESCREEN:
   HI_VOLATILITY > 0.50000
   278 of 504 (55.16 %) of database cases pass
   114 of 252 (45.24 %) of training set cases pass
prescreen
Regression coefficients:
          0.002862  QUA_ATR_5
         -0.000043  CUB_ATR_5
         -0.025968  CONSTANT

Target grand mean = -0.02645
Outer hi thresh = 0.04145  with 17 of 114 cases at or above
(14.91 %)  Mean = 0.35459 versus -0.09323
Outer lo thresh = -0.10434  with 15 of 114 cases at or
below (13.16 %)  Mean = -0.26158 versus 0.00917

MSE = 0.59360   R-squared = 0.00670   ROC area = 0.55687
Buy-and-hold profit factor=0.911 Sell-short-and-hold=1.097
Dual-thresholded outer PF = 3.428
   Outer long-only PF = 6.211   Improvement Ratio = 6.816
   Outer short-only PF = 2.334  Improvement Ratio = 2.126
```

The out-of-sample results then appear:

```
Out-of-sample results...

Target grand mean = 0.02924
Outer hi thresh = 0.04145  with 30 of 164 cases at or above
(18.29 %)  Mean = 0.19140 versus -0.00707
Outer lo thresh = -0.10434  with 20 of 164 cases at or
below (12.20 %)  Mean = 0.35124 versus -0.01549

MSE = 1.31615   R-squared = -0.00201   ROC area = 0.51479
Buy-and-hold profit factor = 1.075   Sell-short-and-hold =
0.930
Dual-thresholded outer PF = 0.944
   Outer long-only PF = 1.582   Improvement Ratio = 1.472
   Outer short-only PF = 0.468  Improvement Ratio = 0.503
```

All of this information is repeated for the model MOD_LO, which specializes in low-volatility regimes.

```
LINREG Model MOD_LO predicting RETURN
PRESCREEN:
   HI_VOLATILITY < 0.50000
   226 of 504 (44.84 %) of database cases pass
   138 of 252 (54.76 %) of training set cases pass
prescreen
Regression coefficients:
         -0.011925  QUA_ATR_5
         -0.011111  QUA_ATR_15
          0.072255  CONSTANT

Target grand mean = 0.06959
Outer hi thresh = 0.19859  with 38 of 138 cases at or above
(27.54 %)  Mean = 0.31771 versus -0.02470
Outer lo thresh = -0.23547  with 14 of 138 cases at or
below (10.14 %)  Mean = -0.52792 versus 0.13705

MSE = 0.38592   R-squared = 0.09210   ROC area = 0.70264
Buy-and-hold profit factor = 1.360   Sell-short-and-hold =
0.735
Dual-thresholded outer PF = 5.984
   Outer long-only PF = 5.271   Improvement Ratio = 3.877
   Outer short-only PF = 7.853  Improvement Ratio = 10.678
Out-of-sample results...

Target grand mean = -0.08242
Outer hi thresh = 0.19859  with 32 of 86 cases at or above
(37.21 %)  Mean = 0.17084 versus -0.23250
Outer lo thresh = -0.23547  with 11 of 86 cases at or below
(12.79 %)  Mean = -0.00337 versus -0.09402
MSE = 0.80462   R-squared = -0.08300   ROC area = 0.55492
Buy-and-hold profit factor = 0.776   Sell-short-and-hold =
1.289
Dual-thresholded outer PF = 1.570
   Outer long-only PF = 1.904   Improvement Ratio = 2.454
   Outer short-only PF = 1.010  Improvement Ratio = 0.784
```

Finally we reach the most interesting result, the performance of the oracle. As a confirmation of how this prediction system operates, look back at the results of training and testing MOD_HI. It had 114 cases in-sample and 164 cases out-of-sample. Model MOD_LOhad 138 cases in-sample and 86 cases out-of-sample Thus, the total number of in-sample cases is 114+138=252, and the total number of out-of-sample cases is 164+86=250. These number are reflected in the oracle results, as well as printed in the walkforward fold header that began this presentation of results. The training and testing cases have been segregated into each model according to the value of the P_VOLATILITY indicator, and then they have been merged by the oracle.

As a point of interest, if one judges overall performance by the dual-thresholded

profit factor, which indicates net return from long and short positions, the low-volatility model did much better (1.570) than the high-volatility model (0.944), and the oracle scored between them (1.226). It's not unusual for trend-based trading systems to perform better in low-volatility regimes than high-volatility.

```
ORACLE  ORAC_HONOR predicting RETURN
Sigma weights:
          0.995467  HI_VOLATILITY
Target grand mean = 0.02614
Outer hi thresh = 0.19859  with 38 of 252 cases at or above
(15.08 %)  Mean = 0.31771 versus -0.02563
Outer lo thresh = -0.13175  with 25 of 252 cases at or
below (9.92 %)  Mean = -0.31227 versus 0.06341

MSE = 0.47987   R-squared = 0.05052   ROC area = 0.65308
Buy-and-hold profit factor=1.109  Sell-short-and-hold=0.902
Dual-thresholded outer PF = 3.897
   Outer long-only PF = 5.271   Improvement Ratio = 4.755
   Outer short-only PF = 2.934  Improvement Ratio = 3.253


Out-of-sample results...

Target grand mean = -0.00917
Outer hi thresh = 0.19859  with 32 of 250 cases at or above
(12.80 %)  Mean = 0.17084 versus -0.03560
Outer lo thresh = -0.13175  with 26 of 250 cases at or
below (10.40 %)  Mean = 0.06952 versus -0.01831

MSE = 1.14019   R-squared = -0.01797   ROC area = 0.50128
Buy-and-hold profit factor=0.976  Sell-short-and-hold=1.025
Dual-thresholded outer PF = 1.226
   Outer long-only PF = 1.904   Improvement Ratio = 1.951
   Outer short-only PF = 0.821  Improvement Ratio = 0.802
```

Example 2: Unguided Specialization

The prior example covered the situation in which the developer had a preconceived notion of how models would best specialize. The PRESCREEN option create specialists based on volatility. But it may be that the developer does not want to force his or her ideas of specialization on the model, or at least not in so strong a manner. Also, the HONOR PRESCREEN option in the oracle forces an all-o nothing approach that may be a bit extreme. This example demonstrates a different approach. In this example, the developer still hypothesizes that volatility may have an impact on the nature of market prediction, and different models may be best under different volatility regimes. But rather than using the PRESCREEN option t split the training and testing data into volatility regimes, we can provide different predictor candidates to different models, train and test each model on the entire dataset, and then use an oracle to differentially weight the models' predictions to come up with a grand prediction. Thus, the oracle's prediction at each bar is a weighted sum of its constituent models' predictions with the weighting varying in accord with the volatility indicator This approach is usually inferior to that of the first example (prescreened models) in those situations in which the developer is confident that a particular predefined specialization is appropriate. However, in some applications it may be best to try multiple models and let the oracle decide which is best in which regimes. Also, by avoiding prescreening, one allows the oracle to consider the predictions of *all* models, giving them different but nevertheless positive weights. This lets all models contribute to some degree, which may be better than the all-or-nothing approach of prescreening.

Here is the script file for this example, omitting the up-front commands already discussed:

```
MODEL MOD_5 IS LINREG [
   INPUT = [ LIN_TREND LIN_ATR_5 QUA_ATR_5 CUB_ATR_5 ]
   OUTPUT = RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;

MODEL MOD_15 IS LINREG [
   INPUT = [ LIN_TREND LIN_ATR_15 QUA_ATR_15 CUB_ATR_15 ]
   OUTPUT = RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;
```

```
ORACLE ORAC_NO_HONOR [
   INPUT = [ MOD_5 MOD_15 ]
   GATE = [ P_VOLATILITY ]
   OUTPUT = RETURN
   MIN CRITERION FRACTION = 0.1
   ] ;

WALK FORWARD BY YEAR 1 2011 ;
```

The first model, MOD_5, uses the regression transform indicator LIN_TREND along with three trend indicators that look back just 5 days. The second model, MOD_15 is identical except that its indicators look back 15 days. No prescreening is done, so no specialization is enforced by the user other than any that might incidentally result from using different lookback distances. Both models are trained on all regimes.

The oracle ORAC_NO_HONOR pools the predictions of these two models, and uses P_VOLATILITY as a gate, so the current volatility will determine the relative importance of the models' predictions. But in this case, the user is not imposing the nature of this differential weighting by prescreening the component models. Rather, if the two models have different capabilities in different volatility regimes as a result of using different indicators, the oracle will automatically discover any such specialization and choose optimal weights when combining the models' predictions.

Here is the (slightly edited) output produced by this script file. As with the prior example, we will break it into easily digestible sections. First we see the walkforward fold header which identifies the out-of-sample year, the number of training cases, and the number of test cases. Then comes the linear regression transform. Note that the regression coefficients are the same as those in the first example, because once again the transform is being trained on all regimes, with no specialization.

```
-------------------------------------------------------
Walkforward test date 2011 training 252 cases, testing 250
-------------------------------------------------------

Transform LIN_TREND regression coefficients:
          0.001045  LIN_ATR_5
          0.000744  LIN_ATR_15
          0.021420  CONSTANT
```

The first model which will feed the oracle comes next. As expected, it is completely different from the first model in the prior example, because it has a different predictor candidate set, and it does not have a PRESCREEN option.

```
LINREG Model MOD_5 predicting RETURN
Regression coefficients:
          0.001365  LIN_ATR_5
          0.023358  CONSTANT

Target grand mean = 0.02614
Outer hi thresh = 0.05223  with 35 of 252 cases at or above
(13.89 %)  Mean = 0.06777 versus 0.01943
Outer lo thresh = -0.00837  with 35 of 252 cases at or
below (13.89 %)  Mean = -0.18889 versus 0.06083

MSE = 0.50471   R-squared = 0.00138   ROC area = 0.53503
Buy-and-hold profit factor=1.109 Sell-short-and-hold=0.902
Dual-thresholded outer PF = 1.621
   Outer long-only PF = 1.396   Improvement Ratio = 1.260
   Outer short-only PF = 1.780  Improvement Ratio = 1.973


Out-of-sample results...

Target grand mean = -0.00917
Outer hi thresh = 0.05223  with 53 of 250 cases at or above
(21.20 %)  Mean = 0.03737 versus -0.02170
Outer lo thresh = -0.00837  with 43 of 250 cases at or
below (17.20 %)  Mean = 0.07337 versus -0.02632

MSE = 1.12414   R-squared = -0.00364   ROC area = 0.47125
Buy-and-hold profit factor=0.976  Sell-short-and-hold=1.025
Dual-thresholded outer PF = 0.972
   Outer long-only PF = 1.127   Improvement Ratio = 1.154
   Outer short-only PF = 0.878  Improvement Ratio = 0.857
```

The second model, which features a longer lookback period than the first, now appears.

```
LINREG Model MOD_15 predicting RETURN
Regression coefficients:
          1.806378  LIN_TREND
         -0.003295  QUA_ATR_15
         -0.022379  CONSTANT
```

```
Target grand mean = 0.02614
Outer hi thresh = 0.09476  with 25 of 252 cases at or above
(9.92 %)  Mean = 0.12817 versus 0.01491
Outer lo thresh = -0.06618  with 26 of 252 cases at or
below (10.32 %)  Mean = -0.25695 versus 0.05871

MSE = 0.50122   R-squared = 0.00827   ROC area = 0.53903
Buy-and-hold profit factor=1.109  Sell-short-and-hold=0.902
Dual-thresholded outer PF = 1.997
   Outer long-only PF = 1.854   Improvement Ratio = 1.673
   Outer short-only PF = 2.085  Improvement Ratio = 2.311


Out-of-sample results...

Target grand mean = -0.00917
Outer hi thresh = 0.09476  with 32 of 250 cases at or above
(12.80 %)  Mean = 0.06346 versus -0.01984
Outer lo thresh = -0.06618  with 22 of 250 cases at or
below (8.80 %)  Mean = 0.02631 versus -0.01260

MSE = 1.12384   R-squared = -0.00338   ROC area = 0.50883
Buy-and-hold profit factor=0.976  Sell-short-and-hold=1.025
Dual-thresholded outer PF = 1.070
   Outer long-only PF = 1.186   Improvement Ratio = 1.215
   Outer short-only PF = 0.941  Improvement Ratio = 0.918
```

Lastly, we have the oracle results. Note that the performance of this oracle is considerably inferior to that of the prior example. This is typical in cases in which we are reasonably certain in advance that the predictable aspects of markets will behave differently in different regimes. Thus, we are usually better off using the PRESCREEN option in the component models and the HONOR PRESCREI option in the oracle that combines the models. However, we must not issue a blanket condemnation of the method shown in this example. It may be the case that we choose to deliberately use radically different families of predictors in the models in the hope that some families will be better predictors than others in different regimes. In this case it is reasonable, or even wise, to trust an oracle to discover and take advantage of any such unforeseen specialization.

```
---> Oracle results <---

ORACLE  ORAC_NO_HONOR predicting RETURN
Sigma weights:
          0.147796  P_VOLATILITY

Target grand mean = 0.02614
Outer hi thresh = 0.06539  with 26 of 252 cases at or above
(10.32 %)  Mean = 0.13192 versus 0.01397
Outer lo thresh = -0.01941  with 28 of 252 cases at or
below (11.11 %)  Mean = -0.19634 versus 0.05395

MSE = 0.50453   R-squared = 0.00173   ROC area = 0.52792
Buy-and-hold profit factor=1.109 Sell-short-and-hold=0.902
Dual-thresholded outer PF = 1.693
   Outer long-only PF = 2.374   Improvement Ratio = 2.141
   Outer short-only PF = 1.529  Improvement Ratio = 1.695


Out-of-sample results...

Target grand mean = -0.00917
Outer hi thresh = 0.06539  with 32 of 250 cases at or above
(12.80 %)  Mean = 0.05947 versus -0.01925
Outer lo thresh = -0.01941  with 31 of 250 cases at or
below (12.40 %)  Mean = 0.20535 versus -0.03954

MSE = 1.12295   R-squared = -0.00258   ROC area = 0.49834
Buy-and-hold profit factor=0.976  Sell-short-and-hold=1.025
Dual-thresholded outer PF = 0.843
   Outer long-only PF = 1.168   Improvement Ratio = 1.197
   Outer short-only PF = 0.629  Improvement Ratio = 0.614
```

Example 3: Triggering on High Volatility

Both of the prior examples processed all regimes during training and testing. They contained several predictive models, each of which specialized in a particular regime. Thus, the entire dataset underwent training and testing, with regime splitting across models happening inside the process. As a result, the performance report covered all regimes.

This third example is very different from the prior two in that triggering is employed so that only high-volatility regimes will be processed. In particular, the linear regression transform will be trained only during periods of high volatility. The model will be trained and tested only during periods of high volatility. Results reported in the AUDIT.LOG file will cover only high-volatility regimes. In other words, this entire development effort, including *all* aspects of training and testing, will be restricted to high volatility by means of triggering. Here is the script file that accomplishes this, omitting the up-front commands already discussed:

```
TRANSFORM HI_VOLATILITY IS EXPRESSION ( 0 ) [
   HI_VOLATILITY = P_VOLATILITY > 0.0
   ] ;

TRANSFORM LO_VOLATILITY IS EXPRESSION ( 0 ) [
   LO_VOLATILITY = P_VOLATILITY <= 0.0
   ] ;


MODEL MOD_ALL IS LINREG [
   INPUT = [ LIN_TREND QUA_ATR_5 QUA_ATR_15 CUB_ATR_5
               CUB_ATR_15 ]
   OUTPUT = RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;

TRIGGER ALL HI_VOLATILITY ;
WALK FORWARD BY YEAR 1 2011 ;
PRESERVE PREDICTIONS ;
TRADE SIMULATOR MOD_ALL GLOBAL DUAL
      (TRAINED 1.0 TRAINED 1.0) ATR 250 ;
WRITE EQUITY "HI_VOL.TXT" ;
```

The first thing done in this file is to define a pair of binary flags that separate the data into two regimes based on the P_VOLATILITY indicator. Actually, the LO_VOLATILITY flag is not used in this example, so this transform does not need to be present. But it will be used in the next example, and by showing both flags in both examples it is made clear that the split is mutually exclusive and exhaustive.

We have just one model in this example, and it includes many predictor candidates, including the linear regression transform LIN_TREND.

All activities are straightforward: We use the TRIGGER ALL command to retai only the high-volatility cases in the database. Then the model is walked forward as done in the prior two examples, and the model's predictions are preserved. This is required because we then invoke the TRADE SIMULATOR command, and writ the resulting equity curve file. There is no need to use the trade simulator in conjunction with a triggering approach to regime specialization. It is done here only because the next and final example will be identical to this example, except that it handles low-volatility regimes. By invoking the trade simulator and writing the equity file, we will then be able to use the PORTFOLIO command to provide ne performance results for both regimes.

We now examine the output produced by this script file, section by section. We will also frequently compare this output to that produced by the first example in order to see which figures remain the same and which change according to the approach (PRESCREEN versus TRIGGER).

The TRIGGER ALL HI_VOLATILITY command produces the following line output:

```
TRIGGER at 279 of 504 cases (55.36 percent)
```

This means that there were a total of 504 days of data in the database prior to the triggering. Of those, 279 of them have high volatility. If you look back at the first example in which MOD_HI was prescreened for high volatility, you will see tha 278 of 504 database cases passed the volatility test. Why is there a discrepancy (279 versus 278)? The reasons is subtle, and not important except for people who wish to understand the innermost workings of the program. There are several cases from 2009 in the dataset. When the prescreened model in the first example was processed, data prior to the first required year, 2010, was ignored for the sake of economy. But when a trigger operation is performed, the entire dataset, including 2009, is processed. In this example it picked up one additional case from 2009 that was not counted in the first example. If you do not understand this distinction, don't worry. It is of no real importance.

The results of training and testing this model in the high-volatility regime data are as follows:

```
-------------------------------------------------------------
Walkforward test date 2011 training 252 cases, testing 250
-------------------------------------------------------------

Transform LIN_TREND regression coefficients:
          0.003026  LIN_ATR_5
          0.000191  LIN_ATR_15
         -0.019709  CONSTANT

LINREG Model MOD_ALL predicting RETURN
Regression coefficients:
          0.002862  QUA_ATR_5
         -0.000043  CUB_ATR_5
         -0.025968  CONSTANT

Target grand mean = -0.02645
Outer hi thresh = 0.04145  with 17 of 114 cases at or above
(14.91 %)  Mean = 0.35459 versus -0.09323
Outer lo thresh = -0.10434  with 15 of 114 cases at or
below (13.16 %)  Mean = -0.26158 versus 0.00917

MSE = 0.59360   R-squared = 0.00670   ROC area = 0.55687
Buy-and-hold profit factor=0.911 Sell-short-and-hold=1.097
Dual-thresholded outer PF = 3.428
   Outer long-only PF = 6.211   Improvement Ratio = 6.816
   Outer short-only PF = 2.334  Improvement Ratio = 2.126


Out-of-sample results...

Target grand mean = 0.02924
Outer hi thresh = 0.04145  with 30 of 164 cases at or above
(18.29 %)  Mean = 0.19140 versus -0.00707
Outer lo thresh = -0.10434  with 20 of 164 cases at or
below (12.20 %)  Mean = 0.35124 versus -0.01549

MSE = 1.31615   R-squared = -0.00201   ROC area = 0.51479
Buy-and-hold profit factor=1.075  Sell-short-and-hold=0.930
Dual-thresholded outer PF = 0.944
   Outer long-only PF = 1.582   Improvement Ratio = 1.472
   Outer short-only PF = 0.468  Improvement Ratio = 0.503
```

There are some important things to notice about this output compared to that for the first example, shown here. Many of these items are crucial to a full understanding of the difference between prescreening and triggering as methods of regime specialization.

- The walkforward header, which shows the out-of-sample year as well as the number of training and test cases, does not reflect the triggering. This information will appear during training and testing, so showing it here would be redundant. Instead, it is more useful to keep the user informed of the

number of days being processed in the folds.

- The linear regression transform LIN_TREND is different here compared t that in the first example. This is because when model prescreening is used, the entire database is processed, and the only regime specialization that takes place is within models. In this triggering example, LIN_TREND is traine only on high-volatility cases, so the transform specializes.

- It happens by coincidence that LIN_TREND was not picked as a predictor fo the model in either example. Thus, we see that model MOD_HI in the firs example and model MOD_ALL in this example are identical. This make perfect sense, because they have the same 114 training cases. In the first example, those 114 cases were selected by the *model* from the complete set of training cases. In this third example, the *database was filtered* to keep only high-volatility cases, and the resulting training set had the same 114 cases as in the first example. So it should be no surprise that we get exactly the same model with the same performance.

- In the same manner, MOD_HI in the first example and model MOD_ALL i this example have identical out-of-sample results. This is because they are testing the same 164 cases. As with the training set discussed in the prior point, these 164 OOS cases were selected by the model's PRESCREEI option in MOD_HI and selected by the TRIGGER command in this thir example.

- If the regression transform LIN_TREND had been picked by MOD_HI MOD_ALL, we would not necessarily obtain the identical results that we saw here. This is because, as discussed in the first point above, LIN_TRENI is different in these two examples. In the first, PRESCREEN example LIN_TREND was computed from the entire dataset. In this third, TRIGGE example, LIN_TREND was computed from only the high-volatility cases. S they produce slightly different indicators.

- In the first and second examples, we were able to examine the oracle results to obtain complete performance figures for all 250 out-of-sample cases. But in this TRIGGER example we can see out-of-sample results for only the 16 high-volatility cases.

As a final note, the TRADE SIMULATOR command (described in the manual; tutorial chapter is pending.) was used to compute and save an equity curve for this example. The reason for doing so is to address the last point made above: when the TRIGGER option is used, we can see results for only the subset of cases that satisfy the regime specification, which here is high volatility. By computing and saving this equity curve, we can then do a separate TRIGGER run on low-volatility cases an

use the PORTFOLIO command (described in the manual; a tutorial chapter i pending.) to produce net results for the entire dataset. This is done in the next example. But for now we'll take a quick look at the TRADE SIMULATOR output:

```
-----> Trade simulator results <-----
Long and short, measuring change relative to ATR(250)

MODEL MOD_ALL

Results for this single market...
Bars=756  Long=30 (3.97%) Short=20 (2.65%)
Total return = -1.283 ATR units
Profit factor = 0.9444
Maximum drawdown = 6.312 on 20110914
   156 bars to bottom and never recovered
```

Naturally we have the same number of long (30) and short (20) trades as we saw in the model's out-of-sample results. We also have the same profit factor (0.944). But here we see that 756 bars were processed, while the model showed 164 cases. This is because the model saw only the 164 out-of-sample cases, while the trade simulator sees the entire market history. Nonetheless, trades are allowed to be executed only on days that are in the out-of-sample set and that pass the TRIGGEF test (high volatility here).

Example 4: Triggering on Low Volatility

This final example is almost identical to the prior example, with just two differences. First, the prior example handled high-volatility regimes, while this one handles low-volatility regimes. The two regimes are defined in these examples to be mutually exclusive and exhaustive. There is no requirement that this be done, but it is usually wise.

Second, the prior example stopped after computing and writing the equity curve for the high-volatility cases. This example will do the same for the low-volatility cases, but it will conclude with a PORTFOLIO command to compute net results fo the high and low-volatility trading systems. Here is the script file for this low-volatility triggering example:

```
TRANSFORM HI_VOLATILITY IS EXPRESSION ( 0 ) [
   HI_VOLATILITY = P_VOLATILITY > 0.0
   ] ;

TRANSFORM LO_VOLATILITY IS EXPRESSION ( 0 ) [
   LO_VOLATILITY = P_VOLATILITY <= 0.0
   ] ;


MODEL MOD_ALL IS LINREG [
   INPUT = [ LIN_TREND QUA_ATR_5 QUA_ATR_15 CUB_ATR_5
               CUB_ATR_15 ]
   OUTPUT = RETURN
   MAX STEPWISE = 2
   CRITERION = PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;

TRIGGER ALL LO_VOLATILITY ;
WALK FORWARD BY YEAR 1 2011 ;
PRESERVE PREDICTIONS ;
TRADE SIMULATOR MOD_ALL GLOBAL DUAL
      (TRAINED 1.0 TRAINED 1.0) ATR 250 ;
WRITE EQUITY "LO_VOL.TXT" ;

FILE PORTFOLIO NET_PERF [
   EQUITY FILE = [ "HI_VOL.TXT" "LO_VOL.TXT" ]
   ] ;
```

Here is the output produced by this script file. We will make some points about it on the following page.

```
------------------------------------------------------------
Walkforward test date 2011 training 252 cases, testing 250
------------------------------------------------------------

Transform LIN_TREND regression coefficients:
         -0.011369  LIN_ATR_5
          0.002348  LIN_ATR_15
          0.110501  CONSTANT

LINREG Model MOD_ALL predicting RETURN
Regression coefficients:
         -0.011925  QUA_ATR_5
         -0.011111  QUA_ATR_15
          0.072255  CONSTANT


Target grand mean = 0.06959
Outer hi thresh = 0.19859  with 38 of 138 cases at or above
(27.54 %)  Mean = 0.31771 versus -0.02470
Outer lo thresh = -0.23547  with 14 of 138 cases at or
below (10.14 %)  Mean = -0.52792 versus 0.13705

MSE = 0.38592   R-squared = 0.09210   ROC area = 0.70264
Buy-and-hold profit factor = 1.360   Sell-short-and-hold =
0.735
Dual-thresholded outer PF = 5.984
   Outer long-only PF = 5.271   Improvement Ratio = 3.877
   Outer short-only PF = 7.853  Improvement Ratio = 10.678


Out-of-sample results...

Target grand mean = -0.08242
Outer hi thresh = 0.19859  with 32 of 86 cases at or above
(37.21 %)  Mean = 0.17084 versus -0.23250
Outer lo thresh = -0.23547  with 11 of 86 cases at or below
(12.79 %)  Mean = -0.00337 versus -0.09402

MSE = 0.80462   R-squared = -0.08300   ROC area = 0.55492
Buy-and-hold profit factor=0.776  Sell-short-and-hold=1.289
Dual-thresholded outer PF = 1.570
   Outer long-only PF = 1.904   Improvement Ratio = 2.454
   Outer short-only PF = 1.010  Improvement Ratio = 0.784
```

```
-----> Trade simulator results <-----
Long and short, measuring change relative to ATR(250)

MODEL MOD_ALL

Results for this single market...
Bars=756  Long=32 (4.23%) Short=11 (1.46%)
Total return = 5.504 ATR units
Profit factor = 1.5696
Maximum drawdown = 4.157 on 20110816
   97 bars to bottom and 148 bars to recovery

COMMAND ---> WRITE EQUITY "LO_VOL.TXT" ;

------------------------------------------------------------
Processing Portfolio NET_PERF
------------------------------------------------------------

Basic profit statistics for 241 records
System  Min     Max     Mean   StdDev Rng/Std Sharpe    PF  Drawdown  Recovery
   1  -3.337   3.294  -0.0053  0.536  12.368  -0.158   0.944   6.312    Never
   2  -1.706   2.087   0.0228  0.324  11.694   1.118   1.570   4.157      148
 All  -3.337   3.294   0.0175  0.627  10.578   0.444   1.129   7.881      188
```

Most of the points to be made about this example are analogous to points made for the prior example, so we will breeze through those quickly.

- The coefficients for the linear regression transform LIN_TREND are differer from any of those seen before because in this example they are computed from only low-volatility cases. In the first two examples they were computed from all cases, and in the third example they were computed from only high-volatility cases.

- The low-volatility prescreened model MOD_LOW in the first example is identical to the model MOD_ALL here because they have the same training and test cases, and neither happened to choose LIN_TREND as a predictor. This is explained in more detail in the prior example. Since the models are identical, their in-sample and out-of-sample performances are also identical.

- The walkforward results shown here are for only low-volatility cases. In the first example we were able to see results for all cases because the oracle processed the entire dataset and employed individual specialist models as needed.

- The oracle in the first example obtained a profit factor of 1.226, while the net equity curve obtained from merging the equity curves of the high-volatility and low-volatility triggering had a profit factor of 1.129. This is because the oracle is treated as a single model with its own optimal trading thresholds, while the portfolio effectively takes every trade produced in each run.

This last point is interesting and important to understand. In the first example, MOD_HI made 30 long trades and 20 short trades in the OOS data. MOD_ALL in the third example did the same, as already discussed. Similarly, in the first example, MOD_LO made 32 long trades and 11 short trades. MOD_ALL in the fourth example did the same, as discussed above.

Here's the crucial part: in the first example, the oracle was treated as a model in and of itself, with its own long and short trading thresholds that were optimized in the training set. As a result, it was able to set more extreme thresholds and be pickier about the trades it took. In particular, even though its component models had 30+32=62 long trades (high plus low volatility), the oracle had only 32 long trades. On the short side, its component models had 20+11=31 trades, but the oracle, with its stricter trading threshold, had only 26 trades. This gives a total of 32+26=58 trades.

On the other hand, the PORTFOLIO command just computes the net performance of its components. Thus, it is taking all 30+32+20+11=93 trades. It must accept the optimal trading thresholds of the individual models in the separate regimes instead

of being able to find stricter optimal thresholds based on the entire dataset. This usually results in more trades and lower profit factor, as compared to using an oracle.

Permutation Training

An essential part of model-based trading system development is training a model (or set of models, committees, et cetera) to predict the near-future behavior of the market. Training involves finding a set of model parameters that maximizes a measure of performance within a historical dataset. Much of the time, the resulting performance will be excellent. However, this superb performance is at least partially due to the fact that the training process treated noise or other unique properties of the data as if they were legitimate patterns. Because such patterns are unlikely to repeat in the future, the performance obtained due to training is optimistic, often wildly so.

This undue optimism requires the responsible developer to answer two separate and equally important questions:

1) What is the probability that if the trading system (model(s) et cetera) were truly worthless, good luck could have produced results as good as those observed? We want this probability, called the *p-value*, to be small. If it turns out that there is an uncomfortably large probability that a truly worthless system could have given results as good as what we obtained just by being lucky, we should be skeptical of the trading system at hand.

2) What is the average performance that can be expected in the future (assuming that the market behavior does not change, which of course may be an unrealistic but unavoidable assumption)?

It is vital to understand that these are different, largely unrelated questions, and a responsible developer will require a satisfactory answer to *both* of them before signing off on real-life trading. It is possible, especially if extensive market history has been tested, for the answer to the first question to be a nicely small probability, even though the expected future performance is mediocre, hardly worth trading real money. It is also possible for the expected future performance to be enticing, but with a high probability of having been obtained by random good luck from a truly worthless system. Thus, we need both properties: it must be highly unlikely that a worthless system would have had sufficient good luck to do as well as we observed, and the expected future performance must be good enough to be worth putting down real money.

The standard method for answering the second question, and sometimes the first as well, is walkforward testing. The performance obtained in the pooled out-of-sample (OOS) period is an unbiased estimate of future performance. (Note that the term *unbiased* is used here in the loose sense of being without prejudice, as

opposed to the much stronger statistical sense.) If the OOS cases are independen (which they would not be if the targets look ahead more than one bar), an ordinary bootstrap test can answer the first question. Even if the cases have serial dependence, special bootstraps can do a fairly good job of answering the first question.

But the problem with the walkforward approach is that it discards a lot of data, often the majority of historical data. Only the pooled OOS data can be used tc answer the two questions. This is a high price to pay. There is an alternative approach to answering the first question, and which can also provide at least a decent hint of an answer to the second question, while using *all* available historical data.

Unfortunately, this alternative approach has a high price of its own: time. Training time is multiplied by a factor of at least 100, and as much as 1000 if a thorough job is to be done. This precludes some analyses. But it is a great addition to the developers toolbox.

This approach is invoked with the following command:

```
TRAIN PERMUTED Nreps ;
```

If this command appears, *no database can be read or appended*. This is because all indicators and targets must be able to be recomputed from the original and permuted market(s). This recomputation would obviously not be possible for precomputed indicators and targets in a database. Also, the REMOVE ZERC VOLUME command describedhere must appear before the READ MARKE' HISTORIES command.

TRAIN PERMUTED operates similarly to the ordinary TRAIN command, exce that in addition to the usual computation of indicators and targets from the data, it also recomputes indicators and targets, and retrains all models, *Nreps*-1 times. Each of these repetitions is done on market histories whose changes have been shuffled. Thus, the permuted markets have exactly the same bar-to-bar changes as the original, unpermuted markets, but in a different order. This random shuffling obviously destroys any predictable patterns in the markets, giving us a large number of examples of truly worthless trading systems; no matter how 'good' a trading system may be, it will be worthless when presented with random market data! Some of these shuffled market histories will allow the trained trading systems to be lucky, and others will not. Thus, by counting how often we obtain a trading system from shuffled data whose performance is at least as good as that obtained from the original, unshuffled history, we can directly answer the first question.

Another way of looking at this algorithm is by realizing that if the trading system

being developed is truly worthless, presenting it with real market data is no better than presenting it with random data. Its performance is no more likely to be outstanding than permuted performance. We would expect its performance to be somewhere in the middle of the heap, better than some and worse than others. On the other hand, suppose our trading system is truly able to detect authentic patterns in market data. Then we would expect its performance on the real data to exceed that of most or all of the systems developed on shuffled data.

In particular, suppose our trading system is truly worthless, and also suppose that we have *Nreps* trading results. One of these is from the original, unpermuted data, and the others are from shuffled data. Then there is a 1/*Nreps* probability that the original will be the best (none of the others exceeded it). There is a 2/*Nreps* probability that it will be at least second-best (at most one other exceeded it) and so forth. This can be visualized by imagining the performance measures laid out in a sorted line and knowing that if the system is worthless, it can land in any position with equal probability. In general, if *k* shuffled performance measures equal or exceed the unshuffled measure, the probability addressed by Question 1 above is (*k*+1)/*Nreps*.

This method of estimating probabilities of performance results can be seen in Figure 16 below. This is a histogram of the long-only profit factors obtained by a model called LIN_LONG when trained on 99 randomly shuffled market histories a well as the original unshuffled history. The profit factor from the original data is shown as a vertical red line. Note that this performance is one of the best, with a probability of 0.06 under the null hypothesis that the model is worthless.

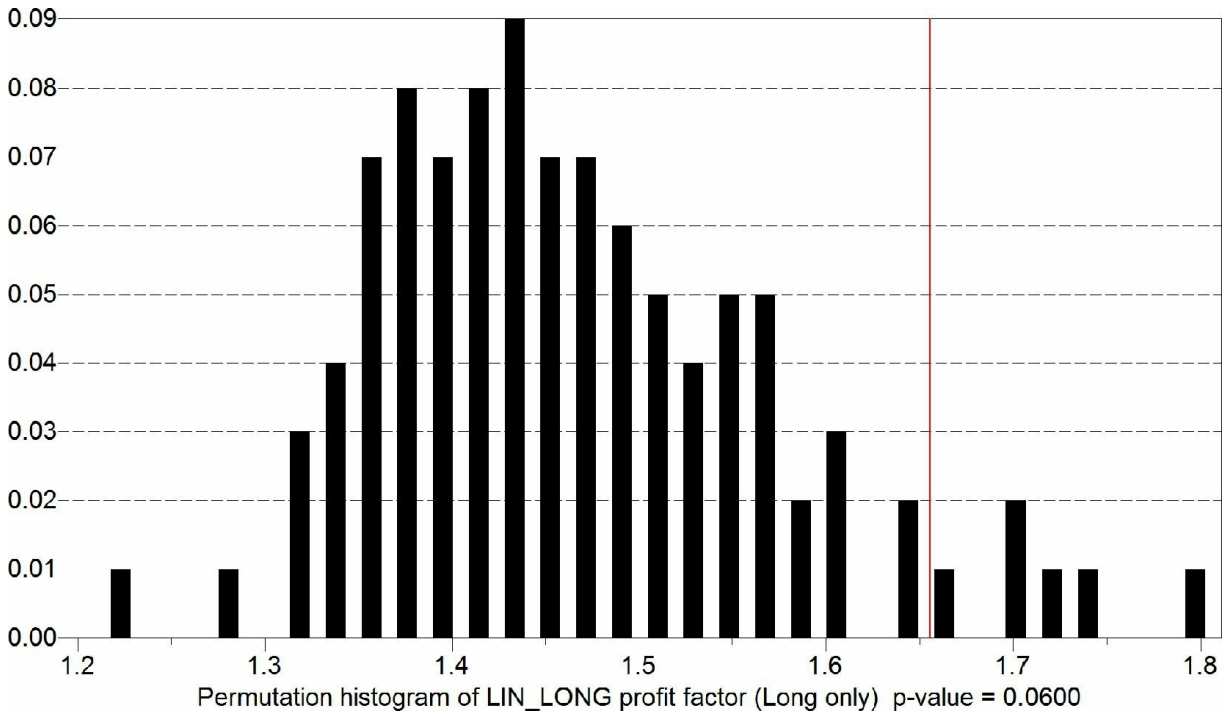

Figure 16: Histogram of TRAIN PERMUTED performance

The Components of Performance

The primary task of permutation training is estimating the probability that a worthless system could benefit from good luck sufficiently to attain the observed performance level. This provides an answer to Question 1 shown at the start of this chapter. However, Question 2 is equally important: What level of performance can we expect in the future? The gold standard for answering this question is walkforward testing. However, as a byproduct of probability estimation we can compute a rough but still useful estimate of one specific measure of future performance: Total return.

The fundamental basis of this estimation is the relationship among four loosely defined aspects of performance:

Total - The total return of the trading system after training

Skill - The true ability of the trading system to find and profit from authentic market patterns

Trend - If the market significantly trends up (or down) and the trading system is unbalanced in favor of long(or short) positions, this unbalance will generate profits on average even without intelligent individual trade decisions. If the market trend is large, and if the trading system development software has the ability to create unbalanced systems (the most common situation), the software will favor unbalanced systems that take advantage of the trend, even if these systems do not make many intelligent individual trades.

Bias - The process of training the system induces *training bias* by tuning its trade decisions to not only the authentic patterns represented in the dataset, but also to patterns that are unique to the historical dataset and may not be repeated in later use. If the system development software employs very powerful models, these models may go to great lengths to capture every pattern they see in the data, authentic and temporary, with the result that apparent performance will be greatly inflated.

Although Equation 9 below is far from a rigorous statement of how these four items are related, it is usually fairly close when these quantities are components of the total return of the trading system, and it is a useful mental tool for understanding the performance of a trading system after it has been trained:

$$Total = Skill + Trend + Bias \quad \textbf{(9)}$$

The market data is permuted in such a way that any underlying trend is almost

exactly preserved. As a result, the training algorithm will have the opportunity to generate unbalanced systems that take advantage of the trend, just as they do with the unpermuted market history. Therefore, we can expect that the *Trend* component of the total return will be about the same for the permuted markets as for the unpermuted market.

Also, the permuted markets will, on average, present to the training algorithm the same opportunity to capitalize on inauthentic patterns as did the unpermuted market. Thus, we can expect that the *Bias* component of the permuted returns will, on average, be about the same as that for the unpermuted return.

The only difference between the returns from the unpermuted and the permuted markets is the *Skill* component, which cannot exist in permuted data. In other words, Equation 10 below represents the average total return obtained from training on permuted markets.

$$TotalForPermuted = Trend + Bias \quad \textbf{(10)}$$

As a point of interest, look back at Figure 16 here. That figure is based on profit factors, while the equations just shown are more applicable to total return. Nonetheless, the principle is the same for either measure of performance. The bars in this histogram represent the values of Equation 10 above (plus the single observation for the unpermuted market, which is also used to compute the histogram). The red vertical bar is the value represented by Equation 9 above. We can thereby see the effect of *Skill* on performance relative to the performance of the trained trading systems when *Skill* is not involved. If the *Skill* component is significant, we would expect the vertical red bar to be on the far right side of the histogram, pushed there by the ability of the trading system to find authentic patterns in the market. Conversely, if the *Skill* component of performance is small, the red bar is more likely to be somewhere in the interior mass of the histogram.

Additional information can be gleaned from the permutation training by analyzing Equation 10 in more detail. When the trading system is trained by maximizing performance on a permuted dataset, the training algorithm will attempt to simultaneously maximize both components of Equation 10: it will try to produce an unequal number of long and short trades (assuming that the user allows unbalanced systems) in order to take advantage of any long-term trend in the market(s), and it will try to take individual trades in such a way as to capitalize on any patterns it finds in the data. Of course, since it is operating on randomly permuted data, any patterns it finds are inauthentic, which is why the *Skill* component plays no role.

Now suppose we have a system that makes only long trades, and it does so randomly; individual trades are made with no intelligence whatsoever. Equation 11 shows its expected return:

$$LongExpectedReturn = \frac{BarsLong}{TotalNumberOfBars} * TotalTargets \quad (11)$$

In this equation, *BarsLong* is the number of bars in which a long position is signaled, and *TotalNumberOfBars* is the number of bars in the entire dataset whose performance is being evaluated. This ratio is the fraction of the number of bars in which a long position is signaled. Looked at another way, this ratio is the probability that any given bar will signal a long position. *TotalTargets* is the sum of the target variable across all bars. This will be positive for markets with a long-term upward trend, negative for those with downward trend, and zero for a net flat market. So if our system is without intelligence, with signals randomly given, we will on average obtain this fraction (the fraction of time a long signal is given) of the total return possible.

A similar argument can be made for a short-only system, although the sign would be flipped because for short systems a positive target implies a loss, and a negative target implies a win. Combining these two situations into a single quantity gives Equation 12, the expected net return (the *Trend* performance component) from a random system that contains both long and short signals.

$$Trend = NetExpectedReturn = \frac{BarsLong - BarsShort}{TotalNumberOfBars} * TotalTargets \quad (12)$$

When we train a trading system using a permuted market, we can note how many long and short trades it signaled and then use Equation 12 to determine how much of its total return is due to position imbalance interacting with long-term trend. Anything above and beyond this quantity is training bias due to the system learning patterns that happen to exist in the randomly shuffled market. This training bias can be estimated with Equation 13.

$$Bias = TotalForPermuted - Trend \quad (13)$$

It would be risky to base a *Bias* estimate on a single training session. However, as long as we are repeating the permutation and training many times to compute the probability of the observed return happening by just good luck, we might as well average Equation 13 across all of those replications. If at least 100 or so replications are used, we should get a fairly decent estimate of the degree to which the training process is able to exploit inauthentic patterns and thereby produce inflated performance results.

Two thoughts are worth consideration at this point:

- *TSSB* allows various BALANCED criteria which force an equal number o long and short positions in multiple markets. In this case, *Trend* will be zero.

- Powerful models will generally produce a large *Bias*, while weak models (not necessarily a bad thing) will produce small *Bias*.

Now that the *Bias* can be estimated, we can go one or two steps further. Equation 9 can be rearranged as shown in Equation 14 below.

$$UnbiasedReturn = Skill + Trend = Total - Bias \quad \textbf{(14)}$$

This equation shows that if we subtract the *Bias* estimated with Equation 13 from the total return of the system that was trained on the original, unpermuted data, we are left with the *Skill* plus *Trend* components of the total return. Many developers will wish to stop here. Their thought is that if a market has a long-term trend, any effective trading system should take advantage of this trend by favoring long or short positions accordingly. However, another school of thought says that since deliberately unbalancing positions to take advantage of trend does not involve actual trade-by-trade intelligent picking, the trend component should be removed from the total return. This can be effected by applying Equation 12 to the original trading system and subtracting the *Trend* from the *UnbiasedReturn*, as shown in Equation 15 below.

$$Skill = BenchmarkedReturn = UnbiasedReturn - Trend \quad \textbf{(15)}$$

This difference, the *Skill* component of Equation 9, is often called the *Benchmarked return* because the *NetExpectedReturn* (*Trend* component) defined by Equation 12 can be thought of as a benchmark against which to judge actual trading performance.

These figures are all reported in the audit log file, with p-values for the profit factor and total return printed first. Here is a sample:

Net profit factor p = 0.0600 return p = 0.0400
Training bias = 52.3346 (67.1255 permuted return minus 14.7909 permuted benchmark)
Unbiased return = 55.4320 (107.7665 original return minus 52.3346 training bias = skill + trend)
Benchmarked return = 39.8372 (55.4320 unbiased return minus 15.5947 original benchmark = skill)

The *Training bias* line is Equation 13, with the term *benchmark* referring to *Trend* in the equation, and these figures averaged over all replications. This figure (52.3346 here) is the approximate degree to which the training process unjustly elevates the system's total return due to learning of inauthentic patterns.

The *Unbiased return* line is Equation 14, the actual return that we could expect in the future after accounting for the training bias. This figure includes both the true

skill of the trading system as well as the component due to unbalanced trades that take advantage of market trend.

The *Benchmarked return* line is Equation 15. This is the pure *Skill* component of the total return.

Permutation Training and Selection Bias

Up to this point we have discussed only training bias, which is usually dominated by models learning patterns of noise as if they were authentic patterns. There are other sources of training bias, such as incomplete representation of all possible patterns in the available history. However, these other sources are generally small and unavoidable. So what we have covered is sufficient to handle training bias in most practical applications.

However, there is another potential source of bias that inflates performance in the dataset above what can be expected in the future. This is *selection bias*, and it occurs when we choose one or more trained models from among a group of competitors. Bias occurs in the selection process because some of the models will have been lucky while others will have been unlucky. Those models that were lucky will be more likely to be selected than those that were unlucky, yet their luck, by definition, will not hold up in the future. Thus, the performance of the best models from among a set of competitors will likely deteriorate in the future as the luck that propelled them to the top vanishes.

If the user has employed any portfolios of type *IS* (in-sample) in the script file, the TRAIN PERMUTED command will properly handle them, and the p-values ar total return performance measures (unbiased and benchmarked) will be correct, or at least as correct as is possible under the loose assumptions of the technique. In other words, the selection bias inherent in the choice of an optimal subset will be accounted for in both the p-values and in the unbiased and benchmarked total returns.

There is one vital aspect of this to consider, though. Recall from the earlier discussion of the IS type of Portfolio that this is a potentially dangerous type. This is because if one or more of the competing models is extremely powerful, its in-sample performance will be unjustifiably excellent. As a result, it will likely be included in a Portfolio. This will, in turn, make this a seriously overfitted Portfolio. Of course, this will be detected in the permutation test, but that is little comfort if you are faced with a poorly performing Portfolio..

The proper way to handle this problem is to use the OOS type of Portfolio Unfortunately, the TRAIN PERMUTED operation cannot handle OOS Portfolio These can be evaluated only in a WALK FORWARD test because they need OO model results for selecting the Portfolio components. A future release of *TSSB* may include a permutation version of this test.

An example of poor IS portfolio creation due to an overfit model can be seen in the example shown below. In this example, five weak models are defined: LIN1

through LIN5. They are all identical, using the EXCLUSION GROUP option guarantee that their solo predictors are different. As stated earlier, this option is excellent for generating committee members, but probably not the best way to generate Portfolio candidates. Only LIN1 is shown here.

A strong model, LIN_POWER, is also defined. Unlike LIN1 through LIN5, whi allow only one predictor, this one allows four predictors, which is almost certainly excessive. Finally, a Portfolio is defined which optimally selects two of these six candidate Models. It is LONG ONLY, so the six models all select their predictors by maximizing the long profit factor.

```
MODEL LIN1 IS LINREG [
   INPUT = [ CMMA_5 CMMA_20 LIN_5 LIN_20 RSI_5 RSI_20
     STO_5 STO_20 REACT_5 REACT_20 PVR_5 PVR_20 ]
   OUTPUT = DAY_RETURN
   MAX STEPWISE = 1
   CRITERION = LONG PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   EXCLUSION GROUP = 1
   ] ;

MODEL LIN_POWER IS LINREG [
   INPUT = [ CMMA_5 CMMA_20 LIN_5 LIN_20 RSI_5 RSI_20
     STO_5 STO_20 REACT_5 REACT_20 PVR_5 PVR_20 ]
   OUTPUT = DAY_RETURN
   MAX STEPWISE = 4
   CRITERION = LONG PROFIT FACTOR
   MIN CRITERION FRACTION = 0.1
   ] ;

PORTFOLIO Port_Power [
   MODELS = [ LIN1 LIN2 LIN3 LIN4 LIN5 LIN_POWER ]
   TYPE = OPTIMIZE 2 IS
   NREPS = 1000
   LONG ONLY
   EQUALIZE PORTFOLIO STATS
   ] ;
```

The Portfolio optimizer selected LIN4 and LIN_POWER (of course) as the be components. Permutation training produced the distribution of profit factors shown on the here.

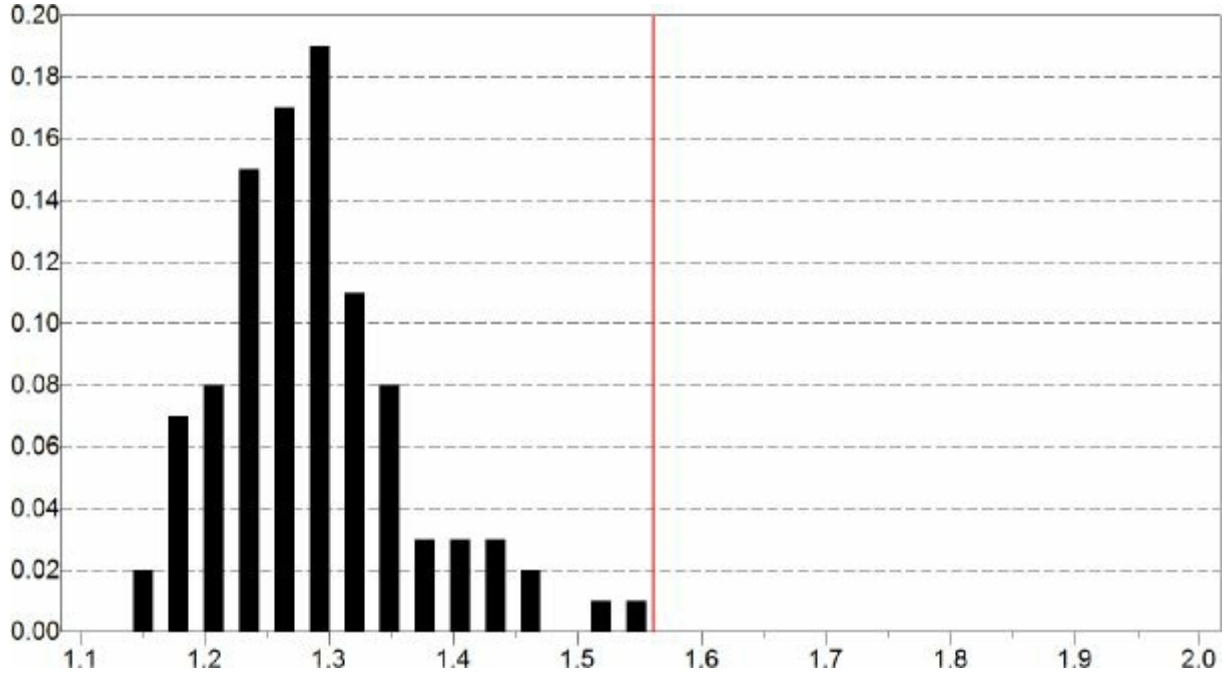

Figure 17: Permutation histogram of the weak model

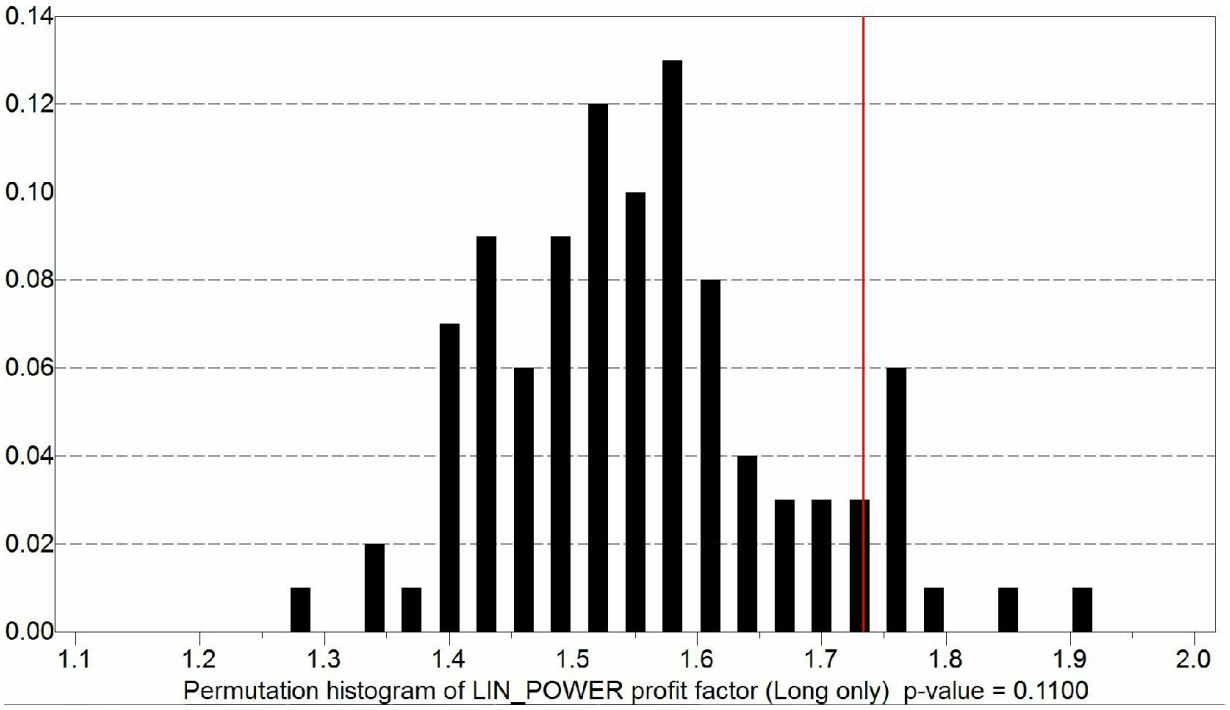

Figure 18: Permutation histogram of the overfitted model

Look at the histograms shown on the prior page. Observe that, as expected, LIN_POWER had a much higher profit factor than LIN4. This is because this is a in-sample computation, and LIN_POWER is a much more powerful model tha LIN4. But along with this greater profit factor for the original data, the profi factors of shuffled data were also greater for LIN_POWER than for LIN4 LIN_POWER's histogram is considerably shifted to the right of that of LIN4. I

fact, LIN_POWER's p-value of 0.11 is very inferior to that of LIN4, 0.01.

Finally, let's explore the performance partitioning for these models and the optimal Portfolio. These are shown on the here. Note first that the p-values for the return have the opposite ordering as those for the profit factor (0.1 versus 0.01). This is not unusual; total return and profit factor have surprisingly little relationship. It may be that one model has few but excellent trades, resulting in a high profit factor and low total return, while another model has many trades that are only moderately good. This model may have a profit factor barely exceeding one, yet have a high total return by virtue of its large number of trades. Experience shows that profit factor is a vastly superior measure of performance than total return.

The most salient figure is the training bias: the weak model LIN4 has a training bias of 37.4672, while the strong model LIN_POWER has a training bias of 58.8356 This great disparity should not be surprising. Interestingly enough, when the bias is subtracted from the performance of each model, the unbiased returns of 59.6410 for LIN4 and 57.8350 for LIN_POWER are almost equal. The benchmarked returns ar also almost equal. So we see that, at least in terms of total return, the weak and the strong model have about the same skill at detecting valid patterns in the market data. It's just that the more powerful model's returns are inflated by its greater ability to learn inauthentic patterns in the data.

Finally, let's look at the Portfolio results. Because its definition includes the EQUALIZE PORTFOLIO STATS option, all of these figures are based on t *average* of its two components, not their *sum*. We see that the Portfolio's training bias is nearly as great as that of LIN_POWER, probably reflecting the fact that thi bias figure includes selection bias as well as the training bias of the two component models. Without the selection bias (if this were a fixed Portfolio), the training bias would have equaled the mean of the training biases of the two component models, or (37.4672 + 58.8356) / 2 = 48.1514. However, this model training bias is inflated further by the selection bias inherent in creating the optimal portfolio. This raises the Portfolio's training bias considerably.

After subtracting the training bias from the raw total return of the Portfolio, we see that the unbiased return is a little less than that of the two component models. This is probably just random variation. The important point is that the Portfolio results are in line with the components because the algorithm has compensated not only for the models' training bias, but the Portfolio's selection bias as well.

MODEL LIN4

Long profit factor p = 0.0100 return p = 0.1000

Training bias = 37.4672 (57.6183 permuted return minus 20.1511 permuted benchmark)

Unbiased return = 59.6410 (97.1082 original return minus 37.4672

training bias = skill + trend)
Benchmarked return = 43.3659 (59.6410 unbiased return minus 16.2751 original benchmark = skill)

MODEL LIN_POWER

Long profit factor p = 0.1100 return p = 0.0100

Training bias = 58.8356 (73.1141 permuted return minus 14.2786 permuted benchmark)
Unbiased return = 57.8350 (116.6705 original return minus 58.8356 training bias = skill + trend)
Benchmarked return = 42.2402 (57.8350 unbiased return minus 15.5947 original benchmark = skill)

PORTFOLIO PORT_POWER

profit factor p = 0.0400 return p = 0.0500

Training bias = 53.7958 (72.4237 permuted return minus 18.6279 permuted benchmark)
Unbiased return = 53.0935 (106.8894 original return minus 53.7958 training bias = skill + trend)
Benchmarked return = 37.1586 (53.0935 unbiased return minus 15.9349 original benchmark = skill)

Multiple-Market Considerations

When the TRAIN PERMUTED command is executed in a multiple-market situation, operation becomes slightly more complicated. The issue is that inter-market correlation must br preserved. A fundamental assumption of permutation tests is that all permutations must be equally likely to have occurred in real life. For example, it is well known that in times of large moves, markets tend to move together. When some world crisis arises, most markets go down. When good economic news is broadcast, most markets rise. If the markets were permuted separately, this coherence would be lost and the permutation test would be invalidated.

This implies that every market must have data available for every date. Otherwise it would be impossible to swap dates with data in some markets and not in others. In order to accomplish this, the first step in permutation training is to pass through the market history and eliminate all dates whose market set is incomplete. This results in a message similar to the following being printed in the audit log file:

```
NOTE... Pruning multiple markets for simultaneous
permutation reduced the number of market-dates from
503809 to 414427.
     The model(s) about to be printed and all permutation
     results reflect this reduced dataset and will differ
     from prior and subsequent results.
```

This message informs the user as to how many market-dates had to be removed from the history files. It also reminds the user that the results about to be shown reflect the market histories *after* this reduction, and hence they will generally be different from results of an ordinary TRAIN operation performed before or after the TRAIN PERMUTED operation.

If any dates had to be eliminated, then after the TRAIN PERMUTED operation complete and all of its results are printed, the dates will be restored and as a convenience for the user the trading system will be trained with the complete dataset. A message similar to the following will be printed to make it clear to the user what is happening:

```
NOTE... Permutation is complete.  Prior to permutation we
     had to reduce  the number of market-dates so that
     every date had all markets.
     The original markets have now been restored.
     All models will now be re-trained with the original
     market data.  The results you are about to see
     reflect this original data, before pruning.
```

Transforms

There are three ways to make indicators and targets available for processing. here we saw how to compute them internally from the *TSSB* library. here we saw that externally generated variables could be read in from disk files. We now explore a third set of methods, *transforms*, which allow us to compute new variables from existing variables and raw market data. Several types of transform are available:

- *Expression transforms* apply common arithmetic operations as well as more sophisticated scalar and vector functions.
- *Linear Regression* transforms find a linear combination of indicators that has maximum correlation with a target variable. Thus, it is a dimensionality-reduction method that takes two or more indicators and turns them into a single, potentially stronger predictor of the target.
- *Quadratic regression* transforms extend linear regression transforms to second-order relationships.
- *Principal components* transforms generate the optionally rotated maximum-variance principal components of a set of indicators. Thus, it is a dimensionality-reduction method for reducing multiple indicators into a smaller set of indicators. However unlike, the linear and quadratic regression transform, it does this without reference to the target variable.
- *Nominal mapping* transforms convert nominal (non-numeric class identifier) indicators into numeric values using an optimal mapping function.
- *ARMA* transforms compute parameter values, shocks, and a variety of other quantities related to fitting an ARMA model to a series.
- *Purification* transforms regress functions of a 'purifier' series onto a series to be purified. The residual of this regression, as well as related values, can serve as valuable indicators.

Transforms should always appear in the script file before models, committees, or oracles, because these may reference as inputs or outputs the variables created by a transform.

The transform name will be the name used for the output variable(s) generated. If the transform produces more than one output variable, the integers 1, 2, 3, ..., will be appended to the transform name to define the names of the output variables.

Some transforms (such as Expression) are fully specified by the user; no training is required. However, some other transforms (such as Principal components) need to be trained on a dataset before they can be invoked. These are sometimes referred to as *trainable transforms*. Transforms are integrated into the training/testing cycle so that training will be done without snooping into an out-of-sample fold. The user, though, is responsible for setting the optional OVERLAP parameter if needed (see here). When a transform is invoked as part of a walkforward or cross validation procedure, it will first be trained on the current training fold (if such training is required), and then all values of the output variable(s) will be computed for all cases in the entire dataset, including both the current training fold and the upcoming test fold.

Expression Transforms

The most commonly used transform is the expression version. This transform allows the user to create new variables by means of simple equations, as well as more sophisticated mathematical functions. It even has several vector functions available.

The syntax for declaring an expression transform is similar to that for models, committees, and oracles. The keyword TRANSFORM appears, followed by name chosen by the user. The maximum name length is 15 characters, with no special characters other than the underscore (_) allowed. The name is followed by the key phrase IS EXPRESSION and the maximum vector length (discussed here) enclosed in parentheses. Finally, the specifications are enclosed in square brackets. This generic form is as follows:

```
TRANSFORM TransformName IS EXPRESSION (MaxLag) [ Specs ] ;
```

There is only one mandatory specification, and in many or most cases, this will be the only specification. This is:

```
TransformName  =  NumberExpression
```

This specification must appear exactly once, and it must be the last specification if more than one specification is used. The *NumberExpression* defines the arithmetic operations. The name of the computed variable is the name of the transform.

These variables are entered into the database as soon as they are defined, so it is legal to reference prior computed values in an expression transform. In other words, it is legal to have several transforms in a script file, and any transform may reference the values computed in a prior transform.

Here is a trivial example of a transform, occasionally useful:

```
TRANSFORM DUMDUM IS EXPRESSION (0) [ DUMDUM = 5 ] ;
```

This will generate a new database variable called DUMDUM which has the value 5 for every case. Interested users may wish to insert this transform into a script file, do a series plot, and observe that a horizontal line is produced, constant at a value of 5.

The only other specification available is used to define a temporary variable (not permanently saved in the database) that may then appear in subsequent specifications. This is:

```
TempVarName = NumberExpression
```

We can extend the trivial example shown above to demonstrate how a temporary variable might be used, although this particular example is pointless as shown. Later we will see how a logical expression could be inserted into an expression like this to make it far more useful, but for now we'll keep it simple. Note that as is the case with models, committees, and oracles, it is conventional though not required to place each specification on a separate line to increase readability for the user.

```
TRANSFORM DUMDUM IS EXPRESSION (0) [
      X = 5
      DUMDUM = X
      ] ;
```

In this example, we set the temporary variable X equal to 5, and then we set DUMDUM equal to X, which of course ultimately means that DUMDUM will have the value 5 for every case. The variable DUMDUM is entered into the database because it is the name of the expression. The variable X is not entered into the database because it is for temporary use only.

Quantities That May Be Referenced

The *NumberExpression* used to define output or temporary variables may reference any of the following quantities:

ExistingVarName - The name of a variable already in the database, or a previously defined expression transform, or a temporary variable name in the current expression

@OPEN : Market
@HIGH : Market
@LOW : Market
@CLOSE : Market - The opening/high/low/closing price of a bar. *Market* names a market, and it (along with the colon) may be omitted if there is only one market. Note that the market must have been read (READ MARKET HISTORIES, here) for this reference to make sense.

@OPEN : THIS
@HIGH : THIS
@LOW : THIS
@CLOSE : THIS- As above, except that in a multi-market situation, the market whose price is used for each record is that record's market as opposed to being fixed at the named market.

Consider the difference between explicitly naming a market, as in the first set of four references above, versus using the keyword THIS, as is done in the second set Suppose, for example, we have two markets in the database, BOL and IBM. If we reference @OPEN:IBM, then the opening price of IBM will be used for all records, both IBM and BOL. If, on the other hand, we reference @OPEN:THIS then the opening price of IBM will be used for IBM records, and the opening price of BOL will be used for BOL records. For tick data, the open, high, low, and close are equal. Thus, OPEN, HIGH, LOW, and CLOSE may be used interchangeably and will give identical results.

Many operations are available within an expression transform. Logical operations always return 1.0 if true and 0.0 if false. Values not equal to zero are treated as true, and zero is false. Common priorities of evaluation are observed, with priorities as shown below. Operations are in groups separated by dashes (------). The groups are listed from the last evaluated to the first evaluated. However, in case of doubt, use of parentheses is advised in order to be clear about the order of evaluation.

||	Logical: Are either of the quantities true?
&&	Logical: Are both of the quantities true?

==	Logical: Are the quantities equal?
!=	Logical: Are the quantities not equal?
<	Logical: Is the left quantity less than the right?
<=	Logical: Is the left quantity less than or equal to the right?
>	Logical: Is the left quantity greater than the right?
>=	Logical: Is the left quantity great than or equal to the right?

+	Sum of the two quantities
-	The left quantity minus the right quantity
---	
*	Product of the two quantities
/	The left quantity divided by the right quantity
%	Both quantities are truncated to integers; the remainder of the left divided by the right
**	The left quantity raised to the power of the right quantity

---

- Negative of the quantity
! Logical: reverses the truth of the quantity

ABS( ) Absolute value of the quantity
SQRT( ) Square root of the quantity
LOG10( ) Log base ten of the quantity
LOG( ) Natural log of the quantity
POWER( ) Ten raised to the power of the quantity
EXP( ) Exponentiation (base *e*) of the quantity
ATAN( ) Arc-tangent of the quantity
LOGISTIC( ) Logistic transform of the quantity
HTAN( ) Hyperbolic tangent of the quantity

So, to illustrate logical operations, suppose X has the value -0.2, Y has the value 56.2, and Z has the value 0.0. Then the following results would be obtained:

X || Z returns 1.0 (true) because X is true (nonzero).
X && Z returns 0.0 (false) because it's not the case that they are both true (nonzero).
X <= Y returns 1.0 (true) because X is less than Y.

Here are a few examples of EXPRESSION transforms:

Compute the difference between two existing predictors, X1 and X2:

```
TRANSFORM DIFF12 IS EXPRESSION (0) [ DIFF12 = X1-X2 ] ;
```

Compute the high minus the low for IBM:

```
TRANSFORM HIGHLOW IS EXPRESSION (0) [
     HIGHLOW = @HIGH:IBM - @LOW:IBM
     ] ;
```

Compute *BIZARRE* as X3 if X1>X2, or X4 otherwise:

```
TRANSFORM BIZARRE IS EXPRESSION (0) [
     X1BIGGER = X1 > x2
     X2BIGGER = ! X1BIGGER
     BIZARRE = X1BIGGER * X3 + X2BIGGER * X4
     ] ;
```

The example just shown might require some explanation. Recall that the result of a logical expression is the number 1.0 if the expression is true, and 0.0 if it is false.

Thus, X1BIGGER will be either 1.0 or 0.0 according to whether X1 exceeds X2 Then X2BIGGER will be the opposite of X1BIGGER, either 0.0 or 1.0. The fin result will be X3 multiplied by either 1.0 or 0.0, plus X4 multiplied by the opposite, 0.0 or 1.0. In other words, this expression transform will return either X3 or X4, depending on the relationship between X1 and X2.

Here is a pair of expressions that illustrate several things simultaneously. It's doubtful that this example would have any practical utility, but it is a great little tutorial. The first expression sets *Exp5* equal to the open of IBM. The second expression subtracts the open of each record's market from *Exp5*. The resulting quantity *Exp6* is zero for IBM and more or less random for all other markets. For IBM records, the open is subtracted from itself, leaving zero. But for other markets that market's open is subtracted from that of IBM, resulting in nonsense. This example demonstrates that an expression transform (*Exp6* here) can reference the result of a prior (but not subsequent!) transform, *Exp5* here. It also clarifies the difference between specifying a market and using THIS for a market quantity.

```
TRANSFORM Exp5 IS EXPRESSION (0) [
      Exp5 = @OPEN:IBM
      ] ;

TRANSFORM Exp6 IS EXPRESSION [
      Exp6 = Exp5 - @OPEN:THIS
      ] ;
```

Why would you choose to specify a market versus using THIS? Suppose you are in a multiple-market situation and you want to use a measure of volatility in some formula that will generate an indicator or perhaps a gate for an oracle. If, for each market, you want to use the volatility of this market when computing your new variable, you would specify THIS. But maybe you believe that it is the volatility of an index such as OEX which is most important because it covers all markets. In this case you would specify OEX.

Vector Operations in Expression Transforms

Database variables and market variables may be referenced with a lag to provide a past value, and they may also be used to generate vectors (strings of values). To lag a variable, append the lag in parentheses. To generate a vector, append the lag and the vector length in parentheses. For example:

SLOPEVAR (5) is the variable *SLOPEVAR* as of five bars ago in this market.

SLOPEVAR (0 , 20) is a vector of the 20 most recent values of *SLOPEVAR* in

this market.

If a lag and/or vector length results in references to values prior to the first value in the database or market, this first value is duplicated as needed to play the role of nonexistent prior values.

When the EXPRESSION transform is declared, the maximum historical lookbac (taking into account lags and vector lengths) must be specified in parentheses. This is an annoyance to the user, but it enables valuable optimization that saves both memory and execution time. Larger values than required are legal but will cause excessive memory use and slower operation.

For example, suppose an expression transform contains a reference to @CLOSE:IBM(5,10). This reference is to a vector of 10 historic values of th close of IBM, beginning with the value 5 bars ago. In other words, this will look a values 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 bars ago. Thus, the declaration must specify maximum lag as at least 14. For example:

```
TRANSFORM VECDEMO IS EXPRESSION (14) [
```

To find the value you should use for the maximum lag, you must look at every variable specification that contains a lag and/or a vector. For each, figure out the maximum possible value that the sum of the lag and the vector length can ever be. If the reference uses only a lag, not a vector length, consider this to be a vector of length one. Subtract one from this, and that will give the maximum lag to specify. In the example just shown, the lag was fixed at 5 and the vector length was fixed at 10, meaning that we specify a maximum lag of 10+5-1=14 (or larger).

When a monadic operator (one number in, one number out) such as SQRT() is applied to a vector, the result is a vector of the same length in which the operator is applied to each component individually.

When a dyadic operator such as + is applied to a pair of vectors, these vectors should be the same length. Results are undefined if the lengths are different. The result is a vector of this same length in which the operator is applied to corresponding elements pairwise.

When a dyadic operator is applied to a scalar and a vector, the result is a vector of the same length in which the scalar is paired with each element of the vector individually. For example, one can add a constant to each element in a vector.

Vector-to-Scalar Functions

The following operations take a vector as an argument and return a scalar (a single number):

@MIN (vec) - The minimum of all values in the vector
@MAX (vec) - The maximum of all values in the vector
@RANGE (vec) - The range (max minus min) of the values in the vector
@MEAN (vec) - The mean of the values in the vector
@STD (vec) - The standard deviation of the values in the vector
@MEDIAN (vec) - The median of the vector
@IQRANGE (vec) - The interquartile range of the vector
@SIGN_AGE (vec)- The number of elements in the vector for which the sign remains the same as that of the first (historically most recent) element. Counting starts at zero. The value zero is treated as negative.
@SLOPE (vec) - The slope (change per element) of a least-squares line fit to the vector
@LIN_PROJ (vec)- The predicted value of the current point based on a linear fit

An Example with the @SIGN_AGE Function

Here is an example of using the *@SIGN_AGE* function in a way that seems reasonable but is not what is intended. This is followed by a demonstration of how to do it correctly.

Suppose you want to count the number of bars that the variable X has recently been on the same side of its 50-bar moving average. When the current value crosses its moving average, this variable is to have the value zero. If it then stays on this side of its moving average for the next bar, the value increments to one. If it stays there yet again, the value is two. This is to continue up to a maximum count of 20, where it remains if the current value continues on the same side of the moving average. (This sort of truncation is vital to preventing rare extreme values of the indicator.)

On first thought, the following transform may seem reasonable. Note that we look back at 50 historical values, including the current bar. Thus, the maximum lag is 49.

```
TRANSFORM AGE_A IS EXPRESSION ( 49 ) [
   AGE_A = @SIGN_AGE ( X(0,20) - @MEAN(X(0,50)) )
   ] ;
```

Look first at the term **@MEAN(X(0,50))**. The **@MEAN** function takes a vector as its argument. In this case, the vector argument is **X(0,50)**, which is the most recent 50 values of X. The **@MEAN** function returns the scalar mean of the vector, which

in this case provides the 50-bar moving average.

This scalar quantity is subtracted from each element of **X(0,20)**, the vector of the 20 most recent values of X. Finally, the **@SIGN_AGE** function counts back from the most recent element of this vector, seeing how far it can get before the sign changes (the value crosses the moving average). The argument to this function is the value of X minus its 50-bar mean. The value returned by this function is the count of how many historical differences have the same sign as the most recent difference.

What's wrong with this transform? It's perfectly legal, and it returns a result that may look reasonable. However, it does not do exactly what you want. This transform computes its count by examining the value of each of the differences between the most recent 20 values of X and the 50-bar moving average based on a window ending *on the current bar*. Your goal, as stated at the beginning of this example, is to examine the difference between a historical case and the 50-bar moving average *behind (prior to) the case being examined* as opposed to the moving average *behind the current bar*.

In order to do this the way you wish, which is also the most sensible way, you need to perform two transforms in sequence. The first computes the difference between each case and its trailing 50-bar moving average. The second counts the sign-change age. This is performed as shown below:

```
TRANSFORM PRICE_DIFF IS EXPRESSION ( 49 ) [
   PRICE_DIFF = X - @MEAN(X(0,50))
   ] ;

TRANSFORM AGE_B IS EXPRESSION ( 19 ) [
   AGE_B = @SIGN_AGE ( PRICE_DIFF(0,20) )
   ] ;
```

Consider the **PRICE_DIFF** transform. For each case, it computes the trailing 50-bar moving average and subtracts this from the current value of X. Note that if you do not want the current case to be included in the moving average, you would use **X(1,50)** instead of **X(0,50)**. In this situation, you must bump up the maximum lag from 49 to 50.

The **AGE_B** transform then examines the series of differences and counts the age of signs.

Logical Evaluation in Expression Transforms

It is possible to evaluate the truth of a logical expression and choose the subsequently executed statements accordingly. This is done with the following expressions:

IF (NumberExp) {
Commands executed if NumberExp is nonzero
}

ELSE IF (NumberExp) {
Commands executed if NumberExp is nonzero and all prior if expressions are zero
}

ELSE {
Commands executed if all prior if expressions are zero
}

In some programming languages such as C++, the curly braces {} are optional if the logical statement encloses just one command. But *TSSB* requires curly braces {} for all logical operations, even if the operation encloses just one command. This requirement reduces the chances of careless mistakes when logical operations are nested.

Remember that the final output statement must be the *last* statement in the transform, so it cannot occur inside a logical expression.

An Example with Logical Expressions

Here is an example of the use of logical expressions. If the long-term trend LIN_ATR_15 and the short-term trend LIN_ATR_5 are both positive, the computed value is to be +2. If the longer trend is positive but the short-term is not, the returned value is +1. Similar rules apply for negative trends. Finally, if the longer trend happens to be zero, the returned value is zero.

```
TRANSFORM DoubleTrend IS EXPRESSION ( 0 ) [
   IF (LIN_ATR_15 > 0) {
      IF (LIN_ATR_5 > 0) {
         RETVAL = 2
         }
      ELSE {
         RETVAL = 1
         }
      }
   ELSE IF (LIN_ATR_15 < 0) {
      IF (LIN_ATR_5 < 0) {
         RETVAL = -2
         }
      ELSE {
         RETVAL = -1
         }
      }
   ELSE {
      RETVAL = 0
      }
   DoubleTrend = RETVAL
   ] ;
```

A More Complex Example

We end this discussion of the expression transform with a somewhat more complex example. Suppose we want a measure of the value of a short-term moving average relative to a long-term moving average, with data that is day bars. Moreover, we want to scale this according to recent daily range. If the high minus the low has been small recently, we want to exaggerate the moving-average difference. Here is an expression transform that we might use:

```
TRANSFORM ScaledMA IS EXPRESSION ( 49 ) [
   HL_DIFF = @MEAN( @HIGH(1,10) - @LOW(1,10) )
   MA_DIFF = @MEAN( @CLOSE(0,10) ) - @MEAN( @CLOSE(0,50) )
   ScaledMA = MA_DIFF / (HL_DIFF + 0.01 * @CLOSE)
   ] ;
```

The term **@HIGH(1,10)** produces a vector of 10 recent highs, lagged by 1 day so that the current day is not included. (There is no mathematical need for this lag. It's here only to demonstrate lagging.) Similarly, **@LOW(1,10)** produces recent lows. When these two quantities are subtracted, we get the vector of 10 daily high-low differences. The **@MEAN** function finds the mean of these 10 numbers. Thus, **HL_DIFF** is the mean of the recent daily high-low differences, a crude measure of volatility.

The term **@CLOSE(0,10)** produces the vector of the 10 most recent closes, and **@MEAN** computes the mean of this vector. We do a similar thing looking back 50 days. Thus, **MA_DIFF** is the 10-day moving average minus the 50-day moving average.

The last line computes the value of this transform by dividing **MA_DIFF** by **HL_DIFF**, resulting in a scaled moving-average difference. We add **0.01 *@CLOSE** to the denominator in order to prevent division by zero or even by a quantity close to zero, which would produce extreme values. By ensuring that the denominator is at least one percent of the closing price, we limit the magnitude of this new indicator.