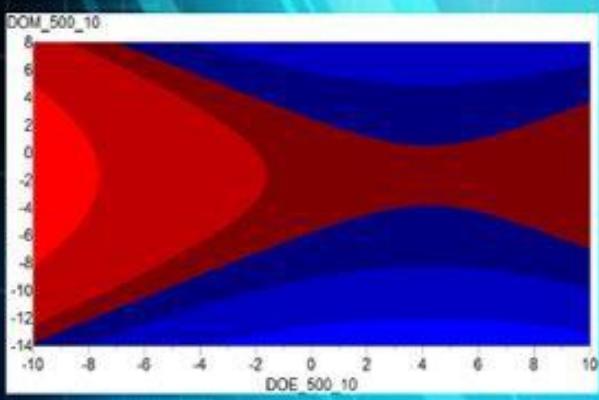
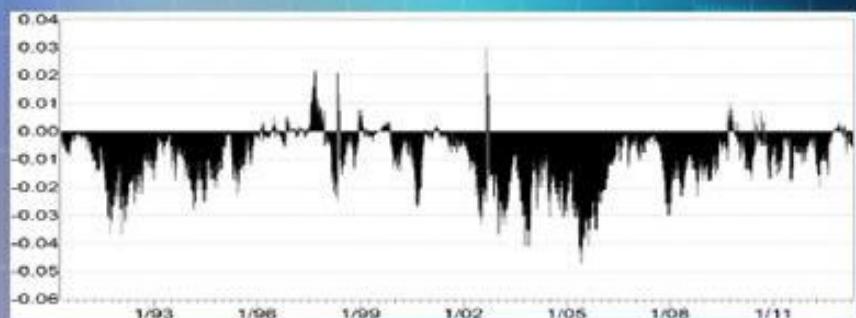
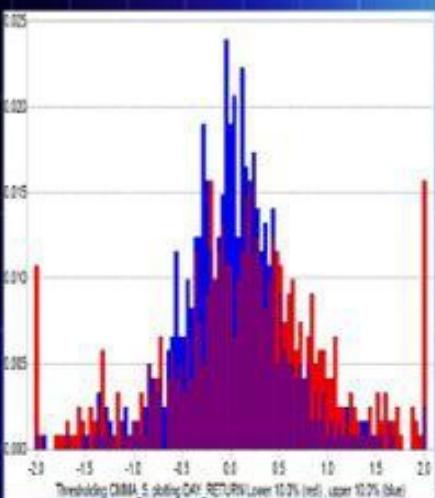


# Statistically Sound Machine Learning for Algorithmic Trading of Financial Instruments

Developing Predictive-Model-Based Trading Systems Using TSSB



David Aronson  
Timothy Masters

Statistically Sound Machine Learning  
for  
Algorithmic Trading of Financial Instruments

Developing Predictive-Model-Based Trading Systems  
Using *TSSB*

David Aronson

with

Timothy Masters, Ph.D.  
Technical Advisor

Edition 1.20

Great effort has been undertaken to ensure that the content of this book, as well as the *TSSB* program, are correct. However, errors and omissions are impossible to avoid in a work of this extent. Neither this book nor the *TSSB* program are meant as providing professional advice. No guarantee is made that these items are free of errors and omissions, and the reader assumes full liability for any losses associated with use of these items.

### **About this book:**

This is a tutorial, with instruction organized from easy to sophisticated. If the reader just skims through the entire text, hoping to gain an idea of how to use the *TSSB* program, the reader will be hopelessly dismayed by the vast complexity of options. The correct approach is to begin with the first, very simple example and implement it. Then progress to the next, and so forth. Each example builds on experience gained from prior examples. In this way, the reader will painlessly become familiar with the program.

The *TSSB* program may be downloaded (free) from the *TSSB* website:  
<http://tssbsoftware.com/>

Copyright © 2013 David Aronson and TimothyMasters  
All rights reserved

ISBN 978-1489507716

## About David Aronson

David Aronson is a pioneer in machine learning and nonlinear trading system development and signal boosting/filtering. He has worked in this field since 1979. His accomplishments include:

- Started Raden Research Group in 1982 where he oversaw the development of PRISM (Pattern Recognition Information Synthesis Modeling).
- Chartered Market Technician certified by The Market Technicians Association since 1992.
- Proprietary equities trader for Spear, Leeds and Kellogg 1997 – 2002.
- Was an Adjunct Professor of Finance, teaching a graduate level course in technical analysis, data mining and predictive analytics to MBA and financial engineering students from 2002 to 2011.
- Author of “Evidence Based Technical Analysis” published by John Wiley & Sons 2006. This was the first popular book to deal with data mining bias and the Monte Carlo Permutation Method for generating bias-free p-values.
- Co-designer of TSSB (Trading System Synthesis and Boosting), a software platform for the automated development of statistically sound predictive-model-based trading systems.
- Developed a method for indicator purification and Pure VIX.
- Innovated the concept of signal boosting: using machine learning to enhance the performance of existing strategies.

## About Timothy Masters

Timothy Masters received a PhD in mathematical statistics with a specialization in numerical computing. Since then he has continuously worked as a consultant for government and industry.

- Early research involved automated feature detection in high-altitude photographs while developing applications for flood and drought prediction, detection of hidden missile silos, and identification of threatening military vehicles.
- Worked with medical researchers in the development of computer

algorithms for distinguishing between benign and malignant cells in needle biopsies.

- Current focus is methods for evaluating automated financial market trading systems.
- Authored five books on prediction, classification, and applications of neural networks:
  - Practical Neural Network Recipes in C++ (Academic Press, 1993)
  - Signal and Image Processing with Neural Networks (Wiley, 1994)
  - Advanced Algorithms for Neural Networks (Wiley, 1995)
  - Neural, Novel, and Hybrid Algorithms for Time Series Prediction (Wiley, 1995)
  - Assessing and Improving Prediction and Classification (CreateSpace, 2013)

More information can be found on the author's website: [TimothyMasters.info](http://TimothyMasters.info)

# Table of Contents

## Introduction

- Two Approaches to Automated Trading
- Predictive Modeling
  - Indicators and Targets
  - Converting Predictions to Trade Decisions
- Testing the Trading System
  - Walkforward Testing
  - Cross Validation
  - Overlap Considerations
  - Performance Criteria
  - Model Performance Versus Financial Performance
  - Financial Relevance and Generalizability
  - Performance Statistics in *TSSB*
- Desirable Program Features

## A Simple Standalone Trading System

- The Script File
- The Audit Log
  - A Walkforward Fold
  - Out-of-Sample Results for This Fold
  - The Walkforward Summary

## A Simple Filter System

- The Trade File
- The Script File
- The Audit Log
  - Out-of-Sample Results for This Fold
  - The Walkforward Summary

## Common Initial Commands

- Market Price Histories and Variables
- Quick Reference to Initial Commands
- Detailed Descriptions
  - INTRADAY BY MINUTE
  - INTRADAY BY SECOND
  - MARKET DATE FORMAT YYMMDD

MARKET DATE FORMAT M\_D\_YYYY  
MARKET DATE FORMAT AUTOMATIC  
REMOVE ZERO VOLUME  
READ MARKET LIST  
READ MARKET HISTORIES  
MARKET SCAN  
RETAIN YEARS  
RETAIN MOD  
CLEAN RAW DATA  
INDEX  
READ VARIABLE LIST  
OUTLIER SCAN  
DESCRIBE  
CROSS MARKET AD  
CROSS MARKET KL  
CROSS MARKET IQ  
STATIONARITY

### A Final Example

## Reading and Writing Databases

Quick Reference to Database Commands  
Detailed Descriptions

RETAIN MARKET LIST  
VARIABLE IS TEXT  
WRITE DATABASE  
READ DATABASE  
READ UNORDERED DATABASE  
APPEND DATABASE  
IS PROFIT

### A Saving/Restoring Example

## Creating Variables

Overview and Basic Syntax  
Index Markets and Derived Variables  
An Example of IS INDEX and MINUS INDEX  
Multiple Indices  
Historical Adjustment to Improve Stationarity  
Centering  
Scaling  
Normalization

## An Example of Centering, Scaling, and Normalization

### Cross-Market Normalization

#### Pooled Variables

MEDIAN pooling

CLUMP60 Pooling

#### Mahalanobis Distance

#### Absorption Ratio

#### Trend Indicators

MA DIFFERENCE ShortLength LongLength Lag

LINEAR PER ATR HistLength ATRlength

QUADRATIC PER ATR HistLength ATRlength

CUBIC PER ATR HistLength ATRlength

RSI HistLength

STOCHASTIC K HistLength

STOCHASTIC D HistLength

PRICE MOMENTUM HistLength StdDevLength

ADX HistLength

MIN ADX HistLength MinLength

RESIDUAL MIN ADX HistLength MinLength

MAX ADX HistLength MaxLength

RESIDUAL MAX ADX HistLength MaxLength

DELTA ADX HistLength DeltaLength

ACCEL ADX HistLength DeltaLength

INTRADAY INTENSITY HistLength

DELTA INTRADAY INTENSITY HistLength DeltaLength

REACTIVITY HistLength

DELTA REACTIVITY HistLength DeltaDist

MIN REACTIVITY HistLength Dist

MAX REACTIVITY HistLength Dist

#### Trend-Like Indicators

CLOSE TO CLOSE

N DAY HIGH HistLength

N DAY LOW HistLength

#### Deviations from Trend

CLOSE MINUS MOVING AVERAGE HistLen ATRlen

LINEAR DEVIATION HistLength

QUADRATIC DEVIATION HistLength

CUBIC DEVIATION HistLength

DETRENDED RSI DetrendedLength DetrenderLength

Lookback

#### Volatility Indicators

ABS PRICE CHANGE OSCILLATOR ShortLen Multiplier  
PRICE VARIANCE RATIO HistLength Multiplier  
MIN PRICE VARIANCE RATIO HistLen Mult Mlength  
CHANGE VARIANCE RATIO HistLength Multiplier  
MIN CHANGE VARIANCE RATIO HistLen Mult Mlen  
ATR RATIO HistLength Multiplier  
DELTA PRICE VARIANCE RATIO HistLength Multiplier  
DELTA CHANGE VARIANCE RATIO HistLength  
Multiplier  
DELTA ATR RATIO HistLength Multiplier  
BOLLINGER WIDTH HistLength  
DELTA BOLLINGER WIDTH HistLength DeltaLength  
N DAY NARROWER HistLength  
N DAY WIDER HistLength

#### Indicators Involving Indices

INDEX CORRELATION HistLength  
DELTA INDEX CORRELATION HistLength DeltaLength  
DEVIATION FROM INDEX FIT HistLength  
MovAvgLength  
PURIFIED INDEX Norm HistLen Npred Nfam Nlooks  
Look1

#### Basic Price Distribution Statistics

PRICE SKEWNESS HistLength Multiplier  
CHANGE SKEWNESS HistLength Multiplier  
PRICE KURTOSIS HistLength Multiplier  
CHANGE KURTOSIS HistLength Multiplier  
DELTA PRICE SKEWNESS HistLen Multiplier DeltaLen  
DELTA CHANGE SKEWNESS HistLen Multiplier  
DeltaLen  
DELTA PRICE KURTOSIS HistLen Multiplier DeltaLen  
DELTA CHANGE KURTOSIS HistLen Multiplier  
DeltaLen

#### Indicators That Significantly Involve Volume

VOLUME MOMENTUM HistLength Multiplier  
DELTA VOLUME MOMENTUM HistLen Multiplier  
DeltaLen  
VOLUME WEIGHTED MA OVER MA HistLength  
DIFF VOLUME WEIGHTED MA OVER MA ShortDist  
LongDist  
PRICE VOLUME FIT HistLength  
DIFF PRICE VOLUME FIT ShortDist LongDist

DELTA PRICE VOLUME FIT HistLength DeltaDist  
ON BALANCE VOLUME HistLength  
DELTA ON BALANCE VOLUME HistLength DeltaDist  
POSITIVE VOLUME INDICATOR HistLength  
DELTA POSITIVE VOLUME INDICATOR HistLen  
DeltaDist  
NEGATIVE VOLUME INDICATOR HistLength  
DELTA NEGATIVE VOLUME INDICATOR HistLen  
DeltaDist  
PRODUCT PRICE VOLUME HistLength  
SUM PRICE VOLUME HistLength  
DELTA PRODUCT PRICE VOLUME HistLen DeltaDist  
DELTA SUM PRICE VOLUME HistLen DeltaDist

### Entropy and Mutual Information Indicators

PRICE ENTROPY WordLength  
VOLUME ENTROPY WordLength  
PRICE MUTUAL INFORMATION WordLength  
VOLUME MUTUAL INFORMATION WordLength

### Indicators Based on Wavelets

REAL MORLET Period  
REAL DIFF MORLET Period  
REAL PRODUCT MORLET Period  
IMAG MORLET Period  
IMAG DIFF MORLET Period  
IMAG PRODUCT MORLET Period  
PHASE MORLET Period  
DAUB MEAN HistLength Level  
DAUB MIN HistLength Level  
DAUB MAX HistLength Level  
DAUB STD HistLength Level  
DAUB ENERGY HistLength Level  
DAUB NL ENERGY HistLength Level  
DAUB CURVE HistLength Level

### Follow-Through-Index (FTI) Indicators

Low-Pass Filtering and FTI Computation  
Block Size and Channels  
Essential Parameters for FTI calculation  
Computing FTI  
Automated Choice of Filter Period  
Trends Within Trends  
FTI Indicators Available in *TSSB*

FTI LOWPASS BlockSize HalfLength Period  
FTI MINOR LOWPASS BlockSize HalfLength LowPeriod  
HighPeriod  
FTI MAJOR LOWPASS BlockSize HalfLength LowPeriod  
HighPeriod  
FTI FTI BlockSize HalfLength Period  
FTI LARGEST FTI BlockSize HalfLength LowPeriod  
HighPeriod  
FTI MINOR FTI BlockSize HalfLength LowPeriod  
HighPeriod  
FTI MAJOR FTI BlockSize HalfLength LowPeriod  
HighPeriod  
FTI LARGEST PERIOD BlockSize HalfLength LowPeriod  
HighPeriod  
FTI MINOR PERIOD BlockSize HalfLength LowPeriod  
HighPeriod  
FTI MAJOR PERIOD BlockSize HalfLength LowPeriod  
HighPeriod  
FTI CRAT BlockSize HalfLength LowPeriod HighPeriod  
FTI MINOR BEST CRAT BlockSize HalfLength  
LowPeriod HighPeriod  
FTI MAJOR BEST CRAT BlockSize HalfLength  
LowPeriod HighPeriod  
FTI BOTH BEST CRAT BlockSize HalfLength LowPeriod  
HighPeriod

### Target Variables

NEXT DAY LOG RATIO  
NEXT DAY ATR RETURN Distance  
SUBSEQUENT DAY ATR RETURN Lead Distance  
NEXT MONTH ATR RETURN Distance  
HIT OR MISS Up Down Cutoff ATRdist  
**FUTURE SLOPE** Ahead ATRdist  
RSQ FUTURE SLOPE Ahead ATRdist

### Screening Variables

#### Chi-Square Tests

Options for the Chi-Square Test  
Output of the Chi-Square Test  
Running Chi-Square Tests from the Menu

#### Nonredundant Predictor Screening

[Options for Nonredundant Predictor Screening](#)  
[Running Nonredundant Predictor Screening from the Menu](#)  
[Examples of Nonredundant Predictor Screening](#)

**Models 1: Fundamentals**

- [Overview and Basic Syntax](#)
- [Mandatory Specifications Common to All Models](#)
  - [The INPUT list](#)
  - [The OUTPUT Specifier](#)
  - [Number of Inputs Chosen by Stepwise Selection](#)
  - [The Criterion to be Optimized in Indicator Selection](#)
  - [A Lower Limit on the Number or Fraction of Trades](#)
  - [Summary of Mandatory Specifications for All Models](#)
- [Optional Specifications Common to All Models](#)
  - [Mitigating Outliers](#)
  - [Testing Multiple Stepwise Indicator Sets](#)
  - [Stepwise Indicator Selection With Cross Validation](#)
  - [When the Target Does Not Measure Profit](#)
  - [Multiple-Market Trades Based on Ranked Predictions](#)
  - [Restricting Models to Long or Short Trades](#)
  - [Prescreening For Specialist Models](#)
  - [Building a Committee with Exclusion Groups](#)
  - [Building a Committee with Resampling and Subsampling](#)
  - [Avoiding Overlap Bias](#)
  - [A Popularity Contest for Indicators](#)
  - [Bootstrap Statistical Significance Tests for Performance](#)
  - [Monte-Carlo Permutation Tests](#)
- [An Example Using Most Model Specifications](#)
- [Sequential Prediction](#)

**Models 2: The Models**

- [Linear Regression](#)
  - [The MODEL CRITERION Specification for LINREG Models](#)
  - [The Identity Model](#)
- [Quadratic Regression](#)
- [The General Regression Neural Network](#)
- [The Multiple-Layer Feedforward Network](#)
  - [The Number of Neurons in the First Hidden Layer](#)
  - [The Number of Neurons in the Second Hidden Layer](#)

- Functional Form of the Output Neuron
- The Domain of the Neurons
- A Basic MLFN Suitable for Most Applications
- A Complex-Domain MLFN
- The Basic Tree Model
- A Forest of Trees
- Boosted Trees
- Operation String Models
  - Use of Constants in Operation Strings
- Split Linear Models for Regime Regression
  - An Ordinary SPLIT LINEAR Model
  - The NOISE Version of the SPLIT LINEAR Model

## Committees

- Model Specifications Used by Committees
- The AVERAGE Committee
- The LINREG (Linear Regression) Committee
- Constrained Linear Regression Committee
- Models as Committees
- Creating Component Models for Committees
  - Exclusion Groups
  - Explicit Specification of Different Indicators
  - Using Different Selection Criteria
  - Varying the Training Set by Subsampling
  - Varying the Training Set by Resampling

## Oracles

- Model Specifications Used by Oracles
- Traditional Operation of the Oracle
- Prescreen Operation of the Oracle
  - The HONOR PRESCREEN Option
  - The PRESCREEN ONLY Option
  - An Example of Prescreen Operation
- More Complex Oracles

## Testing Methods

- Performance for the Entire Dataset
- Walkforward Testing
- Cross Validation by Time Period
- Cross Validation using a Control Variable

Cross Validation by Random Blocks  
Preserving Predictions for Trade Simulation  
Market States as Trade Triggers  
An Example of Simple Triggering  
Triggering Based on State Change  
Triggering Versus Prescreening  
Commands Common to All Four Examples  
Example 1: Model Specialization via PRESCREEN  
Example 2: Unguided Specialization  
Example 3: Triggering on High Volatility  
Example 4: Triggering on Low Volatility

## Permutation Training

The Components of Performance  
Permutation Training and Selection Bias  
Multiple-Market Considerations

## Transforms

Expression Transforms  
Quantities That May Be Referenced  
Vector Operations in Expression Transforms  
Vector-to-Scalar Functions  
An Example with the @SIGN\_AGE Function  
Logical Evaluation in Expression Transforms  
An Example with Logical Expressions  
A More Complex Example  
Principal Component Transforms  
Invoking the Principal Components Transform  
Tables Printed  
An Example  
Linear and Quadratic Regression Transforms  
A Regression Transform Example  
The Nominal Mapping Transform  
Inputs and the Target  
Gates  
Focusing on Extreme Targets  
Declaring the Transform and its Options  
A Nominal Mapping Example  
The ARMA Transform  
The PURIFY Transform

Defining the Purified and Purifier Series  
Specifying the Predictor Functions  
Miscellaneous Specifications  
Usage Considerations  
A Simple Example

## Complex Prediction Systems

### Stacking Models and Committees

## Graphics

Series Plot  
Series + Market  
Histogram  
Thresholded Histogram  
Density Map  
Bivariate and Trivariate Plots  
Trivariate Plots  
Equity  
Prediction Map  
Indicator-Target Relationship  
Isolating Predictability of Direction Versus Magnitude

## Finding Independent Predictors

### A FIND GROUPS Demonstration

## Market Regression Classes

### REGRESSION CLASS Demonstrations

The Hierarchical Method  
The Sequential Method  
The Leung Method

## Developing a Stand-Alone System

### Choosing Predictor Candidates and the Target

Choosing the Target  
Quality Does Not Equal Quantity for Predictors

### Predictor and Target Selection for this Study

Stationarity  
The Problem of Outliers  
Cross-Market Compatibility

Data Snooping: Friend or Foe?  
Checking Stability with Subsampling  
How Long Does the Model Hold Up?  
Finding Models for a Committee  
The Trading System  
The Final Test

Trade Simulation and Portfolios  
Writing Equity Curves  
Performance Measures  
Portfolios (File-Based Version)  
A Portfolio Example

Integrated Portfolios  
A FIXED Portfolio Example  
An OOS Portfolio Example

# Introduction

Many people who trade financial instruments would like to automate some or all of their trading systems. Automation has several advantages over seat-of-the-pants trading:

- Intelligently designed automated trading systems can and often do outperform human-driven systems. An effective data-mining program can discover subtle patterns in market behavior that most humans would not have a chance of seeing.
- An automated system is absolutely repeatable, while a human-driven system is subject to human whims. Consistency of decision-making is a vital property of a system that can consistently show a profit. Repeatability is also valuable because it allows examination of trades in order to study operation and perhaps improve performance.
- Most properly designed automated trading systems are amenable to rigorous statistical analysis that can assess performance measures such as expected future performance and the probability that the system could have come into existence due to good luck rather than true power.
- Unattended operation is possible.

Automated trading systems are usually used for one or both of two applications. *TSSB* is a state-of-the-art program that is able to generate trading systems that perform both applications:

- *TSSB* produces a complete, stand-alone trading system which makes all trading decisions.
- *TSSB* produces a model which may be used to filter the trades of an existing trading system in order to improve performance. It is often the case that by intelligently selecting a subset of the trades ordered by an existing system, and rejecting the other trades, we can improve the risk/reward ratio. *TSSB* can also suggest position sizes according to the likelihood of the trade's success.

## Two Approaches to Automated Trading

Whether the user's goal is development of a stand-alone trading system or a system to filter signals from an existing trading system, there are two common

approaches to its development and implementation: *rules-based* (IF/THEN rules proposed by a human) and *predictive modeling*.

A rules-based trading system requires that the user specify the exact rules that make trade decisions, although one or more parameters associated with these rules may be optimized by the development software. Here is a simple example of an algorithm-based trading system:

**IF** the short-term moving average of prices exceeds the long-term moving average of prices, **THEN** hold a long position during the next bar.

The above algorithm explicitly states the rule that decides positions bar-by-bar, although the exact definition of ‘short-term’ and ‘long-term’ is left open. The developer might use software to find moving-average lookback distances that maximize some measure of performance. Programs such as TradeStation® include a proprietary language (EasyLanguage® in this case) by which the developer can specify trading rules.

With the widespread availability of high-speed desktop computers, an alternative approach to trading system development has become popular. *Predictive modeling* employs mathematically sophisticated software to examine indicators derived from historical data such as price, volume, and open interest, with the goal of discovering repeatable patterns that have predictive power. A predictive model relates these patterns to a forward-looking variable called a *target* or dependent variable. This is the approach used by TSSB, and it has several advantages over algorithm-based system development:

- Intelligent modeling software can discover patterns that are so complex or buried under random noise that no human could ever see them.
- Once a predictive model system is developed, it is usually easy to tweak its operation to adjust the risk/reward ratio to suit applications ranging across a wide spectrum. It can obtain a desired trade off between numerous signals with a lower probability of success and fewer signals with a higher probability of success. This is accomplished by adjusting a threshold that converts model predictions into discrete buy and sell signals.
- Well designed software allows the developer to adjust the degree of automation employed in the discovery of trading systems. Experienced developers can maintain great control over the process and put their knowledge to work creating systems having certain desired properties, while inexperienced developers can take advantage of massive automation, letting the software have majority control.

- In general, predictive modeling is more amenable to advanced statistical analysis than algorithm-based system development. Sophisticated statistical analysis algorithms can be incorporated into the model-generating process more easily than they can be incorporated into systems based on human-specified rules.

# Predictive Modeling

The predictive modeling approach to trading system development relies on a basic property of market price movement: all markets contain patterns that tend to repeat throughout history, and hence can often be used to predict future activity. For example, under some conditions a trend can be expected to continue until the move is exhausted. Under other conditions, a sudden violent move will, more often than not, be followed by a retracement toward the recent mean price. A predictive model studies historical market data and attempts to discover the patterns that repeat often enough to be profitable. Once such patterns are discovered, the model will be on the lookout for their reoccurrence. Based on historical observations, the model will then be able to predict whether the market will soon rise, fall, or remain about the same. These predictions can be translated into buy/sell decisions by applying thresholds to the model's predictions. We expand on notion this below.

## Indicators and Targets

Predictive models do not normally work with raw market data. Rather, the market prices and other series, such as volume, are usually transformed into two classes of variables called *indicators* and *targets*. This is the data used by the model during its training, testing, and ultimate realtime use. It is in the definition of these variables that the developer exerts his or her own influence on the trading system.

Indicators are variables that look strictly backwards in time. When trading in real time, as of any given bar an indicator will be computable, assuming that we are in possession of sufficient historical price data to satisfy the definition of the indicator. For example, someone may define an indicator called *trend* as the percent change of market price from the close of a bar five bars ago to the close of this bar. As long as we know these two prices, we can compute this *trend* indicator. The numerous indicators that *TSSB* can compute will be discussed in detail [here](#).

*Targets* are variables that look strictly forward in time. (In classical regression modeling, the target is often referred to as the *dependent variable*.) Targets reveal the future behavior of the market. We can compute targets for historical data as long as we have a sufficient number of future bars to satisfy the definition of the target. Obviously, though, when we are actually trading the system we cannot know the targets unless we have a phenomenal crystal ball. For example, we may define an indicator called *day\_return* as the percent

market change from the open of the next day to the open of the day after the next. If we have a historical record of prices, we can compute this target for every bar except the last two in the dataset. Targets that *TSSB* can compute are discussed [here](#).

The fundamental idea behind predictive modeling is that *indicators* may contain information that can be used to predict *targets*. The task of a predictive model is to find and exploit any such information. Consider the following hypothetical values of two indicators that we choose to call *trend* and *volatility*, along with a target variable that we will call *day\_return*:

Date	trend	volatility	day_return
19950214	0.251	1.572	0.144
19950215	0.101	1.778	0.055
19950216	-0.167	2.004	-0.013
...			

Suppose we provide several years of this data to a model and ask it to learn how to predict *day\_return* from *trend* and *volatility*. (This process is called model training.) Then, we may at a later date calculate from recent prices that *trend*=0.225 and *volatility*=1.244 as of that day. The trained model may then make a prediction that *day\_return* will be 0.152. (These are all made-up numbers.) Based on this prediction that the market is about to rise substantially, we may choose to take a long position.

## Converting Predictions to Trade Decisions

Intuition tells us that we should put more faith in extreme predictions than in more common predictions near the center of the model's prediction range. If a model predicts that the market will rise by 0.001 percent tomorrow, we would not be nearly as inclined to take a long position as if the model predicts a 5.8 percent rise. This intuition is correct, because in general there is a large correspondence between the magnitude of a prediction and the likelihood of success of the associated trade. Predictions of large magnitude are more likely to signal profitable market moves than predictions of small magnitude.

The standard method for making trade decisions based on predicted market moves is to compare the prediction to a fixed threshold. If the prediction is greater than or equal to an upper threshold (usually positive), take a long position. If the prediction is less than or equal to a lower threshold (usually negative), take a short position. The holding period for a position is implicit in the definition of the target. This will be discussed in detail [here](#).

It should be obvious that the threshold determines a tradeoff in the number of trades versus the accuracy rate of the trades. If we set a threshold near zero, the magnitude of the predictions will frequently exceed the threshold, and a position will be taken often. Conversely, if we set a threshold that is far from zero, predicted market moves will only rarely lie beyond the threshold, so trades will be rare. We already noted that there is a large correspondence between the magnitude of a prediction and the likelihood of a trade's success. Thus, by choosing an appropriate threshold, we can control whether we have a system that trades often but with only mediocre accuracy, or a system that trades rarely but with excellent accuracy.

*TSSB* automatically chooses optimal long and short thresholds by choosing them so as to maximize the profit factor for long systems and short systems separately. (See [here](#) for the definition of profit factor.) In order to prevent degenerate situations in which there is only one trade or very few trades, the user specifies a minimum number of trades that must be taken, either as an absolute number or as a minimum fraction of bars. In addition, *TSSB* has an option for using two thresholds on each side (long and short) so as to produce two sets of signals, one set for ‘normal reliability’ trades, and a more conservative set for ‘high reliability’ trades. Finally, in many applications, *TSSB* prints tables that show performance figures that would be obtained with varying thresholds.

Computation of thresholds and interpretation of trade results based on predictions relative to these thresholds are advanced topics that will be discussed in detail [here](#). For now, the user needs to understand only the following concepts:

- The user specifies *indicator* variables based on recent observed history and *target* variables that portray future price movement.
- *TSSB* is given raw historical market data (prices and perhaps other data, such as volume) and it generates an extensive database of indicator and target variables. One or more models are trained to predict the target given a set of indicators. In other words, the model learns to use the predictive information contained in the indicators in order to predict the future as exemplified by the target.
- Every time a prediction is made, the numerical value of this prediction is compared to a *long* or *upper* threshold. If the prediction is greater than or equal to the long threshold, a long position is taken. Similarly, the prediction is compared to a *short* or *lower* threshold, which will nearly always be less than the *long* threshold. If the prediction is less than or equal to the short threshold, a short position is taken.

- The holding period for a position is inherent in the target variable. This will be discussed in detail [here](#).
- *TSSB* will report results for long and short systems separately, as well as net results for the combined systems.

## Testing the Trading System

*TSSB* provides the ability to perform many tests of a predictive model trading or filtering system. The available testing methodologies will be discussed in detail [here](#). However, so that the user may understand the elementary trading/filtering system development and evaluation presented in the [next chapter](#), we now discuss two general testing methodologies: cross validation and walkforward testing. These are the primary standards in many prediction applications, and both are available in *TSSB* in a variety of forms.

The principle underlying the vast majority of testing methodologies, including those included in *TSSB*, is that the complete historical dataset available to the developer is split into separate subsets. One subset, called the *training set* or the *development set*, is used to train the predictive model. The other subset, called the *test set* or the *validation set*, is used to evaluate performance of the trained model. (Note that the distinction between the terms *test set* and *validation set* is not consistent among experts, so the increasingly common convention is to use them interchangeably. The same is true of *training set* and *development set*.)

The key here is that no data that takes part in the training of the model is permitted to take part in its performance evaluation. Under fairly general conditions, this mutually exclusive separation guarantees that the performance measured in the test set is an *unbiased* estimate of future performance. In other words, although the observed performance will almost certainly not exactly equal the performance that will be seen in the future, it does not have a systematic bias toward optimistic or pessimistic values. Having an unbiased estimate of future performance is one of the two main goals of a trading system development and testing operation. The other goal is being able to perform a statistical significance test to estimate the probability that the performance level achieved could have been due to good luck. This advanced concept will be discussed on Pages [here](#) and [here](#).

In the earliest days of model building and testing, when high speed computers were not readily available, splitting of the data into a training set and a test set was done exactly once. The developer would typically train the model using data through a date several years prior to the current date, and then test the

model on subsequent data, ending with the most recent data available. This is an extremely inefficient use of the data. Modern development platforms should make available *cross validation*, *walkforward*, or both. These techniques split the available data into training sets and test sets many times, and pool the performance statistics into a single unbiased estimate of the model-based trading system's true capability. This extensive reuse of the data for both training and testing makes efficient use of precious and limited market history.

## Walkforward Testing

Walkforward testing is straightforward, intuitive, and widely used. The principle is that we train the model on a relatively long block of data that ends a considerable time in the past. We test the trained model on a relatively short section of data that immediately follows the training block. Then we shift the training and testing blocks forward in time by an amount equal to the length of the test block and repeat the prior steps. Walkforward testing ends when we reach the end of the dataset. We compute the net performance figure by pooling all of the test block trades. Here is a simple example of walkforward testing:

- 1) Train the model using data from 1990 through 2007. Test the model on 2008 data.
- 2) Train the model using data from 1991 through 2008. Test the model on 2009 data.
- 3) Train the model using data from 1992 through 2009. Test the model on 2010 data.

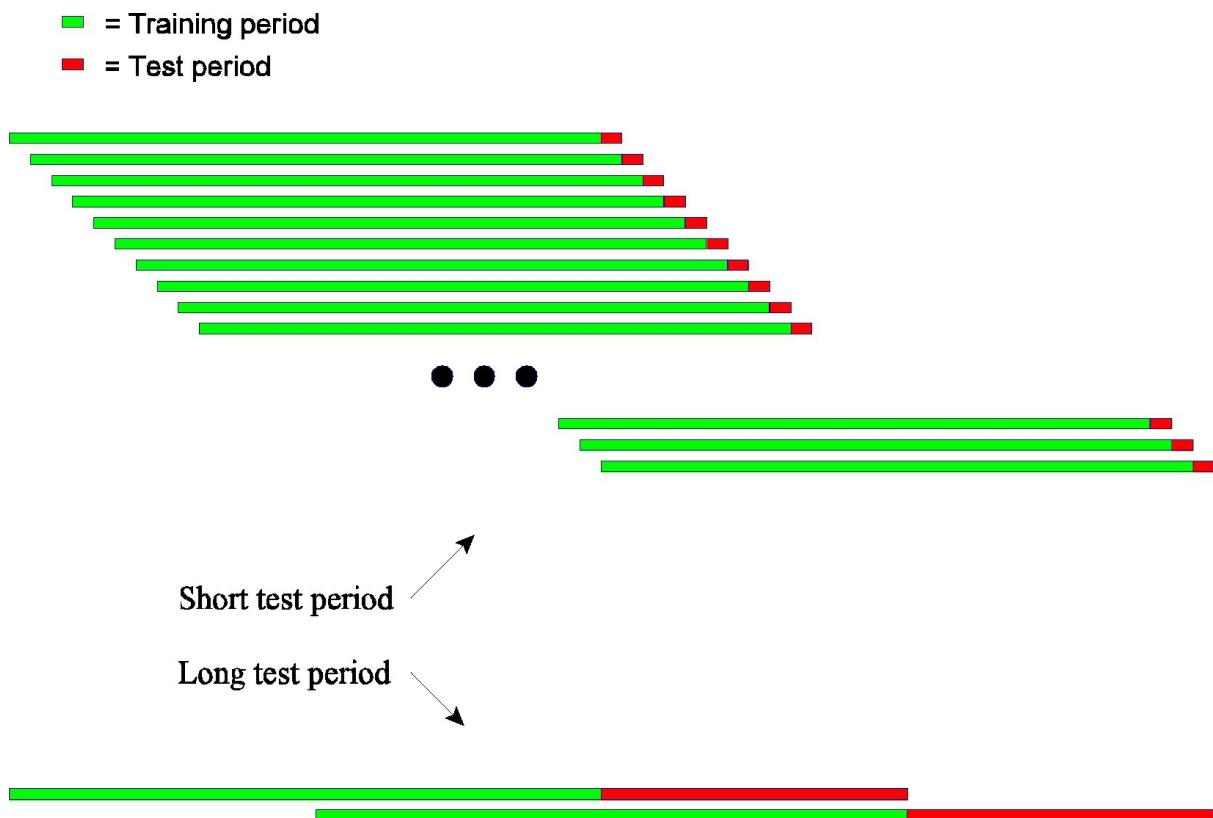
Pool all trades from the tests of 2008, 2009, and 2010. These trades are used to compute an unbiased estimate of the performance of the model.

The primary advantage of walkforward testing is that it mimics real life. Most developers of automated trading systems periodically retrain or otherwise refine their model. Thus, the results of a walkforward test simulate the results that would have been obtained if the system had been actually traded. This is a compelling argument in favor of this testing methodology.

Another advantage of walkforward testing is that it correctly reflects the response of the model to *nonstationarity* in the market. All markets evolve and change their behavior over time, sometimes rotating through a number of different regimes. Loosely speaking, this change in market dynamics, and hence in relationships between indicator and target variables, is called *nonstationarity*. The best predictive models have a significant degree of robustness against such changes, and walkforward testing allows us to judge the robustness of a model.

TSSB's ability to use a variety of testing block lengths makes it easy to evaluate the robustness of a model against nonstationarity. Suppose a model achieves excellent walkforward results when the test block is very short. In other words, the model is never asked to make predictions for data that is far past the date on which its training block ended. Now suppose the walkforward performance deteriorates if the test block is made longer. This indicates that the market is rapidly changing in ways that the model is not capable of handling. Such a

model is risky and will require frequent retraining if it is to keep abreast of current market conditions. On the other hand, if walkforward performance holds up well as the length of the test block is increased, the model is robust against nonstationarity. This is a valuable attribute of a predictive model. Look at [Figure 1](#) on the next page, which depicts the placement of the training and testing blocks (periods) along the time axis.



**Figure 1:** Walkforward testing with short and long test periods

[Figure 1](#) above shows two situations. The top section of the figure depicts walkforward with very short test blocks. The bottom section depicts very long test blocks. It can be useful to perform several walkforward tests of varying test block lengths in order to evaluate the degree to which the prediction model is robust against nonstationarity.

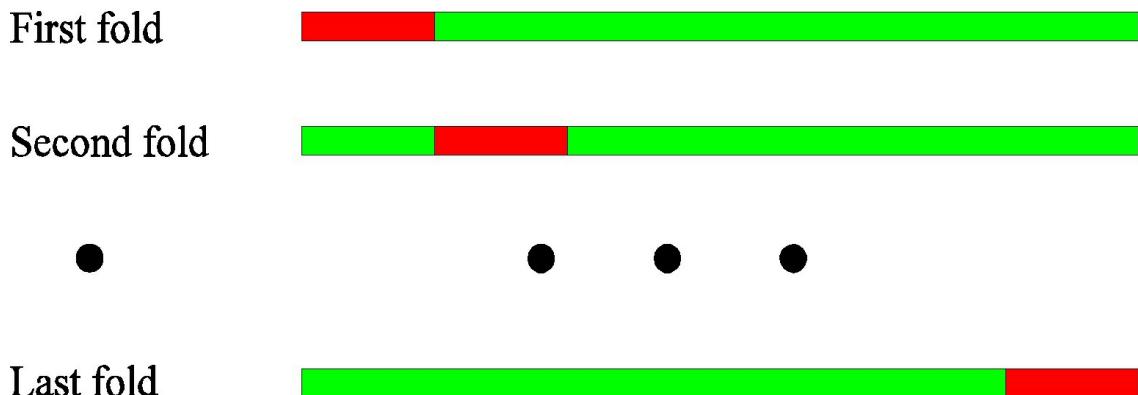
Walkforward testing has only one disadvantage relative to alternative testing methods such as cross validation: it is relatively inefficient when it comes to use of the available data. Only cases past the end of the first training block are ever used for testing. If you are willing to believe that the indicators and targets are reasonably stationary, this is a tragic waste of data. Cross validation, discussed in the [next section](#), addresses this weakness.

## Cross Validation

Rather than segregating all test cases at the end of the historical data block, as is done with walkforward testing, we can evenly distribute them throughout the available history. This is called *cross validation*. For example, we may test as follows:

- 1) Train using data from 2006 through 2008. Test the model on 2005 data.
- 2) Train using data from 2005 through 2008, excluding 2006. Test the model on 2006 data.
- 3) Train using data from 2005 through 2008, excluding 2007. Test the model on 2007 data.
- 4) Train using data from 2005 through 2008, excluding 2008. Test the model on 2008 data.

This idea of withholding interior ‘test’ blocks of data while training with the surrounding data is illustrated in [Figure 2](#) below. In cross validation, each step is commonly called a *fold*.



**Figure 2:** Cross validation

The obvious advantage of cross validation over walkforward testing is that every available case becomes a test case at some point. However, there are several disadvantages to note. The most serious potential problem is that cross validation is sensitive to nonstationarity. In a walkforward test, only relatively recent cases serve as test subjects. But in cross validation, cases all the way back to the beginning of the dataset contribute to test performance results. If the behavior of the market in early days was so different than in later days that the relationship between indicators and the target has seriously changed, incorporating test results from those early days may not be advisable.

Another disadvantage is more philosophical than practical, but it is worthy of note. Unlike a walkforward test, cross validation does not mimic the real-life behavior of a trading system. In cross validation, except for the last fold, we are using data from the future to train the model being tested. In real life this data would not be known at the time that test cases are processed. Some skeptics

will raise their eyebrows at this, even though when done correctly it is legitimate, providing nearly unbiased performance estimates. Finally, overlap problems, discussed in the [next section](#), are more troublesome in cross validation than in walkforward tests.

## Overlap Considerations

The discussions of cross validation and walkforward testing just presented assume that each case is independent of other cases. In other words, the assumption is that the values of variables for a case are not related to the values of other cases in the dataset. Unfortunately, this is almost never the situation. Cases that are near one another in time will tend to have similar values of indicators and/or targets. This generally comes about in one or both of the following ways:

- Many of the targets available in *TSSB* look further ahead than just the next bar. For example, suppose our target is the market trend over the next ten bars. This is the quantity we wish to predict in order to make trade decisions. If this value is high on a particular day, indicating that the market trends strongly upward over the subsequent ten days, then in all likelihood this value will also be high the following day, and it was probably high the prior day. Shifting ahead or back one day still leaves an overlap of nine days in that ten-day target window. Such case-to-case correlation in time series data is called *serial correlation*.
- In most trading systems, the indicators look back over a considerable time block. For example, an indicator may be the market trend over the prior 50 days, or a measure of volatility over the prior 100 days. As a result, indicators change very slowly over time. The values of indicators for a particular day are almost identical to the values in nearby days, before and after.

These facts have several important implications. Because indicators change only slowly, the model's predictions also change slowly. Hence market positions change slowly; if a prediction is above a threshold, it will tend to remain above the threshold for multiple bars. Conversely, if a prediction is below a threshold, it will tend to remain below that threshold for some time. If the target is looking ahead more than one bar, which results in serial correlation as discussed above, then the result of serial correlation in both positions and targets is serial correlation in returns for the trading system. This immediately invalidates most common statistical significance tests such as the t-test, ordinary bootstrap, and Monte-Carlo permutation test. *TSSB* does include several

statistical significance tests that can lessen the impact of serial correlation. In particular, the stationary bootstrap and tapered block bootstrap will be discussed [here](#). Unfortunately, both of these tests rely on assumptions that are often shaky. We'll return to this issue in more detail later when statistical tests are discussed. For the moment, understand that targets that look ahead more than one bar usually preclude tests of significance or force one to rely on tests having questionable validity.

Lack of independence in indicators and targets has another implication, this one potentially more serious than just invalidating significance tests. The legitimacy of the test results themselves can be undermined by bias. Luckily, this problem is easily solved with a *TSSB* option called OVERLAP. This will be discussed [here](#). For now we will simply explore the nature of the problem.

The problem occurs near the boundaries between training data and test data. The simplest situation is for walkforward testing, because there is only one (moving) boundary. Suppose the target involves market movement ten days into the future. Consider the last case in the training block. Its target involves the first ten days after the test block begins. This case, like all training set cases, plays a role in the development of the predictive model. Now consider the case that immediately follows it, the first case in the test block. As has already been noted, its indicator values will be very similar to the indicator values of the prior case. Thus, the model's prediction will also be similar to that of the prior case. Because the target looks ahead ten days and we have moved ahead only one day, leaving a nine-day overlap, the target for this test case will be similar to the target for the prior case. But the prior case, which is practically identical to this test case, took part in the training of the model! So we have a strong prejudice for the model to do a good job of predicting this case, whose indicators and target are similar to the training case. The result is optimistic bias, the worst sort. Our test results will exceed the results that would have been obtained from an honest test.

This boundary effect manifests itself in an additional fashion in cross validation. Of course, we still have the effect just described when we are near the end of the early section of the training set and the start of the test set. This is the left edge of the red regions in [Figure 2](#). But we also have a boundary effect when we are near the end of the test set and the start of the later part of the training set. This is the right edge of each red region. As before, cases near each other but on opposite sides of the training set / test set boundary have similar values for indicators and the target, which results in optimistic bias in the performance estimate.

The bottom line is that bias due to overlap at the boundary between training data

and test data is a serious problem for both cross validation and walkforward testing. Fortunately, the user can invoke the OVERLAP option to alleviate this problem, as will be discussed [here](#).

## Performance Criteria

Prior sections discussed the evaluation of performance using cross validation or walkforward testing. Those sections dealt with the mechanics of partitioning the data into training and test sets. Other considerations will be presented here. Several terms should be defined first:

- A *fold* is a single partitioning of the available data into a training set, a test set, and perhaps some cases from the dataset that are temporarily omitted in order to handle overlap. In Figures 1 and 2, a fold would be one of the horizontal strips consisting of the green training block and the red test block.
- The *in-sample* (or *IS*) performance for a fold is the performance of the model or trading system in the current training set. Because the training process for the model optimized some aspect of its performance, the in-sample performance will usually have an optimistic bias, often called the *training bias*.
- The *out-of-sample* (or *OOS*) performance for a fold is the performance of the model or trading system in the current test set. Because this data did not take part in training the model, it is, for all practical purposes, unbiased. In other words, on average it will reflect the true capability of the model or trading system.

## Model Performance Versus Financial Performance

Performance statistics for model-based trading systems fall into two categories. One is the predictive performance of the model that determines trade decisions. This may include statistics such as the mean squared error of the predictions, or the model's R-squared. The other is the financial performance of the trading system, such as its profit factor or Sharpe ratio. Naturally, there is a degree of correspondence between statistics in these two categories. If a model has excellent performance, the trading system probably will as well. By the same token, a poorly performing model will most likely produce a poorly performing trading system.

On the other hand, the degree of correspondence between model and trade performance may not always be as high as one might expect. For example, the venerable *R-squared* is a staple in many fields. Yet it has a shockingly low relationship with the profit factor of the trading system, a commonly used measure of financial performance. In fact, it is not uncommon for a model-based trading system with respectable financial performance to be driven by a predictive model that has a negative R-squared! (Roughly speaking, a negative R-squared means that the model's error variance exceeds that of just guessing the target mean for every case.) This peculiar behavior happens because trades are signaled only for extremely high or low predictions, which are almost always the most reliable decisions. For less extreme predictions, those lying between the short and long thresholds where no trades are taken, errors can be so large that they overwhelm the predictive quality at the extremes. The result is a negative R-square.

Because of this frequent discrepancy, financial performance statistics for the trading system, such as profit factor, are much more useful than predictive performance of the model. Nonetheless, model accuracy is of some interest and should always be examined, if for no other reason than to check for anomalous behavior.

## Financial Relevance and Generalizability

Performance statistics, especially for the trading system, can be graded on two scales: *financial relevance* and *generalizability*. This is not to say that these two criteria are mutually exclusive, or even at opposing ends of a continuum. Still, they are often independent of one another, and some discussion is warranted.

A *financially relevant* statistic is one that is important from a profit-making or money management perspective. For example, the ratio of annual percent return to average annual drawdown is of great interest to a person responsible for money management.

The *generalizability* of a statistic is a somewhat vague term, though important. It refers to the degree to which its IS (in-sample) value tends to hold up OOS (out-of-sample). The reason this is important is that when we train a model to predict a target, we do so by optimizing some measure of performance. Obviously, our ultimate goal is OOS performance. A model's or trading system's IS performance is of academic interest only; it is the OOS performance that determines whether the system will make money for us. Thus, if the training process optimizes a performance statistic that does not tend to hold up OOS, we

have gained little or nothing. When we choose a performance statistic to optimize during training, we are best off if we choose one whose performance OOS is likely to reflect its IS value. In other words, we want to optimize a performance statistic that has good generalizability.

The generalizability of a statistic is somewhat dependent on the particular model, the indicators, and the target. For this reason, the developer should experiment and assess this property by examining individual fold results for several candidate statistics. However, experience provides a few guidelines. For example, profit factor tends to have good generalizability. In contrast, performance statistics that depend on the temporal order of gains and losses, such as anything involving drawdown, have poor generalizability. Finally, the more trades that go into a statistic, the more likely it is to generalize well.

## Performance Statistics in *TSSB*

The *TSSB* program computes and prints a wide variety of predictive accuracy and financial performance statistics. These are provided for the underlying model(s) as well as the complete trading system that may be based on numerous models, committees, and so forth. We will regularly refer to them throughout the remainder of this document, so rather than repeat definitions every time they appear, they will all be defined here. The program also prints some quantities that are not exactly performance measures, but are related to performance. These quantities are included here as well. The following items are listed in the approximate order that they usually appear in the program's log file.

***Final best crit*** - This depends on the optimization criterion employed by the user. It refers to the stepwise selection of the indicators chosen by the program. This figure is of little or no interest to most users, and is included only for the use of advanced users who are interested in technical details of operation.

***Target grand mean*** - The mean value of the target variable. This is especially useful when examining individual fold results. If the in-sample target mean is very different from the out-of-sample target mean, poor performance on this fold might be excused.

***Outer hi thresh* and *means*** - We saw [here](#) that trade decisions are made by comparing the model's predictions with one threshold for long trades and another (usually lower) threshold for short trades. The *Outer hi thresh* line in the log file shows the optimal long threshold computed by the program, the number of cases that equaled or

exceeded this threshold, the mean target for cases whose predictions lay at or beyond the threshold, and the mean target for the cases below this threshold. We hope that the mean target value of cases beyond the outer hi threshold exceeds that of cases below the threshold. In some applications, *Inner* thresholds and means will also be printed. This is a very advanced concept. See the manual for details.

**Outer lo thresh** and **means** - This is the corresponding information for the lower (sell short trades) threshold.

**Target statistics at various percentages kept** - If the model is effective at predicting the target, one would expect that cases having large predicted values will have large actual target values, and cases having small predictions will have small actual target values. This table lets us assess in a variety of ways the degree to which this is happening. The table contains five columns, corresponding to 10, 25, 50, 75, and 90 percent of the cases having predictions beyond upper (long) and lower (short) thresholds. The various statistics printed in this table are best explained in the context of a specific application with actual numbers, so details are deferred until [here](#).

**MSE** - Mean squared error of the model. In financial applications this figure is nearly meaningless, but it is printed for completeness.

**R-squared** - Fraction of the target variance which is predictable by the model. In pathological cases, which are not uncommon in financial applications, this widely used predictive accuracy measure may be negative. A negative R-squared means that the model's predictions are actually worse than just guessing the target mean for every case. In financial applications this figure is nearly meaningless, but it is printed for completeness.

**ROC Area** - Early communication theory used a *Receiver Operating Characteristic (ROC)* curve to depict the performance of a communication system. The area under this curve is strongly correlated with the ability of the system to separate information from noise. The *TSSB* program generalizes the original communication version to financial modeling. Roughly speaking, the *ROC Area* measures the degree to which large predictions correspond to large target values, and small predictions correspond to small target values. The ROC area ranges from zero (a model that gets its predictions entirely backwards) to one (a perfect model). A

value of 0.5 corresponds to random guessing.

***Buy-and-hold profit factor*** - The profit factor that would be obtained by treating each case as a single trade, taking a long position for every one of them and holding each for the look-ahead distance of the target variable. The buy-and-hold profit factor represents the financial performance of a naive trading system, and thus serves as a basis of comparison for the *long profit factor* of the trained model. TSSB allows targets that may not represent profits. Therefore this value is printed only if the target is interpretable as a profit, or if the user specified a profit variable. See [here](#) for a discussion of this issue.

***Sell-short-and-hold*** - The profit factor that would be obtained by treating each case as a single trade, taking a short position for every one of them, and holding that position for the look-ahead distance of the target. This is the reciprocal of the *Buy-and-hold profit factor*. This serves as a naive-trading-system baseline for comparison with the *short profit factor* of the trained model. This value is printed only if the target is interpretable as a profit, or if the user specified a profit variable. See [here](#) for a discussion of this issue.

***Dual-thresholded outer PF*** - The profit factor that would be obtained by treating each case as a single trade and taking a long position for each case whose prediction equals or exceeds the upper threshold, and taking a short position for each case whose prediction is less than or equal to the lower threshold. The duration of the trade is the look-ahead distance of the target variable. This statistic measures both long and short trading performance. This value is printed only if the target is interpretable as a profit, or if the user specified a profit variable. See [here](#) for a discussion of this issue.

***Outer long-only PF*** and ***Improvement Ratio*** - Outer long-only PF is the profit factor that would be obtained by treating each case as a single trade and taking a long position for each case whose prediction equals or exceeds the upper threshold. The duration of the trade is the look-ahead distance of the target variable. Thus, this statistic measures only the performance of *buy* signals. The *Improvement Ratio* is the *Outer long-only PF* divided by the *Buy-and-hold profit factor*. In other words, this is the factor by which the long-only version of the model improves profit factor over a simple buy-and-hold strategy. These values are printed only if the target is interpretable as a profit, or if the user specified a profit variable. See [here](#) for a

discussion of this issue.

**Outer short-only PF** and **Improvement Ratio** - As above, except for short positions only.

**Balanced (0.01 each side) PF** - This statistic is printed only if the FRACTILE THRESHOLD option ([here](#)) is invoked in a multiple-market situation, such as the stocks in the S&P 500 or some other basket of equities. This is the profit factor that would be obtained by treating each bar as a single trade opportunity in each market (or financial instrument), and taking a long position for each market whose prediction is in the upper (highest ranked) one percent of predictions across all of the markets being traded, and taking a short position for each market whose prediction is in the lower (lowest ranked) one percent of predictions across all of the markets. The idea is that we hold a balanced position, long and short an equal number of markets. This is known as a *market-neutral* strategy. This value is printed only if the target is interpretable as a profit, or if the user specified a profit variable. See [here](#) for a discussion of this issue. See [here](#) for a detailed description of the various *Balanced* optimization criteria and the concept of balanced multiple-market trading.

**Balanced (0.05 / 0.10 / 0.25 / 0.50 each side) PF** - As above, except that the 10, 25, and 50 percentile subsets of predictions are taken across all markets.

**Profit factor above and below various thresholds** - This table is printed only in a cross-validation or walkforward summary, and only if the FRACTILE THRESHOLD option ([here](#)) is invoked in a multiple-market situation. The table contains five columns. The first column lists a variety of thresholds. The second column is the fraction (0-1) of cases whose predicted target is greater than or equal to the threshold for that row. The third is the profit factor one would obtain from taking a long position for those cases. The fourth is the fraction of cases whose predicted target is strictly less than the threshold. The last column is the profit factor one would obtain from taking a short position for those cases. This table is useful for evaluating the tradeoff between the number of trades taken and the profit factor of the corresponding trading system.

## Desirable Program Features

It is possible to use a general-purpose statistical modeling package to develop and test a financial market trading system based on predictive modeling. The best statistical analysis programs contain a data transform language that can be used to create indicator and target variables. Or, these variables can be computed with specialized software. Then, model-building methods in the package can create a predictive model from a specified training set. This model can be applied to a test set, and the predictions exported to a spreadsheet program. With some work, the spreadsheet program can then be used to compute basic financial performance statistics.

It should be obvious that the procedure just described is awkward, tedious, and of limited versatility. The development and testing of predictive-model trading systems is best done with software written specifically for this task. A professional program will do the following, at a minimum:

- Be able to compute a wide variety of indicators and targets, saving the user from the need to write or purchase specialized software to do this.
- Contain a scripting language that will let the user define variables that are not predefined in the program. The language will also enable the user to modify existing variables.
- Be capable of developing and testing both stand-alone trading systems and signal filters for existing systems.
- Handle both daily and intraday data.
- Process multiple markets, including the ability to compute cross-sectional indicators based on the behavior of individual markets in the context of a universe of markets, such as the S&P 500, a variety of currencies, or interest-rate futures.
- Be able to export standard-format databases to other programs, and read externally produced databases.
- Supply a variety of modeling methods so that the user can select the one having the best combination of power, resistance to overfitting, and speed.
- Have the ability to automate selection of indicators from a list of candidates.
- Offer a wide and useful variety of optimization criteria so that the user can choose a criterion that best suits his or her purpose.

- Support regime training and testing to facilitate models that specialize in specific regimes such as high (or low) volatility, up (or down) trends, and so forth.
- Supply both cross validation and walkforward testing at a variety of granularities (day, month, year, and so forth).
- Provide details on the model(s)' predictive accuracy and financial performance statistics for both the training set and the test set individually for every fold, as well as pooled test set results.
- When possible, compute statistical significance levels for financial performance statistics.
- Preserve predictions for examination within the program as well as export to other programs.
- Implement a variety of model-combining committees to enable state-of-the-art prediction capabilities.
- Include built-in graphics capabilities to study variables and their relationships.

*TSSB* includes all of these capabilities, along with many more features that are useful in the development and testing of predictive-model trading systems and signal filters. In the [next chapter](#) we will delve into the program with a specific example of the development and testing of a simple automated trading system based on predictive modeling.

# A Simple Standalone Trading System

To illustrate the first of TSSB's two uses, we begin with the development of a simple standalone trading system. (Development of a system for filtering trades of an existing system will be presented in the [next chapter](#).) This chapter will show and discuss all of the files that are necessary to implement this system, although the discussion of this first system will be limited to an overview. Extensive details will appear later in this document, and we will provide references to these details here to satisfy readers who want to occasionally flip ahead.

## The Script File

All *TSSB* operations are controlled by a script file. The default extension for the file's name is. SCR, although the user is free to use any extension, such as .TXT. The .SCR extension was a standard for script files for many years, but since Microsoft Windows chose it for screen savers, this extension has become problematic on some computers.

The script file for this example is named SIMPLE\_STANDALONE.SCR. Its contents are now listed, and a discussion of each line follows.

```
READ MARKET LIST "SYMBOLS.TXT" ;
READ MARKET HISTORIES "E:\SP100\IBM.TXT" ;
READ VARIABLE LIST "TREND VOLATILITY.TXT" ;

MODEL SIMPLE_MODEL IS LINREG [
    INPUT = [ P_TREND P_VOLATILITY ]
    OUTPUT = DAY_RETURN
    MAX STEPWISE = 0
    CRITERION = PROFIT_FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

WALK FORWARD BY YEAR 10 1999 ;
```

Here is a brief discussion of each line of the script file:

```
READ MARKET LIST "SYMBOLS.TXT" ;
```

*TSSB* needs to know which markets (or tradeable instruments) will take part in the development. These markets are listed in the file SYMBOLS.TXT. Keeping the market list in a separate file facilitates fast and easy changes to the markets that are used for development. Note that like all files named in a script file, the name is enclosed in

quotes and must not include blanks or any other character that Windows deems illegal for file names. The READ MARKET LIST command is discussed in detail [here](#).

```
READ MARKET HISTORIES "E:\SP100\IBM.TXT";
```

The market histories are ASCII files that contain the date, optional time, market symbol, open, high, low, close, and volume. The user must name one of the files (IBM.TXT here) so that TSSB knows where to find all of them. This named file need not appear in the market list specified in the READ MARKET LIST command shown above, but the named file must be present in the specified directory. The READ MARKET HISTORIES command must appear on any line after the READ MARKET LIST command so that the program knows which markets to read. The READ MARKET HISTORIES command is discussed in detail [here](#).

```
READ VARIABLE LIST "TREND_VOLATILITY.TXT";
```

TSSB has the ability to internally compute a wide variety of indicators and targets. The user creates an ASCII text file, often called a *variable definition list* file, that names and defines the indicators and targets that the user wishes to compute. The variable definition list file that we wish to use is specified with this command (READ VARIABLE LIST). This command must appear on any line after the READ MARKET HISTORIES command so that the market data has been read and is ready to be processed. The READ VARIABLE LIST command is discussed in detail [here](#).

```
MODEL SIMPLE_MODEL IS LINREG [
```

The MODEL command specifies the modeling method to be used for model development and assigns a name for the resulting prediction model. Here, the user chose to name the model SIMPLE\_MODEL. A name such as JOSEPH would be allowed as well, although it would probably be confusing to users. Blanks and special characters other than an underscore are not allowed in the name. The type of model in this example is simple linear regression, which is specified by means of the keyword LINREG. Specifications for the model are enclosed in square brackets. Because it is legal (though not required) to spread the specifications across multiple lines, clarity is often enhanced by putting the opening bracket on the model definition line and putting individual specifications on separate lines. The MODEL command is discussed in detail [here](#).

The following five items are specifications for the model developed in this example:

**INPUT = [ P\_TREND P\_VOLATILITY ]**

The INPUT line specifies the indicators that will be used by the model. This indicator list is enclosed in square brackets. The two indicators here, P\_TREND and P\_VOLATILITY, are defined in the variable definition file, which will be discussed soon. The INPUT command is discussed in detail [here](#).

**OUTPUT = DAY\_RETURN**

The OUTPUT line specifies the target variable, the true values of the quantity that the model will be asked to predict. The target here, DAY\_RETURN, is defined in the variable definition file, which will be discussed soon. The OUTPUT command is discussed in detail [here](#).

**MAX STEPWISE = 0**

The MAX STEPWISE specification tells the automated stepwise indicator selection algorithm in TSSB the maximum number of indicators that may be used as inputs to the developed model. By setting it to zero, we tell the program that it is to skip automatic selection, and use every indicator in the INPUT list. The MAX STEPWISE command is discussed in detail [here](#).

**CRITERION = PROFIT FACTOR**

TSSB allows the user to choose from among a variety of optimization criteria (sometimes called the *objective function*) for several aspects of model training. The PROFIT FACTOR criterion tells the program that we are most interested in maximizing the two-sided (long and short) profit factor of the trading system. The CRITERION command is discussed in detail [here](#).

**MIN CRITERION FRACTION = 0.1**

For performance reports as well as some optimization criteria, TSSB generally chooses long and short prediction thresholds that trigger trades in such a way that the profit factor of the trades is maximized. (The actual mechanism can be much more involved, too complex to address here. Further details will be provided in subsequent sections.) The problem with choosing a threshold that maximizes the trade profit factor is that unless constraints are imposed, the program may set such an extreme threshold that only one, or very few, winning trades are executed, leading to a huge or infinite profit factor. The MIN CRITERION FRACTION specifies a minimum fraction of trade opportunities that result in actual trades on each side (long and short, counted separately). (In TSSB, by default each bar in each market is a trade opportunity.) Here, the user has specified that

at least 0.1 (ten percent) of the trade opportunities must result in a long position being taken, and another ten percent must result in a short position being taken. The MIN CRITERION FRACTION command is discussed in detail [here](#).

] ;

This closing bracket (along with the obligatory semicolon that ends commands) completes the model specifications begun with the MODEL command.

**WALK FORWARD BY YEAR 10 1999;**

This command specifies that the trading system will be evaluated by means of a walk forward test. BY YEAR means that the testing window will encompass one calendar year, each training window will encompass the ten years prior to the test window, and the first test window will be in 1999. The WALK FORWARD command is discussed in detail [here](#).

We'll now examine the files that are referenced in the SIMPLE\_STANDALONE.SCRscript file just discussed. The first file referenced is the market list, SYMBOLS.TXT. It is just a list of the trading instruments that will be processed:

**AA  
DELL  
DOW  
GE  
...  
XOM  
XRX**

Next, the script file names a market history file. Here are a few lines from such a file:

```
20110301 16.90 16.94 16.21 16.23 297967
20110302 16.20 16.43 16.13 16.18 201850
20110303 16.37 16.77 16.36 16.63 191486
20110304 16.77 16.80 16.37 16.58 186114
20110307 16.58 16.75 16.16 16.17 119222
```

In the market history file excerpted above, the date appears first as YYYYMMDD. This is followed by the open, high, low, and close prices. Volume appears last on the line.

Finally, the script file references a file called the variable definition list. Here is this file, which defines three variables (two indicators and a target):

```
P_TREND:      LINEAR PER ATR 5 100
P_VOLATILITY: PRICE VARIANCE RATIO 5 4
DAY_RETURN:   NEXT DAY ATR RETURN 250
```

The first line defines a variable called P\_TREND, which will be used as an indicator. This could just as well have been named MARY or PHIL, but it is always good to make the name descriptive. The maximum length of a name is 15 characters. Blanks and special characters other than an underscore () are not allowed. The P\_TREND variable is defined as an indicator that is built into the TSSB library: LINEAR PER ATR. This is the least-squares linear slope divided by ATR (average true range) as a normalizer. The linear slope is fit over a window of 5 bars, and ATR is computed over a window of 100 bars.

The volatility variable (called P\_VOLATILITY here) is the ratio of the variance of the log of price over a 5-bar window, divided by the variance over a  $5 \times 4 = 20$  bar window. This built-in variable will serve as another indicator for the model.

The variable that the model uses as a target is given the name DAY\_RETURN, which is nicely descriptive. It is defined as the TSSB built-in variable NEXT DAY ATR RETURN, which is the price change from tomorrow's open to the next day's open, normalized by the 250-day ATR.

TSSB's built-in variables will be discussed in somewhat more depth [here](#) of this manual. However, the User's Manual is the ultimate reference for all built-in variables.

## The Audit Log

TSSB always writes an ASCII text file called AUDIT.LOG that contains the results of its operations. If the program encountered an error, in many cases the audit log will contain (usually as its last line) a more detailed explanation of the error than appeared on the screen. We now examine the audit log produced by the example script file shown above. For clarity, it will be broken into sections, with each explained separately.

```
COMMAND ---> READ MARKET LIST "SYMBOLS.TXT" ;
User specified 12 markets.
AA DELL DOW GE HAL HNZ IBM JNJ JPM WMT XOM XRX
```

Each command in the script file is echoed to the log file. Here, we see the **READ MARKET LIST** command and its results. The log file tells us that we are processing 12 markets, and they are listed.

```
COMMAND ---> READ MARKET HISTORIES "E:\SP100\IBM.TXT" ;  
  
Reading market histories (*.TXT) from path E:\SP100\  
  
AA had 10393 cases (19700102 000000 through 20110307  
000000)  
Max ratio = 1.29 on 20081010 000000  
  
DELL had 5724 cases (19880623 000000 through 20110307  
000000)  
Max ratio = 1.46 on 19930525 000000  
  
DOW had 6855 cases (19840104 000000 through 20110307  
000000)  
Max ratio = 1.22 on 19871020 000000  
  
.....  
  
XRX had 6855 cases (19840104 000000 through 20110307  
000000)  
Max ratio = 1.42 on 19991008 000000
```

The READ MARKET HISTORIES command is processed. For each market, the following information is printed:

- Name of the market and the number of cases in the history file
- Starting date as YYYYMMDD, and time as HHMMSS. For daily data, the time is zero.
- Ending date in the same format
- Maximum ratio of open prices for any two consecutive bars, and the date/time it occurred. Unusually high ratios indicate a very large change in price from one bar to the next, and while this may be legitimate, it may indicate an error in the market data.

```
COMMAND ---> READ VARIABLE LIST "TREND_VOLATILITY.TXT" ;  
  
Temporary work file is  
D:\BOOSTER\DOC\Tutorial\Examples\BoosterTempFile0.tmp  
  
Database file is  
D:\BOOSTER\DOC\Tutorial\Examples\BoosterTempDatabase.tmp
```

The variable definition file “TREND\_VOLATILITY.TXT” is read. Although it is of little practical importance in most cases, the log file (AUDIT.LOG) names any special temporary work files that are created. These files can be gigantic

(gigabytes for large applications), so the user should confirm that the drive on which the files are created contains sufficient free space. Normally, all work files will be automatically deleted when the program ends, and if any happen to be left from a crashed prior session, they will be deleted when TSSB starts again. However, in rare cases of user error leading to abnormal program termination, enormous work files will remain on the hard drive until *TSSB* is run again. If the hard drive suddenly seems full, it would not hurt to search it for any .tmp files whose name begins with *Booster*. As long as the program is not running, it is safe to delete them.

```
Summary information for variable P_TREND:
Market Fbad Bbad Nvalid First      Last       Min      Max     Mean
    AA   100   0   10293 19700526 20110307 -50.000 49.467 0.719
  DELL   100   0    5624 19881114 20110307 -49.972 49.783 1.007
    DOW   100   0    6755 19840525 20110307 -50.000 49.582 0.821
...
XRX   100   0    6755 19840525 20110307 -50.000 49.721 0.719
```

For each variable in the definition file, summary information is printed for each market. This information is as follows:

**Market** - The symbol for the market

**Fbad** - The number of undefined values at the front (that's what F stands for) of the file. In this case, there are 100 undefined values, which makes sense. The P\_TREND variable is normalized by ATR (average true range) in a 100-bar window. So we need to pass 100 bars before this indicator's initial value can be computed.

**Bbad** - The number of undefined values at the back (end) of the file. For indicators this will always be zero, because indicators look only backwards in time. But targets look forward in time, so as the end of the dataset is approached, targets will become undefined. For example, the DAY\_RETURN target here would have Bbad equal to 2, because it is based on the price change from tomorrow's open to the next day's open. It needs to look two days into the future.

**Nvalid** - The number of cases in this market that have valid (computable) values of this variable. A variable cannot be computed for a case if the case is too early in the database for required history to be available (Fbad), too late in the database for required future prices to be available (Bbad), or if the lookback or lookahead window contains missing or invalid data. The most common cause for invalid data in the interior of the dataset is suspicious price jumps flagged by the

optional CLEAN RAW DATA command described [here](#).

**First** - The date (YYYYMMDD) of the first valid occurrence of this variable.

**Last** - The date (YYYYMMDD) of the last valid occurrence of this variable. It is not guaranteed that all cases within this date range contain valid values of this variable. The most common cause for invalid data in the interior of the dataset is suspicious price jumps flagged by the optional CLEAN RAW DATA command described [here](#).

**Min** - The minimum value of this variable in the market.

**Max** - The maximum value of this variable in the market.

**Mean** - The mean value of this variable in the market.

```
User defined 3 variables
Wrote 87168 records to the database
```

AA	10141
DELL	5472
DOW	6603
GE	6603
HAL	6603
HNZ	6603
IBM	12128
JNJ	6603
JPM	6603
WMT	6603
XOM	6603
XRX	6603

The last thing done by *TSSB* when processing the READ VARIABLE LIST command is to report the number of variables found in the variable definition file, the number of cases (records) placed in the database, and the number of those cases attributed to each market. Note that some markets have more cases because these markets begin at an earlier date.

```
COMMAND ---> MODEL SIMPLE_MODEL IS LINREG [
    INPUT = [ P_TREND P_VOLATILITY ]
    OUTPUT = DAY_RETURN
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;
```

*TSSB* echoes all script file commands to the log file. These lines are the

definition of the model and its specifications, which we already discussed.

## A Walkforward Fold

We continue this tour of the log file AUDIT.LOG with the results of a walkforward fold.

```
COMMAND ---> WALK FORWARD BY YEAR 10 1999 ;
Beginning walk forward by year
```

Walkforward is about to begin. Both IS (in-sample, the training set) and OOS (out-of-sample, the test set) results will be written for each fold. For economy, we will now examine only the first fold. The output generated for the fold will be broken up into sections for clarity. All in-sample results appear first, followed by the out-of-sample results.

```
-----
Walkforward test date 1999 train 30219 cases, testing 3024
-----
```

```
LINREG Model SIMPLE_MODEL predicting DAY_RETURN
Stepwise not used; all predictors available to model
Regression coefficients:
-0.001424 P_TREND
 0.000257 P_VOLATILITY
 0.038171 CONSTANT
```

We see that this walkforward fold is testing the year 1999. The training set for this fold contains 30219 cases, and the test set contains 3024 cases.

The model is linear regression (LINREG), the user named the model SIMPLE\_MODEL, and its target variable is DAY\_RETURN.

Stepwise indicator selection was not used. (Recall that the user specified MAX STEPWISE = 0.) This forces all indicators (often called *predictors* in the context of a predictive model) to be made available to the model being defined. In the case of linear regression, this implies that all variables will be used by the model. Some models, which will be discussed later, may choose to ignore some indicators, even if those indicators are made available to them.

The coefficients of the linear regression model, as well as the constant offset, are printed.

**Target grand mean = 0.03721**

**Outer hi thresh = 0.07134 with 4040 of 30219 cases at or above (13.37 %) Mean = 0.09306 versus 0.02859**

**Outer lo thresh = 0.00428 with 3025 of 30219 cases at or below (10.01 %) Mean = -0.04828 versus 0.04672**

We are still dealing with the training set in this fold. The mean of the target variable, DAY\_RETURN, in the training set is 0.03721.

TSSB found an upper (long trades) threshold of 0.07134 as the prediction threshold that maximizes the profit factor of long trades, subject to the condition that at least ten percent of the potential trades (training cases, which are bars in markets by default) be taken as long trades (MIN CRITERION FRACTION = 0.1). Of the 30219 training cases, 4040 of them, or 13.37 percent, had predictions at or above this threshold and hence signaled long trades. The mean of the target variable, DAY\_RETURN, in these long trades was 0.09306, while the mean of the target for the trades not taken (those trade opportunities in which a trade was not signaled) was 0.02859. Thus, in the in-sample data (training data), the model discriminated well between trade opportunities that offered good buy opportunities and those that did not. This is no surprise.

Corresponding results are reported for the low (short trades) threshold. Note that its optimal threshold came almost right up against the restriction of taking at least ten percent of the trades as short, with 10.01 percent of possible trades signaling as short. As would be expected for in-sample (training set) data, the mean target for cases whose predictions are at or below the low threshold is less than the mean target of cases above the short threshold ( -0.04828 vs. +0.04672).

**Target statistics at various percentages kept...**

<b>Statistic</b>	<b>10</b>	<b>25</b>	<b>50</b>	<b>75</b>	<b>90</b>
<b>Mean target above</b>	<b>0.0807</b>	<b>0.0745</b>	<b>0.0513</b>	<b>0.0508</b>	<b>0.0466</b>
<b>N wins above</b>	<b>1475</b>	<b>3688</b>	<b>7146</b>	<b>10723</b>	<b>12820</b>
<b>Mean win above</b>	<b>0.8105</b>	<b>0.7500</b>	<b>0.7140</b>	<b>0.7015</b>	<b>0.6979</b>
<b>Total win above</b>	<b>1195.5</b>	<b>2765.9</b>	<b>5102.1</b>	<b>7522.6</b>	<b>8947.6</b>
<b>N losses above</b>	<b>1270</b>	<b>3208</b>	<b>6605</b>	<b>9944</b>	<b>11981</b>
<b>Mean loss above</b>	<b>0.7493</b>	<b>0.6868</b>	<b>0.6550</b>	<b>0.6407</b>	<b>0.6410</b>
<b>Total loss above</b>	<b>951.6</b>	<b>2203.1</b>	<b>4326.5</b>	<b>6370.7</b>	<b>7679.7</b>
<b>Profit factor above</b>	<b>1.2563</b>	<b>1.2555</b>	<b>1.1793</b>	<b>1.1808</b>	<b>1.1651</b>

The lines shown above are only half of the complete chart, statistics for cases whose predictions are above a threshold. The second half of the chart, for cases below a threshold, will appear soon. However, this chart contains an enormous amount of information, so it's best to take it one half at a time.

Each of the five columns represents a different threshold. Rather than specifying numeric values, they are specified as percentiles. So, for example, the first column represents whatever threshold results in ten percent of the cases lying at or above the threshold, and the statistics shown in that column are for that ten percent of cases. Similarly, the rightmost column represents a much lower threshold, so low that ninety percent of the cases lie at or above it.

We'll now examine each row statistic:

**Mean target above** - The mean of the target variable (DAY\_RETURN) in this column's percent of the cases. (A case is a trading opportunity, which in most applications is a bar in a market.) Notice the ‘ideal’ behavior: the mean monotonically decreases from 0.0807 to 0.0466 as the number of cases kept increases. In other words, as the threshold for signaling a long trade is loosened (decreased) to produce more trades, the mean win for those trades decreases. The best long trades, with a mean DAY\_RETURN of 0.0807, are obtained by keeping only the top ten percent of predictions. But remember, we are still in the training set! OOS results, which are what we are really interested in, will appear later.

**N wins above** - The number of trades in this ‘at or beyond the threshold’ set which have a positive target value. (A future version of the TSSB program will also express this as a percentage of trades.)

**Mean win above** - The mean target value for winning trades (cases whose target value is positive).

**Total win above** - The total target value of all winning trades. In other words, if the target measures profit then this is the sum of gains on all winning trades.

**N losses above, Mean loss above, Total loss above** - As above, except for losing trades (those whose target value is negative).

**Profit factor above** - The profit factor of the cases whose prediction equals or exceeds the threshold. This is the total wins divided by the total losses, which is the industry standard definition.

The second half of this table does not add much to the prior discussion except for one thing to note: These results apply to short trades, since they involves cases whose predictions are at or below a threshold. Observe ‘ideal’ behavior similar to what was seen in the prior table: the mean target value is at its smallest (which is what we want for short trades) when we keep only the

smallest (usually but not necessarily the most negative) ten percent of predicted values. The target mean monotonically increases as we loosen the threshold (make it larger) so as to produce more short trades.

Statistic	10	25	50	75	90
Mean target below	-0.0478	-0.0035	0.0230	0.0247	0.0324
N wins below	1488	3524	6864	10261	12198
Mean win below	0.7120	0.6718	0.6429	0.6370	0.6384
Total win below	1059.5	2367.6	4412.8	6536.3	7786.6
N losses below	1285	3383	6959	10417	12631
Mean loss below	0.7120	0.6920	0.6841	0.6813	0.6862
Total loss below	915.0	2341.0	4760.5	7096.7	8667.5
Profit factor below	1.1580	1.0113	0.9270	0.9210	0.8984

Here is the final information for the training section of the 1999 fold:

```
MSE = 0.72075 R-squared = 0.00108 ROC area = 0.52323
Buy-and-hold profit factor=1.129 Sell-short-and-hold=0.886
Dual-thresholded outer PF = 1.245
Outer long-only PF = 1.310 Improvement Ratio = 1.161
Outer short-only PF = 1.160 Improvement Ratio = 1.309
```

MSE is the mean squared error of the linear regression model, and R-squared is the corresponding fraction of the target variance accounted for by the model's predictions. It's extremely low, as is the rule in financial applications. In most situations, the best predictions (those most likely to result in winning trades) are in the tails of the model's distribution of predictions, the most extreme large and small values. The bulk of the cases, whose predictions lie in the vast midland, tend to be more or less random. This is why R-squared is usually tiny.

The ROC area, which was briefly discussed [here](#), is of only modest interest. Recall that the value of ROC area ranges from zero (a model that gets every decision exactly wrong) to one (a model that gets every decision exactly correct), with 0.5 corresponding to random guessing. So it is nice, though not unexpected, to see a ROC area in excess of 0.5, which indicates some degree of predictive power. This is the result for the training set, after all, where we expect results to be infected with data mining bias.

The remaining parameters were also discussed in that earlier chapter following the ROC area definition, so we will not dwell on them except to note that the model improved performance over naive holding on both the long and short side. Again, this is the training set (in-sample data) being evaluated here, so we should not be surprised.

## Out-of-Sample Results for This Fold

The prior results concerned the training set (1989-1998) in the first walkforward fold. Results for the test set, 1999, now appear in the log file:

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

```
Target grand mean = 0.02750
```

```
Outer hi thresh = 0.07134 with 430 of 3024 cases at or
above (14.22 %) Mean = 0.13452 versus 0.00976
```

```
Outer lo thresh = 0.00428 with 264 of 3024 cases at or
below (8.73 %) Mean = -0.05468 versus 0.03536
```

The mean of the target variable DAY\_RETURN is a little smaller in the test set (0.02750) than it was in the training set (0.03721). Notice that the thresholds here are the same as those in the training set. (Of course! These are the optimal thresholds that the program found for the training set, so naturally we must keep them. It would be cheating to find new optimal thresholds for the test set!) We see that 14.22 percent of the 1999 trade opportunities, 430 of the 3024, were long trades because their predictions lay at or above the upper threshold. The mean of DAY\_RETURN in these long trades was 0.13452, which nicely exceeds the mean of 0.00976 for the other cases. In other words, the model performed well for this measure of success in the out-of-sample data.

We also see that 8.73 percent of the 1999 trade opportunities, 264 of the 3024, were short trades because their predictions lay at or below the lower threshold. The mean of DAY\_RETURN for these short trades was -0.05468, wonderfully less than the mean of 0.03536 for the other trade opportunities that were not selected as short trades. As for long trades, the model also performed well for this measure of success for short trades.

In the script file, the user set MIN CRITERION FRACTION = 0.1 to impose the restriction that at least ten percent of the trade opportunities be long trades, and another ten percent be short trades. So how can it be that only 8.73 percent of these trades were short? The answer is that any such restriction can apply only to the training data. The program must never be told anything about the test data, so it has no way of knowing how the test set will fare. In fact, it is not unusual, in rapidly changing market conditions, to have no test-set cases produce a trade, or for all cases in the test set to produce a trade. This is yet another manifestation of the non-stationarity problem.

More details are revealed in the table of results that follows. A discussion of these results follows the table.

**Target statistics at various percentages kept...**

Statistic	10	25	50	75	90
Mean target above	0.1697	0.0883	0.0487	0.0404	0.0374
N wins above	162	395	768	1143	1363
Mean win above	0.8046	0.7189	0.6709	0.6514	0.6595
Total win above	130.3	284.0	515.3	744.5	898.9
N losses above	137	347	720	1089	1312
Mean loss above	0.5774	0.6260	0.6135	0.5995	0.6076
Total loss above	79.1	217.2	441.7	652.8	797.2
Profit factor above	1.6479	1.3073	1.1666	1.1404	1.1277
Mean target below	-0.0616	-0.0113	0.0063	0.0072	0.0117
N wins below	154	377	746	1119	1329
Mean win below	0.7391	0.6848	0.6291	0.6200	0.6259
Total win below	113.8	258.2	469.3	693.7	831.9
N losses below	140	360	735	1108	1341
Mean loss below	0.6801	0.6935	0.6515	0.6409	0.6441
Total loss below	95.2	249.6	478.9	710.2	863.8
Profit factor below	1.1954	1.0341	0.9800	0.9769	0.9630

It is interesting and heartening to observe that even though this is out-of-sample data, we have the same monotonic relations among target means for both long and short trades as we had in the training set. The mean target for long trades runs from a high of 0.1697 for the highest ten percent of predictions, to a low of 0.0374 for the highest 90 percent predictions. Similarly, on the short side, the mean DAY\_RETURN for the ten percent smallest predictions is -0.0616 (which is good, because when we are short we want the market to go down). This target mean steadily rises to 0.0117 when we examine the 90 percent smallest predictions.

Just to be clear, there is no way we could in real life choose to trade only some fixed percent of the trade opportunities in the test set. We would need to be able to see into the future to know the predictions for the entire year in order to compute the threshold required to generate trades on at least a specified percentage of trade opportunities. The trading thresholds can only be computed from the training set. Still, after-the-fact examination like this can reveal a lot about the behavior and quality of the trading system.

```
MSE = 0.73187  R-squared = 0.00205  ROC area = 0.53318
Buy-and-hold profit factor=1.091  Sell-short-and-hold=0.916
Dual-thresholded outer PF = 1.352
Outer long-only PF = 1.488  Improvement Ratio = 1.364
Outer short-only PF = 1.166  Improvement Ratio = 1.272
```

These statistics were discussed earlier, so we will not dwell on them here other than to note that improvement ratios of 1.364 for long trades and 1.272 for short trades is quite respectable for a simple trading system like this.

## The Walkforward Summary

After the results for every individual fold have been reported, *TSSB* pools all OOS folds into a single set and computes various performance statistics.

---

**Walkforward is complete. Summary...**

---

**Pooled out-of-sample...**

**Target grand mean = 0.00679**

**4674 of 36732 cases (12.72%) at or  
above outer high threshold (Mean = 0.02957 versus 0.00346)**

**3379 of 36732 cases (9.20%) at or  
below outer low threshold (Mean = -0.00963 versus 0.00845)**

**MSE = 0.51079 R-squared = -0.00101 ROC area = 0.50802  
Buy-and-hold profit factor=1.026 Sell-short-and-hold=0.974  
Dual-thresholded outer PF = 1.075  
Outer long-only PF = 1.099 Improvement Ratio = 1.071  
Outer short-only PF = 1.036 Improvement Ratio = 1.063**

None of these quantities is new. They are the same items that were reported for individual folds and discussed earlier. The only difference is that these results are computed from the pooled decisions of all test (out-of-sample) folds.

There is one aspect of this pooled summary that will not concern casual users, but that may be of interest to those who want to understand the inner workings of the program. For each individual fold, both IS (training set) and OOS (test set) tables are printed that contain detailed performance statistics at each of 5 different percentiles. Such a table appeared [here](#). But no such table appears for the pooled summary. Why?

The answer is that it would not make sense. In the discussion of the table [here](#), it was noted that the table is of marginal meaning for OOS data, because the thresholds to obtain the five percentiles are computed with OOS predictions that would not be known in advance. When the data is pooled across folds, the situation becomes even more inappropriate. Suppose that due to normal market variation, the predictions for some OOS fold are unusually large. Such variation is common. Then, when we compute a threshold based on a percentile of the pooled data, the highest set will be dominated by that unusually high fold. To have the results in a table like this be so dependent on the behavior of the model in one particular fold or small subset of folds would be misleading in the extreme, so the program refrains from printing the table.

Astute readers will then wonder about the veracity of the means of the target variable above and below the long and short thresholds, such as '`Mean = 0.02957 versus 0.00346`' in the example just shown. Would these not suffer from the same problem? No. The reason is that rather than using a single threshold to cover all of the pooled OOS data, these means are based on the separate trading thresholds computed from the training data in each individual fold. Thus, the fact that these thresholds are based on training data, not OOS data, and the fact that each fold's threshold is considered separately, tells us that that these comparative means are legitimate and meaningful.

If you do not understand this explanation, don't worry. It's not crucial to correct use of *TSSB*. Just remember that the difference in means above and below the threshold printed in the pooled summary is an honest measure of the results that would have been obtained in real life had the model been used. Also remember that the table of detailed statistics is not printed for the pooled summary, and there is a good reason for this omission: it would be misleading.

# A Simple Filter System

We now advance to a prediction-model-based method for filtering the trades of an existing trading system. It is assumed that the reader has read and digested the prior chapter that presented a standalone trading system. Repeated explanations will be minimized.

The idea behind filtering is that we already have a trading system that performs fairly well, and we wish to improve its performance by executing only a superior subset of its suggested trades. One could filter a worthless trading system, such as one that is based on coin tosses, and often achieve decent results. However, there is little point in doing so. One would be better off just designing a standalone system.

This chapter will show and discuss all of the files that are necessary to implement a simple filter, although the discussion of this filter system will be limited to an overview of its operation. Extensive details of individual components will appear later in this document, and we will provide references to these details here to satisfy readers who want to occasionally flip ahead.

## The Trade File

In order to build and test filter systems, we must provide *TSSB* with a file that contains the trades produced by the existing system that we wish to filter. This is an ASCII text file in what we call the *TSSB Database Format*. This format has some requirements that will be discussed in detail [here](#). However, we will touch on the requirements here in order to orient the reader to the basic principles involved. Note that this format is a subset of universally recognized database standards, and *TSSB* database files can be easily written and read by common statistical analysis and spreadsheet programs such as Microsoft Excel® and similar packages.

Here are a few early lines from the trade file used in this example:

Date	Market	ATR65	PROFIT	SCALEDPROFIT
19881215	SPY	0.25461501	0.20000000	0.78549898
19890508	SPY	0.25753799	0.10000000	0.38829201
19890615	SPY	0.26400000	0.23000000	0.87121201
19910429	SPY	0.47338501	0.66000003	1.39421499
19910430	QQQQ	0.11830800	0.08000000	0.67620301
19910607	QQQQ	0.11230800	-0.07000000	-0.62328798
19910607	SPY	0.42815399	0.28000000	0.65397000
19910624	QQQQ	0.10753800	0.10000000	0.92989999

The first line of the file lists the fields that will appear on subsequent lines. Here, there are five fields: Date, Market, ATR65, PROFIT, and SCALEDPROFIT. These field names are separated by blanks, although commas or tabs are also legal. Case (upper or lower) is ignored because *TSSB* will automatically convert all letters to upper case. This means that the user can employ whatever capitalization scheme is most clear, and yet be confident that inconsistent capitalization will not cause problems in the program. Because this first line is just a list of names, there is no requirement that they line up with their associated columns, although of course the user can include extra spaces if, for the sake of visual appearance, the user wants them to line up.

Note that ATR65 and PROFIT will not be used in this example. The target variable will be SCALEDPROFIT, as will be seen soon. It is perfectly legal to include any number of extraneous variables in the file.

Also note that this file should be in chronological order. There are ways for *TSSB* to handle database files that are not chronological, but operation is always faster and easier if the file is ordered by date (and time, for intraday data).

We'll now examine one line, the first data line, in this file. The date is specified as YYYYMMDD, so this trade was signaled as of the close on December 15, 1988. This is the date on which all indicators are known, but before the trade is initiated. So in this example, all indicators would have been computed after the market closed on December 15, and the position would be entered the next trading day, or perhaps some time after the close in after-market trading. This timing of 'after indicators are known' as well as 'before the trade is opened' is necessary for any filtering system. We need 'after indicators are known' in order for the filtering system to have access to all of the information it needs to make a decision, and we need 'before the trade is opened' so that the filtering system can make a decision to execute or abort the trade before a position is actually opened.

Continuing on this data line in the file, we see that the trade occurred in the market SPY. The ATR65 variable (average true range over the past 65 days) as of the close of trading that day was 0.25461501 points, and the trading system

made a profit of 0.2 points. Dividing this profit by ATR65 gives a volatility-normalized profit of 0.78549898. This last quantity is the target that this example will use. Note that *TSSB* neither knows nor cares about exactly when or how the system being filtered opened its position. All it has and needs is the profit/loss of the trade and the assurance that the trade was initiated after the close of the market as of the specified date.

When designing a filter system, it is best for the filter to specialize in either long trades or short trades. Thus, if we want to develop a filter for an existing system that executes both long and short trades, we need to create two separate filters and employ the long filter when a long trade is signaled by the existing system, and employ the short filter when a short trade is signaled. In the example presented here, we are considering only long trades. This distinction will be crucial in specifying model options in the script file, and in interpreting results in the AUDIT.LOG results file.

## The Script File

The script file for this example is named SIMPLE\_FILTER.SCR. Its contents are now listed, and a discussion of each line follows.

```
READ MARKET LIST "ETF20.TXT" ;
READ MARKET HISTORIES "E:\ETF\ADDR.TXT" ;
CLEAN RAW DATA 0.65 ;
READ VARIABLE LIST "TREND_VOLATILITY.TXT" ;
APPEND DATABASE "MEANREV.DAT" ;
ScaledProfit IS PROFIT ;

MODEL FILTMOD IS GRNN [
    INPUT = [ P_TREND P_VOLATILITY ]
    OUTPUT = ScaledProfit
    MAX STEPWISE = 2
    CRITERION = LONG PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

WALK FORWARD BY YEAR 5 2007 ;
```

Here is a brief discussion of each line in the script file. Note that most of these lines were already discussed in the prior chapter covering a simple standalone trading system. Repetition will be minimized here.

```
READ MARKET LIST "ETF20.TXT" ;
```

This tells *TSSB* which markets will take part in the development. These markets are listed in the file ETF20.TXT, analogous to the file SYMBOLS.TXT in the SIMPLE\_STANDALONE.SCR script file seen in the prior chapter. The READ MARKET LIST command is discussed in detail [here](#).

```
READ MARKET HISTORIES "E:\ETF\ADDR.TXT" ;
```

The market histories are ASCII files that contain the date, optional time, market symbol, open, high, low, close, and volume. We saw this same command, though naming a different file, in the SIMPLE\_STANDALONE.SCR example in the prior chapter. The READ MARKET HISTORIES command is discussed in detail [here](#).

```
CLEAN RAW DATA 0.65 ;
```

This is an optional command that could have been used in the simple standalone system of the prior chapter, but was omitted to avoid hitting the reader with too many ideas at once. Because this command is almost always employed, we'll introduce it here. The function of the CLEAN RAW DATA command is to protect the study from wild,

likely erroneous prices in the market history files. For the entire price history of each market, the closing price of each bar is divided by the closing price of the prior bar. If this ratio, or its reciprocal, is less than the specified quantity (0.65 in this example) then this bar is flagged as erroneous data and is not used for computing any variable.

```
READ VARIABLE LIST "TREND_VOLATILITY.TXT";
```

This is the same *variable definition list* file that appeared in the simple standalone example in the prior chapter. The READ VARIABLE LIST command is discussed in detail [here](#).

```
APPEND DATABASE "MEANREV.DAT";
```

[here](#) we saw that in a filter application, the user supplies *TSSB* with an ASCII text file that contains the trades produced by the system being filtered. This file is read by means of the APPEND DATABASE command. Note that the prior command, READ VARIABLE LIST, created a database that contains the variables specified in the variable definition list file, TREND\_VOLATILITY.TXT. So at the time the program reads the trade file MEANREV.DAT, a database already exists. Thus, we need to *append* the trade file information to the existing database of internally computed indicators. *TSSB* also supports a READ DATABASE ([here](#)) command which lets the user avoid having to deal with market files and variable definition list files. However, the READ DATABASE command cannot be used when a database already exists, which is the situation in this filtering example. When a database already exists, we need to use the APPEND DATABASE command, which is discussed in detail [here](#).

```
ScaledProfit IS PROFIT;
```

The IS PROFIT command will usually be needed when the target variable (named in the OUTPUT statement) is read from a database file, as opposed to being computed internally. In this simple filter example, the target variable is ScaledProfit, the profit of each trade produced by the existing system that is to be filtered. This variable is read from the database MEANREV.DAT by the APPEND DATABASE command shown above. The reason we need the IS PROFIT command is that *TSSB* contains a safety feature to prevent the user from accidentally misinterpreting results. This safety feature suppresses printing of any performance measure that involves profit (such as profit factors) when the target variable is not actually a profit. Such targets do exist. For example, the trend of prices in the near future can be used as a target being predicted, but it is certainly not a profit. Profit factors based on slope would be meaningless.

When targets are computed internally, *TSSB* knows whether the target is indeed a profit. But when it's read from a database file the program has no way of knowing whether the target is a profit. The IS PROFIT command, described in detail [here](#), informs *TSSB* that the target is a profit so that profit-based performance measures will be printed.

**MODEL FILTMOD IS GRNN [**

The MODEL command defines the modeling methodology used by *TSSB* and allows the user to name the model produced. The simple standalone example in the prior chapter employed a linear regression (LINREG) model. Just for variety, this filter example employs a different modeling approach called the General Regression Neural Network (GRNN). This model type is described [here](#). The MODEL command is discussed in detail [here](#).

**INPUT = [ P\_TREND P\_VOLATILITY ]**

The INPUT line specifies the indicators that will be considered as candidate inputs for the model. The two indicators here, P\_TREND and P\_VOLATILITY, are the same ones that were used in the simple standalone example presented in the prior chapter. The INPUT command is discussed in detail [here](#).

**OUTPUT = ScaledProfit**

The OUTPUT line names the target variable, the true values of the quantity that the model will be asked to predict. The target here, ScaledProfit, was read by the APPEND DATABASE command that appeared above. Note that case (upper or lower) is ignored. *TSSB* converts all letters to upper case, which allows the user to employ case for clarity. The OUTPUT command is discussed in detail [here](#).

**MAX STEPWISE = 2**

The MAX STEPWISE specification tells the stepwise indicator selection algorithm in *TSSB* the maximum number of indicators that it may use. *TSSB* may elect to use fewer than this maximum. This command is discussed in detail [here](#).

**CRITERION = LONG PROFIT FACTOR**

*TSSB* allows the user to choose from among a variety of optimization criteria (sometimes called objective functions) for several aspects of model training. In the simple standalone example of the prior chapter we optimized PROFIT FACTOR, which takes into account performance for both long trades and short trades. In this filtering example, we are considering only long trades signaled by the

existing system that we wish to filter. The MEANREV.DAT file contains only long trades. Thus, it makes sense for the filter we are developing to optimize only the profit factor of long trades. Performance of this example's filter on short trades would involve taking a short position when the existing system being filtered signals a long trade! That would be silly. The CRITERION command is discussed in detail [here](#).

**MIN CRITERION FRACTION = 0.1**

This command, which we also saw in the simple standalone example of the prior chapter, specifies the minimum fraction of trade opportunities that result in actual trades on each side (long and short, counted separately). In signal filtering, a trade opportunity is a trade signaled by the system being filtered. The MIN CRITERION FRACTION command is discussed in detail [here](#).

1 ;

This closing bracket completes the model specifications begun with the MODEL command. The semicolon terminates the MODEL command.

**WALK FORWARD BY YEAR 5 2007;**

This command specifies that the trading system will be evaluated by means of a walkforward test. BY YEAR means that the testing window will encompass one calendar year, each training period will encompass the five years prior to the test window, and the first test window will be in 2007. This command is discussed in detail [here](#).

We examined the market and variable definition list files in the prior chapter, so we will not repeat the examination here.

## The Audit Log

Everything in the audit log for this simple filter example also appeared in the prior chapter when we discussed a simple standalone system. The reader might want to glance back at that chapter for a quick review, as repetition of details will be avoided here. However, because this example is a filtering system, interpretation of some performance measures is somewhat different. We will now focus on these differences.

Here is the first part of the in-sample (training set) results for the first walkforward fold:

```
-----  
Walkforward test date 2007 training 322 cases, testing 83  
-----
```

```
GRNN Model FILTMOD predicting SCALEDPROFIT  
Stepwise based on long pf with min fraction=0.100  
(Final best crit = 1.6797)
```

```
Sigma weights:  
477.578448 P_TREND
```

```
Target grand mean = 0.37588  
Outer hi thresh = 0.37587 with 36 of 322 cases  
at or above (11.18 %) Mean = 0.58157 versus 0.34999
```

```
Outer lo thresh = 0.37586 with 107 of 322 cases  
at or below (33.23 %) Mean = 0.23327 versus 0.44685
```

A detailed explanation of most of these quantities appeared in the prior chapter. The primary exception is that the prior example employed a linear regression (LINREG) model, while in the interest of variety, this example employs a General Regression Neural Network (GRNN) model. Thus, the printed model parameters are different. The GRNN utilizes sigma weights. Large values, such as the 477 seen here, indicate that there is only a slight relationship between the predictor (P\_TREND in this fold) and the target (ScaledProfit). The GRNN model, as well as interpretation of its parameters, will be discussed in detail [here](#).

One other small difference is that the prior chapter's example directly specified the indicators to be used by the model, while this example employs stepwise selection. When stepwise selection is used, the final value of the selection criterion is printed as the *Final best crit*. This is of little interest to most users, although advanced users may be interested in the obtained value, especially if different variables are used for the training target and price performance, as

described [here](#). In this example, the *Final best crit* of 1.6797 is the log of the long profit factor.

Recall that each record in the trade file MEANREV.DAT is the return of a single trade executed by the existing system being filtered. The implication is that in a filtering application such as this example, the Target Grand Mean (0.37588 here) has special meaning: it is the mean return of all of the trades signaled by the existing system being filtered. In other words, it is an indication of the performance of the existing system alone, with no filtering. It has nothing to do with gains produced by TSSB, although knowing this figure for the unfiltered system is always useful.

The means above and below the upper (long) threshold are the first indication we see of the ability of this model to effectively filter an existing system. We see that 36 of the 322 training cases lie at or above the optimal threshold, and these trades had a mean target (ScaledProfit) of 0.58157, as opposed to a mean of 0.34999 for those trades below the threshold. This is not as impressive as it sounds, though, because these are in-sample (training set) results and hence infected with training bias.

The means relative to the lower (short) threshold are of no interest whatsoever in this filtering example. As was discussed earlier in this chapter, any results pertaining to short trades would involve taking a short position when the existing system being filtered signaled a long trade. This would be absurd, so short-side results must be ignored.

This brings up a subtle but important issue. The example of this chapter filters a long system, and for this reason short-side results are meaningless. What about when we filter a short system? Do we examine long-side or short-side results? The answer is that it depends on what the target variable in the trade file represents. If the target is (possibly scaled) market moves (future market returns), a short trade scores a win when the market goes down (the target is negative). For this reason, we would ignore long-side results, and pay attention to only short-side results. On the other hand, if the target in the file is the profit of the short trade, then positive values are wins, so we examine long-side results and ignore the short side. For a system that generates only signals to enter long trades, market moves and profits are the same thing, so this issue does not arise. But for a system that generates only short trades, the profit is the negative of the market move. Thus, for short systems we need to act according to whether the trade file contains market moves or profits.

The next item in the log file is the complex table of performance statistics at various percentiles, and the final item is information on profit factors. We'll skip both of these for now because this is still training set results. We'll cover

these items in the [next section](#) when test set results are presented.

## Out-of-Sample Results for This Fold

The prior results concerned the training set (2002-2006) in the first walkforward fold. Results for the test set, 2007, now appear in the log file:

**Out-of-sample results...**

**Target grand mean = 0.23086**

**Outer hi thresh = 0.37587 with 5 of 83 cases  
at or above (6.02 %) Mean = 0.72830 versus 0.19897**

**Outer lo thresh = 0.37586 with 39 of 83 cases  
at or below (46.99 %) Mean = 0.42834 versus 0.05582**

**Target statistics at various percentages kept...**

Statistic	10	25	50	75	90
<b>Mean target above</b>	-0.0414	-0.2261	0.0128	0.1676	0.2518
<b>N wins above</b>	5	15	30	47	55
<b>Mean win above</b>	0.9991	0.7390	0.9494	0.9217	1.0011
<b>Total win above</b>	5.0	11.1	28.5	43.3	55.1
<b>N losses above</b>	3	6	12	16	20
<b>Mean loss above</b>	1.7756	2.6389	2.3285	2.0477	1.8090
<b>Total loss above</b>	5.3	15.8	27.9	32.8	36.2
<b>Profit factor above</b>	0.9378	0.7001	1.0193	1.3222	1.5220

The target mean in the test set (2007) is considerably less than the target mean in the training set (0.23086 versus 0.37588). This, of course, has nothing to do with *TSSB*. This disparity simply signifies the fact that the system being filtered had much lower mean profit per trade in 2007 than it did in the 2002-2006 training period.

There were 83 cases (trades signaled by the system being filtered) in 2007, of which 5 had predictions that equaled or exceeded the threshold computed during training. The mean profit of these trades was 0.72830, while the mean of the 83-5=78 trades signaled by the existing system but rejected by this filter was just 0.19897. This is impressive performance, considering that this is out-of-sample data. As discussed earlier, results relative to the low (short) threshold are meaningless, because we are filtering a long system.

The table of detailed performance measures reveals something interesting that should temper our enthusiasm over the results that are relative to the optimal training-set threshold. We just saw that when we kept the 5 of 83 (6.02 percent)

of trades having the highest predicted outcomes, we had an impressive mean profit of 0.72830. But this table shows that if we kept just a few more, 10 percent, we would actually incur a loss (-0.0414)! Looking further down the first column, we see that these 8 cases (10 percent of 83) had 5 wins totaling 5.0, and 3 losses totaling -5.3, for a net loss. A reasonable explanation would be that one of these 8 trades was an unusually large loss. Later in this tutorial, [here](#), we will present a graphic capability built into *TSSB* that will let us take a close look at the distribution of wins and losses above and below any threshold.

Finally, the test-set model and profit factor breakdown is as follows:

MSE = 2.30199    R-squared = -0.00922    ROC area = 0.37374

Buy-and-hold profit factor = 1.507  
Sell-short-and-hold = 0.664

Dual-thresholded outer PF = 0.510  
Outer long-only PF = 30.244    Improvement Ratio = 20.068  
Outer short-only PF = 0.371    Improvement Ratio = 0.559

The results just shown present an excellent example of seemingly anomalous but common behavior in financial modeling. R-squared is negative, indicating that overall, the model did worse at predicting trade outcomes than simply guessing the mean trade would have done. Yet despite this, the model did an excellent job of filtering. In particular, the *Buy-and-hold profit factor* of 1.507 is the profit factor that would be obtained by trading the existing system exactly according to its signals, with no filtering. With filtering, we obtain a profit factor of 30.244, an improvement by a factor of 20.068! (As has been mentioned several times, figures involving short trades are meaningless because we are filtering a long system. For this reason, the dual-thresholded profit factor, which includes both long and short trades, is also meaningless.)

So how can we have such phenomenal results, in the out-of-sample set no less, and yet still have a negative R-squared? The performance table shown on the prior page provides a clear illustration of the answer. We obtained a wonderful mean profit per trade when we used a threshold that kept about six percent of the signaled trades, but the mean profit per trade turned negative when we kept ten percent of the trades. A very common effect in financial applications is that most or all of the useful predictive information is out in the tails, the extreme values of indicators and predictions. A good training algorithm will find and capitalize on this information. Meanwhile, the model will perform terribly everywhere else, so badly that its overall performance as measured by R-squared will be tiny or even negative.

## The Walkforward Summary

We conclude this discussion of a simple filter with a look at the walkforward summary. The contents of such a summary were discussed in depth in the prior chapter. Here we will focus on only those aspects of the summary that specially pertain to signal filtering.

```
-----  
Walkforward is complete.  Summary...  
-----
```

Pooled out-of-sample...

Target grand mean = 0.37774

28 of 236 cases (11.86%) at or above outer high threshold  
(Mean = 0.76369 versus 0.32579)

56 of 236 cases (23.73%) at or below outer low threshold  
(Mean = 0.42975 versus 0.36156)

MSE = 1.58079 R-squared = -0.00510 ROC area = 0.35389

Buy-and-hold profit factor = 2.176

Sell-short-and-hold = 0.460

Dual-thresholded outer PF = 0.934

Outer long-only PF = 13.975 Improvement Ratio = 6.423

Outer short-only PF = 0.378 Improvement Ratio = 0.823

After pooling all test periods into one set, the total number of cases (trades signaled by the system being filtered) is 236. Of these, 28 had predictions in the test period that equaled or exceeded the threshold computed in the corresponding training period. The mean target (ScaledProfit) for these select trades was 0.76369, which is more than twice the mean for the trades that the *TSSB* filter rejected.

As we saw in the examination of the first fold, we have a negative R-squared despite having excellent performance as a trade filter. The *Buy-and-hold profit factor* of 2.176 is what the original, unfiltered system obtained across the pooled test period. By accepting only those trades whose predicted value equaled or exceeded the trained optimal value, we obtained a profit factor of 13.975. This greatly reduces the number of trades, as we kept only 28 of 236 signaled trades. Still, this degree of performance is exemplary.

Finally, note that if keeping so few trades is not an option, it is easy to increase the number of trades kept. The following line appeared in the model definition part of the script file:

**MIN CRITERION FRACTION = 0.1**

This tells the training algorithm that when it sets the optimal threshold for accepting or rejecting signaled trades, it must set the threshold such that at least ten percent (0.1) of the trades in the training set are kept. If we raise this to a larger quantity, the threshold will be adjusted accordingly. More trades will be kept in the training set, and almost certainly more trades will also be kept when the model is tested or put to actual use. But be warned... we must never forget the fundamental principle that the best predictive information is almost always in extreme values. If we raise the MINIMUM CRITERION FRACTION too much, we will be accepting a large number of cases (signaled trades) whose predictions are of little value. Performance will suffer, probably a lot.

# Common Initial Commands

In most *TSSB* applications, the first step is to read the market data and compute variables to build a database of indicators and targets. This may be done with every execution of the development script, or it may be done just once, saving the resultant database, and then reading the saved database for subsequent executions. Saving and restoring databases will be discussed on Pages [here](#) and [here](#), respectively. This chapter will discuss reading market files, selecting markets and time periods, and reading variable definition list files. The emphasis of this chapter will be the specific commands used to perform these operations, and the order in which the operations are performed.

## Market Price Histories and Variables

*TSSB* references two types of data: *market price histories* and *variables*. *Market price histories* are the price and (usually) volume information for financial markets. The price generally includes the open, high, low, and close of each bar, although it is perfectly legal for all four of these quantities to be equal if the market information is ticks or some other quantity that the user wants to use as if it were a market.

The other data type, *variables*, comprise quantities that are computed from market price histories, volume, or other financial series. Variables may be things such as measures of trend or volatility, or more sophisticated quantities. They are stored in the *database*, and they may be used as indicators or targets. The database of variables may be written to or read from a text file to avoid having to recompute them every time a study is performed. This text file of variables may also be used to transfer data to and from third-party programs such as Excel and statistical packages.

*TSSB* has built-in functions that can operate on market price histories and variables. It contains a library of over 100 predefined indicator and target families from which the user can create variables from market histories. This topic will be discussed [here](#). The program also allows the user to apply built-in functions and transformations, including fully general arithmetic scalar and vector operations, to both market price histories and variables. This is so even for variables created by other programs and read in as a database. These operations, called *expression transforms*, are discussed [here](#).

In summary, the terminology essential to understanding this chapter is as follows:

- *Market price histories* are the price and volume information for markets.
- *Variables* are quantities computed from market prices histories. They reside in the *database* and may be used as *indicators* and *targets*.

## Quick Reference to Initial Commands

We begin with a quick reference guide to the commands that will be presented in this chapter. Every command here is related in some way to creation of the database of indicators and targets from market files and a variable definition list file. This list is presented in an order which would be legal and common in *TSSB*. However, some other orderings are also legitimate. See the notes for individual commands regarding order.

**INTRADAY BY MINUTE** - Specifies that all market files are intraday, with time resolved to minutes. If used, this must precede the READ MARKET HISTORIES command. See [here](#).

**INTRADAY BY SECOND** - Specifies that all market files are intraday, with time resolved to seconds. If used, this must precede the READ MARKET HISTORIES command. See [here](#).

**MARKET DATE FORMAT YYMMDD** - Alters the date format in market files from the default YYYYMMDD to YYMMDD. In this case, the legal date range is 1920 through 2019, inclusive. If used, this must precede the READ MARKET HISTORIES command. See [here](#).

**MARKET DATE FORMAT M\_D\_YYYY** - Alters the date format in market files from the default YYYYMMDD to Month/Day/Year. If used, this must precede the READ MARKET HISTORIES command. See [here](#).

**MARKET DATE FORMAT AUTOMATIC** - Automatically, for each market file, determines whether the date format is YYYYMMDD or Month/Day/Year. If used, this must precede the READ MARKET HISTORIES command. See [here](#).

**REMOVE ZERO VOLUME** - Removes all market history records having zero volume. If used, this must precede the READ MARKET HISTORIES command. See [here](#).

**READ MARKET LIST** - Reads a text file that lists the markets that will be used in the study. This must precede the READ MARKET HISTORIES command. See [here](#).

**READ MARKET HISTORIES** - Reads all of the market history files that were listed in the file specified with the READ MARKET LIST command. See [here](#).

**MARKET SCAN** - Scans one or all markets for unusual price jumps that may indicate errors in the market file. This optional command must not appear until after the market(s) have been read with the READ MARKET HISTORIES command. See [here](#).

The preceding commands are all associated with reading one or more market history files into TSSB. The following commands are associated with using these market histories to compute a database of indicator and/or target variables.

**RETAIN YEARS** - Specifies that only a specified range of years be read from the market file(s) and/or computed for the database. If used, this must precede the READ VARIABLE LIST command. Its position before or after the READ MARKET HISTORIES command is significant, and in most cases it should follow this command. See [here](#).

**RETAIN MOD** - Specifies that every  $n$ 'th database date (and time, if intraday) be kept. This is useful for temporarily reducing the size of the database to speed operation during early development phases. (The term *MOD* comes from the algebraic operation of *modulo arithmetic*.) If used, this must precede the READ VARIABLE LIST command. See [here](#).

**CLEAN RAW DATA** - Decrees that suspiciously large market price jumps be flagged as invalid data so that no variable will be computed using this data. If used, this must precede the READ VARIABLE LIST command. See [here](#).

**INDEX IS** - Names a particular market as an *index*. Index markets have special uses when computing indicators. If used, this must precede the READ VARIABLE LIST command. See [here](#).

**READ VARIABLE LIST** - Reads the variable definition list file and builds a database of all variables. See [here](#).

**OUTLIER SCAN** - Scans one or all variables in the computed database for outliers that may indicate errors in the market file or unstable market conditions. This command must not appear until after the database of variables has been computed with the READ VARIABLE LIST command. See [here](#).

**DESCRIBE** - Prints a statistical summary of a variable. See [here](#).

**CROSS MARKET AD** - Performs an Anderson-Darling test for conformity of a variable across multiple markets. See [here](#).

**CROSS MARKET KL** - Performs a Kullbach-Liebler test for conformity of a variable across multiple markets. See [here](#).

**CROSS MARKET IQ** - Performs an interquartile range overlap test for conformity of a variable across multiple markets. See [here](#).

**STATIONARITY** - Performs a suite of tests to identify nonstationary behavior

in a variable. See [here](#).

## Detailed Descriptions

This section provides more detailed descriptions of the commands that read market history files and create a database of indicators and targets.

### INTRADAY BY MINUTE

The format for this command is as follows:

```
INTRADAY BY MINUTE ;
```

By default, there is no time field in a market history file. Each record is for a single day, so only the date appears. However, if the INTRADAY BY MINUTE command appears before the READ MARKET HISTORIES command, it is assumed that in each market record the date is followed by the time of each bar. (If the INTRADAY BYMINUTE command appears *after* the READ MARKET HISTORIES command, it is ignored and has no effect.) This time is encoded as HHMM in 24-hour format. Here are a few typical records in an intraday market history file that resolves to the closest minute. The remaining fields on each record are the price open, high, low, and close, and finally the volume.

```
20110301 0830 16.90 16.94 16.21 16.23 297967  
20110301 0831 16.20 16.43 16.13 16.18 201850  
20110301 0832 16.37 16.77 16.36 16.63 191486  
20110301 0833 16.77 16.80 16.37 16.58 186114  
20110301 0834 16.58 16.75 16.16 16.17 119222
```

### INTRADAY BY SECOND

This is identical to the INTRADAY BY MINUTE option just described, except that the time field includes seconds, as HHMMSS. A typical intraday market history file that resolves to the second might have records that look like the following lines:

```
20110301 083000 16.90 16.94 16.21 16.23 297967  
20110301 083001 16.20 16.43 16.13 16.18 201850  
20110301 083002 16.37 16.77 16.36 16.63 191486  
20110301 083003 16.77 16.80 16.37 16.58 186114  
20110301 083004 16.58 16.75 16.16 16.17 119222
```

## **MARKET DATE FORMAT YYMMDD**

The format for this command is as follows:

```
MARKET DATE FORMAT YYMMDD ;
```

By default, the date in a market history file contains the full 4-digit year. In other words, the date is expressed as YYYYMMDD. If the file expresses the date using only two digits as YYMMDD, then the TWO DIGIT YEAR command must appear before the READ MARKET HISTORIES command. In this case, the legal date range is 1920 through 2019, inclusive. A file that uses two digit years might contain records that look like those shown below. These records are for the year 2011.

```
110301 16.90 16.94 16.21 16.23 297967  
110301 16.20 16.43 16.13 16.18 201850  
110301 16.37 16.77 16.36 16.63 191486  
110301 16.77 16.80 16.37 16.58 186114  
110301 16.58 16.75 16.16 16.17 119222
```

## **MARKET DATE FORMAT M\_D\_YYYY**

The format for this command is as follows:

```
MARKET DATE FORMAT M_D_YYYY ;
```

This changes the date format for all market files to the usual American standard. The month and the day can be one or two digits. The year must be four digits.

```
3/1/2011 16.90 16.94 16.21 16.23 297967
```

## **MARKET DATE FORMAT AUTOMATIC**

The format for this command is as follows:

```
MARKET DATE FORMAT AUTOMATIC ;
```

This allows different market files to have different date formats. The first record of the file will be read and the date format determined from that. At this time, only YYYYMMDD and M\_D\_YYYY are allowed.

## REMOVE ZERO VOLUME

Sometimes the market history supplied by a vendor may include records that have a volume of zero. For example, on 9/27/1985 there was a severe weather situation that interfered with trading in New York markets. Many issues did not trade at all that day. Nonetheless, since this was a legal trading day and some markets did trade, instead of simply deleting this date from market histories, some vendors include it but set the volume to zero. This has a minor impact on most operations. However, it severely impacts the TRAIN PERMUTED operation ([here](#)). For this reason, the following command is provided:

```
REMOVE ZERO VOLUME ;
```

This command causes market records having zero volume to be skipped as their history files are read. (In Version 1 of *TSSB*, zero-volume records are always kept.) If used, this command must precede the READ MARKET HISTORIES command. This command is required if a TRAIN PERMUTED command appears in the script file, even if the market(s) read do not contain any zero-volume records.

## READ MARKET LIST

The user may wish to repeat the same study on several different collections of markets. The easiest way to facilitate this versatility is to allow the user to list all of the desired markets in a single text file, and then reference this text file in the script file that controls the study. Such a file is, not surprisingly, called a *market list file*. So, for example, the user might have one market list file that lists the components of the Dow Jones composite index, another that lists the markets in the S&P 500, another that lists numerous bank stocks, and so forth. The lines in a market list file might look like this:

```
AA  
DELL  
DOW  
GE  
...
```

The READ MARKET LIST command specifies the name of the market list file, enclosed in double quotes (""). The file name may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will assume that the file is in the current working directory as defined by the Windows operating system. The format of this command is as follows:

```
READ MARKET LIST "FileName" ;
```

By listing the markets in a single file, the user can easily repeat the study in different sets of markets by changing only the name of the market list file in the READ MARKET LIST command. For example, the following command would perform the study on a set of markets that is made up of utilities on the east coast (assuming that the user named the file intelligently!):

```
READ MARKET LIST "EastCoastUtilities.txt" ;
```

## READ MARKET HISTORIES

All of the commands prior to this one are preparatory; they tell *TSSB* about the format of the market history file(s) and specify which files are to be read. It is the READ MARKET HISTORIES command that actually reads the files. For this reason, if any of the preceding commands appear *after* the READ MARKET HISTORIES command, they are ignored; it's too late.

This command must name one market history file in the directory in which all history files reside. The file need not be in the market list. It only needs to be present in the directory. The format of this command is as follows:

```
READ MARKET HISTORIES "AnyMarketFile" ;
```

The file name is enclosed in double quotes (""). The file name may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will assume that the file is in the current working directory as defined by the Windows operating system.

Why name a single file instead of simply naming the directory where the files are located? There are several reasons. First, this is a convenient way to simultaneously specify the extension. Otherwise, the user would have to use a separate command to tell *TSSB* what extension to use. So naming a file simplifies things for the user and makes the script file one line shorter. Also, it is a bit of insurance against carelessness by the user. By imposing the requirement that the user name a file that exists in the directory, *TSSB* is able to generate a clear error message for the user in case the directory or extension is not correct. In other words, it can display the file name, including path and extension, so that it's obvious where the program is looking and what it's looking for.

*TSSB* keeps the same path and extension for all market files, and just substitutes the market names as listed in the market list file. For example, suppose the

market list file contains three markets as follows:

```
IBM  
T  
BOL
```

Suppose also that the following READ MARKET HISTORIES command is used:

```
READ MARKET HISTORIES  "\MyMarkets\CIT.TXT"  ;
```

In this case, *TSSB* will look in the directory **\MyMarkets** for all market history files, and it will use the **.TXT** extension for all of them. In particular, it will attempt to read the following three files:

```
\MyMarkets\IBM.TXT  
\MyMarkets\T.TXT  
\MyMarkets\BOL.TXT
```

## MARKET SCAN

This command scans one or all markets for unusual price jumps that may indicate errors in the market file. This optional command must not appear until after the market(s) have been read with the READ MARKET HISTORIES command.

If a market is named in this command, the scan will be done for only that one market. If no market is named, all markets will be scanned. For example, the following command will scan only IBM:

```
MARKET SCAN  IBM  ;
```

For each bar, the scanning algorithm computes the high minus the low and the absolute difference between the open and the prior day's close. The greater of these two quantities is divided by the lesser of today's close and yesterday's close, and the quotient is multiplied by 100 to express the difference as a percent.

If only one market is scanned, the largest 20 differences, along with their dates, are listed in descending order. If all markets are scanned simultaneously, each market's 20 worst dates are listed in descending order, as is the case for one market. However, the market results are sorted according to the magnitude of each market's worst gap. Thus, results for the market having the single worst

gap appears first, followed by the market having the second-worst gap, and so forth. This command can also be accessed through the menu system.

The preceding commands are all associated with reading one or more market history files into *TSSB*. The following commands are associated with using these market histories to compute a database of indicator and/or target variables.

## RETAIN YEARS

It is often the case that the user may wish to perform a study on only a subset of the complete market history. Perhaps the user wants to implement a simple virgin-data development strategy, in which the most recent year or few years are held back from development of the trading system. Then later, after a trading system is in hand, it is tested on the recent data that was excluded from the development phase.

More commonly, the user simply wants the study to run quickly while models and options are decided upon. It can be frustrating to tweak a study when each run requires several hours of computer time! In this case, limiting the market history to a short time period can greatly speed the tweaking process. Once the models and options are decided upon, the entire historical dataset can be employed for the final study.

The format of the RETAIN YEARS command is as follows:

```
RETAIN YEARS FirstYear THROUGH LastYear ;
```

The specified years are inclusive. Thus, the following command would retain the years 1997, 1998, and 1999:

```
RETAIN YEARS 1997 THROUGH 1999 ;
```

The position of this command in the script file has a subtle but important effect. It must always precede the READ VARIABLE LIST command if variables are computed internally, or the READ DATABASE command (described [here](#)) if variables are read from an external file. If the RETAIN YEARS command *follows* these commands it will be ignored; it's too late, because the database is already created.

But the important distinction arises according to the position of the RETAIN YEARS command relative to the READ MARKET HISTORIES command. In the vast majority of applications, the RETAIN YEARS command should *follow*

the READ MARKET HISTORIES command. This way, the entire history of every market will be read, which allows the internal variables to be based on the full extent of the available data. If instead the RETAIN YEARS command *precedes* the READ MARKET HISTORIES command, then only the specified years of market history will be read. This will deny some otherwise available data to *TSSB* when it computes variables.

For example, suppose that one of the indicators the user wishes to create is a linear trend that looks back 100 days. Suppose also that the user has data back to 1990 and wants the study to run from 1992 through 1999. The following command is used:

```
RETAIN YEARS 1992 THROUGH 1999 ;
```

If this command appears *after* the READ MARKET HISTORIES command and (as required) before the READ VARIABLE LIST command, when the value of the 100-day trend is computed for the first trading day of 1992, which is where the database will begin, it will have two years of prior data to work with. This is because the market histories were read *before* the RETAIN YEARS command appeared. These two years (1990 and 1991) are obviously more than enough to look back 100 days to compute the trend.

On the other hand, suppose the RETAIN YEARS command appears *before* the READ MARKET HISTORIES command. Now, all market history prior to 1992 is rejected. Thus, when *TSSB* tries to compute the value of the 100-day trend indicator for the first trading day of 1992, it will have no history to look at. It will have to let the first 100 days of 1992 pass before it can begin computing the trend indicator.

The same effect applies to the ending date and targets, which look into the future. So it should be obvious that when a variable definition list contains a variety of lookback and look-ahead distances, the results will be complex. The bottom line is that in nearly all cases, the RETAIN YEARS command should appear *after* the READ MARKET HISTORIES command so that all available history is available to *TSSB*.

The only exception might be if one or more market history files go back much, much further than is needed, and the user wants to save computer memory by reading only part of the history. In this case, it is fine to use *two* RETAIN YEARS commands, one prior to the READ MARKET HISTORIES, and one after. For example, suppose the most distant lookback among indicators is two years, and the most distant look-ahead is one year. Then we might do this:

```
RETAIN YEARS 1992 THROUGH 2003 ;
READ MARKET HISTORIES  "\MyMarkets\CIT.TXT" ;
RETAIN YEARS 1994 THROUGH 2002 ;
```

But please note that this degree of complexity is almost never necessary, let alone desirable. Unless the market history files are gigantic and computer memory is limited, just put the RETAIN YEARS command after the READ MARKET HISTORIES command. Keep things simple.

## RETAIN MOD

This command directs that the database be decimated by keeping one date, skipping several, keeping one, skipping several, and so forth. This is useful for temporarily reducing the size of the database to speed operation during early development phases. If used, this must precede the READ VARIABLE LIST or READ DATABASE ([here](#)) command. The syntax of this command is as follows:

```
RETAIN MOD  Divisor = Offset ;
```

In this command, *Divisor* is the factor by which the size of the database is reduced, and *Offset* determines where the kept records begin relative to the first record in the database. *Offset* must be greater than or equal to zero and less than *Divisor*. The offset parameter facilitates testing consistency of study results.

For example, suppose we have two markets. Cases in the database may look like the following lines. In these lines, the ellipsis (...) represents the indicators and targets for the given date and market.

```
19870103 IBM ...
19870103 BOL ...
19870104 IBM ...
19870104 BOL ...
19870105 IBM ...
19870105 BOL ...
19870106 IBM ...
19870106 BOL ...
19870107 IBM ...
19870107 BOL ...
19870108 IBM ...
19870108 BOL ...
19870109 IBM ...
19870109 BOL ...
```

Suppose we had included the following line in the script file before the READ

VARIABLE LIST command:

```
RETAIN MOD 3 = 0 ;
```

This command specifies that we will keep every third date (divisor=3) and begin with the first (offset=0). So the resultant database will look like this:

```
19870103 IBM ...
19870103 BOL ...
19870106 IBM ...
19870106 BOL ...
19870109 IBM ...
19870109 BOL ...
```

Now suppose we had instead included the following line in the script file before the READ VARIABLE LIST command:

```
RETAIN MOD 3 = 1 ;
```

This command specifies that we will keep every third date (divisor=3) and begin with the second (offset=1). So the resultant database will look like this:

```
19870104 IBM ...
19870104 BOL ...
19870107 IBM ...
19870107 BOL ...
```

The obvious use for the RETAIN MOD command is to shrink the database so that studies execute quickly while models and options are being finalized by the user. But there is another interesting application of this command. Suppose we are developing models that look just one day ahead for the target, a common operation. As is well known, markets have daily changes that have very low, almost (though not quite!) negligible serial correlation. We could use the first of the following two commands in a study, and then repeat the study with the second.

```
RETAIN MOD 2 = 0 ;
RETAIN MOD 2 = 1 ;
```

The two studies would be trained and tested on datasets that cover the same total time period, and hence experience roughly the same large-scale movements. However, the targets in these two datasets would be almost independent. If the user's development strategy is stable, similar results should be obtained in both cases. If the results are wildly different, the user should be suspicious of his or her methodology.

If basing the datasets on adjacent days makes the user a little nervous because of the fear of small but significant serial correlation in the targets, the distance could easily be expanded. The following two options, one for one study and the other for a separate study, would cut the size of the usable dataset in half compared to the example just shown, but it would space the cases two days apart instead of just one.

```
RETAIN MOD 4 = 0 ;
RETAIN MOD 4 = 2 ;
```

Finally, note that there is nothing special about requiring the target to look ahead just one day, other than the fact that we can make efficient use of the historical data. Suppose the target looks ahead five days. Then we could use the following commands:

```
RETAIN MOD 10 = 0 ;
RETAIN MOD 10 = 5 ;
```

The above commands, used in separate studies, would offset the dates by five days, which is the lookahead distance, and hence make the datasets nearly independent.

## CLEAN RAW DATA

Market histories are rarely perfect. Spurious price jumps are more common than most vendors are willing to admit. Also, some users may wish to disqualify price jumps that, while correct, are so unusual that they may be considered atypical and hence should be ignored. The CLEAN RAW DATA command facilitates removal of records with unusually large price changes from the market history file(s). If used, this must precede the READ VARIABLE LIST command.

The format for this command is as follows:

```
CLEAN RAW DATA Fraction ;
```

Each bar in each market's history is checked. If the ratio of a day's close to the prior day's close is less than the specified fraction (0-1), or greater than the specified fraction's reciprocal, the current day for the suspect market is flagged as erroneous. This date in this market will not be used to compute any variable (indicator or target) in the database.

This can have more far-reaching implications than is immediately obvious. Most

indicators look back in time for a considerable distance. Even seemingly short lookback indicators may use a long stretch of history for normalization. For example, the LINEAR PER ATR indicator (discussed [here](#)) normally uses extensive history for normalization. The user may specify that this indicator reflect trend over the most recent ten days, but most users would normalize the trend with ATR (Average True Range) computed over a much longer time frame, perhaps as long as a year. In this case, if the CLEAN RAW DATA command results in a date being flagged as invalid, the indicator will be omitted for the next year! Even one piece of bad market history in the lookback period of an indicator, or the look-ahead period of a target, will suppress computation of that value. For this reason, large (strict) values of this parameter can result in ridiculously few cases in the database of indicators and targets. Only rarely would values as large as 0.7 be appropriate, and 0.4 or so would usually be sufficient.

## INDEX

The use of index markets will be discussed in detail [here](#). Here we provide an introduction. The basic idea is that many sets of markets are officially characterized by a weighted average of their component markets, and we can make use of this fact. Perhaps the most famous index is the Dow Jones Industrial Average. Another common index ‘market’ is the S&P 100, with the symbol OEX. TSSB treats index markets in the same way as component markets. The user does not have to declare an index market as an INDEX unless any of the special uses of indices are specified in the variable list file. These special uses are briefly discussed below and will be discussed in detail [here](#).

Up to 16 markets may be declared as index markets. However, it is very rare that the user would employ more than one index, so this discussion will focus on application of a single index. Multiple index markets will be discussed [here](#). The syntax for declaring a market as an index is as follows:

```
INDEX IS MarketName ;
```

So, for example, we would declare OEX as an index as follows:

```
INDEX IS OEX ;
```

Any index declarations must precede the READ VARIABLE LIST command because index markets will be referenced in the definitions of indicators.

There are two uses for index markets when computing indicators from the built-

in *TSSB* library:

- 1) An indicator can be based strictly on the index market rather than individual component markets. Thus, on any given date (and time, if intraday), the value of this indicator will be the same for all markets. This provides the predictive model with the ‘average’ behavior of all markets as opposed to the behavior of a specific market.
- 2) An indicator can be based on the departure of a specific market from the equivalent behavior of the index market. For example, we may be interested in the value of the RSI indicator in IBM relative to the RSI in the S&P 500 index. This tells the predictive model if a particular market is ‘out of step’ with the average behavior of the other markets in the index.

Both of these uses will be discussed in detail [here](#).

## READ VARIABLE LIST

The commands discussed prior to this one specify options for computation of built-in indicators and targets. The READ VARIABLE LIST command does the work: It reads the variable definition list file and builds the database of variables (indicators and targets) according to the definitions specified by the user in that file.

The READ VARIABLE LIST command specifies the name of the variable list file, enclosed in double quotes (""). The file name may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will assume that the file is in the current working directory as defined by the Windows operating system. The format of this command is as follows:

```
READ VARIABLE LIST "FileName" ;
```

The contents of the variable list file are quite complex, so an entire chapter will be devoted to a detailed description of variable definitions. This chapter begins [here](#). For the moment, understand that each line of this text file defines a single variable that is available in *TSSB*’s built-in library. Each defined variable may then be used as an indicator or target in subsequent studies.

## OUTLIER SCAN

After the database of variables has been computed by means of the READ VARIABLE LIST command, or read in via the READ DATABASE command ([here](#)), the user may be interested in assessing the validity of the cases in this database. The OUTLIER SCAN command scans one or all variables in the database for outliers that may indicate errors in the market file or unstable market conditions.

If the user names a single variable in this command, only the named variable will be scanned. If no variable is named, all variables are scanned. The format for these two options is as follows:

```
OUTLIER SCAN ;  
OUTLIER SCAN FOR VariableName ;
```

The following information is printed to the AUDIT.LOG file (separately for each variable if all are scanned):

- Name of the variable
- Minimum value, and the market in which the minimum occurred
- Maximum value, and the market in which the maximum occurred
- Interquartile range
- Ratio of the range (maximum minus minimum) to the interquartile range. Large values of this ratio indicate outliers.
- Relative entropy, which ranges from zero (worthless) to one (max possible)

If all database variables are scanned, at the end of the report they will be listed sorted from worst to best, once by ratio and once by relative entropy.

The ratio of the range to the interquartile range is an extremely useful measure of the degree to which a variable contains extreme values. The interquartile range is the difference between the 75'th percentile and the 25'th percentile. It is a stable and accurate measure of the ‘spread’ of the variable because it examines only the central (least extreme) fifty percent of the distribution. The range is the difference between the largest and the smallest values of the variable. If a case contains an unusually large or unusually small value of this variable, the range will be large, and hence the ratio of the range to the interquartile range will also be large.

One nice property of this ratio as a measure of the degree to which one or more extreme values are present is that it is immune to scaling and offset. In other words, if one were to rescale a variable by multiplying it by a constant, and/or by adding a constant, the ratio of the range to the interquartile range will not change. This scaling immunity is an excellent property.

Entropy is much more difficult to explain and justify intuitively. Roughly speaking, entropy measures how well a variable is ‘spread around’ its range. Still roughly speaking, entropy is an upper bound on the amount of information that a variable can contain. An ideal indicator will usually have its values scattered equally throughout its range, resulting in high entropy. An indicator whose values lie in one or a few narrow clumps will have low entropy and probably contain little useful predictive information. Raw entropy is not immune to transformations, so the value printed here is relative entropy, which ranges from zero to one.

Entropy does not paint a complete picture, because it may be that some variable contains little information, but all of the information it does contain is useful for predicting a target. Conversely, another variable may contain an enormous amount of information, but this abundant information may be useless for predicting the particular target we are interested in. Nonetheless, entropy is a decent tool for assessing the potential utility of an indicator. At a minimum, unusually low entropy may be used as a red flag that a variable needs refinement.

## DESCRIBE

The DESCRIBE command can be used to obtain numerous statistics about a single variable. If multiple markets are present, the user can choose whether to separate results by market or pool all markets into a single collection. The format of the command for these two options is as follows:

```
DESCRIBE VariableName IN MarketName ;  
DESCRIBE VariableName ;
```

The following statistics are printed:

- Mean
- Standard error of the mean
- t-score for the mean
- Number of cases
- Variance
- Standard deviation
- Skewness
- Standard error of the skewness
- Kurtosis
- Standard error of the kurtosis
- Median
- Interquartile range

25'th and 75' percentiles  
Range  
Minimum and maximum values  
Range divided by interquartile range  
Relative entropy and number of bins used to compute this entropy

## CROSS MARKET AD

This test, officially called the *Anderson-Darling test*, would be employed only when the application is using more than one market. When the user is trading multiple markets with a single model-based trading system, it is crucial that every indicator and target have at least approximately the same distribution in every market. Suppose some indicator ranges from -40 to 10 in one market, and this same variable ranges from 15 to 70 in another market. This can easily happen if variables do not properly compensate for natural variations in market behavior such as extended trends and volatility that impact markets differently. In such a situation, no model can do a good job of predicting both markets. Sometimes the model will be good in one market and poor in the other. More often, the model will be useless in both. The CROSS MARKET AD command performs a sophisticated test of how well variables have similar distributions in all markets.

The user may choose to perform this test on just one variable, or perform it on all variables in the database. The format of this command for these two options is as follows:

```
CROSS MARKET AD FOR VariableName ;  
CROSS MARKET AD ;
```

All markets are pooled to produce a single ‘generic’ distribution of a variable. Then the distribution of each individual market is tested for equality with the pooled distribution. Anderson-Darling statistics and their associated p-values are printed twice, once ordered by the appearance of markets in the Market List File, and a second time sorted from best fit to worst.

These p-values are for testing the null hypothesis that the distribution of a variable in a given market is equal to the pooled distribution of this variable. Thus, low p-values indicate the distribution for the market in question does not conform to the pooled distribution. This is undesirable. However, when numerous cases are present (the usual situation), these p-values, as well as the A-D statistic itself, can be excessively sensitive to differences in the distributions. A tiny p-value does not necessarily imply that discrepancies will

be problematic, only that it is unlikely that if the distributions were identical we would have seen a p-value as small as that observed. When there are numerous cases, all it takes is a tiny, harmless discrepancy in the distributions to produce a very significant p-value. Thus, the best use of this test is for revealing the *worst* performers (lowest p-values). These variables/markets should be subjected to visual examination by histograms or other tests.

Note that the number of lines printed by this test in the audit log is proportional to the product of the number of variables and the number of markets. This output can become large quickly.

## CROSS MARKET KL

This test, officially known as the *Kullback-Liebler* test, is similar to the Anderson-Darling test in that it tests the degree to which the distribution of a variable is similar across multiple markets. However, unlike the A-D test, the number of cases in the dataset has no impact on the test statistic. On the other hand, the K-L test is no more intuitive in its result than the A-D test. Therefore, one should not judge the values of the K-L test statistic in isolation. Instead, one should use this statistic to identify the most problematic variables and markets so that they can be given additional attention. As with the *AD* test, small p-values indicate potential problems, and the variables and markets having the smallest p-values should be given close inspection.

The user may choose to perform this test on just one variable, or perform it on all variables in the database. The format of this command for these two options is as follows:

```
CROSS MARKET KL FOR VariableName ;  
CROSS MARKET KL ;
```

## CROSS MARKET IQ

Like the CROSS MARKET AD and CROSS MARKET KL tests, the CROSS MARKET IQ test checks the degree to which the distribution of a variable is similar across multiple markets. The user may choose to perform this test on just one variable, or perform it on all variables in the database. The format of this command for these two options is as follows:

```
CROSS MARKET IQ FOR VariableName ;  
CROSS MARKET IQ ;
```

In this test, IQ stands for interquartile range. The IQ test roughly measures the degree to which the interquartile range of each tested market overlaps the interquartile range of the pooled data. It ranges from zero, meaning that the interquartile ranges are completely disjoint (no overlap at all) to one, meaning that the interquartile ranges overlap completely.

The IQ test has two nice properties. Like the KL test but unlike the AD test, the IQ test is not sensitive to the number of cases in the dataset. Unlike both of these earlier tests, the IQ test is strongly intuitive. The test statistic, defined above, has a meaning that is easy to grasp. Thus, one can judge each individual result (for a variable and a market) by the numeric value of the test statistic.

Unfortunately, the IQ test does have one disadvantage compared to the A-D and K-L tests: it is not very sensitive to mismatches of distributions in the tails. The IQ test looks only at the central half of the data. It is conceivable that a market will closely match the pooled distribution in this central half, resulting in an excellent IQ score, while being very different in one or both tails. Nonetheless, this phenomenon is not common. For this reason, the IQ test is probably the best test to perform if one is going to rely on just one cross-market conformity test. On the other hand, it is advisable to perform all three tests whenever possible, because each has its own strengths and weaknesses. A variable/market pair that scores relatively poorly on even one of the three tests should invite close scrutiny.

## STATIONARITY

A time series is *stationary* if and only if its statistical properties do not change as time passes. The expected value (mean) of a stationary series will not change, nor will its variance, nor its serial correlation, nor any of an infinite number of other properties. Since variation in *any* of an infinite number of properties destroys stationarity, strict testing for stationarity is impossible. Such a test would require an infinite number of component tests. Luckily, in practice it is only the grossest, usually most visible aspects of stationarity (a stable mean and variance) that are important. A suite of tests that covers the largest issues is usually sufficient.

It is at least desirable, and perhaps crucial, that all variables used in a trading system be reasonably stationary across the time period of interest. It is vital that the statistical properties of the variables will remain the same when the system is placed in operation as when it was trained and tested. Suppose, for example, you define a crude trend variable as a five-day moving average of closing price minus a ten-day moving average, and attempt to use this variable for predicting

the S&P500. Decades ago, when SP was trading at prices less than 100, a one-point difference was significant. But today, with SP well over 1000, a one-point move is trivial. This sort of nonstationarity would be devastating.

The situation goes beyond just conformity between variables in the training period and the testing or implementation period. If the statistical properties of a variable change significantly during the training period, models will find it difficult to identify the subtle bits of predictive information buried in the data. It's like looking for a needle in a haystack when someone keeps taking hay out from one place and adding new hay to another place.

Because of the importance of stationarity, *TSSB* contains an extensive suite of stationarity tests. These tests can be invoked in several ways. The most versatile method is to use the menu system, because this way the user can select which tests to perform. A simpler but less versatile method is to invoke the test suite from within the script file. In this case, a default set of tests is performed. If a variable is specified, the tests are performed on that variable only. If no variable is specified, the tests are performed on all variables. The syntax for these two options is as follows:

```
STATIONARITY OF VariableName IN MarketName ;  
STATIONARITY IN MarketName ;
```

The simplest and fastest stationarity tests are based on crude cell counts. Imagine plotting a time series of the variable, using a single dot to represent the variable's value at each bar. Place a rectangular grid over the plot and count the number of cases that fall into each section of the grid. For example, suppose we use a 5 by 3 grid, dividing the time extent into five periods and dividing the range of the variable into three sections: large, medium, and small. If the series is stationary, the vertical distribution of counts should be similar across time (columns of the grid). If, on the other hand, we see an unusually large count of the 'large' variable category in one time period, but an unusually small count of the 'large' category in another time period, we suspect that the series is nonstationary. This decision can be made rigorous by using an ordinary chi-square test.

Another family of tests searches for a single point in time at which the nature of the series changes dramatically, a *structural break*. This might happen, for example, if a market moves from open-outcry trading to electronic trading, or if a new government regulation changes the nature of trading. A significant market change of this sort can be devastating to an automated trading system. *TSSB* contains several algorithms that attempt to determine if such a changeover point exists. The program identifies that point in time at which the largest such change in statistical properties occurs, and when possible attempts to assign a p-value

to the change. This p-value is the probability that, if there is truly no structural break, we would have seen an apparent break as large as that found. Thus, very small p-values are bad, indicating a likely structural break.

It is well known that variables associated with financial markets often have distributions that have heavy tails (a high probability of extreme values), are skewed (more likely to have unusually large than small values, or the opposite), or have other properties that violate assumptions of a normal distribution. For this reason, *TSSB* contains many stationarity tests that are based on order statistics and hence do not assume normality. These latter tests are vastly preferable because they are robust against outliers, which can destroy conventional tests.

Theories abound that markets tend to behave differently in some months of the year than in other months, *seasonality patterns*. If any variable has such a property to a significant extent, this inconsistent behavior will hamper a model's ability to find predictive information in the data. Month-to-month changes may swamp out more subtle but important behavior. Thus, *TSSB* contains numerous tests that attempt to determine if some months behave differently from other months.

*TSSB* provides the user with a unique and valuable feature for any of the tests. If the stationarity test is invoked through the menu system (GUI), a Monte-Carlo Permutation Test (MCPT) can be performed on the test statistic. These MCPTs provide two valuable pieces of information. First, they provide an alternative estimate of the p-value of the test. The ***Single pval*** is the estimated probability that the test statistic would be as large as or larger than the obtained value if the distribution of the values of the variable was independent of time (stationary). The Monte-Carlo ***Single pval*** statistic is, in many cases, more accurate than p-values computed with conventional methods.

The Monte-Carlo Permutation Test provides a second statistic that can be enormously helpful in interpreting results. In nearly all applications, the user will be simultaneously testing a (possibly large) set of candidate indicator and target variables. Those having a small p-value will be singled out for more detailed study. However, even if all of the variables happen to be nicely stationary, the luck of the draw will practically guarantee that one or more variables will have an unjustifiably small p-value, especially if a large number of variables are present. The *Grand pval* is the probability that, if *all* of the variables happen to be truly stationary, the obtained value of the test statistic that is the greatest among the variables would equal or exceed the obtained value. This allows us to account for the selection bias inherent in focusing our attention on only those variables that are most suspicious. If we see that the *p-*

*value* or *Single pval* is suspiciously small, but the *Grand pval* is relatively large, our degree of suspicion would be lessened. Of course, failure to reject a null hypothesis does not mean that we can safely accept it. Hence, relatively large values of the *Grand pval* do not mean that we can conclude that the variable is stationary. On the other hand, tiny values of the *Grand pval* should raise a big red flag.

Understand that although tiny values of the various p-values indicate nonstationarity, this does not automatically imply that the degree of nonstationarity is serious enough to degrade performance of the trading system or signal filter. If the dataset contains a large number of cases, even trivially small amounts of nonstationarity may result in small p-values. Many of the tests provide an additional statistic that indicates, at least roughly, the practical extent of any nonstationarity. These other statistics can be quite heuristic, and should be interpreted with great caution. Still, they are useful.

This section has provided an overview of the stationarity tests provided by *TSSB*. Because the program contains a large number of tests, some of which are quite complex and sophisticated, the subject merits an entire chapter of its own. This chapter will be provided in a later edition of the tutorial, and a summary is currently in the manual.

## A Final Example

This chapter has discussed the commands that are employed to read market price history files and create a database of variables that may be used as indicators and targets for a subsequent study. We end this discussion with an ‘all the bells and whistles’ example that combines the bare essentials with as many options as possible. The following commands read a set of market files, compute a database of variables, and perform several optional statistical tests.

```
READ MARKET LIST "SYMBOLS.TXT" ;
READ MARKET HISTORIES "E:\SP100\IBM.TXT" ;
MARKET SCAN ;
RETAIN YEARS 2001 THROUGH 2006 ;
CLEAN RAW DATA 0.65 ;
READ VARIABLE LIST "TREND_VOLATILITY.TXT" ;
OUTLIER SCAN ;
DESCRIBE P_TREND ;
CROSS MARKET AD ;
CROSS MARKET KL ;
CROSS MARKET IQ ;
STATIONARITY OF P_TREND IN IBM ;
```

The market scan produced the following results (the first few lines from the audit log):

---

-----  
Market scan for large percentage price changes  
-----

Market HAL largest percent differences...

Date	Time	Difference
20011207	000000	82.500
20020724	000000	31.648
19871019	000000	25.955
19871020	000000	22.930

The market HAL is listed first in the log because its maximum percent price jump, 82.5 percent, was the largest among the markets. This jump occurred on December 7, 2001. The other markets also appear, in decreasing order of worst jump size. They are not listed here.

The outlier scan produced the following output in the log file, shown here in its entirety:

-----  
Outlier / Entropy scan  
-----

Variable	Min	Mkt	Max	Mkt	IQ	Rng	Ratio	Entropy
P_TREND	-49.989	IBM	49.336	JNJ	23.508	4.2	0.888	
P_VOLATILITY	-49.507	HNZ	49.726	JPM	49.364	2.0	0.992	
DAY_RETURN	-5.226	JNJ	3.974	AA	0.775	11.9	0.593	

Sorted by ratio, worst to best

Variable	Ratio	Entropy
DAY_RETURN	11.9	0.593
P_TREND	4.2	0.888
P_VOLATILITY	2.0	0.992

Sorted by entropy, worst to best

Variable	Ratio	Entropy
DAY_RETURN	11.9	0.593
P_TREND	4.2	0.888
P_VOLATILITY	2.0	0.992

We see that the variable P\_TREND had a minimum value of -49.989, which occurred in the market IBM. Its maximum of 49.336 happened in JNJ. Its interquartile range was 23.508, the ratio of its range to its interquartile range was 4.2, and its relative entropy was 0.888. When the two measures of distribution quality are sorted from worst to best, we see that the three variables end up in the same order, a fairly common occurrence.

The DESCRIBE P\_TREND command produced the following self-explanatory output:

```
Descriptive statistics for P_TREND in Pooled data
Mean = 0.460128 (std err = 0.129125 t = 3.563)
N = 17834 Variance = 297.336151
Standard deviation = 17.243438
Skewness = -0.085221 (std err = 0.029002)
Kurtosis = -0.191218 (std err = 0.073369)
Median = 0.558860
Interquartile range = 23.5057 (-11.0866 to 12.4191)
Range = 99.3247 (-49.9888 to 49.3359) Range/IQ = 4.226
20-bin relative entropy = 0.888071
```

The CROSS MARKET AD test produced extensive output. Here are the first two sections:

**Cross market Anderson-Darling test for variable P\_TREND**

<b>AA</b>	<b>2.415</b>	<b>(p=0.05489)</b>
<b>DELL</b>	<b>1.555</b>	<b>(p=0.16390)</b>
<b>DOW</b>	<b>0.913</b>	<b>(p=0.40617)</b>
<b>GE</b>	<b>2.858</b>	<b>(p=0.03233)</b>
<b>HAL</b>	<b>4.462</b>	<b>(p=0.00520)</b>
<b>HNZ</b>	<b>2.124</b>	<b>(p=0.07860)</b>
<b>IBM</b>	<b>1.303</b>	<b>(p=0.23153)</b>
<b>JNJ</b>	<b>0.555</b>	<b>(p=0.69166)</b>
<b>JPM</b>	<b>1.332</b>	<b>(p=0.22217)</b>
<b>WMT</b>	<b>3.454</b>	<b>(p=0.01619)</b>
<b>XOM</b>	<b>4.298</b>	<b>(p=0.00625)</b>
<b>XRX</b>	<b>1.425</b>	<b>(p=0.19535)</b>

**Cross market Anderson-Darling test for variable P\_TREND,  
sorted worst to best**

<b>HAL</b>	<b>0.00520</b>
<b>XOM</b>	<b>0.00625</b>
<b>WMT</b>	<b>0.01619</b>
<b>GE</b>	<b>0.03233</b>
<b>AA</b>	<b>0.05489</b>
<b>HNZ</b>	<b>0.07860</b>
<b>DELL</b>	<b>0.16390</b>
<b>XRX</b>	<b>0.19535</b>
<b>JPM</b>	<b>0.22217</b>
<b>IBM</b>	<b>0.23153</b>
<b>DOW</b>	<b>0.40617</b>
<b>JNJ</b>	<b>0.69166</b>

The first table lists the markets in the order in which they appeared in the market list file. It shows the raw Anderson-Darling statistic, which has no simple intuitive meaning, along with the associated p-value. Because small p-values indicate a low probability of obtaining a score this extreme if the market has good conformity, small p-values are bad. Thus, it makes sense to focus on the p-values and sort based on them. The second table does this. We see that HAL is a serious oddball market when it comes to conformity of P\_TREND among this set of markets, while JNJ is a solid conformist.

These pairs of tables are repeated for the other two variables. Finally, these two summary tables appear:

**Median Anderson-Darling across markets, worst to best...**

P_TREND	0.12125
DAY_RETURN	0.19107
P_VOLATILITY	0.25621

**Median Anderson-Darling across variables, worst to best...**

XOM	0.00065
HAL	0.00520
GE	0.04263
WMT	0.07537
AA	0.12825
DELL	0.16390
JPM	0.22217
IBM	0.23153
XRX	0.26103
HNZ	0.27253
DOW	0.56027
JNJ	0.84289

The first table shows that when one considers the median A-D p-value across markets, P\_TREND has the worst cross-market conformity, while P\_VOLATILITY has the best. If we instead consider the median across variables and compare markets, XOM is the worst non-conformer, with a median p-value of 0.00065. A result this significant indicates that we might be wise to take a closer look at XOM to see why it behaves so differently from other markets, at least as far as these variables are concerned.

The CROSS MARKET KL and CROSS MARKET IQ tests produced tables similar to those from the A-D test. However, these two statistics do not have p-values, so the sorting is based on the test statistics themselves. It is worth looking at the IQ test summary tables:

**Median IQ range overlap across markets, worst to best...**

P_TREND	0.96019
DAY_RETURN	0.96218
P_VOLATILITY	0.98540

**Median IQ range overlap across variables, worst to best...**

XOM	0.91561
HAL	0.92772
JPM	0.92911
IBM	0.93044
GE	0.94769
DELL	0.95565
AA	0.95676
JNJ	0.97401
HNZ	0.98087
WMT	0.98220
DOW	0.98352
XRX	0.98485

These results are quite remarkable in how much cross-market conformity is obtained. When we look at the median IQ overlap across markets, once again P\_TREND is the worst variable, but it has more than 96 percent overlap of interquartile ranges! If we look at the median across variables, once again XOM is the worst, but it still manages to have a median overlap of almost 92 percent.

So... the A-D test gave a worst p-value of 0.00065 for XOM, which invited a warning to check on this market. Yet we see that its median interquartile range overlap with the pooled distribution was almost 92 percent, which is respectable. One or both of two things are responsible for this conflict:

- 1) There is some true non-conformity, but the large number of cases in the dataset (17,834) exaggerated the statistical significance of the difference, which in practice may be negligible.
- 2) The nonconformity lies mostly in one or both tails of the distribution. The A-D test is extremely sensitive to tail behavior, while the IQ test practically ignores tails.

Finally, the stationarity test produced voluminous output in the log file. However, this output is so complex that it is best deferred to the chapter on stationarity tests, to appear in a later edition of the tutorial.

# Reading and Writing Databases

The prior chapter discussed how to read one or more market history files and create a database of variables that may be used as indicators and targets. In most cases, computing variables requires a significant amount of computer time; repeating this computation for every study would be an exercise in frustration. For this reason, *TSSB* has the ability to write a database to the hard drive, and later read it back in a very fast and efficient operation.

Besides speed, another reason for reading a database file is to bring into *TSSB* data that has been computed by other programs. It may be that the user needs indicators or targets that are not available in *TSSB*'s built-in library. We also saw in [Chapter 3](#), which dealt with developing signal filters for existing trading systems, that trade information for the system being filtered is provided to *TSSB* by means of a database. Finally, the user may sometimes want to use *TSSB*'s built-in library to generate variables, but then perform studies of these variables with another program. For all of these reasons, it is handy to be able to read and write database files.

Because we will often be interfacing with other programs, either to read database files produced by them, or to write database files that will be read by them, *TSSB* does not use a proprietary database format. Rather, it uses a subset of internationally recognized database standards. The exact nature of this format will be described [here](#). For now, understand that database files created by *TSSB* can be read by all popular spreadsheet and statistical analysis programs. Also, most if not all such programs are capable of writing database files that can be read by *TSSB*.

# Quick Reference to Database Commands

This section contains a list of all of the commands related to reading and writing databases, along with a brief description of each. Further details are presented in other sections.

**RETAIN YEARS** - Specifies that only a range of years be kept from the database file. This command was discussed in detail [here](#). If used, this option must precede any command that reads a database.

**RETAIN MOD** - Specifies that every n'th date be kept from the database file. This command, which is useful for temporarily shrinking the database during initial development, was discussed in detail [here](#). If used, this option must precede any command that reads a database.

**RETAIN MARKET LIST** - Specifies that only records from certain markets be read from the database. If used, this option must precede any command that reads a database. See [here](#).

**VARIABLE IS TEXT** - Specifies that a particular variable is text as opposed to numeric. If used, this option must precede any command that reads a database. See [here](#).

**WRITE DATABASE** - Writes the database to a disk file. See [here](#).

**READ DATABASE** - Reads a database that is in chronological order. See [here](#).

**READ UNORDERED DATABASE** - Reads a database that need not be in chronological order. See [here](#).

**APPEND DATABASE** - Reads a database (which must be in chronological order) and appends its variables to an existing database. If used, this command must not appear until a database already exists, either from internal generation (READ VARIABLE LIST) or reading from a disk file (READ DATABASE). See [here](#).

**IS PROFIT** - Specifies that a variable is a measure of profit. If used, this command must not appear until the variable is present in an existing database, so in general it would appear *after* a READ VARIABLE LIST or READ DATABASE command. See [here](#).

## Detailed Descriptions

This section provides more detailed descriptions of the commands associated with reading and writing a database. Several commands that were discussed in the prior chapter (RETAIN YEARS, RETAIN MOD) will not be repeated in detail here.

## RETAIN MARKET LIST

This command is the database analogue of the READ MARKET LIST command that named the markets for which variables are to be computed. The syntax is similar:

```
RETAIN MARKET LIST "FileName" ;
```

*FileName* specifies the name of the market list file, enclosed in double quotes (""). The file name may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will assume that the file is in the current working directory as defined by the Windows operating system. See [here](#) for an example of a market list file.

This command, which if used must appear before a database is read, causes only a subset of markets to be read from the database file. The primary use for this command is to allow the user to keep one master database of all markets of interest, but select only a subset of the master database for a particular study. For example, we may use the following two commands to read a huge database but keep only data from certain markets:

```
RETAIN MARKET LIST "EastCoastUtilities" ;
READ DATABASE "AllSP500Equities" ;
```

## VARIABLE IS TEXT

By default, all variables in a database file except the market name are numeric. However, *TSSB* can occasionally make use of category variables that are identified by names. For example, we may have a season variable with values ‘planting’, ‘growing’, and ‘harvest’. More often, the user will be presented with a database produced by another program, and this database will contain one or more variables that are text. We will probably want to ignore these variables in the study, but they are nonetheless present in the file, and hence must be accommodated.

The following syntax is used to declare a variable as text:

```
VARIABLE VarName IS TEXT ;
```

This command, if used, must precede any command that reads the database. The values of this text variable must not contain a space or any special character except the underscore (\_). Text variables are almost never used in *TSSB*, so we will defer further explanation until a later edition of the tutorial.

## WRITE DATABASE

Any time a database is present, it can be written to a disk file. A database will be present as a result of internal variable computation (READ VARIABLE LIST) or reading a database. The database file is an ordinary ASCII text file which is readable not only by the *TSSB* program, but by most spreadsheet and statistical programs.

There are two commands for writing a database. The first command shown below writes only the database file, and the second also writes a *family file*. A family file contains supplementary information that may be useful to the *TSSB* program and will be briefly discussed later in this section.

```
WRITE DATABASE "DatabaseName" ;
WRITE DATABASE "DatabaseName" "FamilyFileName" ;
```

The file names are enclosed in double quotes (""). The names may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will write the file into the current working directory as defined by the Windows operating system.

In order to conform to recognized database standards, the first record written in the file will list the fields in the order that they appear on subsequent lines. The first item is the date as YYYYMMDD. If the application is intraday, the next item is the time as HHMM or HHMMSS. The next item is the name of the market. Variables follow. Blanks are used as delimiters when the file is written by *TSSB*, although tabs and commas are also legal. Here are the first few records in a typical database file:

Date	Market	LIN_ATR_5	LIN_ATR_15	RETURN
19621228	IBM	4.89092588	-1.36788785	-0.31357768
19621231	IBM	-0.92078519	1.70236468	-0.47036651
19621231	SP500	5.16105413	4.43815613	-0.38055974
19630102	IBM	-10.57968712	1.92275798	1.25470793
19630102	SP500	0.11760279	5.16873121	0.95639825
19630103	IBM	-5.70287132	1.73097610	0.00000000

The first line says that on subsequent lines, the first field will be the date, then the market, then three variables. The database file is always written in chronological order.

The user may optionally specify that a *family file* is to be written to accompany the database file. This file contains information about any variables that were computed from *TSSB*'s internal library. This information concerns things such as the name of the family (discussed in the [Variables chapter](#)), how far the variable looks back or ahead in history, whether it involves an index, and whether it is an indicator (looks back in history) or a target (looks forward in history). Here are the first few lines of a typical family file:

```
LIN_ATR_5 "LINEAR PER ATR" 5 0 0 0 0 0 0 0 0 0  
LIN_ATR_15 "LINEAR PER ATR" 15 0 0 0 0 0 0 0 0 0  
RETURN "NEXT DAY ATR RETURN" -1 0 0 0 0 0 0 0 0 1
```

The first line says that the internally computed variable “LIN\_ATR\_5” is a member of the “LINEAR PER ATR” family and looks back five days in history. The other items are very advanced, and the user need not be concerned with them. [here](#) we will see how it can be useful to write a family file and then read the family file later, when the database is read. However, there is no reason why the average user would ever need to examine a family file. This file is intended strictly for internal use by the *TSSB* program, so we will not pursue the meaning of individual components of this file.

## READ DATABASE

Computing variables from *TSSB*'s internal library can require a large amount of computer time. For this reason, it is recommended that whenever possible, all potential indicators and targets be computed just once and saved as a database file using the WRITE DATABASE command just presented. This database can then be rapidly read back using the READ DATABASE command.

This command can also be used to read a file produced by another program. The database file is an ordinary ASCII text file. The first line must specify the names of all fields. The first field must be ‘Date’ and, if the data is intraday, the next field must be ‘Time’. The next field must be ‘Market’, with market names limited to a maximum of five characters. Subsequent fields must be variable names. These names must not contain spaces or special characters other than the underscore (\_).

The maximum length of a variable name is 15 characters. See [here](#) for a sample database file. Note that spaces, commas, or tabs may be used as delimiters.

The records in a database read with the READ DATABASE command must be in chronological order. If the records are not in order, the READ UNORDERED DATABASE command (described in the [next section](#)) must be used.

Just as there are two commands for writing a database, there are also two commands for reading it back. The first command shown below reads only the database file, and the second also reads a *family file*. A family file contains supplementary information that may be useful to the *TSSB* program. Refer back to [here](#) for a discussion of the family file. See [here](#) for an example application in which the family file plays a vital role.

```
READ DATABASE "DatabaseName" ;
READ DATABASE "DatabaseName" "FamilyFileName" ;
```

The file names are enclosed in double quotes (""). The names may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will read the file from the current working directory as defined by the Windows operating system.

## READ UNORDERED DATABASE

The READ DATABASE command described in the prior section assumes that the records in the file are in chronological order. This allows reading to operate with great efficiency. If the database file is not chronological, then it must be read with the READ UNORDERED DATABASE command. Note that this command requires much more computer time, disk space, and memory than the READ DATABASE command, especially if the file is large. Therefore, it should be used only if necessary.

Like the WRITE DATABASE and READ DATABASE commands, this has two forms, depending on if a family file is to be read in addition to the database file. The syntax of these two forms is as follows. See [here](#) for a discussion of family files.

```
READ UNORDERED DATABASE "DatabaseName" ;
READ UNORDERED DATABASE "DatabaseName" "FamilyFileName" ;
```

## APPEND DATABASE

The user may wish to compute some variables using *TSSB*'s internal library, and then read others from an external database file. Or there may be a need to

merge several external database files. These operations can be accomplished with the APPEND DATABASE command.

The syntax of this command is as follows:

```
APPEND DATABASE "FileName" ;
```

The file name is enclosed in double quotes (""). The name may not include blanks or special characters other than the underscore (\_). If a full path name is not specified, *TSSB* will read the file from the current working directory as defined by the Windows operating system.

The appended file must be in chronological order. At this time there is no UNORDERED option for appending files. However, there is a simple trick for working around this limitation if the file to be appended is not ordered: Just write a two line script that reads the unordered file and then writes it back out again. Writing a database always outputs the file in chronological order. Such a script might look like this:

```
READ UNORDERED DATABASE "UnorderedFile" ;
WRITE DATABASE "OrderedFile" ;
```

Then, subsequent scripts would append “OrderedFile” instead of the original unordered file.

Also note that at this time the APPEND DATABASE command is not able to read a family file. This is almost never a problem, because appended databases are usually created by other programs. The ability to append a family file may be added to a future version of *TSSB*.

The APPEND DATABASE command operates by merging records in the appended database with records already present. A *record* is the set of all variables for a particular market at a single date (and time, if intraday). Thus, for each record in the appended database, *TSSB* searches its existing database for the same market and date/time and, if it finds a corresponding record, appends the new variable(s) to the existing variable(s) to create a new record.

We'll look at a simple example using just one variable in each database. Suppose the following database exists at the time the APPEND DATABASE command appears:

Date	Market	X1
20010601	SP	17.2
20010601	IBM	22.5
20010602	SP	44.1
20010603	SP	37.8
20010603	IBM	51.2
20010604	IBM	46.6

Now suppose that the appended database looks like this:

Date	Market	X2
20010601	SP	42.7
20010601	IBM	66.7
20010602	IBM	10.4
20010603	SP	13.7
20010603	IBM	48.9
20010604	SP	62.4
20010604	IBM	68.4

Then the merged database, which contains only those records which have the same date and market in both the original and appended databases, looks like this:

Date	Market	X1	X2
20010601	SP	17.2	42.7
20010601	IBM	22.5	66.7
20010603	SP	37.8	13.7
20010603	IBM	51.2	48.9
20010604	IBM	46.6	68.4

Notice in this example that no record remains for the date 20010602 because on that date the existing database contained only a record for SP, while the appended database has only a record for IBM on that date. On 20010604, the appended database has records for both SP and IBM, but the existing database has a record for only IBM. Thus, no record for SP remains for this date after appending.

One implication of this merging algorithm is that if the existing and the appended databases have few markets in common, the resulting merged database could be very sparse. Similarly, if the two databases are highly disjoint in time (their time periods have little overlap), the resulting merged database would cover only the short interval in which their time periods overlap. Finally, if the appended database is for a single market that does not exist in the current database, the result will be an empty database, one with no records at all! The most common use for the APPEND DATABASE command is when the current and the appended databases cover approximately the same time period and have most or all of their markets in common.

## IS PROFIT

*TSSB* contains a safety feature to prevent the user from accidentally misinterpreting results. This safety feature suppresses printing of any performance measure that involves profit (such as profit factors) when the target variable is not a profit. Such targets do exist. For example, the trend of prices in the near future can be used as a target being predicted, but it is certainly not a profit. Profit factors based on slope would be meaningless. When targets are computed from the internal library,

*TSSB* knows whether the target is a profit. But when it's read from a database file the program has no way of knowing whether the target is a profit unless the user informs it that this is the case.

The following syntax is used to declare a variable as profit:

```
VarName IS PROFIT ;
```

The IS PROFIT command will usually be needed when the target variable of a model, committee, or oracle is read from a database file, as opposed to being computed internally. Of course, if the target variable is not a measure of profit, you would not want to use this command. But most targets are measures of profit, so this command will then be needed so that profit-based performance measures will be printed.

## A Saving/Restoring Example

[here](#) we saw an example of a simple standalone trading system. That example script computed the indicators and target, and then immediately tested a prediction model. We now show an approach that is usually more efficient: split the script into two parts. The first part computes and saves the indicators and target, and the second part reads the saved database and tests the prediction model. Here is the script that computes the indicators and targets and saves them as a database file:

```
READ MARKET LIST "SYMBOLS.TXT" ;
READ MARKET HISTORIES "E:\SP100\IBM.TXT" ;
READ VARIABLE LIST "TREND_VOLATILITY.TXT" ;
WRITE DATABASE "EXAMPLE_DBASE.DAT" "EXAMPLE_DBASE.FAM" ;
```

The author's personal convention is to use the .DAT extension for all database files. This makes it easy to look at a directory listing and know which file(s) are databases. Other users may wish to use .TXT because the database really is a

text file. This is purely a personal preference; *TSSB* does not care what you use for an extension. Similarly, the author uses .FAM for family files. Again, the family file is a text file, so .TXT would be fine also.

This script did not need to save the family file, because the second script, which tests a prediction model, does not make use of family information. Still, it's a good habit to save the family file whenever possible, as you never know when you might change the application in a way that needs family information. The family file is very small and fast to write to disk, so saving it costs nothing and may prove handy later.

Here is the second part of the script, which reads the previously saved database and tests the prediction model:

```
READ DATABASE "EXAMPLE_DBASE.DAT" "EXAMPLE_DBASE.FAM" ;
MODEL SIMPLE_MODEL IS LINREG [
    INPUT = [ P_TREND P_VOLATILITY ]
    OUTPUT = DAY_RETURN
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

WALK FORWARD BY YEAR 10 1999 ;
```

# Creating Variables

Several prior chapters provided an overview of the process of creating a database of variables (indicators and targets) that are based on the extensive library built into *TSSB*. In particular, the control script reads a list of markets with the READ MARKET LIST command, reads the market price histories with the READ MARKET HISTORIES command, and reads the list of variables to be computed with the READ VARIABLE LIST command. It is this final step that will be the focus of this chapter.

## Overview and Basic Syntax

The variable definition list is an ASCII text file containing one variable definition per line. The syntax is simple for basic variables, and it can become quite complex when more sophisticated options are invoked. Rather than trying to present the complete set of all syntax variations in one general statement, we will focus here on only the most basic form, and deal with extensions to the syntax in separate sections.

A variable definition in its simplest form consists of three items: a hopefully descriptive name chosen by the user, the family selected from the *TSSB* library, and any parameters that are needed to fully define that family member. (An indicator family refers to indicators that all have the same computational form but differ in their values of user-specified parameters.) For example, we may want to define an indicator that we choose to call **MA\_XOVER**. For this variable, we select the **MA DIFFERENCE** family that is a member of the built-in library. This variable, which will be discussed later, subtracts a long-term moving average from a short-term moving average and normalizes the result. It requires three parameters: the short-term history length, the long-term history length, and the number of bars to lag the long-term history. Members of this family differ only in these three parameters. The line in the variable definition list file to implement this variable could be as follows:

**MA\_XOVER: MA DIFFERENCE 5 20 5**

The name chosen by the user is the name that will appear in all studies. It should be reasonably short, yet descriptive to the user. The maximum length is 15 characters. It may contain only letters, numbers (though the name cannot begin with a number), and the underscore character (\_).

The user name is followed by a colon (with or without a space between) and then the definition of the family. Legal families will be discussed soon. Most but not all families require one or more parameters to immediately follow.

Blank lines, which can help separate groups of variables, are legal. Any text after a semicolon (;) is ignored and can be used for comments.

## Index Markets and Derived Variables

Many index ‘markets’ are commercially available. These may be actual baskets of securities, or they may be computed from weighted averages of numerous equity prices and traded indirectly by means of options or futures. A useful

property of most indices is that because they are derived from numerous securities, they reflect an average behavior of the market. Because they in a sense summarize the state of an entire market or large segment of the universe of markets, they are often useful in the computation of indicators.

If the user wishes to employ one or more indices in the variable definition list, the index (indices) must be declared before the variable definition list file is read. If the user will employ only one index, the declaration is in the first form shown below. The second form is used to declare multiple individual indices, up to a maximum of 16.

```
INDEX IS MarketName ;  
INDEX Number IS MarketName ;
```

The *Number* of the index must be an integer from 1 through 16. There is no requirement that numbering start at one, although that would probably reduce confusion.

The following command declares that OEX (the S&P 100 index) will be referenced as an index in the variable definition list file:

```
INDEX IS OEX ;
```

TSSB allows the user to use indices in two ways when creating variables. The most common use is to cause a variable to be computed from the index instead of individual markets. In normal operation, the value of a variable for a particular market at a particular date/time is based on the price history of that market. Naturally. But if the variable definition is followed by the keyword IS INDEX, that variable is computed from the price history of the index, regardless of the market.

A second use for indices is to compute the deviation of a variable for a market from the same variable for an index. For example, we may want to compute the trend of prices in a market, also compute the trend of an index, and define our variable of interest as their difference. If a market is trending the same as the index, the value of this variable would be near zero. If the market were trending upward more strongly than the index, the variable would be positive, and if the opposite were true the variable would be negative. This lets us measure how much a market is conforming to ‘average’ behavior. In order to do this, we place the keyword MINUS INDEX after the variable definition (but before any parameters).

## An Example of IS INDEX and MINUS INDEX

Here is an example of both uses of an index. The LINEAR PER ATR family of indicators fits a straight line to a specified length of history, computes ATR (Average True Range) over a specified length, and scales the slope of the trend line by dividing by ATR. The first of the following three variables (RAWVAR) computes this variable by fitting the trend line over 20 bars and fitting ATR over 250 bars. Because no index is referenced, the variable is computed based on the price history of each market in the market list. The second variable (INDEXVAR) bases the value on the index for all markets in the market list. The third variable (DIFFVAR) is the value for the market minus the value for the index.

```
RAWVAR: LINEAR PER ATR 20 250
INDEXVAR: LINEAR PER ATR IS INDEX 20 250
DIFFVAR: LINEAR PER ATR MINUS INDEX 20 250
```

Some records in the resulting database might look like the following:

Date	Market	RAWVAR	INDEXVAR	DIFFVAR
19970301	IBM	37.2	32.0	5.2
19970301	BOL	31.5	32.0	-0.5
19970302	IBM	24.6	20.1	4.5
19970302	BOL	18.5	20.1	-1.6

Observe that on any given date, the IS INDEX variable, INDEXVAR, has the same value for each market. The difference variable, DIFFVAR, is the value computed from the market price history minus that computed from the index history. So for the first record,  $37.2 - 32.0 = 5.2$ . This tells us that on this date, IBM had a stronger upward trend than the index.

Note that this example took a simplified approach in order to provide the reader with exact numbers for the differences. If this example were run in *TSSB*, we would probably see that the difference variable DIFFVAR does not exactly equal the difference between the market and index variables. This is because nearly all variables computed in *TSSB* are subjected to a nonlinear squashing function as a final operation so as to fix them in a common range and suppress outliers. In the case of the MINUS INDEX version, the subtraction is done before the transformation so that the quantities being differenced are in conformity. The net result is that the differences may not always be exact, depending on where the various values lie.

Also note that the index market did not appear in this example database. There is a subtle reason for this. If a MINUS INDEX variable appears anywhere in the variable list file, then the index will be omitted from the database. This is an

annoying consequence of the fact that values of the MINUS INDEX variable would be effectively undefined for the index. Mathematically, the value is zero, but if a variable were identically zero for all cases of a market, modeling results would be nonsensical and all sorts of problems would arise during system development. So this quandary is resolved by simply omitting the index market when the MINUS INDEX option is used.

## Multiple Indices

In the vast majority of applications, one index is sufficient. However, *TSSB* allows the user to declare up to 16 markets as indices and use them individually. We saw how to declare index markets [here](#). Suppose we have declared two index markets as follows:

```
INDEX 1 IS OEX ;
INDEX 2 IS DJIA ;
```

We might then use them as follows:

```
OEXVAR: LINEAR PER ATR IS INDEX1 20 250
DJIAVAR: LINEAR PER ATR IS INDEX2 20 250
```

Note in this example that the declaration of an index requires a space between the INDEX keyword and the number, while later variable references require that no space separate them. When an index is referenced, its number (1-16) is part of the name.

Also note that INDEX1 is synonymous with INDEX. Thus, the user could declare an index with the command INDEX IS OEX and later reference it as INDEX1. Similarly, the user could declare an index with the command INDEX 1 IS OEX and later reference it as INDEX. However, for the sake of clarity, it is suggested that the user be consistent in the script and variable definition list in order to avoid confusion.

## Historical Adjustment to Improve Stationarity

Indicators generally fall into one of two categories: those whose actual value at the moment is of primary importance in and of itself, and those whose primary importance is based on their current value relative to recent values. Many of the indicators built into *TSSB* have this relativity built into them. However, for those that do not, the program includes the ability for the user to adjust current values in several ways that normalize the value according to recent values. This capability will now be discussed.

Why would one want to adjust the current value of an indicator according to recent values? The basic reason is that such adjustment is an excellent way of forcing a great degree of stationarity on the indicator. In most cases, stationarity improves the accuracy of predictive models. (Recall that, roughly speaking, stationarity means that the statistical properties of an indicator do not change over time.) As long as the historical lookback period for the adjustment is made long relative to the frequency of trading signals, important information is almost never lost, and the improvement in stationarity can be enormous.

*TSSB* supports three types of historical adjustment. *Centering* subtracts the historical median from the indicator. *Scaling* divides the indicator by its historical interquartile range. Full *normalization* does both: first it centers the indicator by subtracting the median, and then it scales it by dividing by the interquartile range. This is roughly equivalent to traditional standardization, in which data is converted to Z scores by subtracting the mean and dividing by the standard deviation. However, by using the median and interquartile range instead of the mean and standard deviation, we avoid problems with extreme values. Each of these options will now be presented.

### Centering

Centering can be useful for stabilizing slow-moving indicators across long periods of time. For example, suppose we have a large-scale trend indicator, and we devise a trend-following trading rule that opens long positions when this indicator is unusually positive. Also suppose we hope for trades that have a duration of a few days to perhaps a few weeks, and we would like to obtain such trades regularly. If a market spends long periods of time in an upward trend, and other long periods in a downward trend, this indicator will flag an enormous number of trades in the up periods, and then shut off in the flatter and down periods. By subtracting a historical median of this trend indicator, these long periods of alternating performance will be reduced. In a long period of

upward trend, the indicator will no longer describe the trend. Rather, it will tell us whether the current trend is up versus down relative to what it has been recently. This will more evenly distribute trades across time. Many applications find this useful.

Centering is invoked by following the family name and its parameters with a colon (:), the word CENTER, and the historical lookback period for computing the median which centers the data. Here is an example using the LINEAR PER ATR trend indicator that we saw in the prior section:

```
TREND_OSC: LINEAR PER ATR 20 250 : CENTER 100
```

The definition above computes the LINEAR PER ATR indicator using 20 days to define the trend and 250 days for the Average True Range scaling. Then, it finds the median of the prior 100 values of this quantity and subtracts it from the current value. This provides a powerful ‘return to zero’ force that prevents the computed value from staying above or below zero for too long. In effect, it converts an ‘absolute’ measurement into a variable that oscillates about its mean with guaranteed regularity.

## Scaling

Sometimes centering a variable based on its historical values destroys vital information. In such cases, the sign and magnitude of the variable at the moment is of paramount importance, and centering would destroy this information. Yet we may want to compensate for varying volatility. It may be that a value considered ‘large’ in a time period during which it has low variation would be ‘small’ in a high-variation period. Thus, we may want to divide the value of the variable by a measure of its recent variation. The interquartile range is an ideal measure of variation because it is not impacted by extremely large or small values the way a standard deviation would be.

Scaling is particularly useful for ensuring cross-market conformity of an indicator. If we want to pool indicator data from several different markets, it is vital that the statistical properties of the indicators be similar for all markets. Otherwise, markets with large variation will dominate models, while those with small variation may be essentially ignored. By scaling the indicator according to its historical volatility, we produce a variable that is likely to have similar variability across markets. Scaling is invoked the same way as centering, except that the keyword SCALE is used instead of CENTER.

In addition to division by the interquartile range, the final value is transformed

with a nonlinear function that compresses outliers and produces a variable lying in the fixed range of -50 to 50. In the following equation, which defines the scaling option,  $\Phi(\bullet)$  is the standard normal CDF. Also,  $F_{25}$ ,  $F_{50}$ , and  $F_{75}$  are the 25'th, 50'th, and 75'th percentiles, respectively, of the historical values of the indicator.

$$V = 100 * \Phi\left(0.25 * \frac{X}{F_{75} - F_{25}}\right) - 50 \quad (1)$$

## Normalization

In situations in which centering is appropriate, it often makes sense to simultaneously scale the indicator. This is almost like converting each observation into a standard normal Z score based on recent history, except that here we use the median instead of the mean and the interquartile range instead of the standard deviation. Normalization is the ultimate in imposing stationarity on an indicator. This is invoked the same way as centering except that the keyword NORMALIZE is used instead of CENTER.

In addition to subtraction of the median and division by the interquartile range, the final value is transformed with a nonlinear function that compresses outliers and produces a variable lying in the fixed range of -50 to 50. In the following equation, which defines the normalization option,  $\Phi(\bullet)$  is the standard normal CDF. Also,  $F_{25}$ ,  $F_{50}$ , and  $F_{75}$  are the 25'th, 50'th, and 75'th percentiles, respectively, of the historical values of the indicator.

$$V = 100 * \Phi\left(0.5 * \frac{X - F_{50}}{F_{75} - F_{25}}\right) - 50 \quad (2)$$

## An Example of Centering, Scaling, and Normalization

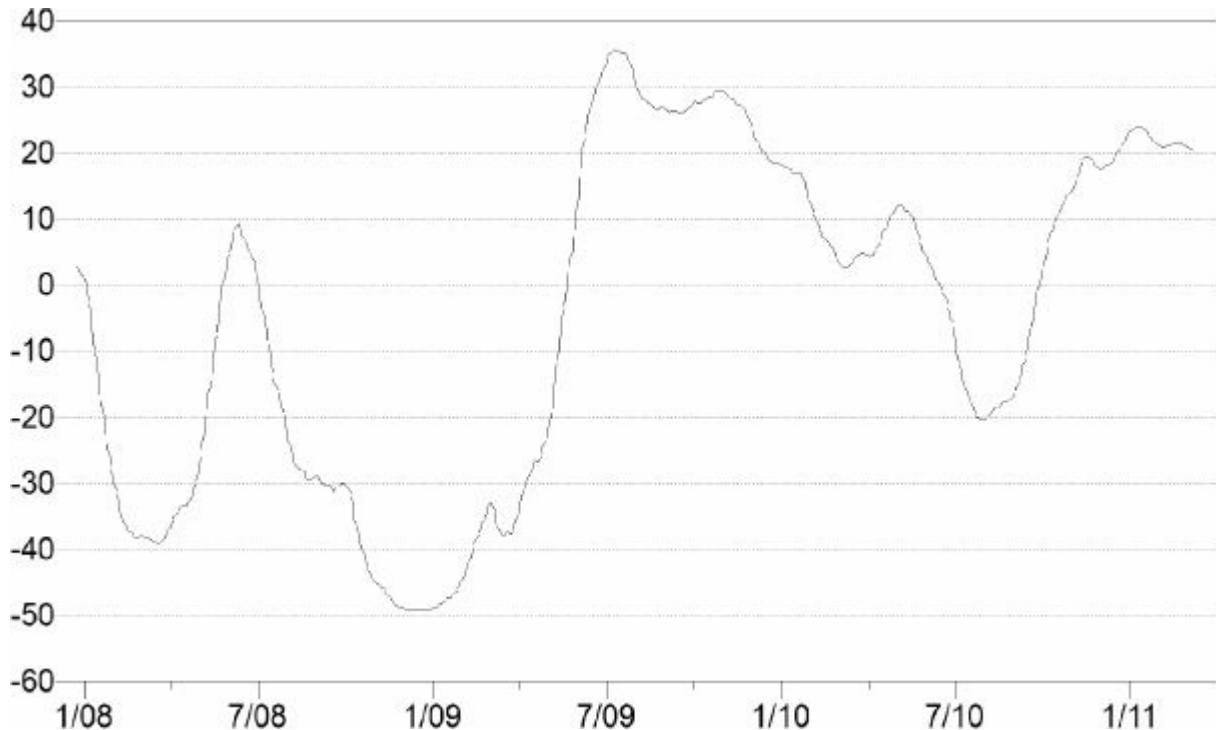
An illustration of these historical adjustments may make them more clear. [Figure 3](#) on the next page illustrates a 100-day trend variable LINEAR PER ATR, scaled per its ATR at a lookback of 1000 days. (This scaling is part of the definition of LINEAR PER ATR, and it has nothing to do with the scaling option discussed in this section.) Notice how, when it moves far upward around the middle of the time period, it stays up there for a long time.

[Figure 4](#) shows this trend variable centered with a lookback period of 200 days. Notice that it does not stay high for long after the rise in the middle time period.

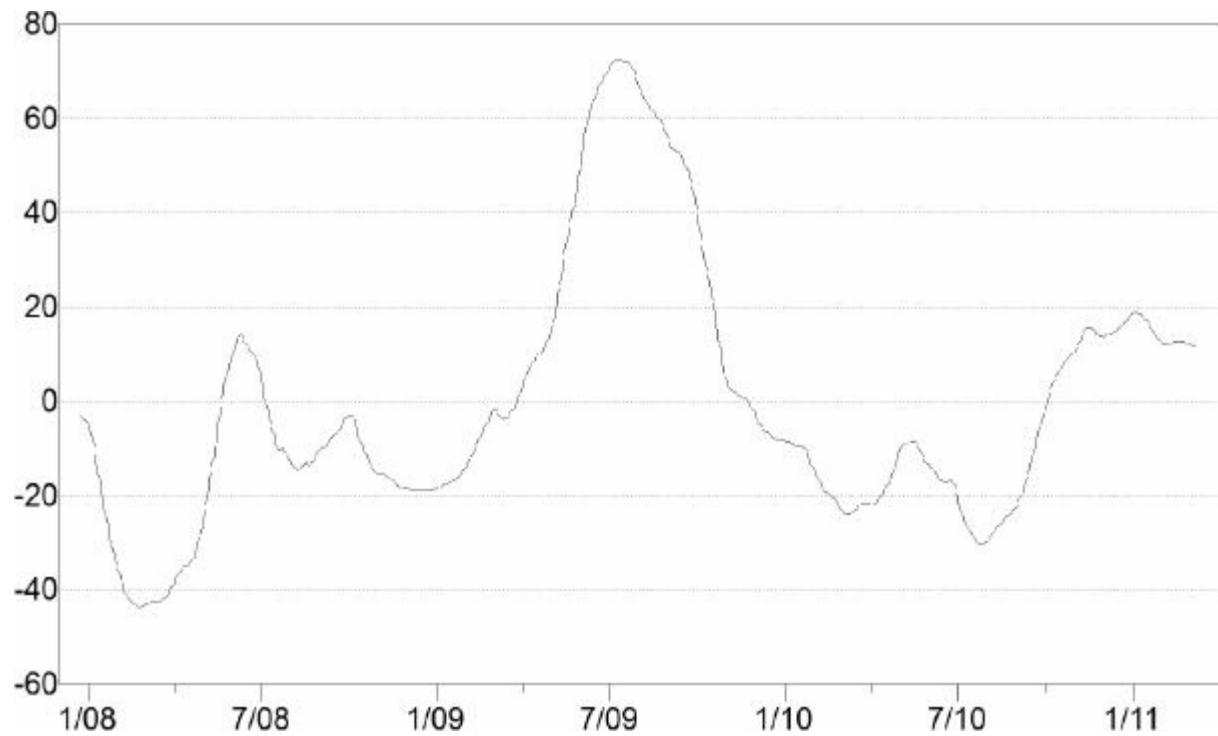
The centering pulls it back toward zero.

[Figure 5](#) shows the trend variable with the scale option applied. It keeps the same sign, but otherwise its behavior is quite different. Beginning around 7/09, the strong positive trend takes a sharp dip because the steep rise in trend just prior to 7/09 is interpreted (rightly or wrongly) as a sudden jump in volatility.

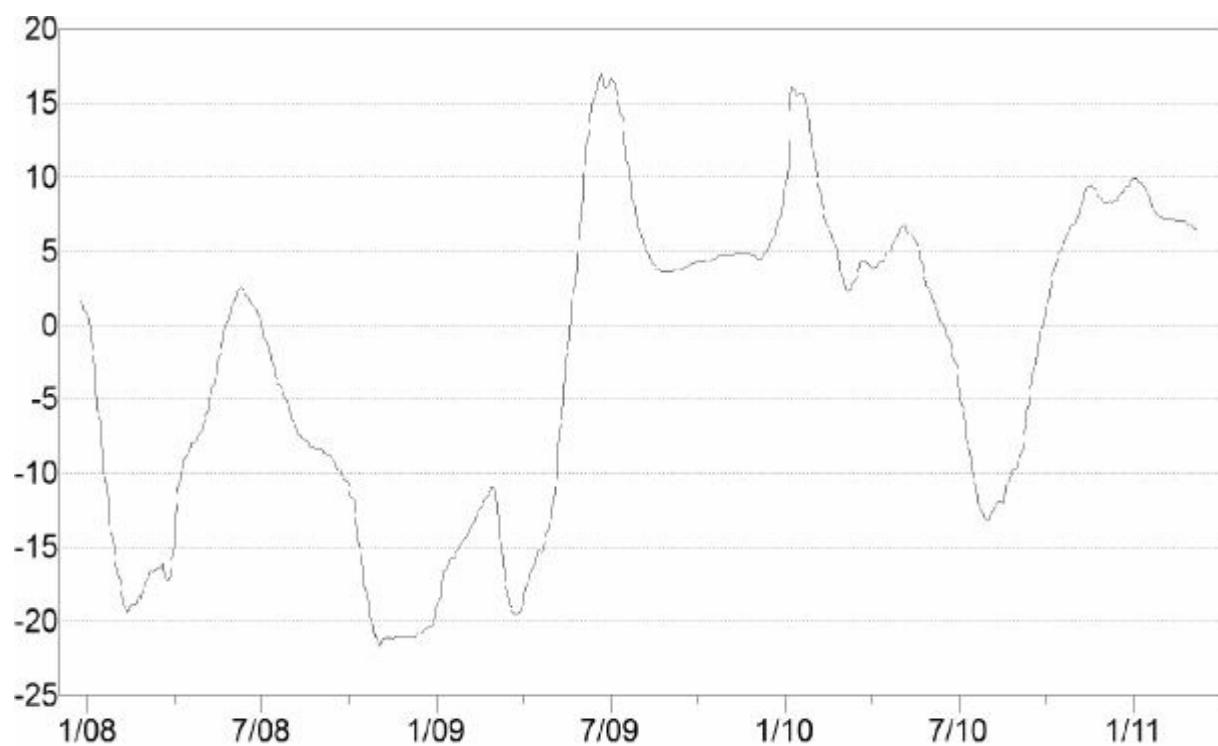
[Figure 6](#) shows the trend variable with the normalize option used. The influence of normalization (centering plus scaling) is prominent.



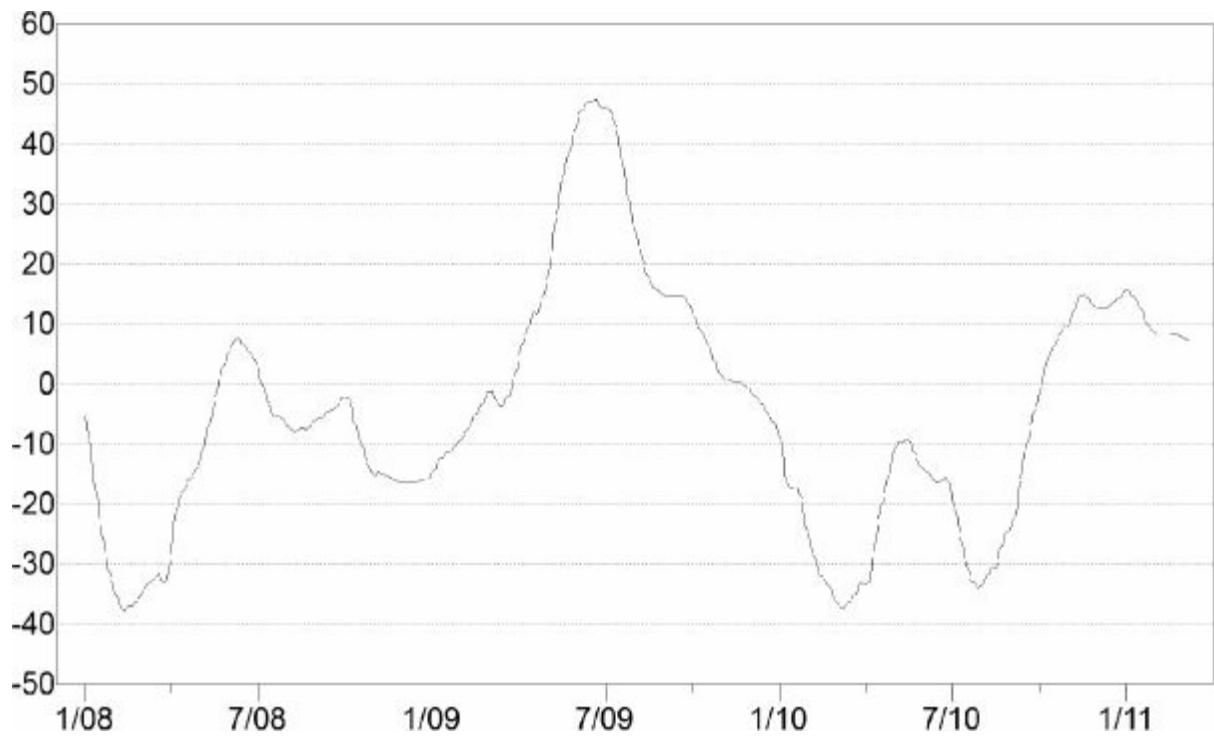
**Figure 3:** A trend indicator



**Figure 4:** Centered trend



**Figure 5:** Scaled trend



**Figure 6:** Normalized trend

## Cross-Market Normalization

When you are trading many markets simultaneously, it can be useful to know how the various markets rank in terms of an indicator. For example, suppose we compute a trend variable for each market, and we are now considering whether to open a position in, say, IBM. If the trend in IBM is the largest among all markets, that may well mean something important. Similarly, if we know that IBM has the minimum trend of all of the markets, that may mean something entirely different. Cross-market normalization ranks markets according to an indicator's value in each. This can be particularly useful in a long/short market-neutral trading system.

The indicator being cross-market normalized (which may or may not also have centering, scaling, or combined normalization as described in the prior section) is computed for every market for which data is available. Each day (or bar), the values are ranked across all of the markets, and the percentile rank for each market is computed. The final value of this variable is defined by subtracting 50 from the percentile. Thus, a cross-market-normalized variable ranges from -50 to 50. The market having minimum value of the variable is assigned a value of -50. The maximum market gets 50. Markets near the middle of the range will obtain values near zero.

Naturally, in order for this rank variable to be defined, we must have at least two markets with valid data for this indicator. Usually we would want many more than that! For this reason, the user specifies a fraction 0-1 which is the minimum fraction of the markets that must be present in order for this value to be computed. Some markets may begin their history later than other markets, or end earlier. Even in the interior, some market may have missing data. For this reason, we cannot count on every market having valid data on every bar. If too few markets are present on a given date/time, cross-market normalization would not have much meaning. Thus, if the minimum fraction is not met, the variable is not computed and is recorded as missing.

In order to invoke cross-market normalization, follow the variable definition with an exclamation point (!) and the minimum fraction of markets required. For example, the first line below computes the cross-normalized value of the LINEAR PER ATR indicator we've seen before, and the second line centers the variable before cross-market normalization. In both cases, we require that at least 60 percent (0.6) of the markets be present on a date in order to produce a valid value.

```
P_TREND : LINEAR PER ATR 20 250 ! 0.6  
P_TREND_C: LINEAR PER ATR 20 250 : CENTER 100 ! 0.6
```

## Pooled Variables

Another technique for extracting information about the behavior of an indicator across multiple markets is to use a *Pooled Variable*. This is done by appending the keyword POOLED followed by the type of pooling to be done, followed by the minimum fraction (0-1) of markets that must be present, as in computing cross-sectional normalization. These specifications must come last on a line, after the variable definition and any normalization.

All pooled variables are scaled and slightly transformed so that in most situations their natural range will be approximately -50 to 50. Pooling and cross-market normalization may not be performed simultaneously. The following pooling types are available:

**MEDIAN** - The median of the indicator across all markets. This provides a general ‘consensus’ value for the indicator in the complete set of all markets.

**IQRANGE** - The interquartile range of the indicator across all markets. This reveals the degree of ‘spread’ of values of this indicator across the universe of markets. A large IQRANGE means that the indicator has a large diversity of values within the set of markets. A small value means that the markets all tend to have about the same value for the indicator.

**SCALED MEDIAN** - The median divided by the interquartile range. Like the MEDIAN, this provides a consensus view of the indicator across markets. However, this consensus is scaled by the variation of the indicator across markets. Thus, if the indicator has a wide spread across markets, the actual value of the median will be scaled back. Conversely, if the indicator has similar values across the markets, the median will be accentuated in computing this variable. This often makes sense, because if a median is far from zero but the individual values for markets are all over the place, the median is less noteworthy than if all markets have about the same value for this indicator.

**SKEWNESS** - A nonparametric measure of skewness. If the indicator has a perfectly symmetric distribution, this value will be zero. Positive SKEWNESS means that there are a preponderance of relatively large positive values of the indicator, while negative SKEWNESS means that large negative values prevail. So, for example, suppose that on some day the SKEWNESS of a trend indicator is very large.

This means that most markets had similar trends, but there were a few markets that had unusually large positive trends, and there were few or no markets with an unusually small trend.

**KURTOSIS** - A nonparametric measure of kurtosis. If the indicator has a roughly normal bell-shaped curve, the value of this will be near zero. A large KURTOSIS means that one or both tails of the indicator are unusually heavy (numerous outliers). A small KURTOSIS means that the distribution of values of this indicator across markets is compact, with few or no ‘oddball’ markets that stand out from the crowd. KURTOSIS says nothing about the width of the distribution. It measures only the degree to which extreme values outside the majority range of the indicator are present.

**CLUMP60** - If the 40'th percentile is positive (meaning that at least 60 percent of the markets have a positive value of this indicator) CLUMP60 is the 40'th percentile. If the 60'th percentile is negative, CLUMP60 is the 60'th percentile. Otherwise it is zero. This variable measures the degree to which the markets are moving in conformity. This variable is discussed in detail later in this section.

## MEDIAN pooling

Here is a quick example of using the MEDIAN pooling option. With the exception of the unusual CLUMP60 pooling which will be covered separately, the other options should be clear once MEDIAN pooling is understood. Once again we will use the LINEAR PER ATR trend variable that appeared in prior examples. Suppose the following two variable definitions appear in a situation involving three markets:

```
P_TREND:    LINEAR PER ATR 20 250  
P_TREND_M:  LINEAR PER ATR 20 250 POOLED MEDIAN 0.6
```

We may see that the generated database includes records that look like this:

Date	Market	P_TREND	P_TREND_M
20010601	IBM	5.2	7.3
20010601	BOL	7.3	7.3
20010601	IFF	9.1	7.3
20010602	IBM	6.2	6.2
20010602	BOL	4.4	6.2
20010602	IFF	6.5	6.2
20010603	IBM	5.5	6.8
20010603	BOL	7.2	6.8
20010603	IFF	6.8	6.8

Notice that for each date, the value of P\_TREND\_M is the same in all markets. It is the median of P\_TREND in the three markets.

## CLUMP60 Pooling

MEDIAN pooling provides a general consensus of what an indicator is doing in the universe of markets being studied. The median is the ‘center’ value of the indicator across these markets. Thus, if the median is positive, we know that at least half of the markets have a positive value of the indicator, and probably more than half of them do, because there likely are some markets whose value of the indicator is less than the median but greater than zero. Similarly, if the median is negative, we know that at least half of the markets have a negative value of the indicator, and probably more than half of them do.

The problem with using the median as a consensus figure for the universe is the word ‘probably’ in the prior paragraph. Even if we obtain a large positive median, we have no guarantee that more than half of the markets have a positive value of the indicator, and similarly for a negative median. The markets may well be split 50-50 between positive and negative. If our goal in pooling is to detect an informative consensus among the markets, MEDIAN pooling may not be adequate. A 50-50 split is hardly a consensus!

An obvious solution to this problem is to create a ‘dead zone’ in the area around a 50-50 split, so that if there is little or no consensus, the value of the pooled variable will be zero. This way, if we obtain a nonzero value for the pooled variable, we know that the split is outside the dead zone, well away from a tie between positive and negative values of the indicator. There is a significant degree of consensus among the markets.

CLUMP60 pooling accomplishes this. The number 60 in the name means that the dead zone is sized so that the pooled variable will have a nonzero value only if at least 60 percent of the markets have the same sign on the indicator for which CLUMP60 is being computed. This is easy to do:

If the 40'th percentile of the indicator is positive, then by definition at least 60 percent of the markets have a positive value of the indicator. So in this case we set the CLUMP60 pooled variable equal to the 40'th percentile of the indicator across markets. The positive sign signifies that a substantial majority of the markets have positive values for this indicator, and the magnitude of CLUMP60 gives an indication of the magnitude of the indicators.

Similarly, if the 60'th percentile of the indicator is negative, then by definition at least 60 percent of the markets have a negative value of the indicator. So in this case we set the CLUMP60 pooled variable equal to the 60'th percentile of the indicator across markets. The negative sign signifies that a substantial majority of the markets have negative values for this indicator, and the magnitude of CLUMP60 gives an indication of the magnitude of the indicators.

Finally, if neither of the above conditions are true, then the distribution of indicator values across markets is in the dead zone, near a tie, with no clear consensus. In this case we set the CLUMP60 pooled variable to zero to flag the lack of consensus.

## Mahalanobis Distance

A surprising world event or some other destabilizing influence can cause a sudden dramatic change in market behavior. This can be measured for an individual market by a spike in a traditional volatility indicator. But in a multiple-market scenario we can compute a more sophisticated turbulence measure by taking into account not only how much market prices change relative to their historical norm, but also by how their interrelationships change. For example, suppose two markets normally move together. If they suddenly diverge, this may indicate a major market upheaval of some sort.

A standard mathematical technique for quantifying change in terms of not only individual components but interrelationships as well is the *Mahalanobis Distance* that separates a vector (in this context, a set of market price changes as of a single bar) from its historical mean, with historical covariance taken into account. This can be accomplished for any variable by following the variable definition with the keyword MAHALANOBIS and the number of bars to look back when computing the mean and covariance.

In Kritzman and Li (“Skulls, Financial Turbulence, and Risk Management” in *Financial Analysts Journal*, September/October 2010, Vol. 66), Mahalanobis distance is based on daily price changes, which is the most intuitive choice. In this case you would use the CLOSE TO CLOSE variable. For example, suppose you are working with daily data, and you want a mean/covariance lookback window of approximately one year. The variable definition line be this:

**YEAR\_MAHAL: CLOSE TO CLOSE MAHALANOBIS 250**

Don't feel limited to daily price changes, though. Other indicators, such as trend or volatility, may prove to be useful raw material for measuring turbulence. Here is a short example of how this variable might be generated. For this example, assume that these two market have been highly correlated over the window period.

Date	Market	Change	Mahal
19950601	C	23.1	0.21
19950601	JPM	21.4	0.21
19950602	C	10.3	0.32
19950602	JPM	9.2	0.32
19950603	C	-11.5	0.25
19950603	JPM	-12.7	0.25
19950604	C	-14.9	48.60
19950604	JPM	8.8	48.60

Observe that the Mahalanobis distance is small when the markets are moving

together, and it jumps when the markets move in opposite directions.

## Absorption Ratio

Many experts believe that financial markets are most stable when the individual components are moving relatively independently. At such times investors tend to be making decisions independently, and market efficiency is likely to be high. However, when investors as a group become focused on the same market driving factors, independent decision-making declines and herding emerges. During such periods, when the components become locked together in consistent patterns, a situation called *coherence*, the market becomes unstable.

The degree of market coherence can be measured by examining the eigenvalues of the covariance matrix of the market components. If all components are completely independent, so that all correlations are zero, the eigenvalues are equal. As correlation among components increases, variance tends to shift so that it is concentrated in some eigenvalues and depleted in others. In the extreme, if all components move exactly together (all correlations are plus or minus one), all variance is contained in one eigenvalue; the other eigenvalues are zero. Intermediate degrees of coherence cause degrees of concentration of variance in few eigenvalues.

This leads to an intuitive and effective way to measure coherence: compute the fraction of the total variance that is contained in the few largest eigenvalues. We must specify in advance the number of largest eigenvalues that will be kept. The fraction of the total variance contained in these few kept eigenvalues will range from a minimum of *number kept/number of markets* when all components are independent, to 1.0 when all components move in perfect lockstep. ABSRATIO (absorption ratio) is a family of variables to accomplish this, and its close relative *Absorption Shift* may do so as well.

In order to compute the absorption ratio or absorption shift for a variable, follow the variable definition with the keyword ABSRATIO and then the following four parameters:

- The number of bars to include in the moving window used to compute the covariance matrix.
- The fraction (0-1) of the eigenvalues to use for computing the absorption ratio. Kritzman et al use 0.2.
- The short-term moving-average lookback for computing absorption shift (discussed soon), or zero to compute the absorption ratio.
- The long-term moving-average lookback for computing absorption shift (discussed soon), or zero to compute the absorption ratio.

If those last two parameters, the moving-average lookbacks, are zero, the value computed is the absorption ratio. If they are nonzero, the variable is what Kritzman et al call the *Absorption Shift*. This is the short-term moving average of the absorption ratio, minus the long-term, scaled by the standard deviation of the absorption ratio during the long-term time period. Note that it is illegal for one of the lookbacks to be zero but not the other.

Any predefined *TSSB* variable can be used to compute the absorption ratio, but in order to implement the algorithm of Kritzman et al, the CLOSE TO CLOSE variable should be used. Here are two examples. Both use a lookback window of 250 bars and the fraction 0.2 of the eigenvalues. The first example computes the absorption ratio, and the second computes the shift using a short-term MA of 20 bars and a long-term MA of 100 bars.

```
YEAR_ABSRT: CLOSE TO CLOSE ABSRATIO 250 0.2 0 0  
YEAR_SHIFT: CLOSE TO CLOSE ABSRATIO 250 0.2 20 100
```

Note that in order to compute the absorption ratio for a case, valid data for all markets must be present for every date in the lookback period. This is a severe restriction. There are two major ways in which the lookback period might have missing data:

- 1) Markets might exist for non-overlapping time periods. For example, suppose you are processing the components of an index such as the S&P 100. Many of its current components did not exist a few years ago. Thus, the first date on which all markets could have data will be the date on which the most recently ‘born’ market has its first price information. This could easily leave only a small subset of the data that would otherwise be available. For this reason it is best to check the starting and ending dates of all markets and process only markets that have substantial overlap. All dates outside the time period in which *all* markets have data will be excluded.
- 2) The CLEAN RAW DATA command ([here](#)) will eliminate data which has a suspiciously large bar-to-bar price jump. If even one market has missing data due to this command for any date in the lookback period behind the current date, the absorption ratio will not be computed for the current date. For this reason, the CLEAN RAW DATA command should use as small a parameter value as possible, perhaps 0.4 or even less.

## Trend Indicators

This section discusses indicators that respond to trends in the market price. In all cases, the relationship is monotonic; large values correspond to upward trends and small values correspond to downward trends. Most but not all of these are shifted, scaled, and compressed in such a way that they have a natural range of about -50 to 50. Most of these indicators are self-explanatory. The more complex indicators are explained in detail in later supplementary sections.

### MA DIFFERENCE ShortLength LongLength Lag

A short-term moving average and a long-term moving average, the latter lagged by the specified amount, are computed for closing prices. Typically, the lag will equal the short length, thus making the two windows disjoint, although many users will want to set the lag to zero so that the windows overlap. The long-term MA is subtracted from the short-term MA, and this difference is divided by the average true range measured across the *LongLength+Lag* history. Finally, this normalized difference is slightly compressed to a range of -50 to 50. For example:

**MADIFF: MA DIFFERENCE 10 100 10**

The preceding definition creates a moving average difference with a short-term lag of 10 bars and a long-term lag of 100 bars. The long-term moving average is lagged by 10 bars so that it does not overlap the short-term average. Thus, the total lookback period is 110 bars.

### LINEAR PER ATR HistLength ATRlength

This indicator computes a least-squares straight line over the specified length of historical data. The data which is fit is the log of the mean of the open, high, low, and close. It also computes the Average True Range over the specified ATR history length. The returned value is the slope of the line divided by the ATR. This is sometimes called price velocity. A small amount of compression and rescaling is applied to restrain values to the interval -50 to 50. For example:

**LINFIT: LINEAR PER ATR 50 250**

The preceding definition fits a least-squares straight line to the most recent 50 bars and divides the slope by the ATR for the most recent 250 bars.

## QUADRATIC PER ATR HistLength ATRlength

This indicator computes a least-squares fit of a second-order Legendre polynomial (an orthogonal family) over the specified length of historical data. The data which is fit is the log of the mean of the open, high, low, and close. It also computes the Average True Range over the specified ATRhistory length. The returned value is the (quadratic) polynomial coefficient divided by the ATR. This is sometimes called price acceleration. A small amount of compression and rescaling is applied to restrain values to the interval -50 to 50. The quadratic coefficient measures the *change in trend* of historical prices. This value will be positive if the trend is increasing and negative if the trend is decreasing. Note that this indicator is not affected by the trend itself, only the *change* in the trend.

For example:

**QUADFIT: QUADRATIC PER ATR 50 250**

The preceding definition fits a least-squares quadratic polynomial to the most recent 50 bars and divides the coefficient by the ATR for the most recent 250 bars.

## CUBIC PER ATR HistLength ATRlength

This indicator computes a least-squares fit of a third-order Legendre polynomial (an orthogonal family) over the specified length of historical data. The data which is fit is the log of the mean of the open, high, low, and close. It also computes the Average True Range over the specified ATR history length. The returned value is the (cubic) polynomial coefficient divided by the ATR. A small amount of compression and rescaling is applied to restrain values to the interval -50 to 50. This indicator measures the rate of change in acceleration, the rate at which the quadratic term is changing. See QUADRATIC PER ATR. Note that this indicator is independent of trend (price velocity), as well as the quadratic term (acceleration, or rate of change in trend). For example:

**CUBEFIT: CUBIC PER ATR 50 250**

The preceding definition fits a least-squares cubic Legendre polynomial to the most recent 50 bars and divides the coefficient by the ATR for the most recent 250 bars.

## RSI HistLength

This is the ordinary Relative Strength Indicator proposed by J. Welles Wilder, Jr. It is computed as described in *The Encyclopedia of Technical Market Indicators* by Colby and Meyers, with one exception. That reference uses an ordinary moving average for smoothing, while most modern references claim that exponential smoothing give superior results, so that's what is done here. Actually, it's hard to tell the two apart, so in reality the difference is probably inconsequential inmost applications. For example:

```
RSI_WEEK: RSI 5
```

The preceding definition computes the RSI for the most recent five bars. No transformation of any sort is done; this produces the ordinary RSI which varies from zero through one hundred. Note that because exponential smoothing is done, the CLEAN RAW DATA command ([here](#)) is ignored. This is because exponential smoothing has a very long (theoretically infinite) lookback distance.

## STOCHASTIC K HistLength

## STOCHASTIC D HistLength

These are the first-order smoothed (K) and second-order smoothed (D) Lane Stochastics as proposed by George C. Lane. The algorithm used here is from *The Encyclopedia of Technical Market Indicators* by Colby and Meyers. For example:

```
STO_K_YEAR: STOCHASTIC K 250
```

The preceding definition computes the K stochastic looking back approximately one year (250 days, assuming we have day bars). No transformation is done; this produces the ordinary stochastic which varies from zero through one hundred. Note that because a form of exponential smoothing is done, the CLEAN RAW DATA command ([here](#)) is ignored. This is because exponential smoothing has a very long (theoretically infinite) lookback distance.

## PRICE MOMENTUM HistLength StdDevLength

This measures the price today relative to the price *HistLength* days ago (or bars, for intraday data). It is normalized by the standard deviation of daily price changes. A small amount of compression and rescaling is applied to restrain

values to the interval -50 to 50. For example:

**PMOM: PRICE MOMENTUM 20 250**

The preceding definition computes the price momentum over the most recent 20 bars and then normalizes it by the standard deviation of the most recent 250 bars.

## ADX HistLength

The ADX trend indicator proposed by J. Wells Wilder is computed for the specified history length. Unlike most other indicators in the *TSSB* library, ADX is neither scaled nor transformed. It retains its defined range of 0-100. Remember that ADX does not indicate the direction of a trend, only its strength (degree of directionality or persistence). For example:

**ADX\_MONTH: ADX 21**

The preceding definition computes the ADX trend strength indicator for the most recent 21 bars, which is approximately a month for day bars.

## MIN ADX HistLength MinLength

The ADX trend indicator with a lookback of *HistLength* is computed for the current bar as well as prior bars, for a total of *MinLength* times. The minimum value across these times is found. For example:

**MIN\_AXD\_M2M: MIN ADX 21 42**

The preceding definition computes the 21-bar ADX trend strength indicator for the most recent bar, and the second-most recent, and the third-most recent, et cetera, a total of 42 times. The minimum of the current value and the 41 historic values is found.

## RESIDUAL MIN ADX HistLength MinLength

The ADX trend indicator with a lookback of *HistLength* is computed for the current day (or bar for intraday data) as well as prior days, for a total of *MinLength* times. The minimum value across these times is found. This is the

MIN ADX as described above. The final variable is today's ADX minus the minimum. For example:

```
RESMIN_ADX_M2M: RESIDUAL MIN ADX 21 42
```

The preceding definition first computes the MIN ADX indicator as described [here](#). It then computes ADX for the current bar and subtracts the former from the latter.

## MAX ADX HistLength MaxLength

The ADX trend indicator with a lookback of *HistLength* is computed for the current bar as well as prior bars, for a total of *MaxLength* times. The maximum value across these times is found. For example:

```
MAX_ADX_M2M: MAX ADX 21 42
```

The preceding definition computes the 21-bar ADX trend strength indicator for the most recent bar, and the second-most recent, and the third-most recent, et cetera, a total of 42 times. The maximum of the current value and the 41 historic values is found.

## RESIDUAL MAX ADX HistLength MaxLength

The ADX trend indicator with a lookback of *HistLength* is computed for the current day as well as prior days, for a total of *MaxLength* times. The maximum value across these times is found. This is the MAX ADX as described above. The final variable is the maximum minus today's ADX. For example:

```
RESMAX_ADX_M2M: RESIDUAL MAX ADX 21 42
```

The preceding definition first computes the MAX ADX indicator as described [here](#). It then computes ADX for the current bar and subtracts the latter from the former.

## DELTA ADX HistLength DeltaLength

The ADX trend indicator with a lookback of *HistLength* is computed for the current bar and for the bar *DeltaLength* ago. The final variable is the former

minus the latter, slightly transformed and scaled to a range of -50 to 50. This measures the rate of change of ADX, its velocity. For example:

```
ADXVEL: DELTA ADX 21 10
```

The preceding definition computes the 21-bar ADX for today and for 10 days ago. It subtracts the lagged value from the current value and slightly transforms/compresses the result.

## ACCEL ADX HistLength DeltaLength

The ADX trend indicator with a lookback of *HistLength* is computed for today and for the day *DeltaLength* ago and for the day  $2 * \text{DeltaLength}$  ago. The final variable is today's value plus the doubly lagged value, minus twice the single-lagged value, slightly transformed and scaled to a range of -50 to 50. This measures the curvature or acceleration of the ADX function, the rate at which ADX velocity is changing. For example:

```
ADXACC: ACCEL ADX 21 8
```

The preceding definition computes the 21-bar ADX for today, for 8 days ago, and for 16 days ago. It doubles the lag-8 value and subtracts that from the sum of the current and lag-16 values. Finally, it slightly transforms/compresses the result.

## INTRADAY INTENSITY HistLength

The smoothed intraday intensity statistic is returned. (The concept underlying this indicator was proposed by David Bostian. The version is TSSB is a modified version.) First, the true range is computed as the greatest of (today's high minus today's low), (today's high minus yesterday's close), and (yesterday's close minus today's low). (For intraday data, these quantities refer to bars, not days.) Then today's change is computed as today's close minus today's open. The intraday intensity is defined as the ratio of the latter to the former. This quantity is computed over the specified history length, and the moving average returned after being slightly transformed and rescaled to the range -50 to 50. For example:

```
INTENS_20: INTRADAY INTENSITY 20
```

The preceding definition computes the 20-bar-smoothed intraday intensity and

returns the transformed/compressed result. Note that out of deference to tradition, this definition referred to ‘day’ bars, but it applies to any bar definition.

## DELTA INTRADAY INTENSITY HistLength DeltaLength

This is the difference between today’s *INTRADAY INTENSITY* and that of the specified number of days ago. For example:

**D\_INTENS\_20\_10: DELTA INTRADAY INTENSITY 20 10**

The preceding definition computes the *INTRADAY INTENSITY* with a 20-bar moving-average window, and also does the same for 20 bars ago. The latter is subtracted from the former.

## REACTIVITY HistLength

This is the *REACTIVITY* technical indicator computed over the specified history length. It is slightly compressed and transformed to the range -50 to 50. Reactivity is primarily used when trading cycles. The *HistLength* lookback period should be approximately half of the period of the cycle the trader is exploiting. The reactivity is the price change over the history length, normalized by a function of the smoothed range and volume. For example:

**REAC\_20: REACTIVITY 20**

The preceding definition computes reactivity over the most recent 20 bars, which will capture information about the status of the market in regard to a 40-bar cycle.

The Reactivity Indicator was proposed by Gietzen as part of trading approach described in Advanced Cycle Trading (Irwin Professional Publishing, 1995). It combines price rate-of-change (PROC) with price range and trading volume to overcome a deficiency associated with rate-of-change indicators based on price alone. PROC extremes (over-bought and over-sold) signal peaks and troughs in price during non-trending periods. However, during strong trends PROC extremes are premature. Gietzen’s intuition is that when the reactivity indicator exceeds a critical high or low threshold, a strong up or down trend is in effect and is therefore more likely to continue than reverse. At such times PROC extremes should be ignored.

To define reactivity, a look-back window (*HistLength*) must be specified. It should be approximately one half of the duration of the cycle the trader is interested in exploiting. For example, if the trading cycle's duration is estimated to be 24 days, *HistLength* would set to 12. *HistLength* is used to compute the three factors that are combined to produce reactivity: price change, price range, and trading volume. These factors are defined as follows:

**Price Change** is the most recent value of the closing price minus the closing price *HistLength* bars ago.

**Price Range** is the maximum price bar high minus the minimum price bar low determined over *HistLength*.

**Trading Volume** is the total volume occurring during *HistLength*.

Price range and trading volume can be combined into a single quantity call the aspect ratio, defined as Price Range / Trading Volume. The aspect ratio is inspired by the work of Richard Arms discussed in Volume Cycles in the Stock Market (Dow Jones Irwin, 1983). However, the raw ratio is problematic for two reasons: (1) range and volume both vary considerably over time. A value that would be considered large at one time might be insignificant at another time, and (2) the quantities range and volume are denominated in different units, points and number of shares. To solve both problems, Gietzen suggests normalizing range and volume by their smoothed averages. The normalized price range is the price range divided by its smoothed value, and the normalized volume is the volume divided by its smoothed value. This modified aspect ratio is defined as follows:

$$\text{AspectRatio} = \frac{\text{Range} / \text{SmoothedRange}}{\text{Volume} / \text{SmoothedVolume}} \quad (3)$$

Gietzen uses exponential smoothing, and his heuristic for choosing a smoothing constant is a time period of four times the length of the trading cycle. Thus a trading cycle of 25 days calls for the exponential smoothing equivalent to a 100 day moving average. In particular, if  $n$  is the number of days of a simple moving average, the (very roughly) equivalent exponential smoothing constant is  $2 / (n+1)$ . In this example of a 100-day moving average, the exponential smoothing constant would be  $2/101 \approx 0.02$ .

The aspect ratio (which is a dimensionless constant) is multiplied by the price change ( $M$ ) computed over *HistLength*, which is denominated in price points. This causes raw reactivity to be scaled in price points. In particular:

$$M = Price_0 - Price_{HistLength} \quad (4)$$

$$RawReactivity = M * AspectRatio \quad (5)$$

To obtain a dimensionless (pure) value for reactivity, Gietzen suggests dividing raw reactivity by the same exponentially smoothed range used above to obtain the aspect ratio. Thus Reactivity is defined as follows:

$$Reactivity = RawReactivity / SmoothedRange \quad (6)$$

The resulting pure number is interpreted as follows: values greater than one indicate a strong uptrend, values less than minus one indicate a strong downtrend, and intermediate values indicate a trading range. However, *TSSB* applies a compressing transform to the traditional reactivity to rescale it a range of -50 to 50 and thereby improve its compatibility with other indicators in the built-in library, so the original interpretation does not apply.

## DELTA REACTIVITY HistLength DeltaDist

This is the *REACTIVITY* computed over the *HistLength* most recent bars, minus the same quantity computed for the bar *DeltaDist* bars earlier. It is slightly compressed and transformed to the range -50 to 50. For example:

**DREACT\_20\_15: DELTA REACTIVITY 20 15**

The preceding definition computes REACTIVITY using a lookback of the most recent 20 bars, as well as the same thing at a lag of 15 bars. The latter is subtracted from the former and slightly compressed and transformed to a uniform range.

## MIN REACTIVITY HistLength Dist

This is the minimum of *HistLength REACTIVITY* with the minimum taken over *Dist* days. It is slightly compressed and transformed to the range -50 to 50. For example:

**MINREAC: MIN REACTIVITY 20 15**

The preceding definition computes 'REACTIVITY 20' for the most recent 20 bars, 20 bars lagged one bar, 20 bars lagged two bars, and so forth. A total of 15 such computations are done. The indicator is the minimum of these 15 values.

## MAX REACTIVITY HistLength Dist

This is identical to MIN REACTIVITY except that the maximum is taken instead of the minimum.

## Trend-Like Indicators

The prior section presented indicators that depict multiple-bar price trends in obvious ways. This section covers indicators whose relationship to trend is close but not quite so obvious.

### CLOSE TO CLOSE

This rapidly changing indicator is 100 times the log ratio of the current bar's close to the prior bar's close. It would rarely, if ever, be used as an input to a prediction model. It is extremely unstable and nonstationary. Also, it has very poor cross-market conformity (its distribution varies widely for different markets). However, it is the most common indicator used for Mahalanobis Distance ([here](#)) and Absorption Ratio ([here](#)) computation. See those sections for examples of its use.

### N DAY HIGH HistLength

Let  $N$  be the number of bars one has to go back in order to find a price higher than the current bar's high. Search only  $HistLength$  bars back from the current bar. If this many bars are examined and no higher high is found, set  $N=HistLength+1$ . Then this variable is defined as  $100 * (N-1) / HistLength - 50$ . Note that this variable is highly unstable. Suppose that over the most recent  $HistLength$  bars the highs have steadily increased. Then this indicator will attain its maximum value of 50. But now suppose the same steady increase is true, except that the current bar's high drops off by one tick lower than the prior bar's high. It may even be that this bar closes higher than the prior bar's close. Nevertheless, this indicator will attain its minimum value of -50!

For example:

#### **NDH10: N DAY HIGH 10**

The preceding definition examines the most recent 10 bars. It compares the high of the current bar to the highs of the prior 10 bars. If the immediately prior high is greater than the current bar's high, the value of this indicator is -50. If none of the prior highs exceed that of the current bar, the value of this indicator is 50. Intermediate locations of the most recent higher high will result in intermediate values of this indicator.

## N DAY LOW HistLength

This is identical to N DAY HIGH except that we search for lower prices. For example:

**NDL10: N DAY LOW 10**

The preceding definition examines the most recent 10 bars. It compares the low of the current bar to the lows of the prior 10 bars. If the immediately prior low is less than the current bar's low, the value of this indicator is -50. If none of the prior lows is less than that of the current bar, the value of this indicator is 50. Intermediate locations of the most recent lower low will result in intermediate values of this indicator.

## Deviations from Trend

The prior section discussed indicators that capture trend information. This section presents indicators that quantify the degree to which the current value of the market deviates from recent trend. We also include in this category the indicator that compares the current price to the moving average, because this is a deviation that is similar to deviations from trend.

### CLOSE MINUS MOVING AVERAGE HistLen ATRlen

This indicator measures today's market price relative to its recent history, normalized by recent average true range. The close of the current bar is divided by the current moving average. The log of this ratio is divided by the average true range (based on log changes, of course). The result is transformed and slightly compressed to a range of -50 to 50. *HistLen* is the lookback period for the moving average, and *ATRlen* is the lookback period for the ATR calculation. For example:

**CMMA\_10: CLOSE MINUS MOVING AVERAGE 10 250**

The preceding definition computes the average close of the most recent 10 bars and divides the close of the current bar by this average. The log of this ratio is found, and this is divided by the average true range of the most recent 250 bars. The result is transformed and compressed to a uniform range.

### LINEAR DEVIATION HistLength

A least-squares line is fit to the most recent *HistLength* log prices, including that of today (or the current bar, if this is intraday data). The data which is fit is the log of the mean of the open, high, low, and close. The returned value is today's price minus the fitted line's value for today, divided by the standard error of the fit. A modest compressing transform is applied to limit the range to the interval -50 to 50. For example:

**LINDEV\_20: LINEAR DEVIATION 20**

The preceding definition fits a straight line to the most recent 20 bars. This provides an estimate of what the price would be for the current bar if the trend line were exactly followed. This quantity is subtracted from the current price in order to quantify the degree to which the current price deviates from the trend

line.

## QUADRATIC DEVIATION HistLength

A least-squares quadratic polynomial is fit to the most recent *HistLength* log prices, including that of today. The data which is fit is the log of the mean of the open, high, low, and close. The returned value is today's price minus the fitted line's value for today, divided by the standard error of the fit. A modest compressing transform is applied to limit the range to the interval -50 to 50. This indicator is similar to LINEAR DEVIATION except that curvature (a second-order term) is allowed in the fit. For example:

### **QUADDEV\_20: QUADRATIC DEVIATION 20**

The preceding definition does a least-squares fit of a parabola (a quadratic polynomial) to the most recent 20 bars. This provides an estimate of what the price would be for the current bar if the curving trend line were exactly followed. This quantity is subtracted from the current price in order to quantify the degree to which the current price deviates from that predicted by the parabolic fit.

## CUBIC DEVIATION HistLength

A least-squares cubic (third degree) polynomial is fit to the most recent *HistLength* log prices, including that of today. The data which is fit is the log of the mean of the open, high, low, and close. The returned value is today's price minus the fitted line's value for today, divided by the standard error of the fit. A modest compressing transform is applied to limit the range to the interval -50 to 50. This indicator is similar to LINEAR DEVIATION except that curvature of second and third order is allowed in the fit. For example:

### **CUBEDEV\_20: CUBIC DEVIATION 20**

The preceding definition does a least-squares fit of a cubic polynomial to the most recent 20 bars. This provides an estimate of what the price would be for the current bar if the curving trend line were exactly followed. This quantity is subtracted from the current price in order to quantify the degree to which the current price deviates from that predicted by the cubic fit.

## DETRENDED RSI DetrendedLength DetrenderLength Lookback

Ordinary RSI at the two specified lengths are computed and a least-squares regression line is fit for predicting RSI at *DetrendedLength* from RSI at *DetrenderLength*. This least-squares fit is based on these pairs of RSIs over the specified *Lookback* period. The value returned is the *DetrendedLength* RSI minus its predicted value. The only exception is that when *DetrendedLength* is 2, an inverse logistic function is applied to that RSI, and all computations (linear fit and computation of residual) are based on this transformed value. This function approximately linearizes what is otherwise a highly nonlinear relationship. No other compression or transformation is done. Note that *DetrendedLength* must be less than *DetrenderLength*. For example:

**DETRSI: DETRENDED RSI 4 20 50**

The preceding definition causes RSI at lengths of 4 and 20 to be computed for the 50 most recent bars. A least-squares line is computed for predicting 'RSI 4' from 'RSI 20' using these 50 pairs of RSI values. The current value of 'RSI 20' is plugged into this linear equation to predict the value of 'RSI 4' at the current bar. This indicator is then defined as the actual value of 'RSI 4' at the current bar minus its predicted value. In this way we can assess the direction and degree to which short-term RSI is currently deviating from its longer-term trend.

The intuition behind this indicator is the widely recognized tendency for RSI to oscillate in a range that reflects the longer-term trend. During non-trending periods the typical extremes of RSI are 30 and 70. However, in a strong long-term up-trend RSI tends to be contained in the range 40 to 80 while in strong long-term down-trends the fluctuation bounds are 20 to 60.

## Volatility Indicators

An enormous number of popular measures of market volatility exist. This section presents many of them, as well as a few relatively obscure indicators. There is a common thread running through most of them: they are not absolute quantities. Rather, they measure recent volatility relative to ‘historical norms’ of volatility. There are at least two reasons for this:

- 1) Markets often experience very long-term, slow evolution in volatility. Thus, the same absolute volatility may be ‘high’ when it occurs in a low-volatility period of history, and ‘low’ when it occurs in a high-volatility period of history.
- 2) *TSSB* is often called upon to work with datasets that pool multiple markets. Different markets often have inherently different volatilities. If we measured absolute volatility, we would not have conformity of meaning across markets, which would make it impossible to find universally effective prediction models. By measuring current volatility in the context of ‘normal’ volatility for each market, we can achieve good conformity across markets.

## ABS PRICE CHANGE OSCILLATOR ShortLen Multiplier

A *ShortLen* moving average of absolute log daily price changes is computed. The same is done for a length of *ShortLen* times *Multiplier*. The long-term MA is subtracted from the short-term MA, and this difference is divided by the average true range measured across the longer history. Finally, this normalized difference is transformed and slightly compressed to a range of -50 to 50. For example:

**PCO\_10\_20: ABS PRICE CHANGE OSCILLATOR 10 20**

The preceding definition computes the mean of the absolute values of log of closing price changes (ratio of current to prior) over the most recent 10 bars and the most recent 200 bars. The latter is subtracted from the former, divided by the average true range across the most recent 200 bars, and then transformed and scaled to a uniform range.

## PRICE VARIANCE RATIO HistLength Multiplier

This is the ratio of the variance of the log of closing prices over a short time period to that over a long time period. It is transformed and scaled to a range of

-50 to 50. The short time period is specified as *HistLength*, and the long time period is *HistLength* times *Multiplier*. For example:

```
PVR_10_20: PRICE VARIANCE RATIO 10 20
```

The preceding definition computes the variance of the log of closing prices over the most recent 10 bars and the most recent 200 bars. The former is divided by the latter, and then transformed and scaled to a uniform range.

## MIN PRICE VARIANCE RATIO HistLen Mult Mlength

This is the minimum of the *PRICE VARIANCE RATIO* over the prior *Mlength* observations. For example:

```
MNPVR: MIN PRICE VARIANCE RATIO 10 20 50
```

The preceding definition computes "PRICE VARIANCE RATIO 10 20" for the most recent bar, the second-most recent, third-most recent, and so forth, for a total of 50 bars back in history. The value produced is the minimum of these 50 quantities.

## MAX PRICE VARIANCE RATIO HistLen Mult Mlength

This is identical to MIN PRICE VARIANCE RATIO except that the maximum is taken.

## CHANGE VARIANCE RATIO HistLength Multiplier

This is identical to *PRICE VARIANCE RATIO* except that the quantity whose variance is computed is the log ratio of each day's close to that of the prior day. In other words, this is operating on changes rather than prices. For example:

```
CVR_10_20: CHANGE VARIANCE RATIO 10 20
```

The preceding definition computes the variance of the log of closing price changes over the most recent 10 bars and the most recent 200 bars. The former is divided by the latter, and then transformed and scaled to a uniform range.

## MIN CHANGE VARIANCE RATIO HistLen Mult Mlen

This is the minimum of the *CHANGE VARIANCE RATIO* over the prior *Mlength* observations.

**MNCVR: MIN PRICE CHANGE RATIO 10 20 50**

The preceding definition computes "CHANGE VARIANCE RATIO 10 20" for the most recent bar, the second-most recent, third-most recent, and so forth, for a total of 50 bars back in history. The value produced is the minimum of these 50 quantities.

## MAX CHANGE VARIANCE RATIO HistLen Mult Mlength

This is identical to MIN CHANGE VARIANCE RATIO except that the maximum is taken.

## ATR RATIO HistLength Multiplier

This is the ratio of the Average True Range over a short time period to that over a long time period. It is transformed and scaled to a range of -50 to 50. The short time period is specified as *HistLength*, and the long time period is *HistLength* times *Multiplier*. For example:

**ATTRAT\_10\_20: ATR RATIO 10 20**

The preceding definition computes the Average True Range over the most recent 10 bars and the most recent 200 bars. The former is divided by the latter, and then transformed and scaled to a uniform range.

## DELTA PRICE VARIANCE RATIO HistLength Multiplier

This is the difference between the *PRICE VARIANCE RATIO* for the current bar minus that *HistLength* times *Multiplier* bars ago. For example:

**DPVR: DELTA PRICE VARIANCE RATIO 10 20**

The preceding definition computes "PRICE VARIANCE RATIO 10 20" for the

current bar as well as for that 200 bars earlier in history. The latter is subtracted from the former.

## DELTA CHANGE VARIANCE RATIO HistLength Multiplier

This is the difference between the *CHANGE VARIANCE RATIO* for the current bar minus that *HistLength* times *Multiplier* days ago. For example:

```
DCVR: DELTA CHANGE VARIANCE RATIO 10 20
```

The preceding definition computes "CHANGE VARIANCE RATIO 10 20" for the current bar as well as for that 200 bars earlier in history. The latter is subtracted from the former.

## DELTA ATR RATIO HistLength Multiplier

This is the difference between the *ATR RATIO* for the current bar minus that *HistLength* times *Multiplier* days ago. For example:

```
DARRAT: DELTA ATR RATIO 10 20
```

The preceding definition computes "ATR RATIO 10 20" for the current bar as well as for that 200 bars earlier in history. The latter is subtracted from the former.

## BOLLINGER WIDTH HistLength

The mean and standard deviation of closing prices is computed for the specified history length. The value of this variable is the log of the ratio of the standard deviation to the mean. The effect of this division is to make the standard deviation be relative to the moving average. Note that this variable is extremely poorly behaved in nearly every regard. For this reason, historical normalization (full normalization, including both centering and scaling) as discussed beginning [here](#) is strongly recommended. For example:

```
BOLLWIDTH: BOLLINGER WIDTH 40 : NORMALIZE 200
```

The preceding command computes the BOLLINGER WIDTH variable for the

current bar, the prior bar, and so forth, for a total of 200 bars back in history. It computes the median and interquartile range of these 200 values, subtracts this median from the current value, and divides by the interquartile range. A transformation is also applied, as described in the section beginning [here](#). This normalization converts the highly unstable BOLLINGER WIDTH to a nicely stable oscillator.

## DELTA BOLLINGER WIDTH HistLength DeltaLength

This is the difference between the BOLLINGER WIDTH for the current bar minus that *DeltaLength* bars earlier. Again, historical normalization (full normalization, including both centering and scaling) as discussed beginning [here](#) is strongly recommended. For example:

```
DBOLLWIDTH: DELTA BOLLINGER WIDTH 40 20 : NORMALIZE 200
```

The preceding command computes the DELTA BOLLINGER WIDTH variable for the current bar, the prior bar, and so forth, for a total of 200 bars back in history. It computes the median and interquartile range of these 200 values, subtracts this median from the current value, and divides by the interquartile range. A transformation is also applied, as described in the section beginning [here](#). This normalization converts the highly unstable DELTA BOLLINGER WIDTH to a nicely stable oscillator.

## N DAY NARROWER HistLength

Let  $N$  be the number of days one has to go back in order to find a true range less than the current bar's true range. True range is defined as the maximum of the current bar's high minus its low, the current bar's high minus the prior bar's close, and the prior bar's close minus the current bar's low. Search only *HistLength* bars back. If after this many bars are checked, no smaller true range is found, set  $N=HistLength+1$ . Then this variable is defined as  $100 * (N-1) / HistLength - 50$ . Note that this variable is highly unstable. Suppose that over the most recent *HistLength* bars the true range has steadily decreased. Then this indicator will attain its maximum value of 50. But now suppose the same steady decrease is true, except that the current bar's true range is one tick greater than that of the prior bar. This indicator will attain its minimum value of -50!

For example:

```
NDN10: N DAY NARROWER 10
```

The preceding definition examines the most recent 10 bars. It compares the true range of the current bar to that of the prior 10 bars. If the immediately prior bar's true range is less than that of the current bar, the value of this indicator is -50. If none of the prior true ranges are less than that of the current bar, the value of this indicator is 50. Intermediate locations of the most recent lesser true range will result in intermediate values of this indicator.

## N DAY WIDER HistLength

This is identical to N DAY NARROWER except that we search for greater true range. For example:

**NDW10: N DAY WIDER 10**

The preceding definition examines the most recent 10 bars. It compares the true range of the current bar to that of the prior 10 bars. If the immediately prior bar's true range is great than that of the current bar, the value of this indicator is -50. If none of the prior true ranges are greater than that of the current bar, the value of this indicator is 50. Intermediate locations of the most recent greater true range will result in intermediate values of this indicator.

## Indicators Involving Indices

We saw [here](#) that one or more markets can be declared as *index* markets. Usually, an index market is a ‘summary’ market, an average of the markets in a defined set. We also saw that the IS INDEX modifier can be used to clone a variable for an index to all markets, and the MINUS INDEX modifier can be used to compute the deviation of a variable for a certain market from its value in an index market.

The IS INDEX and MINUS INDEX modifiers are general-purpose tools that can be applied to nearly all indicators in *TSSB*’s built-in library. In this section we will explore several special-purpose indicators that involve an index market.

### INDEX CORRELATION HistLength

The ordinary correlation coefficient is computed between the log price (mean of open, high, low, and close) of the market under consideration and the log price of the index market. The correlation is over the specified history length, including the current bar. This variable is the correlation times 50, which provides a range of -50 to 50. If the user employs more than one index market, INDEX1 is used for this variable. Currently there is no way to specify any other index, although this feature could be added later if it becomes necessary. For example:

```
INDCORR: INDEX CORRELATION 20
```

The preceding definition computes the correlation between the market under consideration and the index market (such as OEX or the S&P500 index) over the most recent 20 bars. If the two markets (that under consideration and the index) happen to be perfectly correlated, the value of this variable will be 50. If they are perfectly negatively correlated (highly unlikely!), the value will be -50. The value will be zero if the two markets are totally uncorrelated.

### DELTA INDEX CORRELATION HistLength DeltaLength

This is the INDEX CORRELATION of the current bar minus that *DeltaLength* bars ago. For example:

```
DINDCORR: DELTA INDEX CORRELATION 20 10
```

The preceding definition computes INDEX CORRELATION for the current bar

and for the bar 10 bars ago. It subtracts the latter from the former.

## DEVIATION FROM INDEX FIT HistLength MovAvgLength

A least-squares line is computed for predicting the log price (mean of open, high, low, and close) of the market under consideration from the log price of the index market. The fit is over the specified history length, including the current bar. This variable is the current bar's log price minus its predicted value, normalized by the standard error of the fit and slightly compressed to the range -50 to 50. If *MovAvgLength* is greater than one, a moving average of the variable is taken before compression. To avoid a moving average being taken, specify *MovAvgLength* as 0 or 1. If the user employs more than one index market, INDEX1 is used for this variable. Currently there is no way to specify any other index, although this feature could be added later if it becomes necessary. For example:

**DEVFIT: DEVIATION FROM INDEX FIT 50 0**

The preceding definition finds a least-squares linear equation for predicting the log price of the market under consideration from the log price of the index market. It uses the most recent 50 bars to fit this line. It then uses this equation to predict the log price of the current bar in the market under consideration, and subtracts this predicted value from the actual value. This deviation from the prediction is normalized and compressed.

## PURIFIED INDEX Norm HistLen Npred Nfam Nlooks Look1 ...

A linear model is used to predict values of an index variable, and these predictions are subtracted from the true value of the index. This difference is the value of the PURIFIED INDEX indicator. No compression is done. Typically, the index 'market' will not be an actual market. Rather, it will be a sentiment indicator such as VIX (or Investors Intelligence Survey). The predictors are based on recent market dynamics. In particular, linear trend, quadratic trend, and volatility of the market may be used as predictors in the linear model that predicts the sentiment index.

The user can specify up to ten lookback distances for computing the market-based predictors. For each lookback, linear trend, quadratic trend, and volatility

may be measured. Thus, we may have as many as  $10*3=30$  predictors available for the linear model. The user must specify whether the model will use one or two of these candidates. If the user specifies that one be used, the best predictor will be chosen. If two, all possible pairs of predictors will be examined and the best pair chosen. This process of choosing the best predictor(s) and training the linear model to predict the sentiment index is repeated for every bar.

The following parameters must be specified by the user:

**Normalization** - Either or both of two normalization schemes can be used. One scheme is to multiply the predicted value by R-square before it is subtracted from the index. This has the effect of de-emphasizing the prediction when the model does a poor job of predicting the index. The price paid for this R-square normalization is less centering of the purified value. The other scheme is to divide the purified index by the standard error of the prediction. This stabilizes the variance of the purified index and has the effect of shrinking the value toward zero when the model does a poor job of prediction. The normalization parameter has the value 0, 1, 2, or 3:

- 0** - No normalization
- 1** - R-square normalization
- 2** - Standard error normalization (This is the method in David Aronson's paper)
- 3** - Both normalizations

**HistLen** - The number of recent observations that will be used to train the predictive model.

**Npred** - The number of predictors that will be used by the model. This must be 1 or 2.

**Nfam** - The number of families of predictors for each lookback. This must be 1, 2, or 3.

- 1** - Use linear trend only
- 2** - Use linear and quadratic trend
- 3** - Use linear and quadratic trend, and volatility

**Nlookbacks** - The number of lookback distances to use for the trends and volatility. This must be at least one, and at most ten.

**Lookback1** - The first lookback distance

**Lookback2** - The second lookback distance, if used. A total of *Nlookbacks* of

these appear.

For example, the following definition would employ Type 2 normalization and use the 50 most recent values of linear trend, quadratic trend, and volatility to train the model. These 50 do not include the current bar. The linear model that predicts the sentiment index would employ two predictors. Four different lookbacks will be used for the trend and volatility indicators: 10, 20, 40, and 80 bars. Thus, the model will have  $3 \times 4 = 12$  candidate predictors.

```
PURE1: PURIFIED INDEX  2   50   2   3   4   10  20  40  80
```

Because the optimal model is recomputed for every bar, it is not practical to print for the user the predictors chosen for the model. They can and do change frequently. However, TSSB does print a summary of how often each candidate is chosen, which can be interesting. It also prints the mean R-square. Here is a sample such output, showing the use of all three families for lookbacks of 10 and 50 bars:

```
PURIFIED INDEX results for PURIF232 in OEX
Mean R-square = 0.927
10 Linear      28.7 %
10 Quadratic    8.4 %
10 Volatility   10.7 %
50 Linear       25.2 %
50 Quadratic    13.2 %
50 Volatility   13.8 %
```

*Note that this indicator, while still valid, has been deprecated by the PURIFY Transform.*

## Basic Price Distribution Statistics

Sometimes it can be useful to know whether recent prices or price changes have an ‘unusual’ statistical distribution. Here, ‘unusual’ means that the prices or changes are unusually skewed (with extremely large or small values), or they have unusually heavy tails (outliers). The problem with using the traditional measures of skewness and kurtosis is that with financial data, a certain number of outliers are the norm. If one were to use the ordinary moment-based measures, skewness and kurtosis would be inflated and unstable. Instead, the indicators presented in this section use rank-based statistics so as to minimize this problem.

### PRICE SKEWNESS HistLength Multiplier

This computes a measure of the skewness of the price distribution with a lookback period of *HistLength* days. This variable ranges from -50 to 50, with a value of zero implying symmetry. Positive values imply right skewness (some unusually larger prices) and negative values imply left skewness (some unusually smaller prices). If a *Multiplier* greater than one is specified, the skewness with a lookback period of *HistLength* times *Multiplier* is also computed, and the variable is the skewness of the shorter period relative to that of the longer period. A recent increase in skewness results in a positive value for this variable. To compute just the current skewness, specify a *Multiplier* of 0 or 1. Consider the following two examples:

```
PSKEW: PRICE SKEWNESS 50 0
PRSKEW: PRICE SKEWNESS 50 3
```

The first of the two preceding commands computes the price skewness for the most recent 50 bars. The second example computes the price skewness for the most recent 50 and the most recent 150 bars. It then compares the two skewness values and returns a positive value if skewness is increasing and a negative value if skewness is decreasing.

### CHANGE SKEWNESS HistLength Multiplier

This is identical to *PRICE SKEWNESS* except that the quantity evaluated is the daily price changes (one bar’s close relative to the prior bar’s close) rather than the prices themselves.

## PRICE KURTOSIS HistLength Multiplier

This is identical to *PRICE SKEWNESS* except that the kurtosis (tail weight) rather than the skewness is measured.

## CHANGE KURTOSIS HistLength Multiplier

This is identical to *CHANGE SKEWNESS* except that the kurtosis (tail weight) rather than the skewness is measured.

## DELTA PRICE SKEWNESS HistLen Multiplier DeltaLen

This computes the difference between the current value of the PRICE SKEWNESS and its value *DeltaLength* days ago. For example:

```
DPSKEW: DELTA PRICE SKEWNESS 50 3 200
```

The preceding definition computes 'PRICE SKEWNESS 50 3' for the current bar and also for the bar 200 bars earlier. It then subtracts the latter from the former. So, for example, a positive value for this variable implies that skewness is increasing faster right now than it did 200 bars ago.

## DELTA CHANGE SKEWNESS HistLen Multiplier DeltaLen

This is identical to DELTA PRICE SKEWNESS except that it is based on the daily price changes (one bar's close relative to the prior bar's close) rather than the prices themselves.

## DELTA PRICE KURTOSIS HistLen Multiplier DeltaLen

This is identical to DELTA PRICE SKEWNESS except that the kurtosis (tail weight) is measured, instead of skewness.

## DELTA CHANGE KURTOSIS HistLen Multiplier DeltaLen

This is identical to DELTA PRICE KURTOSIS except that it is based on the daily price changes (one bar's close relative to the prior bar's close) rather than the prices themselves.

## Indicators That Significantly Involve Volume

All of the indicators seen so far are based on price information only. The indicators presented in this section also incorporate volume as a major component of their information conveyance.

### VOLUME MOMENTUM HistLength Multiplier

This computes the *Histlength* moving average of volume, as well as that over a length of *HistLength* times *Multiplier*. The variable is the ratio of the former (short term) to the latter (long term), transformed and compressed to a range of -50 to 50. Thus, this variable measures the degree to which volume is increasing or decreasing. For example:

**VMOM: VOLUME MOMENTUM 20 4**

The preceding definition compares the average volume over the most recent 20 bars to the average volume over the most recent 80 bars. If they are equal, the result is zero. If the former exceeds the latter, the result is positive, and vice versa.

### DELTA VOLUME MOMENTUM HistLen Multiplier DeltaLen

This is the VOLUME MOMENTUM for the current bar minus that *DeltaLen* bars ago, transformed to a range of -50 to 50. For example:

**DVMOM: DELTA VOLUME MOMENTUM 20 4 100**

The preceding definition computes 'VOLUME MOMENTUM 20 4' for the current bar as well as for the bar 100 bars ago. It subtracts the latter from the former and then transforms/compresses it to a uniform range.

### VOLUME WEIGHTED MA OVER MA HistLength

This is the log of the ratio of the volume-weighted moving average to the ordinary moving average. It is slightly compressed and transformed to a range of -50 to 50. The volume-weighted moving average is computed by multiplying each closing price in the *HistLength* lookback period by the volume for that bar,

summing, and then dividing that sum by the total volume for the lookback period. In this way, bars with high volume are given more emphasis than bars with low volume. If the volume is equal for all bars, the ratio will be 1.0 and the value of this variable will be zero. A positive value of this variable implies that high-volume bars tended to have higher closing prices than low-volume bars. A negative value means the opposite is true. For example:

#### **VWMAMA: VOLUME WEIGHTED MA OVER MA 50**

In the preceding definition, the volume-weighted moving average of the most recent 50 bars is computed, as well as the ordinary moving average. The former is divided by the latter, the log is taken, and the result is transformed and compressed to a uniform range.

### **DIFF VOLUME WEIGHTED MA OVER MA ShortDist LongDist**

This is ‘VOLUME WEIGHTED MA OVER MA *ShortDist*’, minus that over *LongDist*. It is slightly compressed and transformed to a range of -50 to 50. For example:

#### **DVWMAMA: DIFF VOLUME WEIGHTED MA OVER MA 20 100**

The preceding definition computes VOLUME WEIGHTED MA OVER MA with lookbacks of 20 and 100 bars. The latter is subtracted from the former, and the result is transformed and compressed to a uniform range.

### **PRICE VOLUME FIT HistLength**

This is the slope of the least-squares regression line for predicting log closing price from log volume. It is slightly transformed to the range -50 to 50. If there is no relationship between volume and price, this variable is zero. A positive value implies that higher prices and higher volumes tend to occur together, while a negative value implies that higher volumes are associated with lower prices. For example:

#### **PVF\_50: PRICE VOLUME FIT 50**

The preceding definition examines the most recent 50 bars and fits a regression line for predicting the closing price of each bar from the volume of that bar. The slope of this line is compressed and transformed to a uniform range.

## DIFF PRICE VOLUME FIT ShortDist LongDist

This is the *PRICE VOLUME FIT* computed over *ShortDist* days, minus that over *LongDist* days. It is slightly transformed to the range -50 to 50. For example:

**DIFPVF\_20\_100: DIFF PRICE VOLUME FIT 20 100**

The preceding definition computes PRICE VOLUME FIT with lookbacks of 20 and 100 bars. The latter is subtracted from the former, and the result is transformed and compressed to a uniform range.

## DELTA PRICE VOLUME FIT HistLength DeltaDist

This is the *PRICE VOLUME FIT* computed over *HistLength* days minus the same quantity computed at a lag of *DeltaDist* days. It is slightly transformed to the range -50 to 50. For example:

**DELPVF\_20\_30: DELTA PRICE VOLUME FIT 20 30**

The preceding definition computes PRICE VOLUME FIT for the current bar with a lookback of 20 bars. It then computes PRICE VOLUME FIT with a lookback of 20 bars for the bar that is 30 bars ago. The latter is subtracted from the former, and the result is transformed and compressed to a uniform range.

## ON BALANCE VOLUME HistLength

This is the *ON BALANCE VOLUME* technical indicator computed over *HistLength* days. It is slightly transformed to the range -50 to 50. In order to compute this indicator, two quantities are cumulated. The total volume is the sum of the volumes of the *HistLength* most recent bars. The signed volume is the sum of the volumes of all bars whose close exceeds the close of the prior bar, minus the sum of the volume of all bars whose close is less than that of the prior bar. Thus, high volumes on bars having increasing price will push up the signed volume, while high volumes on bars having decreasing price will push down the signed volume. Bars with low volume will have relatively little impact on the signed volume. The ON BALANCE VOLUME indicator is the ratio of the signed volume to the total volume, compressed and transformed to a range of -50 to 50. For example:

**OBV50: ON BALANCE VOLUME 50**

The preceding definition computes the total and the signed volumes over the most recent 50 bars. The latter is divided by the former, and the ratio is compressed and transformed to a uniform range.

## DELTA ON BALANCE VOLUME *HistLength* *DeltaDist*

This is the *ON BALANCE VOLUME* computed over *HistLength* days, minus the same quantity computed at a lag of *DeltaDist* days. It is slightly transformed to the range -50 to 50. For example:

**DOBV50: DELTA ON BALANCE VOLUME 50 45**

The preceding definition computes the ON BALANCE VOLUME for the most recent 50 bars, as well as for the bar 45 bars earlier. The latter is subtracted from the former, and the difference is compressed and scaled to a uniform range.

## POSITIVE VOLUME INDICATOR *HistLength*

This is the average relative price change (the difference between the current bar's close and the prior bar's close, divided by the prior bar's close) over the specified number of bars. (Actually, it examines *HistLength*+1 bars, because it considers *HistLength* changes.) Only changes corresponding to increased volume are cumulated into the sum for finding the mean. Price changes that correspond to constant or decreased volume are treated as zero for the purposes of finding the average, thus reducing the average. In order to provide cross-market conformity (which is terrible in its raw form), this average is normalized by dividing by the standard deviation of price changes taken over a history of  $2 * \text{HistLength}$  or 250 days, whichever is longer. It is slightly compressed and transformed to the range -50 to 50. For example:

**POSVOL: POSITIVE VOLUME INDICATOR 40**

The preceding definition examines the most recent 41 bars, which implies examination of the 40 most recent relative price changes. Those changes that correspond to increasing volume are summed, and the sum is divided by 40 to find the mean. This mean is then divided by the standard deviation of price changes over the most recent 250 bars. This normalized quantity is compressed and transformed to a uniform range.

## DELTA POSITIVE VOLUME INDICATOR HistLen DeltaDist

This is the *POSITIVE VOLUME INDICATOR* computed over *HistLen* changes minus the same quantity computed at a lag of *DeltaDist* bars. It is slightly transformed to the range -50 to 50. For example:

**DPOSVOL: DELTA POSITIVE VOLUME INDICATOR 40 35**

The preceding definition computes ‘POSITIVE VOLUME INDICATOR 40’ for the current bar as well as for the bar 35 bars earlier. The latter is subtracted from the former, and the difference is compressed/transformed to a uniform range.

## NEGATIVE VOLUME INDICATOR HistLength

This is identical to *POSITIVE VOLUME INDICATOR* except that only bars having decreasing volume are considered.

## DELTA NEGATIVE VOLUME INDICATOR HistLen DeltaDist

This is identical to *DELTA POSITIVE VOLUME INDICATOR*, except that price changes are considered only when they correspond to decreasing volume.

## PRODUCT PRICE VOLUME HistLength

For each bar in the history, the ‘precursor’ to this indicator is computed in three steps:

- 1) The current bar’s volume is normalized by dividing it by the median of the prior 250 bar’s volumes. Thus, if the current bar’s volume is ‘average’ the result will be one. If the current bar’s volume is unusually small, the result will be near zero, and if unusually large, the result will be much greater than one.
- 2) The current bar’s price change (log of the ratio of the current close to the prior close) is normalized by subtracting the median of this quantity over the prior 250 bars and dividing by the interquartile range.

3) The precursor to the *PRODUCT PRICE VOLUME* indicator is computed as the product of the normalized price of the current bar times the normalized volume of the current.

The net result of these three steps is that bar price changes that correspond to unusually large relative volume will be amplified, while price changes corresponding to relatively small volumes will be diminished. However, this precursor value varies greatly from bar to bar. It needs smoothing. So...

The final value of the *PRODUCT PRICE VOLUME* indicator is computed by means of a moving average of the precursor described above. The moving average is taken over *HistLength* bars. It is slightly compressed and transformed to the range -50 to 50.

For example:

#### **PPV: PRODUCT PRICE VOLUME 25**

For each bar, the preceding definition evaluates the normalized volume and price changes over the most recent 25 bars, computes their average product, and compresses/transforms the result to a uniform range.

## **SUM PRICE VOLUME HistLength**

Each bar's price change and volume are normalized as in the *PRODUCT PRICE VOLUME* indicator. The precursor to the *SUM PRICE VOLUME* indicator is computed as the sum of the normalized volume and the absolute value of the normalized price change. If the normalized price change is negative, the sign of the sum is flipped. This produces a result that is vaguely similar to what was had in the *PRODUCT PRICE VOLUME* case, in that extreme results are obtained when volume is relatively high and the price change is extreme. However, in the case of *SUM PRICE VOLUME*, an extreme value of either volume or price change alone can cause the result to be extreme, while for *PRODUCT PRICE VOLUME* they both must be extreme. The final value is computed by averaging this sum over the most recent *HistLength* bars. For example:

#### **SPV: SUM PRICE VOLUME 25**

For each bar, the preceding definition evaluates the normalized volume and price changes over the most recent 25 bars, computes their average signed sum, and compresses/transforms the result to a uniform range.

## DELTA PRODUCT PRICE VOLUME HistLen DeltaDist

This is the *PRODUCT PRICE VOLUME* indicator computed over *HistLen* bars minus the same quantity computed at a lag of *DeltaDist* bars. It is slightly compressed and transformed to the range -50 to 50. For example:

**DPPV: DELTA PRODUCT PRICE VOLUME 40 35**

The preceding definition computes ‘PRODUCT PRICE VOLUME 40’ for the current bar as well as for the bar 35 bars earlier. The latter is subtracted from the former, and the difference is compressed/transformed to a uniform range.

## DELTA SUM PRICE VOLUME HistLen DeltaDist

This is the *SUM PRICE VOLUME* indicator computed over *HistLen* bars minus the same quantity computed at a lag of *DeltaDist* bars. It is slightly compressed and transformed to the range -50 to 50. For example:

**DSPV: DELTA SUM PRICE VOLUME 40 35**

The preceding definition computes ‘SUM PRICE VOLUME 40’ for the current bar as well as for the bar 35 bars earlier. The latter is subtracted from the former, and the difference is compressed/transformed to a uniform range.

## Entropy and Mutual Information Indicators

The exact mathematical definition of information is beyond the scope of this text. However, it is not too different from the intuitive meaning of the term. When a variable carries information, this variable tells us something about the state of the process on which it is based.

Two variables may share some information. A simple example might be two measures of volatility. Each of them might contain information about some general aspect of volatility, while at the same time each might also respond to some aspect of volatility that is specific to that variable. Shared information is called *mutual information*. The mutual information indicators available in TSSB do not involve pairing of variables. Rather, they relate to information on price changes at various short lags. This will be made explicit in the definitions of these indicators.

On a simple level, *entropy* is a measure of disorder. A variable with high entropy appears to be highly disordered. It contains a large number of states. Conversely, a variable with low entropy is very ordered. It contains relatively few distinct states.

This section describes indicators that are based on entropy and mutual information. They are all based on a simple partitioning of the historical data. Two specifications are important. The *word length* is a small number, typically 1-5 or so, which is the number of contiguous bars considered together to define a single unit of relationship. Each bar in a word defines a binary quantity: either a bar closes higher than the prior bar, or it does not. Thus, the total number of possible patterns is two to the power of the number of comparisons made. This will become more clear when illustrated in the context of specific indicators.

The other important specification is the *window length*. This is the number of historical bars that are examined in order to compute the value of an indicator for a single date/time. The user cannot set the window length. It is predefined to be ten times the number of possible patterns discussed above. As a result, the average number of cases that fit each binary pattern is ten.

### PRICE ENTROPY WordLength

This computes the binary entropy of price changes, transformed and slightly compressed to the range -50 to 50. The number of historical bars used in the computation (window length) is ten times two to the power *WordLength*. For example:

## **PENT: PRICE ENTROPY 2**

The preceding definition compares the closing price of the current bar with the closing price of the prior bar. Either the price increased or it did not, a binary outcome. Then it compares the closing price of the prior bar with the closing price of the bar just before it. Again, either the price increased or it did not. This gives us four possible relationships which can be visualized as a 2 by 2 arrangement of four cells. Whichever cell corresponds to the pattern of these two price changes has its count incremented. Then this block of adjacent comparisons is moved back in time by one bar. The same pair of comparisons is made, and the appropriate cell count is incremented. This is repeated  $10^4=40$  times. Finally, the entropy of this 2 by 2 set of cells is computed and transformed/compressed to a uniform range. If the patterns vary widely in this set of 40 historical bars, the entropy will be high. Conversely, if a relatively small set of patterns predominate, the entropy will be low.

## **VOLUME ENTROPY WordLength**

This computes the binary entropy of volume changes, transformed and slightly compressed to the range -50 to 50. Computation is identical to PRICE ENTROPY except that the volume of each bar is used instead of closing price.

## **PRICE MUTUAL INFORMATION WordLength**

This computes the binary mutual information between the current bar's price change (prior close to current close), and the *WordLength* prior bars' price changes. This quantity is transformed and slightly compressed to the range -50 to 50. The number of bars used in the computation is ten times two to the power [one plus *WordLength*]. Note that for ENTROPY, the unit of relationship is *WordLength* bars, but for MUTUAL INFORMATION the unit of relationship is *WordLength+1* bars. This is because the mutual information is between a set of *WordLength* contiguous bars (lagged one bar behind the current bar) and the current bar. For example:

## **PMI: PRICE MUTUAL INFORMATION 2**

The preceding definition compares the closing price of the current bar with the closing price of the prior bar. Either the price increased or it did not, a binary decision. This is one of the two variables whose mutual information is computed. The other variable has four states (two to the power *WordLength*). These states are defined exactly as was done for entropy: The close of the lag-1

bar is compared to the close of the lag-2 bar. Either the price increased or it did not. Similar, the close of the lag-2 bar is compared to the close of the lag-3 bar. Either it increased or it did not. Recalling that the current bar also has two states, we thus have a total of  $2*2*2=8$  cells. The cell count corresponding to the combined state is incremented. Then this block of  $1+2=3$  contiguous bars is moved back one bar and the operation is repeated. This counting operation is done a total of  $10*2^3=10*8=80$  times. The mutual information between the binary current bar price change and the 4-state prior two-bar price changes is computed and transformed/compressed to a uniform range. If over this 80-bar window there is a substantial relationship between the 4-state price pattern of the prior two bars and the binary price pattern of the current bar, the mutual information indicator will be large, toward 50. Conversely, if there is little such relationship within this 80-bar window, the mutual information indicator will be small, toward -50. One would tend to see large mutual information when the market behavior is organized, with good short-term predictability. Conversely, periods of highly random market behavior will result in small mutual information.

## VOLUME MUTUAL INFORMATION WordLength

This computes the mutual information of volume changes, transformed and slightly compressed to the range -50 to 50. Computation is identical to PRICE MUTUAL INFORMATION except that the volume of each bar is used instead of closing price.

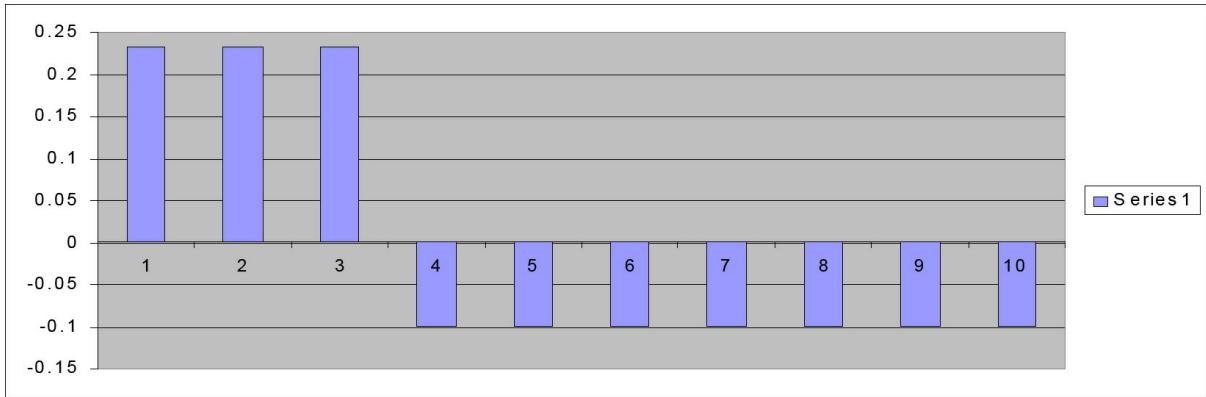
## Indicators Based on Wavelets

A wavelet decomposition analyzes a time series in two dimensions: location and frequency (or, equivalently, period). Features in the time series are identified by their location in time and their approximate frequency. So, for example, we may (very roughly speaking) say that on a certain date the time series contained an event whose dominant frequency/period was a certain value.

Note that frequency and period have an inverse relationship and are equivalent ways of specifying the repetition rate of a periodic event. An event with a period of  $n$  bars has a frequency of  $1/n$  cycles per bar. Because of a mathematical limitation called the Nyquist limit, the maximum frequency we can measure is 0.5 cycles per bar. Equivalently, the minimum period we can measure is 2 bars.

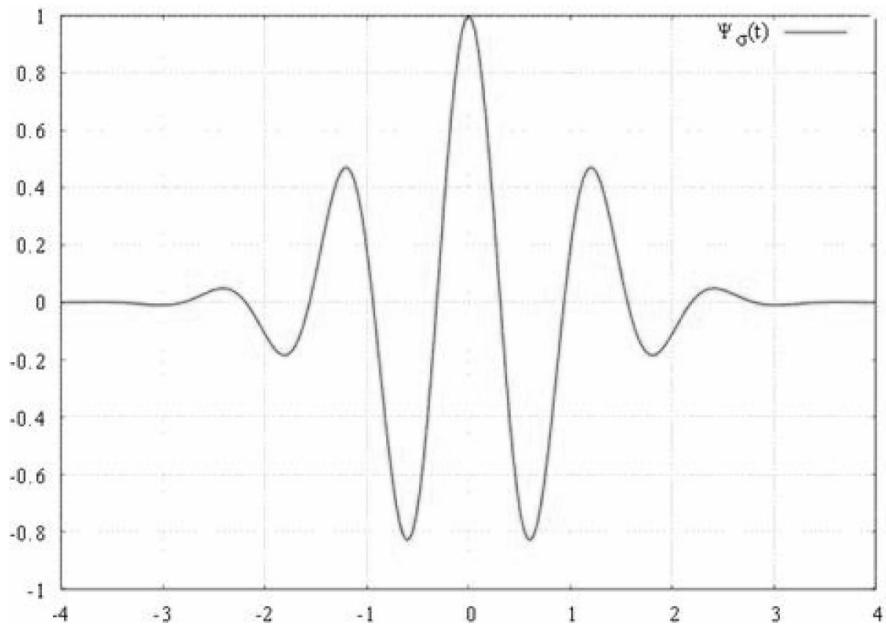
Although wavelets are new to many market analysts they are similar to indicators that are commonly used: moving average oscillators. An  $n$ -period simple moving average is derived by adding up  $n$  prices and dividing by  $n$ . An alternative way of looking at it is to multiply each of  $n$  prices by a weight of  $1/n$  and then sum the products. Thinking of a moving average in these terms makes clear the idea of a weight function, a set of weights that are multiplied against a set of prices. If one visualizes a set of equally valued weights as a sequence of numbers along the time axis, the weights have a rectangular shape. This is the shape of the weight function used to produce a simple moving average. Other types of moving averages use weight functions that have other shapes, such as a triangle or weights that decrease in value exponentially. So long as the sum of the weights equals 1.0 you obtain some form of moving average.

A common indicator based on subtracting a short term moving average from a long-term moving average is a price oscillator. For example, we may have the 3-day moving average minus the 10-day moving average. Price oscillators vary around a mean value of zero. This is because the weights sum to zero. Thus, the moving average oscillator is the result of applying a weight function to prices. In this example, three weights, each having a value of 0.3333, form the 3-day average, while 10 weights each having a value of 0.10 form the 10-day moving average. When the weights for the 10-day MA are subtracted from the weights for the 3-day MA, the resulting weights are pictured in [Figure 7](#) below. Note that it has both positive and negative weights, which sum to zero.



**Figure 7:** Weights for a moving-average oscillator

Wavelets can be thought of as a sophisticated type of price oscillator that uses a more intelligently designed weight function. By altering the shape of the weight function we can obtain an oscillator that more precisely responds to the price events in which we are most interested. [Figure 8](#) below is a weight function for a particularly useful wavelet, the Morlet Wavelet. Note that it has both positive and negative values, and they sum to zero.



**Figure 8:** A Morlet Wavelet

There are an infinite number of wavelet decompositions (types of wavelet families) that are theoretically possible. Several tradeoffs are involved in choosing the best wavelet for a particular application. One of the two primary considerations is the Heisenberg Uncertainty Principle. This says that we cannot accurately identify an event in terms of both its period and its time of occurrence. If we demand high accuracy in finding the time, we will have to settle for low accuracy in its period. Conversely, if we need to locate an event

of a certain narrowly specified period, we will not be able to pinpoint when it occurred with high accuracy. It is usually in our best interest to use a wavelet that does the best possible job of dealing with this unfortunate compromise.

There is a second consideration for our choice of wavelet type. This issue is usually of lesser importance than the Heisenberg time/period compromise, but not always. The issue is redundancy. In some applications we will want to detect wavelet events at a wide range periods and at frequent intervals, probably every bar. This can be a lot of information for a prediction model to process, so we want to use as few indicators as possible to satisfy our goal of being thorough in our capture wavelet information. Ideally, we want each wavelet indicator to carry its own unique set of information so that we convey the maximum possible amount of information in the minimum possible number of indicators in order to avoid overfitting due to the so-called *curse of dimensionality*. The best possible wavelet (in terms of redundancy) would be able to (theoretically, at least) perfectly reproduce the original time series with fewer indicators than any other wavelet.

Unfortunately, these two criteria (simultaneous location in period and time, and redundancy) are in direct conflict. The best ‘simultaneous locators’ are terribly redundant, and the least redundant wavelets do an abysmal job of simultaneously locating an event in terms of period and time of occurrence.

Because of this conflict, *TSSB* provides two wavelet families that are at opposite ends of this continuum. *Morlet wavelets* perfectly attain the Heisenberg Uncertainty Principle limit; no other wavelet does a better job of simultaneously locating events in terms of period and time. But members of the Morlet wavelet family are seriously redundant. A huge number of Morlet wavelets would be required to come even close to encapsulating all of the information in a time series. Supplying enough Morlet wavelet wavelets to predictive models so as to capture all of the information in a price series would create massive overfitting. Thus, the user must be judicious in choosing Morlet wavelets as indicators.

At the opposite extreme, *Daubechies wavelets* have zero redundancy. No other wavelet captures as much information about the time series in as few indicators. The price paid is exceptionally poor localization in terms of period and time. In most financial applications, we want to know the time of an event with maximum precision. For this reason, Morlet wavelets are almost always preferred to Daubechies wavelets. However, there are some exceptions, so both types are available.

Keep in mind that because future leak must be strictly avoided, all wavelet indicators operate at a lag. For Morlet wavelets, this lag is exactly twice the

user-specified period. In other words, any time we compute a Morlet wavelet indicator, we are actually measuring (and hence providing to the prediction model) the value of the indicator two periods ago. There is no way to compute Morlet wavelets of less lag without making major sacrifices in frequency response, although the DIFF and PRODUCT indicators discussed soon do so as part of their operation. For Daubechies wavelets, the exact lag is not so easily specified because of their very poor time localization.

## REAL MORLET Period

The real (in-phase) component of a Morlet wavelet is computed for the log of closing prices. It is slightly compressed and transformed to a range of -50 to 50. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. REAL MORLET roughly measures the position of the price within a periodic waveform of approximately the specified period. The period must be greater than or equal to 2.0. For example:

**RMORLET: REAL MORLET 5**

The preceding definition computes the real component of a wavelet having a repetition period of 5 bars. The indicator computed for a given bar is the value of the real component  $2*5=10$  bars earlier. Roughly speaking, this means that if we were to isolate only the component of the market series that repeats with a period of 5 bars, and ignore all other repetitive components and noise, this value is the price due to that component at that time.

## REAL DIFF MORLET Period

The real component of a Morlet wavelet is computed for the log of closing prices. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. The same quantity is computed for twice the period, though the lag is the same. The long-period quantity is subtracted from the short-period quantity. This roughly measures whether the price value due to fast motion is above or below that due to the slow motion. The period must be greater than or equal to 2.0. For example:

**RDMORLET: REAL DIFF MORLET 5**

The preceding definition computes the real component of a wavelet having a repetition period of 5 bars, and also the real component of a wavelet having a period of 10 bars. The pair of indicators computed for a given bar are the

values of the real components  $2*5=10$  bars earlier. The 10-bar wavelet is subtracted from the 5-bar wavelet, and the difference is transformed/compressed to a uniform range.

## REAL PRODUCT MORLET Period

The real component of a Morlet wavelet is computed for the log of closing prices. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. The same quantity is computed for twice the period, though the lag is the same. If the two quantities have opposite signs, the value of the variable is zero. Otherwise, the value is their product with their sign preserved. This roughly measures whether the price value due to fast motion and that due to slow motion are in agreement, with the sign telling their position within the common motion. The period must be greater than or equal to 2.0. For example:

### **RPMORLET: REAL PRODUCT MORLET 5**

The preceding definition computes the real component of a wavelet having a repetition period of 5 bars, and also the real component of a wavelet having a period of 10 bars. The pair of indicators computed for a given bar are the values of the real components  $2*5=10$  bars earlier. If the 10-bar and 5-bar wavelets have opposite signs, the value of this indicator is set to zero because the price positions of the two periodic components are in conflict. Otherwise, the 10-bar wavelet is multiplied by the 5-bar wavelet. The result is given the sign of the wavelets, and the difference is transformed/compressed to a uniform range.

## IMAG MORLET Period

The imaginary (in-quadrature) component of a Morlet wavelet is computed for the log of closing prices. It is slightly compressed and transformed to a range of -50 to 50. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. IMAG MORLET roughly measures the velocity of the price within a periodic waveform of approximately the specified period. The period must be greater than or equal to 2.0. For example:

### **IMORLET: IMAG MORLET 5**

The preceding definition computes the imaginary component of a wavelet having a repetition period of 5 bars. The indicator computed for a given bar is the value of the imaginary component  $2*5=10$  bars earlier. Roughly speaking, this

means that if we were to isolate only the component of the market series that repeats with a period of 5 bars, and ignore all other repetitive components and noise, this value is the price velocity (signed rate of change) due to that component at that time.

## IMAG DIFF MORLET Period

The imaginary component of a Morlet wavelet is computed for the log of closing prices. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. The same quantity is computed for twice the period, though the lag is the same. The long-period quantity is subtracted from the short-period quantity. This roughly measures whether the velocity of the fast motion is above or below the velocity of the slow motion. The period must be greater than or equal to 2.0. For example:

**IDMORLET: IMAG DIFF MORLET 5**

The preceding definition computes the imaginary component of a wavelet having a repetition period of 5 bars, and also the imaginary component of a wavelet having a period of 10 bars. The pair of indicators computed for a given bar are the values of the imaginary components  $2*5=10$  bars earlier. The 10-bar wavelet is subtracted from the 5-bar wavelet, and the difference is transformed/compressed to a uniform range.

## IMAG PRODUCT MORLET Period

The imaginary component of a Morlet wavelet is computed for the log of closing prices. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. The same quantity is computed for twice the period, though the lag is the same. If the two quantities have opposite signs, the value of the variable is zero. Otherwise, the value is their product with their sign preserved. This roughly measures whether the velocities of the fast motion and the slow motion are in agreement, with the sign telling the direction of the common motion. The period must be greater than or equal to 2.0. For example:

**IPMORLET: IMAG PRODUCT MORLET 5**

The preceding definition computes the imaginary component of a wavelet having a repetition period of 5 bars, and also the imaginary component of a wavelet having a period of 10 bars. The pair of indicators computed for a given bar are the values of the imaginary components  $2*5=10$  bars earlier. If the 10-bar and 5-

bar wavelets have opposite signs, the value of this indicator is set to zero because the price velocities of the two periodic components are in conflict. Otherwise, the 10-bar wavelet is multiplied by the 5-bar wavelet. The result is given the sign of the wavelets, and the difference is transformed/compressed to a uniform range.

## PHASE MORLET Period

This is the *rate of change* of the phase (not the phase itself) of a wavelet having the specified period, transformed and scaled to a range of -50 to 50. The wavelet is centered  $2 * \text{Period}$  bars prior to the current bar. Roughly speaking, this tells us how rapidly the actual phase of the wavelet is changing relative to what one would expect from a pure (noiseless) wave. For example:

### **PHMORLET: PHASE MORLET 5**

The preceding definition computes a Morlet wavelet with a repetition period of 5 bars, centered  $2*5=10$  bars prior to the current bar. The phase at that bar ten bars prior to the current bar is computed, as well as the phase at the bar just prior to it (11 bars before the current bar). The phase difference is computed and transformed/compressed to a uniform range.

## DAUB MEAN HistLength Level

A Daubechies wavelet is computed for the log of bar close ratios (current bar's close divided by the prior bar's close) over the most recent *HistLength* bars. *HistLength* must be a power of two. If not, it is increased to the next power of two. The level must be 1, 2, 3, or 4, with larger values resulting in more smoothing and noise elimination. Two to the power of (*Level*+1) must be less than or equal to *HistLength*. Li, Shi, and Li recommend that *HistLength* be approximately three times the number of bars we will predict into the future. They also recommend that *Level* be two, because a value of one results in too much noise being retained, while values larger than two result in loss of useful information. The user may wish to do his/her own experiments to choose optimal values. The *DAUB MEAN* variable is the mean of the smooth (parent wavelet) coefficients, with the detail coefficients ignored. This mean is compressed and transformed to a range of -50 to 50. For example:

### **DAUBMEAN: DAUB MEAN 16 2**

The preceding definition computes a level-2 Daubechies wavelet over the most

recent 16 bars. The mean of the parent wavelet's coefficient is found and transformed/compressed to a uniform range.

## DAUB MIN HistLength Level

This is identical to *DAUB MEAN* except that the minimum rather than the mean is returned.

## DAUB MAX HistLength Level

This is identical to *DAUB MEAN* except that the maximum rather than the mean is returned.

## DAUB STD HistLength Level

This is identical to *DAUB MEAN* except that the standard deviation rather than the mean is returned.

## DAUB ENERGY HistLength Level

This is identical to *DAUB MEAN* except that the sum of squared coefficients rather than the mean is returned.

## DAUB NL ENERGY HistLength Level

This is identical to *DAUB MEAN* except that the sum of squared differences between neighbors (adjacent coefficients of the parent wavelet) rather than the mean is returned.

## DAUB CURVE HistLength Level

This is identical to *DAUB MEAN* except that the sum of absolute differences between neighbors (adjacent coefficients of the parent wavelet) rather than the mean is returned.

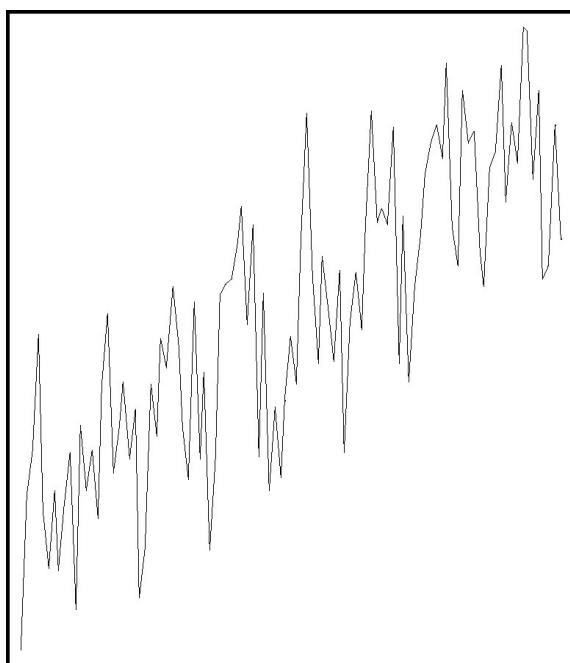
## Follow-Through-Index (FTI) Indicators

Look at the hypothetical market price graphs shown in Figures 9 and 10 at the bottom of this page. Which market do you think would be easier for a prediction model to handle? In fact, even if you were just sitting at a terminal, watching the market, and trading by the seat of your pants, which market do you think would be easier for you to successfully trade?

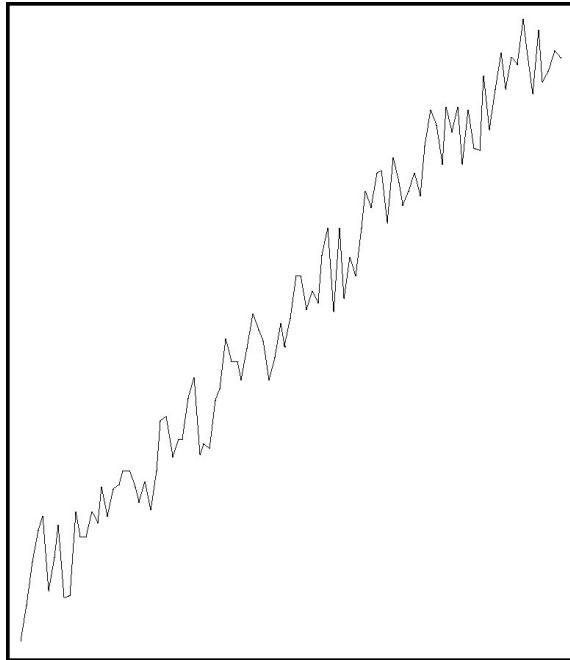
Surely, most people would prefer a market that looks like that in [Figure 10](#). The dominant movement of the market stands out well above the noise. One might say that prices *follow through* on their motion better in [Figure 10](#) than they do in [Figure 9](#). In that latter figure, prices keep bouncing back and forth in wild swings.

Thus, it is useful to be able to measure the degree to which longer-term price movement continues marching onward, overshadowing smaller local movement that is usually just noise. When a market is in a period of high follow-through we would probably be inclined to trade it, while if the follow-through is low we might want to sit out for a while.

There are many simple ways to measure long-term net change relative to noise. The ordinary Sharpe Ratio comes to mind, as well as ADX and many variance-adjusted trend indicators. However, G. S. Khalsa, in “New Concepts in Identifying Trends and Categorizing Market Conditions” proposes an extremely sophisticated measure that he calls the *Follow Through Index* or *FTI*. The exact computations are far too complex to present in full detail here. However, the basic idea is discussed starting on the [here](#).



**Figure 9:** Market with small follow-through



**Figure 10:** Market with large follow-through

## Low-Pass Filtering and FTI Computation

The concept of a low-pass filter is central to FTI computation, so it is important that the reader be clear on this topic. An example of a simple low-pass filter is the ordinary moving average. The term *low* refers to the frequency of the oscillations that remain (i.e., are passed) by the filter. In the time domain, with which many market technicians are more familiar, low-frequency oscillations are long-duration changes in trend. Applying a moving average to price data results in a smooth curve that retains the long-term trend changes but removes the shorter-term movements around the trend. The term *cutoff* refers to the frequency of the shortest-duration trend change that will show up in the filtered (smoothed) data. Trend changes of shorter duration than the cutoff will not show up in the behavior of the filtered data.

## Block Size and Channels

The developer of the FTI indicator family comes from a signal-processing background, so his terminology is somewhat foreign to many market traders. We will stay with his terminology. However, as an aid to translating terminology from the signal-processing domain to the market-trading domain, please keep

these two issues in mind:

- 1) Computation of the FTI indicators is done by moving a window across the market history. This is similar to most or all of the indicators previously discussed in this document. In order to compute FTI indicators at a point in time, we examine a block of history extending backwards from that point in time. The length (lookback) of this moving-window block is the *Block Size* that will be frequently referred to here.
- 2) Due to the mechanical aspects of filtering, the entire block of *Block Size* historical bars cannot be used for the most critical aspects of FTI indicator computation. Only the most recent subset of the total block size may be used. This subset of the entire block (the moving window) is called the *Channel*. Only bars in the channel are used for most aspects of FTI indicator computation. Later, when we see illustrations of FTI operations, the reader will see how the concept of an FTI channel is similar to channels of other types that are already familiar to market technicians. In particular, FTI channels often appear strikingly similar to Bollinger Bands.

## Essential Parameters for FTI calculation

The user must specify three parameters in order to compute the FTI measure of price follow-through:

**BlockSize** is the length of the moving-window block that marches through the market history. This is exactly the same concept as in virtually all indicators: when we compute the value of an indicator at the current point, we examine a block of the most recent points in order to do this calculation. For most other indicators described in this document, the length of this block is called *HistLength*. However, we call this quantity *BlockSize* here in order to conform to convention in the FTI literature.

**Period** is the number of bars for the cutoff of a lowpass filter that is applied to the data for much of the FTI calculation. Consider that any measure of follow-through is highly dependent on where we draw the line between noise and valid price movement. If we are liberal and label all but the smallest, most frantic market moves to be valid, we will probably get a very different measure of follow-through than if we are conservative and label every move as noise unless it is long and large. In the Khalsa algorithm, price changes that remain after lowpass filtering are used to compute net price movement, and the

differences between the unfiltered and the filtered (smoothed) prices are used for noise calculation. Our choice of a period for the lowpass filter is application dependent and may require some experimentation. Khalsa also has a method available in *TSSB*, described later, for automatically computing an ‘optimal’ filter period based on the characteristics of the block of market prices being analyzed. This is a convenient way to avoid the need for specifying a period, but the utility of this algorithm is open to debate.

***HalfLength*** is more a computational issue than an experimental issue. This is the number of points on each side of the center point that are used to compute the lowpass filter. Choosing a reasonable value for the *HalfLength* is discussed soon.

These three quantities interact in various ways, so careful choice of their values is critical. Here is a summary of the relevant issues and rules:

- In most cases, it is best to begin by choosing a *Period* for the filter. Khalsa processes day bars, and he uses periods ranging from 5 days up to 65 days in his demonstrations. Try different values, or use the automated selection algorithm described later.
- *HalfLength* must be greater than or equal to half of the *Period*. In most cases it is best to make it somewhat greater, as strict equality produces a lowpass filter of marginal quality. Larger values of *HalfLength* produce better filters and hence more accurate FTI indicators. On the other hand, larger values of *HalfLength* also ‘waste’ data, as described in the next bullet point.
- The FTI indicator is based on filtered and unfiltered prices in a window (called the *channel* in FTI nomenclature) extending back from the current bar a total of *BlockSize* - *HalfLength* bars. The more bars in the channel, the more accurate the measure of follow-through. This inspires one to make *HalfLength* as small as possible, in direct conflict with the prior bullet point. The solution is to make *BlockSize* as large as possible, remembering that excessive lengths will look so far back in history that response to recent changes in market behavior may be masked. This is the universal indicator-definition conflict between wanting to use long lookback windows for stability, while wanting to use short lookback windows for rapid response to current market conditions. Unfortunately, for FTI calculation we have one more monkey wrench thrown into the mix, the need for a filter *HalfLength* that is as large as possible! In many cases, setting the *BlockSize* equal to twice the *HalfLength* is a reasonable

choice. The channel length (*BlockSize* minus *HalfLength*) certainly should be at least 20 or so. An error will be generated if the channel length is less than 2, an absurdly small (but legal) amount.

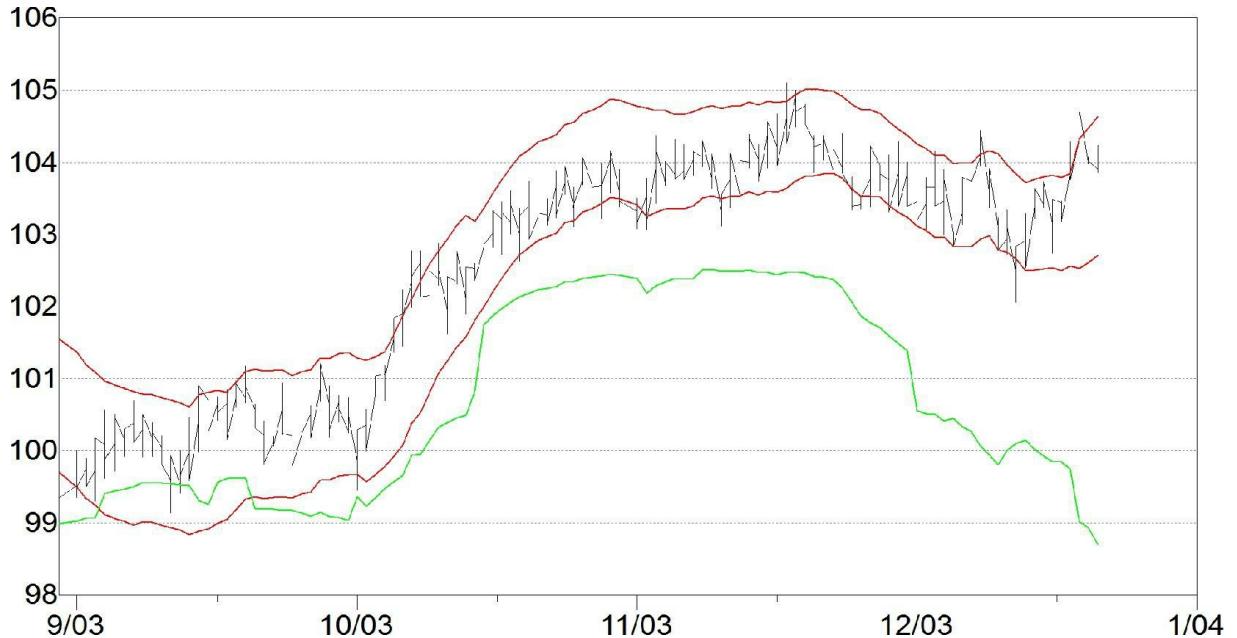
- In summary, a reasonable procedure is to choose the *Period* first. Then choose *HalfLength* to be somewhat greater than half of *Period*. Finally, set *BlockSize* to twice *HalfLength*, larger if needed to make *BlockSize* - *HalfLength* at least 20, or smaller if that difference is at least 20 and you are worried about looking too far back in history.
- Of interest only to readers familiar with signal processing: You are probably thinking that the channel length is not *BlockSize* - *HalfLength* but rather is *BlockSize* - 2 \* *HalfLength*. The reason it works out this way is that Khalsa has a clever method for using just half of the complete filter for the most recent data point, half of the filter plus one for the second-most recent, and so forth. The result is a correct and realizable filter with zero lag for all data points up to the *HalfLength* bar, and slowly deteriorating quality as it approaches the most recent bar. The filter at these recent points still has zero lag, but its frequency response suffers (unavoidably, of course). As long as the channel is reasonably long compared to the *HalfLength*, this error is of little consequence, and the fact that the filter has zero lag is immensely useful.

Lag can be a real problem in market analysis. Lowpass filters such as moving averages are known to lag the price data. Troughs and peaks in the moving average show up later than troughs and peaks in the price. The lag for a simple moving average is the moving average span minus 1 divided by 2 (e.g., 11 day simple MA has a lag of 5). Some applications of smoothing require that the smoothed day be plotted so that its peaks and trough line up with those in the raw price data. The disadvantage however, is that there will be no moving average values for the most recent  $(n-1)/2$  days where  $n$  is the span of the moving average. The lowpass filter used in generating this family of indicators is superior to a moving average but it too has a lag in its basic form. Khalsa developed a clever way to eliminate the lag and thereby obtain a current value for the lowpass smoother. The price is suboptimal performance of the filter (a considerable quantity of undesirable high-frequency components leak through), but it is often a price worth paying.

## Computing FTI

The procedure for computing FTI is quite complex and will not be presented in detail. However, for those interested in the basics, here is a rough overview of the process:

- 1) For each bar in the channel (the most recent  $BlockSize - HalfLength$  bars) compute the lowpass filtered version of the log of the closing prices. Only the bars in the channel will take part in subsequent calculations. The  $HalfLength$  oldest bars in the block are ignored from now on. Their only function was to take part in the lowpass filtering.
- 2) Partition the filtered price moves into up legs and down legs. Examine the distribution of absolute leg lengths and use a rule to set a threshold for distinguishing between legitimate legs and noise legs. Compute the mean length of the legitimate legs.
- 3) For each bar in the channel, compute the difference between the filtered (smoothed) log price and the actual log price. Examine the distribution of these absolute differences and thereby compute the *channel width*, a measure which defines a pair of boundaries above and below each bar such that these boundaries enclose the majority (though not necessarily all) of the price moves within the channel. An example of such a channel is shown bounding the prices in [Figure 11](#) below.
- 4) Divide the mean length of the legitimate legs found in Step 2 by the channel width found in Step 3. This is the FTI measure for the current bar. In [Figure 11](#) below, FTI is graphed in green below the bounded market prices. Notice that the FTI value is declining over time. This is telling us that the follow-through of the price trend in question is decreasing.



**Figure 11:** Noise bounds around a market, and FTI

## Automated Choice of Filter Period

Khalsa states an important point about his Follow-Through Index, somewhat paraphrased here: *If a trend exists, then the FTI for some filter period will be significantly greater than the FTI for other nearby filter periods.* Two things to note about this statement are:

- 1) His rule begins *if a trend exists*. Markets do not always trend.
- 2) The dominant FTI should achieve a significant local peak for some filter period.

What does ‘significant’ mean here? Khalsa does not address this issue at all, leaving it to a human to examine a table of FTI values for various filter periods and make the decision. Later, we will present some TSSB FTI indicators that automatically choose the ‘optimal’ filter period. They do this in a very naive manner: the user specifies a range of periods to try, and the program picks the period that has the largest FTI.

Note that the chosen period is highly data dependent, with the result that as we move from bar to bar through the historical dataset, the chosen period can vary widely, often jumping by a large amount even from one bar to the next. Thus, if this automated choice is used, it is probably best for the user to specify a relatively narrow range of periods to try in order to avoid dramatic changes in indicator behavior from bar to bar.

What about the decision of whether there even is a trend? Actually, experience indicates that the value of FTI obtained is itself a decent indicator of the existence of a trend, regardless of how ‘significantly’ it stands out from its neighbors. The larger the FTI, the stronger is the trend.

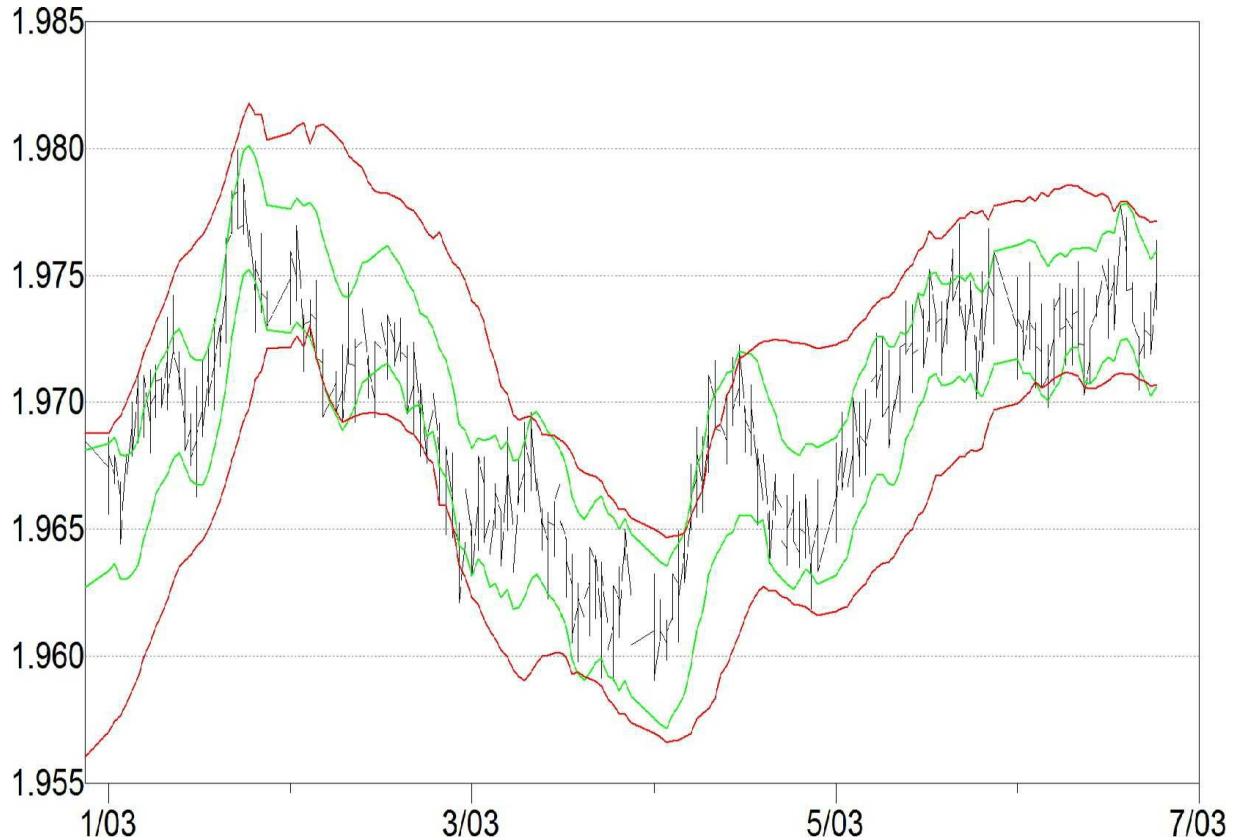
## Trends Within Trends

The Follow-Through Index is the most important and useful aspect of Khalsa’s work. However, he does present a second measure of market state that is closely related to FTI. This is the *Channel Width Ratio*.

The idea behind the channel width ratio is that we find two different filter periods, each of which produces a locally maximum, large FTI. In other words, we find the two trends with the highest FTI values within the span searched. The FTI having the larger period is called the *major trend* and the one having the smaller period is called the *minor trend*.

To be specific, *TSSB* finds every period within a user-specified range whose FTI is greater than or equal to the FTI of its two neighbors (periods one greater and one less). These are the locally maximum FTIs. The two largest such FTIs are found. The one having the greater period defines the major trend within the channel, while the one having the lesser period defines the minor trend within the channel. The actual values of the FTIs are ignored.

For each of these two periods, we compute the channel width as described [here](#). The ratio of the width of the minor trend channel to that of the major trend channel is called the *Channel Width Ratio*, and it can be a useful indicator of the current market state. To see this double channel in action, look at [Figure 12](#) below. The (usually) wider channel is for the major trend.



**Figure 12:**FTI Major and Minor Trend channels

## FTI Indicators Available in *TSSB*

We now list the FTI indicators that exist in the *TSSB* library. These all begin with ‘FTI’ for clarity, and they all require the parameters that have been discussed in the prior sections. Recall that in most situations, the easiest approach to setting parameters is to decide on the desired *Period* (or pair of periods for those that require two periods or a range) first. Then choose the filter *HalfLength* to be at least half of the *Period* (at least half of the larger period, if two periods are specified). Finally, decide on how many bars you want in the channel and add this quantity to the *HalfLength* in order to get the *BlockSize*.

## FTI LOWPASS BlockSize HalfLength Period

Gavinda Khalsa’s zero-lag lowpass filter is applied to the log of closing prices. In other words, this produces the filtered (smoothed) log prices that form the basis of subsequent FTI computations. This can make for an interesting and informative plot. One possible application for this quantity would be as part of an oscillator crossover system as a substitute for a moving average. Note,

however, that the price paid for zero lag is poor frequency response for recent bars. Because this indicator is the filtered value at the current bar, this deterioration in frequency response will be at its maximum. Therefore, this quantity is probably inferior to an ordinary moving average for use in crossover systems. Its zero-lag property is not important enough to overshadow its poor frequency response. In all likelihood, the only legitimate use of this indicator is for visual examination of its plot. For example:

**FTILOW: FTI LOWPASS 6 4 6**

The preceding definition applies Khalsa's zero-lag lowpass filter to the log close of each bar. The *Period* is 6 bars. The filter *HalfLength* is 4, which is legal because it is at least half of the *Period*. The *Blocksize* is 6, which is the minimum legal amount. Recall from the rules [here](#) that the *BlockSize* must be at least 2 more than the *HalfLength*. There is no point in making the channel any longer, because this indicator returns only the current value. Earlier values in the channel would be used for FTI calculations, but this indicator does not do any! It just returns the filtered log closes, so all channel entries except the most recent are ignored.

## FTI MINOR LOWPASS BlockSize HalfLength LowPeriod HighPeriod

This is similar to the FIT LOWPASS indicator just described in that it returns the zero-lag lowpass filtered value of the log prices. The only difference is in how the period of the filter is chosen. In the prior indicator, FTI LOWPASS, the user explicitly specifies the filter period. In this indicator, FTI MINOR LOWPASS, the user instead specifies an inclusive range of possible periods, *LowPeriod* through *HighPeriod*. The program then uses the algorithm described [here](#) to find the period that characterizes the *minor trend*. This is the period chosen for the lowpass filter. In all likelihood, this quantity is useful for display and diagnostic purposes only. As explained earlier, the automated selection of the *minor period* and *major period* is highly unstable and of limited utility. For example:

**FTIMINLP: FTI MINOR LOWPASS 26 6 5 10**

The preceding definition applies lowpass filters having periods ranging from 5 through 10 bars. The *HalfLength* of each filter is 6. This is legal because it exceeds half the period of the maximum (half of 10 is 5, and 6 equals or exceeds 5). Because FTI values will be computed, we make the channel contain 20 points by adding 20 to 6, giving us a *BlockSize* of 26. In accordance with the

algorithm described [here](#), the program computes FTI for every period from 5 through 10. It chooses the two highest FTIs that are local maxima and chooses the smaller of the two associated periods (the minor trend). This period is used for the lowpass filter whose output defines this indicator.

## FTI MAJOR LOWPASS BlockSize HalfLength LowPeriod HighPeriod

This is identical to FTI MINOR LOWPASS except that the major trend's period is selected to define the lowpass filter whose output defines this indicator. In all likelihood, this quantity is useful for display and diagnostic purposes only.

## FTI FTI BlockSize HalfLength Period

This returns the FTI value for the specified parameters. This may be the single most useful FTI indicator. It allows the user to bypass the unstable and often unpredictable automated period selection algorithm and directly specify a period that is tailored to the application. For example:

```
FTI10: FTI FTI 36 6 10
```

The preceding definition computes FTI for a filter period of 10 bars. The *HalfLength* of the lowpass filter is 6, which satisfies the requirement of being at least half of the *Period*. A *BlockSize* of 36 provides a *channel length* of  $36 - 6 = 30$  bars, which is decently long.

## FTI LARGEST FTI BlockSize HalfLength LowPeriod HighPeriod

This returns the value of the largest FTI within the specified range of periods (inclusive). Note that this does not use the automated period selection algorithm. That algorithm finds *two* local maxima and defines the major and minor trends according to the size of the associated periods. FTI LARGEST FTI finds the *single* global maximum. If the user feels uncomfortable choosing a period, then FTI LARGEST FTI may be a useful alternative to FTI FTI because it gives the program the freedom to search a range of periods of the largest FTI, but it does not have suffer the instability due to choosing a *pair* of local maxima. For example:

```
FTIBIG: FTI LARGEST FTI 36 6 5 10
```

The preceding definition computes FTI for periods from 5 through 10 bars. The *HalfLength* of the lowpass filter is 6, which satisfies the requirement of being at least half of the *Period* for all trial periods. A *BlockSize* of 36 provides a *channel length* of  $36-6=30$  bars, which is decently long. The largest FTI found among these trial periods is returned as the computed indicator.

## FTI MINOR FTI BlockSize HalfLength LowPeriod HighPeriod

The user specifies an inclusive range of possible periods, *LowPeriod* through *HighPeriod*. The program uses the algorithm described [here](#) to find the period that characterizes the *minor trend*. This returns the FTI for the minor (smaller period) filter. For example:

```
FTIMIN: FTI MINOR FTI 36 6 5 10
```

The preceding definition computes FTIs using lowpass filters having periods ranging from 5 through 10 bars. The *HalfLength* of each filter is 6. This is legal because it exceeds half the period of the maximum (half of 10 is 5, and 6 equals or exceeds 5). We make the channel contain 30 points by adding 30 to 6, giving us a *BlockSize* of 36. In accordance with the algorithm described [here](#), the program computes FTI for every period from 5 through 10. It chooses the two highest FTIs that are local maxima and chooses the smaller of the two associated periods (the minor trend). This period is used to compute FTI, which is this indicator.

## FTI MAJOR FTI BlockSize HalfLength LowPeriod HighPeriod

This is identical to FTI MINOR FTI as described above, except that it uses the major trend, not the minor.

## FTI LARGEST PERIOD BlockSize HalfLength LowPeriod HighPeriod

This returns the period corresponding to the largest FTI within the specified range of periods (inclusive). Note that this does not use the automated period

selection algorithm. That algorithm finds *two* local maxima and defines the major and minor trends according to the size of the associated periods. FTI LARGEST PERIOD finds the *single* global maximum FTI and returns its filter period as the value of this indicator. For example:

```
FTIBIGPER: FTI LARGEST PERIOD 36 6 5 10
```

The preceding definition computes FTI for periods from 5 through 10 bars. The *HalfLength* of the lowpass filter is 6, which satisfies the requirement of being at least half of the *Period* for all trial periods. A *BlockSize* of 36 provides a *channel length* of  $36-6=30$  bars, which is long enough to get stable FTI values. The period corresponding to the largest FTI found among these trial periods is returned as the computed indicator. It is unlikely that this would ever be of any use as an indicator for a prediction model. However, series and histogram plots of this variable can sometimes be informative.

## FTI MINOR PERIOD BlockSize HalfLength LowPeriod HighPeriod

The user specifies an inclusive range of possible periods, *LowPeriod* through *HighPeriod*. The program uses the algorithm described [here](#) to find the period that characterizes the *minor trend*. This period is the value of the indicator. For example:

```
FTIMINPER: FTI MINOR PERIOD 36 6 5 10
```

The preceding definition computes FTIs using lowpass filters having periods ranging from 5 through 10 bars. The *HalfLength* of each filter is 6. This is legal because it exceeds half the period of the maximum (half of 10 is 5, and 6 equals or exceeds 5). We make the channel contain 30 points by adding 30 to 6, giving us a *BlockSize* of 36. In accordance with the algorithm described [here](#), the program examines FTI for each period. It chooses the two highest FTIs that are local maxima and chooses the smaller of the two associated periods (the minor trend). This smaller period is the indicator. It is unlikely that this would ever be of any use as an indicator for a prediction model. However, series and histogram plots of this variable can sometimes be informative.

## FTI MAJOR PERIOD BlockSize HalfLength LowPeriod HighPeriod

This is identical to FTI MINOR PERIOD, except that the major period is returned.

## FTI CRAT BlockSize HalfLength LowPeriod HighPeriod

The minor/major *channel width ratio* for the specified pair of periods is returned. (See [here](#) for a description of the *channel width ratio*.) Note that unlike the indicators that contain the keywords MAJOR or MINOR, this indicator does not use the major/minor definition algorithm described [here](#). Rather, the user explicitly specifies the periods for the two channels, thus avoiding the instability of the automated algorithm. *LowPeriod* is the filter period for the minor channel, and *HighPeriod* is that for the major channel. This indicator nicely complements the FTI FTI variable, because the exact periods can be provided to best fit the application. In most cases, if the user is employing both FTI FTI and FTI CRAT as indicators, the same low and high periods would be used in both definitions. For example:

**CRAT\_5\_10: FTI CRAT 36 6 5 10**

The preceding definition computes the channel width for a filter period of 5 bars as well as for a period of 10 bars. The *HalfLength* of the lowpass filter is 6, which satisfies the requirement of being at least half of the longest *Period*. A *BlockSize* of 36 provides a *channel length* of  $36-6=30$  bars, which is decently long. The channel width at a period of 5 bars is divided by the channel width at a period of 10 bars in order to compute this indicator.

## FTI MINOR BEST CRAT BlockSize HalfLength LowPeriod HighPeriod

The minor/major *channel width ratio* is returned. (See [here](#) for a description of the *channel width ratio*.) The major period is fixed at *HighPeriod*. All periods from *LowPeriod* up to (but of course not including) *HighPeriod* are searched for local maxima in FTI. The largest local maximum in this range is used for the minor period. This is what Khalsa does in the FTI paper. He fixes the major period at 65 and finds the best associated minor period. The keywords MINOR BEST are used in this definition because we are searching for the best minor trend, with the major trend's filter period fixed by the user. For example:

**MINBESTCRAT: FTI MINOR BEST CRAT 120 35 5 65**

The preceding definition computes the channel width using a filter period of 65 bars. The *HalfLength* of this filter is 35, which is legal because it exceeds half of 65. The channel length is a generous  $120-35=85$  bars. Then it tries all smaller periods, down to and including 5. It chooses the one having the largest local maximum FTI and computes its channel width. This quantity is divided by the channel width at a period of 65 in order to compute the indicator.

## FTI MAJOR BEST CRAT BlockSize HalfLength LowPeriod HighPeriod

This is identical to the FTI MINOR BEST CRAT described above, except that the minor period is fixed at *LowPeriod*. Periods up to and including *HighPeriod* are tried, and the one having the largest local maximum is used for the major trend period.

## FTI BOTH BEST CRAT BlockSize HalfLength LowPeriod HighPeriod

The minor/major *channel width ratio* is returned. (See [here](#) for a description of the *channel width ratio*.) The program uses the algorithm described [here](#) to find the periods that characterize the minor trend as well as the major trend. Compare this with the two prior indicators. FTI MINOR BEST lets the user fix the major trend period, and it finds the best minor trend period. FTI MAJOR BEST lets the user fix the minor trend period, and it finds the best major trend period. But this indicator, FTI BOTH BEST CRAT, uses the automatic algorithm to find both periods, restricting the search to the inclusive range *LowPeriod* through *HighPeriod*. For example:

**FTIBOTHCRAT: FTI BOTH BEST CRAT 120 35 5 65**

The preceding definition computes FTIs using lowpass filters having periods ranging from 5 through 65 bars. The *HalfLength* of each filter is 35, which is legal because it exceeds half of 65. The channel length is  $120-35=85$  bars. In accordance with the algorithm described [here](#), the program examines FTI for each period. It chooses the two highest FTIs that are local maxima and lets the smaller of the two associated periods define the minor trend. The larger period defines the major trend. The minor/major *channel width ratio* is returned as the indicator.

## 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:

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

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:

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

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) to minimum. 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.

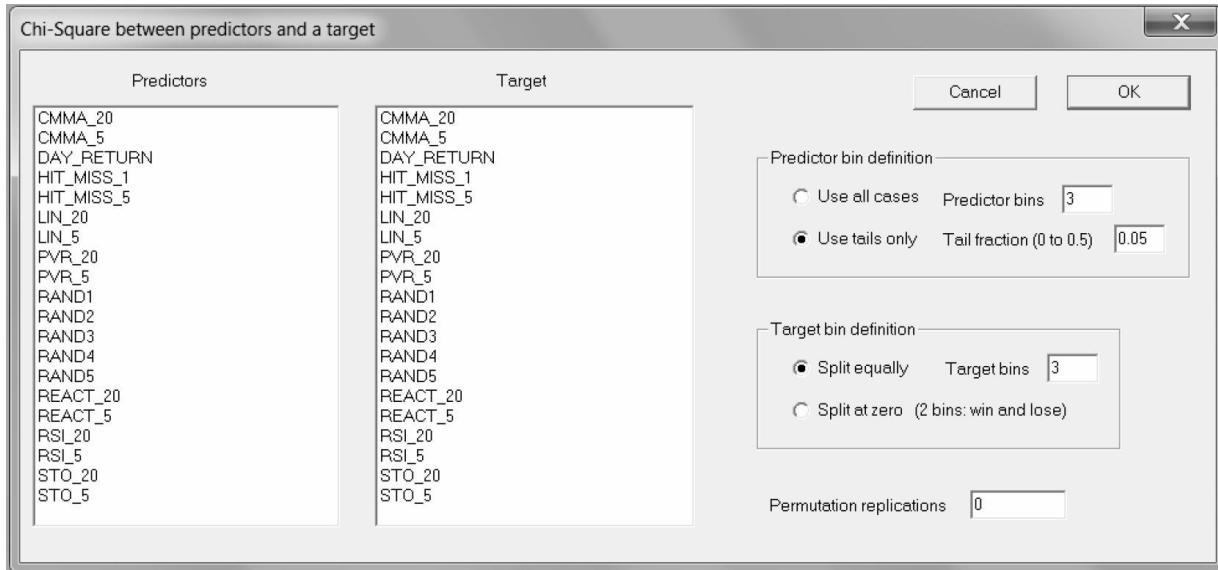
The most important thing to note is that for targets that have little or no serial correlation, the unbiased p-value, which takes selection bias into account, can (and usually does) greatly exceed the solo p-value.

If 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 variables 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:

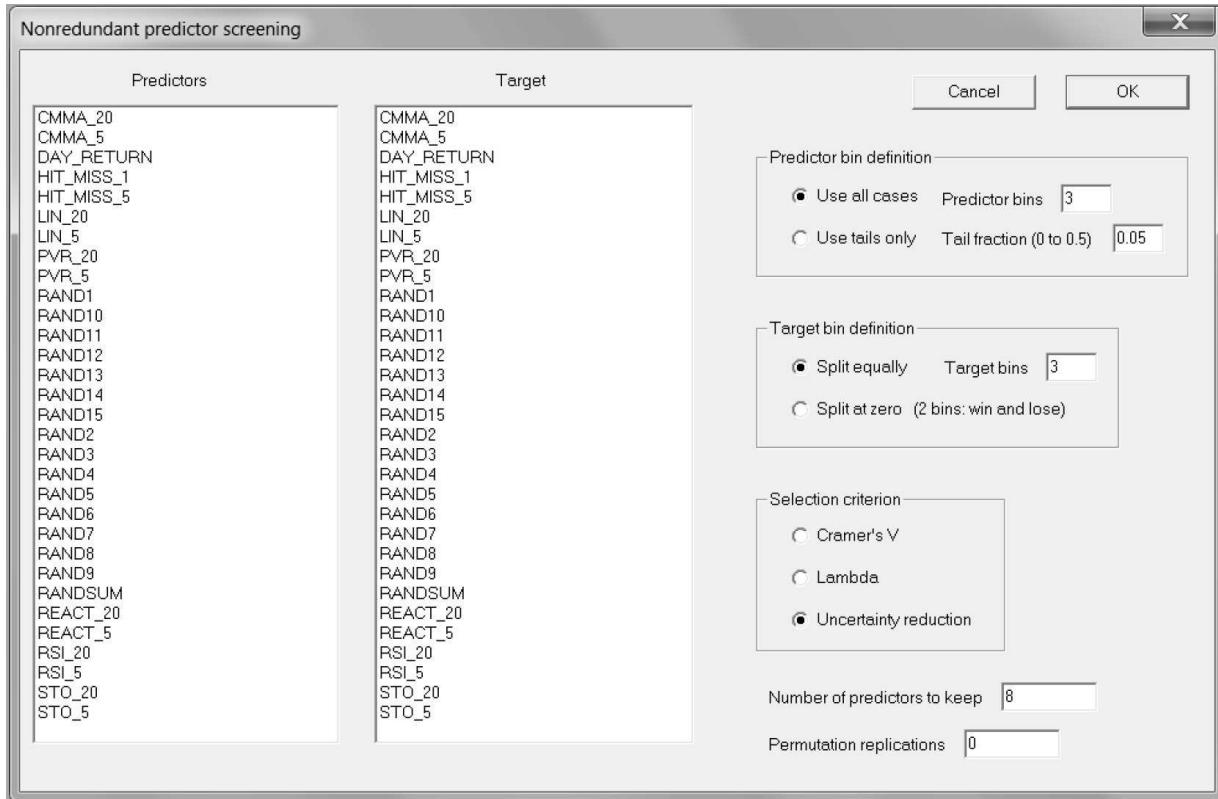
```
NONREDUNDANT PREDICTOR SCREENING  
[ X1 X2 X3 ] ( 3 ) WITH RET ( 3 ) MCPT=100 ;
```

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 variables 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 not 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 others 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	
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*** - Ordinary multiple 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 its 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 user 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 of 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 the 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's 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 above 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 criteria 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. This 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 AREA looks at the entire range of predicted values, while the PROFIT FACTOR criteria look at only the tail area(s), cases beyond an 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 discussed 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 the

threshold. Instead, the threshold is explicitly computed based on the distribution of predicted values. For example, the MEAN ABOVE 10 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 XVAL 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 the mean profit obtained from keeping 90 percent of trading opportunities. Similarly, when used for a short system, the MEAN ABOVE 10 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 in which one wishes to keep 90 percent of all potential trades. MEAN BELOW 10 would be appropriate for a long system that keep 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 are good at eliminating the worst trades in a long system. Note that each of these MEAN BELOW criteria has an analog in the MEAN ABOVE 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 the 90 percent most likely large winning trades. Or you could use MEAN\_BELOW\_10, which would focus on eliminating the 10 percent 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.

See 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 of the selected cases, while the MEAN\_ABOVE criteria are based on 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\_BELOW criteria.

See 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 of the selected cases, while the MEAN\_BELOW criteria are based on 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. But PF\_ABOVE\_90 and PF\_BELOW\_10 are different due to profit factors involving ratios, not sums.

***BALANCED\_01***

***BALANCED\_05***

***BALANCED\_10***

***BALANCED\_25***

***BALANCED\_50*** - The BALANCED family of criteria are most useful for the development of market-neutral trading systems. They are valid only if the FRACTILE THRESHOLD option ([here](#)) 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 thresholds *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 market neutrality. They do this by setting upper and lower thresholds that are balanced. BALANCED\_01 decrees that the highest one percent of 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 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 the 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 complete 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 idiosyncrasies 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 by

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 STEPWISE RETENTION or XVAL STEPWISE to improve on default indicator selection, the 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 profit 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 and RSQ FUTURE SLOPE? These measure the future trend direction of the market, but 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  $\text{prediction} = 2 * \text{fractile} - 1$ . 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 short 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 be trained using only those cases that satisfy the specification. This takes the following form:

```
PRES SCREEN VariableName Relationship Number
```

The *Relationship* can be:

<	<i>Less than</i>
<=	<i>Less than or equal</i>
>	<i>Greater than</i>
>=	<i>Greater than or equal</i>

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:

```
PRES SCREEN VOLATIL_25 < 12
```

It is legal to use multiple PRESCREEN commands in a model definition. For 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:

```
PRES SCREEN TREND_50 >= -20
PRES SCREEN 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 log 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 GROUP 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 be 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 next 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 accomplish 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 shown 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 the same indicators, they will still make quite different predictions and hence be useful to a committee. By specifying a different EXCLUSION GROUP for the two GRNNs (2 for the GRNN; 1 for the LINREG) we allow re-use of the indicators. 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 the 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 3 250. 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 the 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 training case would be the market change from time 1 to that at time 4. But the OVERLAP 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 RETENTION 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 an 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 PREDICTED 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 of 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, we would 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 discussed [here](#). 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 parenthesis, 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)** - The Martin 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 linear 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 actual 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 of 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 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, is 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 modest 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 all applications.

## The Domain of the Neurons

The vast majority of the time, we want the entire MLFN to operate in the real 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 layer 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 the 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 hence 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 and 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. If 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 produced 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 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 in 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 available 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  $X_1$  by 3.7, subtract  $X_2$ , 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 that 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 may be used in any operations. The CONSTANTS IN COMPARE option 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 COMPARE 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 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 in the model definition.

- 2) A less restrictive use of constants also allows comparisons with intermediate results. For example, an operation string might inquire whether  $(X1 + X2) \leq -5.0$ . In order to allow this level of usage, specify the CONSTANTS IN COMPARE 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  $\langle \rangle$ .

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 the 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 (0.999999999) = <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 some 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 place 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 to 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 for 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 LINEAR MINIMUM FRACTION (described soon) specification when the 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 SPLIT LINEAR MAXIMUM INDICATORS times the number of regimes. It 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 SPLIT LINEAR SPLITS = NOISE option is used, it may not always be 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 ensures 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 in 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 two 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 the 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 SPLIT LINEAR MAXIMUM INDICATORS = 2. As it happens, the stepwise procedure 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 different

version of the SPLIT LINEAR model will be employed. In the usual version, one of 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 ]***

This mandatory specification 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 all 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 TREE. 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 already described. Models other than AVERAGE and CONSTRAINED are either too 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 to 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 (the 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 you have reason to believe that there might be considerable differences of predictive ability among the models, the CONSTRAINED committee will 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 GROUP option described [here](#). The other method is to employ different selection criteria (the CRITERION specification described [here](#)). 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 described [here](#)) 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 RESAMPLE 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 those 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 described [here](#) 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 ]***

This mandatory 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 any 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 option ([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 to disregard the predictions of models whose PRESCREEN option(s) disqualify the 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 cited, 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 to 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. The 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 option 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 inappropriate 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’s 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 these 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 of 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 prescreen 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 defined***. 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 or APPEND DATABASE command. Here are a few forms of specialized cross 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 our 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 model 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 prescreening 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 committee 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, CROSS VALIDATE, and WALK FORWARD commands will operate on this reduced set, those cases whose value of *VarName* exceeds 0.5.

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

It is also possible to ‘undo’ the effect of a TRIGGER command. The following 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 with 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 TRIGGER 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 are 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 to specialize in cases for which HIFLAG is true (> 0.5, the result of ADX being greater than 35). Similarly, modelMOD\_LO specializes in LOFLAG being true.

Here is the output for MOD\_HI after the first TRAIN command. That command processes the entire database, but the PRESCREEN option in MOD\_HI causes the 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 that 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 PRESCREEN 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 a 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 we can use a TRIGGER command before training, cross validating, or walking 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 considered 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\_10 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 when 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\_10 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 retain 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 at 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 FIRST 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 log 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" ;

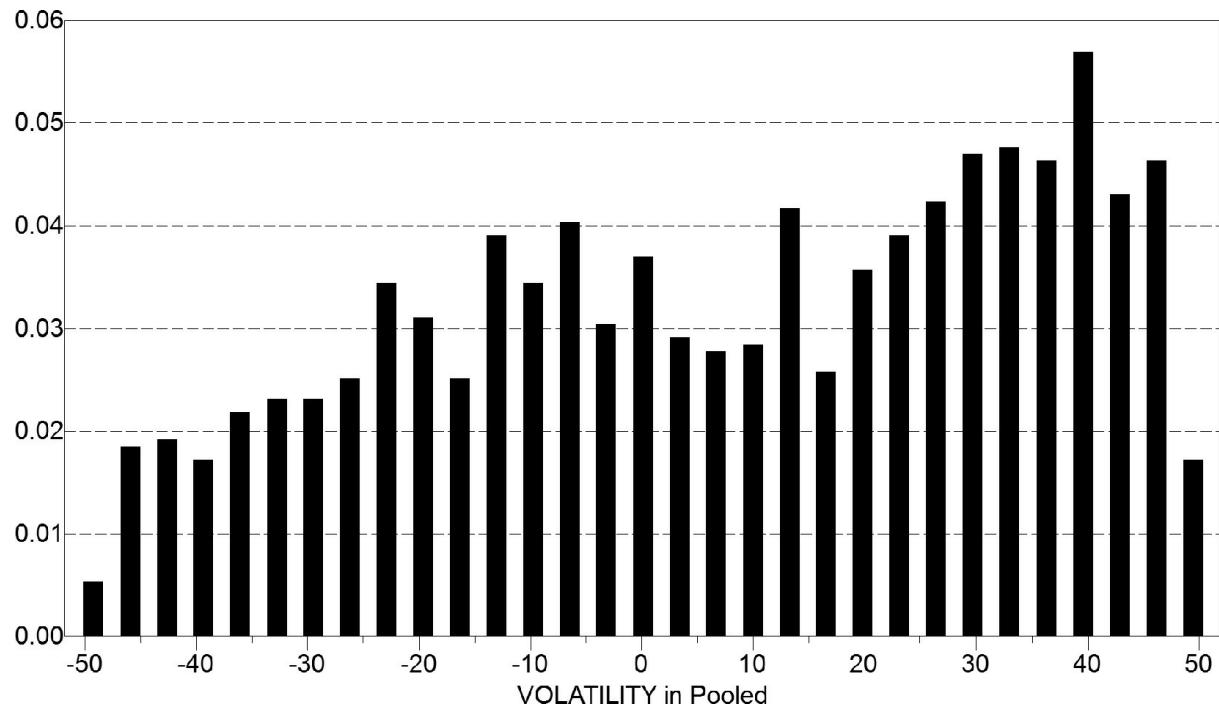
TRANSFORM LIN_TREND IS LINEAR REGRESSION [
    INPUT = [ LIN_ATR_5 LIN_ATR_15 ]
    TARGETVAR = RETURN
] ;

```

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 chosen 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 linear 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 that there is no requirement that the PRESCREEN variable be binary. These models could just as well reference the P\_VOLATILITY indicator directly instead of going 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, does 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 two models. Because the HONOR PRESCREEN option is used, the choice of model is based on the PRESCREEN variable: when a model's PRESCREEN is false, 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 linear 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 created 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-or-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 to 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 it 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 PRESCREEN 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 retain 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 write 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 net 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 of 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 that 278 of 504 database cases passed the volatility test. Why is there a discrepancy (279 versus 278)? The reason 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 to 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 trained only on high-volatility cases, so the transform specializes.
- It happens by coincidence that LIN\_TREND was not picked as a predictor for the model in either example. Thus, we see that model MOD\_HI in the first example and model MOD\_ALL in this example are identical. This makes 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 in 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 PRESCREEN option in MOD\_HI and selected by the TRIGGER command in this third example.
- If the regression transform LIN\_TREND had been picked by MOD\_HI or 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\_TREND is different in these two examples. In the first, PRESCREEN example, LIN\_TREND was computed from the entire dataset. In this third, TRIGGER example, LIN\_TREND was computed from only the high-volatility cases. So 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 164 high-volatility cases.

As a final note, the TRADE SIMULATOR command (described in the manual; a 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 and use the PORTFOLIO command (described in the manual; a tutorial chapter is 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 TRIGGER 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 for 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 different 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 independent (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 to 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 ;
```

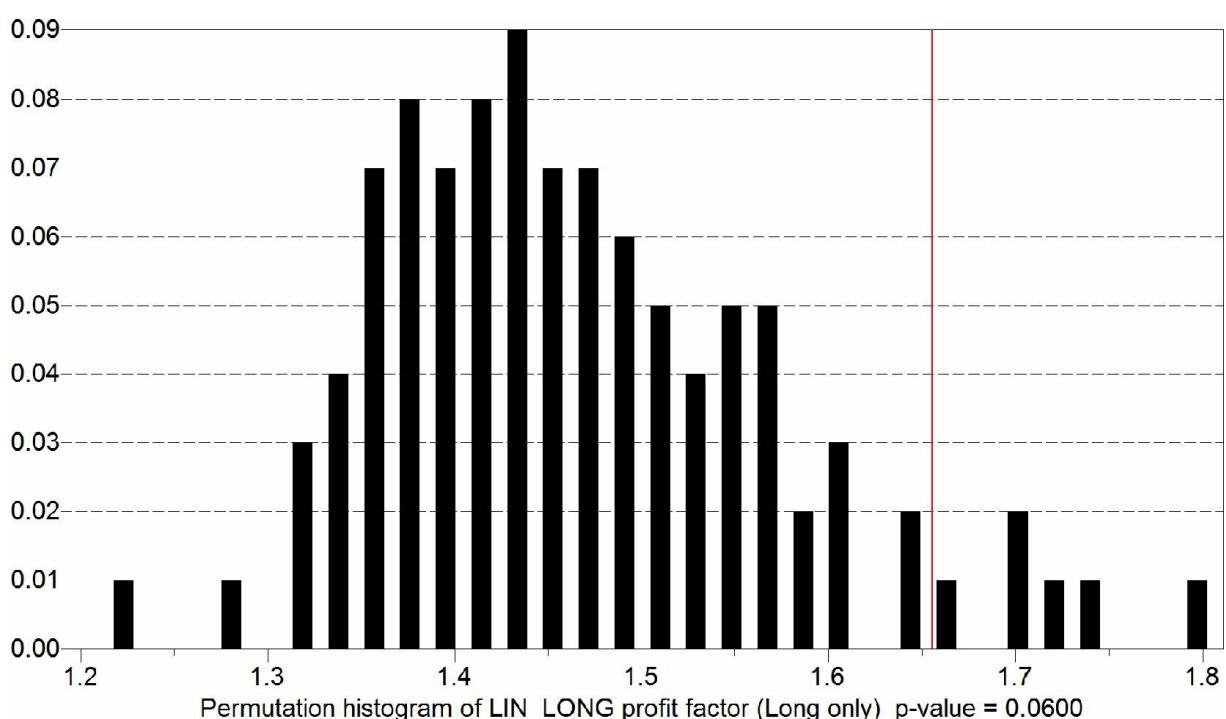
If this command appears, *no database can be read or appended*. This is because all indicators and targets must be able to be recomputed from the original and permuted market(s). This recomputation would obviously not be possible for precomputed indicators and targets in a database. Also, the REMOVE ZERO VOLUME command described [here](#) must appear before the READ MARKET HISTORIES command.

TRAIN PERMUTED operates similarly to the ordinary TRAIN command, except 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 as 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:

$$\text{Total} = \text{Skill} + \text{Trend} + \text{Bias} \quad (9)$$

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.

$$TotalForPermuted = Trend + Bias \quad (10)$$

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:

$$\text{LongExpectedReturn} = \frac{\text{BarsLong}}{\text{TotalNumberOfBars}} * \text{TotalTargets} \quad (11)$$

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.

$$\text{Trend} = \text{NetExpectedReturn} = \frac{\text{BarsLong} - \text{BarsShort}}{\text{TotalNumberOfBars}} * \text{TotalTargets} \quad (12)$$

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.

$$\text{Bias} = \text{TotalForPermuted} - \text{Trend} \quad (13)$$

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 of 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.

$$\text{UnbiasedReturn} = \text{Skill} + \text{Trend} = \text{Total} - \text{Bias} \quad (14)$$

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.

$$\text{Skill} = \text{BenchmarkedReturn} = \text{UnbiasedReturn} - \text{Trend} \quad (15)$$

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 and 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 Portfolios. These can be evaluated only in a WALK FORWARD test because they need OOS 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 to 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, which 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.

```

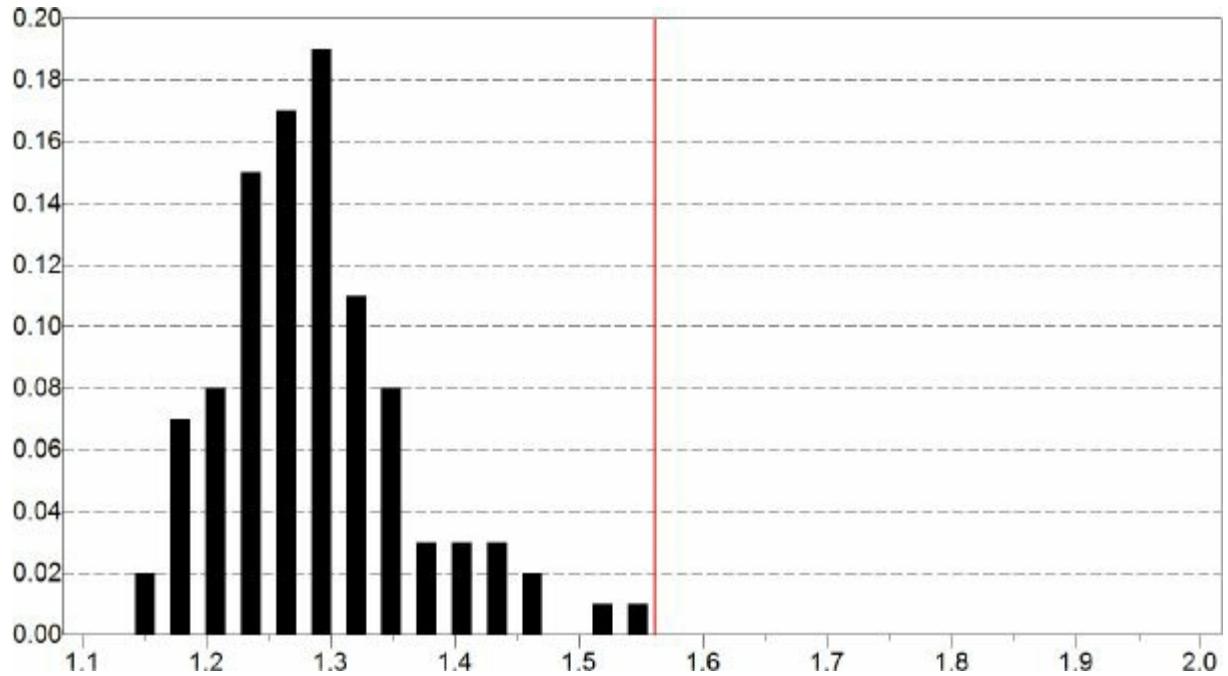
MODEL LIN1 IS LINREG [
  INPUT = [ CMMA_5 CMMA_20 LIN_5 LIN_20 RSI_5 RSI_20
            STO_5 STO_20 REACT_5 REACT_20 PVR_5 PVR_20 ]
  OUTPUT = DAY_RETURN
  MAX STEPWISE = 1
  CRITERION = LONG PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
  EXCLUSION GROUP = 1
] ;

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
] ;

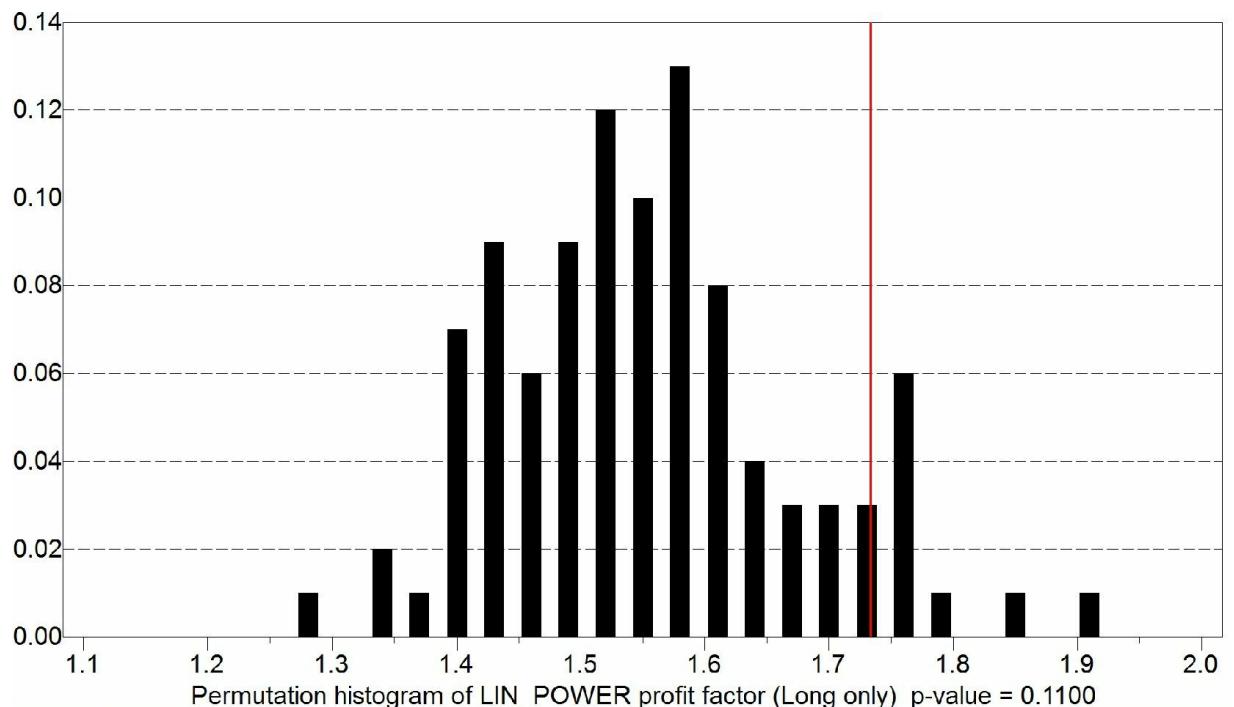
PORTFOLIO Port_Power [
  MODELS = [ LIN1 LIN2 LIN3 LIN4 LIN5 LIN_POWER ]
  TYPE = OPTIMIZE 2 IS
  NREPS = 1000
  LONG ONLY
  EQUALIZE PORTFOLIO STATS
] ;

```

The Portfolio optimizer selected LIN4 and LIN\_POWER (of course) as the best 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 an in-sample computation, and LIN\_POWER is a much more powerful model than LIN4. But along with this greater profit factor for the original data, the profit 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. In 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 are 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 the *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 this 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 be 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 is 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 a 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 final 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 lookback (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 the close of IBM, beginning with the value 5 bars ago. In other words, this will look at 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.

# Principal Component Transforms

The principal components of a set of variables are an equal number of new variables that are linear combinations of the original variables and that have a special property: the first principal component is the linear combination that has the maximum variance of all possible linear combinations. Subsequent principal components also have maximum variance, but subject to the restriction that they be uncorrelated with all prior principal components. In other words, the principal components of a set of indicators have two nice properties that the original indicators almost never have:

- 1) They are uncorrelated with one another.
- 2) Any subset consisting of the first principal component, the second, the third, et cetera, carries the maximum variation inherent in the original indicators that is possible to carry in a subset of that size. Roughly speaking, they carry the maximum variation in the minimum number of new indicators. This technique is an excellent way to reduce dimensionality and redundancy, two important goals in predictive modeling

As a simple example, suppose we have two indicators: trend over the most recent 10 bars, and trend over the most recent 20 bars. Obviously, these two indicators will be highly correlated. Now suppose we compute the principal components of these two indicators. Almost certainly, the first principal component will be something akin to the mean of the two indicators, a more or less generic measure of trend. The second principal component will be the suitably weighted difference between the two trend indicators, a measure of how much or little the trend has changed in the most recent 10 bars relative to what it has been over the last 20 bars.

That example is poor in that using just two indicators does not do justice to principal components. It would probably be as good to just supply the two indicators to the model, rather than bothering with principal components. But their real value comes to light when we have many members of a family. For example, we might measure volatility at four different lookbacks and three different types of volatility at each lookback. This gives us 12 volatility indicators. We may find that just the first three or four principal components capture the large majority of the variation observed in the original 12 indicators. Thus we end up with not only fewer indicators (which reduces the chance of overfitting), but as a nice bonus they are also uncorrelated!

It should be mentioned that capturing the majority of *variation* in the indicators is not necessarily equivalent to capturing the majority of the *predictive*

*information* that they contain. It may be that the most useful information is contained in a small-variance component that we reject. So by computing principal components and keeping the highest-variance subset, we might end up discarding vital information. On the other hand, much practical experience indicates that in most applications, the largest principal components do carry the most predictive information, while the smallest tend to be mostly or entirely noise.

Once the principal components have been computed and a maximum-variance subset of them has been retained, the user can optionally request Varimax rotation of the reduced space. This rotation scheme preserves exactly the same information as is contained in the subset before rotation. Hence, if the indicators will *all* be fed to a *linear model*, the choice of whether to rotate is unimportant because they will give identical results (except for trivial rounding errors in computation). However, if the reduced set of indicators is given to a linear model via stepwise selection, or if they are given to a nonlinear model, rotation can have an impact on performance. This is because the ordering of most-to-least variation is sacrificed in order to often obtain composite indicators that are easier to interpret than principal components. These rotated components tend to have correlations with the original indicators that are either near +/- one, or near zero. In other words, rotation tends to produce new indicators that represent subsets of the original indicators. This may or may not be a good thing as far as model performance is concerned.

## Invoking the Principal Components Transform

This transform is invoked with the following command:

```
Transform TransformName IS PRINCIPAL COMPONENTS [Specs] ;
```

The transform name may consist of letters, numerals, and the underscore character (\_) and may be at most 15 characters. The following mandatory and optional specifications may be included:

**INPUT = [ VarName1 VarName2 ... ]** - This mandatory specification names two or more indicators that are the source for the transformation.

**N KEPT = Integer**

**N KEPT = ALL** - One or the other of these specifications is mandatory. This specifies how many principal components are kept. If more than 8 are specified, coefficients for only the first 8 are printed.

**ROTATION = VARIMAX** - This option decrees that Varimax rotation will be applied to the principal components in order to drive weights toward zero or +/-1. If this option is not specified, ordinary principal components will be output.

The *N KEPT* principal components will be entered into the database. They will be given the name of the transform with the numerals 1, 2, 3, et cetera appended.

So, for example, we might declare a principal components transform as shown below. This declaration uses the indicators X1 through X10 as inputs, and it keeps the first (largest) three principal components. These will be named DemoTran1, DemoTran2, and DemoTran3. These new variables behave just like any other variables in the database. In particular, they can be used as model inputs and plotted with the various menu-accessed plotting functions.

```
TRANSFORM DemoTran IS PRINCIPAL COMPONENTS [
    INPUT = [ X1 - X10 ]
    N KEPT = 3
] ;
```

## Tables Printed

The principal components transform will cause several sets of numbers to be printed in the audit log file. A specific example will appear in the [next section](#). But first, here is a description of what will appear.

The first table printed is called the *factor loadings*. This is the correlation between each principal component and each original indicator. Thus, all entries in this table will be in the range -1 to 1. There is a row for each original indicator and a column for each principal component. This table is headed with two additional rows. The top row is the percent of the total variance which is accounted for by that principal component, and the second row is the cumulative percentage for that column and all prior columns.

At the far right of each row is a quantity called the *communality*. This is the percent of the corresponding indicator's variance that is accounted for by the principal components that are kept. If this quantity is near 100 percent, then the principal components kept capture almost all of the variation in this indicator. Conversely, if this quantity is small, then the retained principal components have kept almost none of the variation in this indicator. This missing variation resides in principal components that were discarded. (If you keep all principal components, the communalities will all be 100 percent.)

If the user specified the ROTATION=VARIMAX option in the principal components declaration, a second table of factor loadings (correlations with the original indicators) is printed. This is the loadings for the rotated factors.

The final table printed is called *Score Coefficients*. Again, there is a row for each original indicator and a column for each principal component kept. This table is of little interest to most people, but it is printed for completeness. These are the coefficients for the linear combination of standardized indicators which would be used to compute the (optionally rotated) corresponding principal component. In other words, if we were to standardize the original indicators by subtracting their means and dividing by their standard deviations, we could then multiply these standardized values by the printed score coefficients and sum them to get the principal component, rotated if so specified.

## An Example

Nine volatility indicators were chosen for this example. PVARRAT\_5, 10, and 20 are the PRICE VARIANCE RATIO at lookbacks of 5, 10, and 20 days. The CVARRAT indicators are the CHANGE VARIANCE RATIO. The ATTRAT indicators are the ATR RATIO. These are all described in the [Variables chapter](#). Here is the principal components declaration, as well as the declaration of a model that uses the newly created variables, along with a single trend indicator:

```
TRANSFORM PCO_ROTATE IS PRINCIPAL COMPONENTS [
    INPUT = [ PVARRAT_5 PVARRAT_10 PVARRAT_20
              CVARRAT_5 CVARRAT_10 CVARRAT_20
              ATTRAT_5 ATTRAT_10 ATTRAT_20 ]
    N KEPT = 3
    ROTATION = VARIMAX
] ;

MODEL ROTATE IS LINREG [
    INPUT = [ LIN_ATR_7 PCO_ROTATE1 PCO_ROTATE2 PCO_ROTATE3
]
    OUTPUT = DAY_RETURN_1
    MAX STEPWISE = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;
```

Observe that in addition to the trend indicator LIN\_ATR\_7, the model specifies as inputs the three principal components produced by the transform. These are the name of the transform, PCO\_ROTATE, with the numerals 1, 2, and 3 appended.

The transform produces the following table of factor loadings (before rotation):

#### Factor loadings...

	1	2	3	Comm
<b>Percent</b>	<b>35.9</b>	<b>20.8</b>	<b>12.7</b>	
<b>Cumulative</b>	<b>35.9</b>	<b>56.7</b>	<b>69.4</b>	
<b>PVARRAT_5</b>	<b>0.322</b>	<b>0.611</b>	<b>-0.437</b>	<b>66.9</b>
<b>PVARRAT_10</b>	<b>0.452</b>	<b>0.080</b>	<b>0.297</b>	<b>29.9</b>
<b>PVARRAT_20</b>	<b>0.248</b>	<b>-0.464</b>	<b>0.740</b>	<b>82.4</b>
<b>CVARRAT_5</b>	<b>0.509</b>	<b>0.597</b>	<b>0.156</b>	<b>64.0</b>
<b>CVARRAT_10</b>	<b>0.834</b>	<b>0.077</b>	<b>-0.066</b>	<b>70.6</b>
<b>CVARRAT_20</b>	<b>0.676</b>	<b>-0.535</b>	<b>-0.278</b>	<b>82.1</b>
<b>ATRRAT_5</b>	<b>0.604</b>	<b>0.539</b>	<b>0.286</b>	<b>73.6</b>
<b>ATRRAT_10</b>	<b>0.835</b>	<b>-0.113</b>	<b>0.043</b>	<b>71.2</b>
<b>ATRRAT_20</b>	<b>0.625</b>	<b>-0.568</b>	<b>-0.354</b>	<b>83.8</b>

Notice that even though there are nine volatility indicators, the first principal component alone accounts for over one-third of the total variance of these nine indicators. Also note that this first principal component has positive correlation, generally large, with all nine indicators. Thus, we can consider the first principal component to be a broad measure of overall volatility.

The second principal component accounts for another 20.8 percent of the total variance, meaning that these two alone account for 56.7 percent of the total variance. This principal component has high positive correlation with lookbacks of 5 days, almost no correlation with lookbacks of 10 days, and high negative correlation with lookbacks of 20 days. In other words, this second principal component measures the contrast between 5-day and 20-day volatility lookbacks, the degree to which the total volatility is produced by the short-term versus the long-term indicator. It is fascinating to note that just these two new indicators alone account for well over half of the total variability of the nine indicators we began with.

The third principal component is more difficult to interpret. We'll leave that to the reader! Also, because it includes only 12.7 percent of the total variance, it is debatable whether it's even worth bothering with.

It's somewhat interesting that these three principal components capture the majority of the variance in all of the original indicators except for one: PVARRAT\_10. The communality of this indicator is just 29.9, barely over one-quarter. This means that the three principal components that we kept are missing the majority of the variation in this indicator. This may or may not be consequential.

Because the declaration specified the ROTATION=VARIMAX option, the rotated loadings are also printed. This table is as follows:

	1	2	3	Comm
<b>Percent</b>	<b>27.4</b>	<b>26.7</b>	<b>15.3</b>	
<b>Cumulative</b>	<b>27.4</b>	<b>54.1</b>	<b>69.4</b>	
<b>PVARRAT_5</b>	0.062	0.417	-0.701	66.9
<b>PVARRAT_10</b>	0.163	0.479	0.207	29.9
<b>PVARRAT_20</b>	0.146	0.186	0.876	82.4
<b>CVARRAT_5</b>	-0.032	0.774	-0.200	64.0
<b>CVARRAT_10</b>	0.585	0.597	-0.085	70.6
<b>CVARRAT_20</b>	0.902	0.034	0.082	82.1
<b>ATTRAT_5</b>	0.018	0.856	-0.058	73.6
<b>ATTRAT_10</b>	0.650	0.526	0.112	71.2
<b>ATTRAT_20</b>	0.913	-0.051	0.037	83.8

It is crucial to understand the difference that rotation makes. Recall that the whole point of rotation is to keep exactly the same information as in the unrotated principal components, but drive the coefficients toward -1, 0, or +1 as much as possible so as to emphasize and de-emphasize the importance of individual indicators. This may or may not be a good thing, depending on the application and the goals of the developer.

With this in mind, it is not surprising to see that the percent of variance captured by the three principal components is the same, 69.4, before and after rotation. Also, the communalities remain the same. This makes sense, because no information has been gained or lost by rotation. But the distribution of variance has changed. The three components are now more equal in their contributions. Also, we no longer have the clear labeling of the first component as overall volatility and the second as the contrast between short-term and longer-term lookbacks. Instead, we see that the first rotated principal component is primarily made up of CVRRAT\_20 and ATTRAT\_20, with secondary contributions from CVRRAT\_10 and ATTRAT\_10. The other five indicators are practically ignored. The second rotated component primarily picks up the 5-day lookbacks. The third rotated component features a contrast between a 5-day and a 20-day lookback, but only for PVARRAT. The other seven indicators make little or no contribution. So rotation has done exactly what it is supposed to do: give up the clear ranking of variance from greatest to least, and replace it with maximum conceptual clarity between indicators. Neither is universally superior to the other, and both convey the same information. The choice of which to use is often a shot in the dark, although engineers do tend to prefer avoiding rotation.

# Linear and Quadratic Regression Transforms

The principal components transform described in the prior section is a common and effective means of reducing the number of indicators by distilling their information down into a much smaller set of derived indicators. However, it suffers from one worrisome drawback: there is no guarantee that the largest-variance principal components, the ones kept, carry the most useful predictive information. It may be that the information we would most like to have lies in one or more of the smaller-variance components that are discarded. The good news is that in practice, this unfortunate situation seems to be the exception rather than the rule. But the bad news is that it can and does happen.

There is a simple way to get around this potential weakness: train a multiple-input model to predict a target that is equal or similar to our ultimate target, and use the predictions of this model as our ‘distilled’ indicator. The problem with this approach is that we get only one indicator out of the transform, whereas with principal components we can get multiple indicators. But there is a bright side: the one indicator that we do get is guaranteed in some reasonable sense to optimally capture the predictive information in the multiple input indicators that will be most useful to us. This can be a lot more reassuring than using principal components and crossing our fingers, hoping that the information we want does not lie in the components we discarded.

TSSB provides two methods for accomplishing this optimally predictive transform. One is simple linear regression, and the other is quadratic regression, the principle of which is described [here](#). These transforms are invoked with the following commands, respectively:

```
Transform TransformName IS LINEAR REGRESSION [Specs] ;  
Transform TransformName IS QUADRATIC REGRESSION [Specs] ;
```

The transform name may consist of letters, numerals, and the underscore character () and may be at most 15 characters. The following two specifications are required:

**INPUT = [ VarName1 VarName2 ... ]** - Names two or more indicators that are the source for the transform.

**OUTPUT = TargetVar** - Names an indicator that will be the target for the transform. The generated variable will be an optimal fit to this target.

All input variables are used, and for the quadratic model, all possible terms up to second order are used. No stepwise selection or internal cross validation is

performed. This is because the point is to create an optimal mapping, not to engender optimal out-of-sample predictions. That is the task of the ultimate prediction model into which the transformed indicators will be fed. Thus, we want to capture all available information.

## A Regression Transform Example

Here is an example of using regression transforms. We use two linear regressions and one quadratic regression. The two linear regressions use different indicator sets, and the quadratic regression uses a different target. It is legitimate to use the same target for a regression transform as for the ultimate model, but it is often better to use a related but not identical target so as to provide variety. Finally, the targets look ahead by 10 days, so all declarations include the OVERLAP=9 option. See [here](#) for details on this option.

```
TRANSFORM LIN1 IS LINEAR REGRESSION [
  INPUT = [ CMMA_5 CMMA_10 CMMA_20 LIN_ATR_7 LIN_ATR_15 ]
  TARGETVAR = FUTSLP_10
  OVERLAP = 9
] ;

TRANSFORM LIN2 IS LINEAR REGRESSION [
  INPUT = [ PENT_1 - PMUTINF_3 ]
  TARGETVAR = FUTSLP_10
  OVERLAP = 9
] ;

TRANSFORM QUAD IS QUADRATIC REGRESSION [
  INPUT = [ CMMA_5 CMMA_10 CMMA_20 LIN_ATR_7 LIN_ATR_15 ]
  TARGETVAR = RSQFUTSLP_10
  OVERLAP = 9
] ;

MODEL REGDEMO IS LINREG [
  INPUT = [ LIN1 LIN2 QUAD ]
  OUTPUT = DAY_RETURN_10
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
  OVERLAP = 9
] ;
```

All regression coefficients are printed in the audit log file. Here is the output for these three transforms:

Training LINEAR REGRESSION TRANSFORM LIN1  
LIN1 read 11187 cases for training

Transform LIN1 regression coefficients:

-0.061653	CMMA_5
0.116553	CMMA_10
-0.185353	CMMA_20
0.024601	LIN_ATR_7
0.112782	LIN_ATR_15
3.027141	CONSTANT

Training LINEAR REGRESSION TRANSFORM LIN2  
LIN2 read 11187 cases for training

Transform LIN2 regression coefficients:

0.014179	PENT_1
-0.016498	PENT_2
0.006523	PENT_3
0.046827	PENT_4
-0.021096	PMUTINF_1
-0.063403	PMUTINF_2
0.029090	PMUTINF_3
2.773316	CONSTANT

Training LINEAR REGRESSION TRANSFORM QUAD  
QUAD read 11187 cases for training

Regression coefficients:

-0.056716	CMMA_5
0.053772	CMMA_10
-0.091056	CMMA_20
0.019502	LIN_ATR_7
0.056980	LIN_ATR_15
0.007442	CMMA_5 Squared
0.009810	CMMA_10 Squared
0.008363	CMMA_20 Squared
0.005146	LIN_ATR_7 Squared
0.010850	LIN_ATR_15 Squared
-0.008801	CMMA_5 * CMMA_10
-0.012168	CMMA_5 * CMMA_20
0.001762	CMMA_5 * LIN_ATR_7
0.013164	CMMA_5 * LIN_ATR_15
0.001154	CMMA_10 * CMMA_20
-0.014178	CMMA_10 * LIN_ATR_7
0.000042	CMMA_10 * LIN_ATR_15
0.005368	CMMA_20 * LIN_ATR_7
-0.019181	CMMA_20 * LIN_ATR_15
-0.003459	LIN_ATR_7 * LIN_ATR_15
0.653100	CONSTANT

# The Nominal Mapping Transform

The vast majority of predictive models, including all of those available in *TSSB*, take one or more numeric values (indicators) as inputs and produce a numeric predicted output. But occasionally we may have an indicator that is *nominal*, meaning that it names classes rather than having a numerical value. For example, the month of the year may be useful as a predictor, but it certainly does not have a numerical value. Even if we code the month as an integer from 1 through 12, that does not make it numeric. It may be a literal number, but it's still just a label.

In many cases there may be a discoverable relationship between a nominal indicator and future behavior of the market. *TSSB* contains a trainable *Nominal Mapping Transform* that computes an optimal (in a reasonable sense) mapping from two or more classes to a single indicator with multiple numeric values. This numeric value can then be used as an input to predictive models.

## Inputs and the Target

There are two different methods for presenting a nominal value to the transform. The most straightforward way is to use the NOMINAL INPUT option (described in the option summary later) to name a single variable whose values define classes. Each unique value of this variable in the database defines a class. This must be a TEXT variable (Page ?).

The other method for presenting the nominal value to the transform is to use the FLAG INPUT option (described in the option summary later) to name two or more variables. Each variable corresponds to a class, and the class of a case will be defined by whichever variable has the maximum value. This is a general method for handling a common technique: a class coding scheme may be binary, in which all variables have the value '0' except for one, which has the value '1' and thereby identifies the class membership of the case. However, *TSSB* takes the more versatile approach of defining the class by the variable having the largest value. This, of course, subsumes the common 0/1 coding scheme.

The user must also name a target variable via the TARGETVAR specification described in the option summary later. This is the variable that determines the ordering of the classes, and it will generally be a variable that looks into the future, although this is not required. For example, suppose we have three classes. Also suppose that the target variable tends to be unusually large for members of Class 2 and unusually small for members of Class 1. Cases in Class 3 are middle-of-the-road. Then the mapping will be computed in such a way that

the new indicator has relatively small values for Class 1, large values for Class 2, and middle values for Class 3. In this way, the nominal indicator that gives the class membership provides a new numeric variable that correlates with the target variable. If the target variable looks into the future, which will nearly always be the intelligent choice, then this new indicator will hopefully have predictive power and be usable as an input to a predictive model.

The output of the nominal mapping transform is computed in one of two ways. The temptation is to just use the mean of each class. However, the heavy tails of many ‘future return’ variables renders this idea useless. It would be more reasonable to use the median of the target in each class. However, much experimentation revealed that even this can be problematic. A more stable method is to rank the target variable across the entire training set and then use the mean target percentile for each class. This preserves the order relationship of the targets while being fairly immune to distribution problems. Percentiles are constrained to 0-100.

The mapping method just described is performed by specifying the MEAN PERCENTILE option described in the option summary later. However, in rare instances the user may wish to remove the mapping rule even further from the target distribution. This can be done by specifying the RANKED PERCENTILE option. In this case, the mean percentiles are computed as just described, and then they themselves are ranked. The map output is thus the percentile of the mean percentile.

## Gates

The nominal transform includes an advanced option that allows subdivision into more categories based on one or two additional inputs called *gates*. Gates are premised on the idea that the assignment of a transform output value to each input class may depend on other conditions. For an analogy, consider the popular ADX indicator. It measures the strength of a trend, but it does not indicate the direction of the trend. So if ADX were used to predict price movement in the near future, the relationship between ADX and future price would be different according to whether the market were trending up or trending down. This is not a perfect analogy, because ADX is continuous while the nominal mapping transform applies to nominal variables, but the idea is the same. We may have one mapping of classes to new indicator values when the market is in one state (category), and a completely different mapping when the market is in a different state. Thus we may have two different categories of class mappings.

We may employ one or two gate variables. A positive value of a gate variable defines one mapping, and a nonpositive value defines another mapping. Thus, if we use one gate variable we will have two different mapping categories, and if we use two gate variables we will have four different mappings.

The idea of gating can be taken a step further. It may be that one or more values of a gate variable signify an ‘undefined’ condition, one which is believed to contribute no useful information. To handle this situation, we need three possible mappings, the choice of which depends on the value of a gate variable. Two of these are the mappings already described, while the third is a ‘neutral’ mapping in which the output value of the transform is not influenced by the presumably uninformative input class. Rather, the output is always set to a ‘neutral’ value, the median of the target, for all input classes. Thus, we can use a gate to tell the transform to ignore the input class for one or more values of the gate variable. This is done by means of the IGNORE option, described in the option summary later. If two gates are employed, the IGNORE option applies only to the first. The program contains no provision for ignoring values of the second gate, if used.

The audit log will contain a table showing the mean target percentile for each category. If a gate is used, the name of the class will be followed by a plus sign to indicate the class corresponding to positive values of the gate, and a negative sign for nonpositive values. If two gates are used, the first sign applies to the first gate, and the second sign applies to the second gate.

## Focusing on Extreme Targets

Mapping from classes to numeric values is a dicey operation when the target variable is extremely noisy, as is the case in market price prediction. If there are more than a very few classes, it is almost certain that noise will cause some errors in assigning numeric values to classes. These errors will probably be of little consequence, because faulty ranking will usually be confined to local areas of the ranking. Maybe the correct ranking should be ABCDEF but the computed ranking turns out to be BADCFE. All that happens is that a class gets switched with a near neighbor. But if local errors are likely, why bother resolving the mapping to such a fine level? We gain little or nothing, for we are only propagating input noise to the output.

It is also well known in market price modeling that the most predictive power usually lies in the tails, the extreme values of the indicators. Central values of the indicators often have little predictive power. This is the motivation for the KEEP option, which will appear in the option summary later.

With the KEEP option, the user specifies a minimum percent of the cases that must be mapped at each extreme output value. Then the program maps as few classes as possible at each extreme, subject to the user's restriction. All other classes (the classes corresponding to central values of the target) are mapped to the median target percentile.

For example, suppose the user specifies that we are to KEEP 30 PERCENT. The program will compute the mapping as described [here](#). Then it will see how many cases are in the class having the minimum output value. If this is at least 30 percent of the total number of cases, it will finish dealing with the low end of the outputs and move on to the high end. But if not, it will advance to the class having the second-smallest output value, and add its cases to the number in the lowest class. This process continues, moving upwards until 30 percent of the cases have been accounted for. At that point, it will repeat the process, but beginning with the class having the largest output value. All of the extreme classes, at both the low and high end, retain their original mapping. But the interior cases are mapped to the median percentile.

## Declaring the Transform and its Options

The syntax for declaring a nominal mapping transform is as follows:

```
Transform TransformName IS NOMINAL MAPPING [ Specs ] ;
```

The transform name may consist of letters, numerals, and the underscore character (\_) and may be at most 15 characters. The following specifications (in addition to the usual OVERLAP option) are available:

**FLAG INPUT = [ Var1 Var2 ... ]** - Lists two or more input variables that will be used as class flags. In typical operation, all of these variables will have the value zero except for one, which will have the value one. This identifies the class membership of the case. However, the program actually defines class membership more generally by saying that the class is determined by whichever variable has the largest value. Either this specification or the NOMINAL INPUT specification must appear.

**NOMINAL INPUT = Variable** - Names a single variable whose value defines the nominal class membership. This must be a TEXT variable (Page ?). Either this specification or the FLAG INPUT specification must appear.

**TARGETVAR = *Variable*** - Names a variable, usually a forward-looking variable, that will be used to train the mapping. The nominal-to-ordinal mapping will be defined so as to maximize a nonparametric measure of correlation between the mapped values and the target variable. This specification is mandatory.

**GATEVAR = *Variable***

**GATEVARS = *Variable1 Variable2*** - It may be the case that a single mapping does not suffice for all possibilities in the application. Specifying a single gate variable allows for two separate mappings, one for positive values of the gate variable, and another for nonpositive values. Specifying two gate variables allows for four mappings as defined by the four possible combinations of positive versus nonpositive for the first and second gates. These are optional.

**IGNORE = (*Value1 Value2 ...*)** - It may be that one or more values of the first gate variable indicate an ‘undefined’ condition, such as a missing input measurement. For example, a popular market pundit may offer one of several prognostications most mornings. You may not believe his/her predictions of upcoming market direction, but you may nevertheless consider them potentially useful, even if they may be anti-predictive! Furthermore, you may believe that these pronouncements map to future market movement in one manner when the market is in a high-volatility state, and in another manner when the market is in a low-volatility state. So you would use a gate variable to specify volatility. But what if some mornings a substitute prognosticator appears, someone whom you believe has no sense of the market. When this happens, you would specify a value for the gate variable that signals to TSSB that the substitute pundit’s prediction is to be ignored. This value (or more than one if needed for complex situations) can be listed in the IGNORE specification. For cases whose value of the first gate variable equals a value in this list, the output variable is set equal to a neutral median value. Values of the second gate variable, if used, cannot be ignored. This option may be used only if at least one gate is employed.

**MAPPING = *MEAN PERCENTILE*** - For each category (nominal class combined with any gates) the output value generated is equal to the mean percentile rank of all target values in that category. This is usually the preferred mapping, although it does assume that the target mapping carries some interval information. (Most standard statistics references define interval scaling.) This is usually a valid assumption. Either this or the MAPPING = RANKED

PERCENTILE specification must appear.

**MAPPING = RANKED PERCENTILE** - The MEAN PERCENTILE mapping is computed, but the category values are then ranked across all categories. The output value is the percentile rank across categories. This is less powerful than the MEAN PERCENTILE method, but it should be used if no interval information in the target can be assumed.

**KEEP Number PERCENT** - It may be that only categories that produce extreme values of the transform are considered useful. Those that produce only a small deviation of the target from its median should be considered noise. This number, which must be specified less than 50 percent, keeps as few classes as possible at each extreme such that at least this percent of cases at each extreme are represented.

## A Nominal Mapping Example

The following script file uses two nominal mapping transforms to demonstrate a useful approach as well as most options. The four VLM indicators used in this example are all members of the VOLUME MOMENTUM family described [here](#). Both of these examples use the FLAG INPUT specification to determine which of the four lookbacks is dominant. This is not a classic binary example, with the flag variables explicitly naming class membership. Instead, this illustrates the more general ability of the transform to choose the largest value from among competing indicators, which is an indirect assessment of class membership.

Here are the transform and model declarations. The OVERLAP=4 option ([here](#)) is used because the target looks ahead five days.

```
TRANSFORM MapDemo1 IS NOMINAL MAPPING [
  FLAG INPUT = [ VLM_3_4 VLM_5_4 VLM_10_4 VLM_20_4 ]
  GATEVARS = LIN_ATR_7 LINDEV_10
  TARGETVAR = DAY_RETURN_5
  MAPPING = MEAN PERCENTILE
  OVERLAP = 4
] ;

TRANSFORM MapDemo2 IS NOMINAL MAPPING [
  FLAG INPUT = [ VLM_3_4 VLM_5_4 VLM_10_4 VLM_20_4 ]
  GATEVARS = LIN_ATR_7 LINDEV_10
  TARGETVAR = DAY_RETURN_5
  MAPPING = MEAN PERCENTILE
  KEEP 20 PERCENT
  OVERLAP = 4
] ;

MODEL Mod1b IS LINREG [
  INPUT = [ MapDemo1 MapDemo2 ]
  OUTPUT = DAY_RETURN_5
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  RESTRAIN PREDICTED
  MIN CRITERION FRACTION = 0.1
  OVERLAP = 4
] ;
```

The class input for these transforms considers only volume, not trend direction. Thus, we would probably not expect for there to be a meaningful relationship between the class membership and the *direction* of the future price movement. With no supplementary information about *historical* trend, this transform would be useless for predicting *future* trend. The volume information provided by the

class input may be useful for predicting the *strength* of the future trend, which in turn may be useful to a predictive model. But if we could also coax direction information from the transform we would be providing something even more useful to the model. This is where the two gate variables come into play.

LIN\_ATR\_7 is a simple trend variable that looks back 7 days. LINDEV\_10 looks back 10 days and uses a linear projection to estimate the most recent closing price. This indicator is the deviation of the actual value from the predicted value, so it detects a sudden departure from recent trend. These two indicators provide a lot of information about the price behavior of the market.

There is a second aspect of gating that is relevant. It would be unreasonable to assume that the relationship between the volume-based input class and future price movement is the same for all trend conditions. It could well be that when the market is trending upward, one volume class corresponds to an upcoming reversal, while when the market is trending downward a different volume class corresponds to an upcoming reversal. We should not count on everything being perfectly symmetric. This is why the nominal mapping transform computes a separate, independent mapping for each gate category.

The output produced by the *MapDemo1* transform appears on the [here](#). There are four input classes, two categories for the first gate, and two categories for the second gate, making a total of  $4*2*2=16$  bins altogether. The output values corresponding to each category are listed in ascending order, along with the number of cases that fall into each bin. The signs enclosed in parentheses after each input ‘class’ name depict the gate status for the class, with the first sign being the first gate and the second sign being the second gate.

For example, here is the third line in the table:

VLM_5_4 (+-)	687	47.455
--------------	-----	--------

This line corresponds to the indicator VLM\_5\_4 having the greatest value of the four input class indicators, LIN\_ATR\_7 being positive, and LINDEV\_10 being negative or zero. There were 687 cases that satisfied this set of conditions. In the future, any case satisfying this set of conditions will be assigned a value of 47.455, which is the mean of the percentiles of targets within this category.

```
Training NOMINAL MAPPING MAPDEMO1
Gate 1 is LIN_ATR_7
Gate 2 is LINDEV_10
MAPDEMO1 read 11187 cases for training
```

**Number of cases and mean target percentile for each class**

Class	N	Mean pctile
VLM_10_4(++)	561	46.060
VLM_20_4(++)	1037	46.874
VLM_5_4(+-)	687	47.455
VLM_10_4(--)	343	47.983
VLM_20_4(+-)	917	48.678
VLM_10_4(+-)	696	48.956
VLM_3_4(+-)	529	49.315
VLM_3_4(++)	1284	49.809
VLM_5_4(+-)	336	49.884
VLM_5_4(++)	537	50.469
VLM_20_4(--)	802	50.823
VLM_10_4(--)	597	51.621
VLM_3_4(--)	805	52.075
VLM_20_4(--)	1115	52.406
VLM_3_4(--)	513	54.415
VLM_5_4(--)	428	54.744

Nothing notable pops out of this chart in regard to the volume-bases input classes, except perhaps the remarkable *lack* of consistency. For example, VLM\_10\_4 has the smallest output in the ++ and -+ gate conditions, but this same input ‘class’ is well above the median for the -- gate condition.

On the other hand, something quite remarkable is revealed for the first gate, LIN\_ATR\_7, a measure of short-term trend. With only one exception, the fourth line in the chart, the smallest outputs correspond to a positive value of this trend. Also with only one exception, the largest transform outputs correspond to negative (or zero) values of this trend. In other words, we clearly see a mean reversion going on here, as opposed to trend persistence.

The second nominal mapping transform, *MapDemo2*, is identical to the first, except that the KEEP 20 PERCENT option is used. This results in the chart of outputs shown below. The 20 percent outer class values are identical to those in the chart just seen, while the central vales are all changed to the mean percentile, 50.004. Twenty percent of 11187 (the total number of cases) is 2237. The lower ‘unchanged’ set contains  $561+1037+687=2285$  cases, and the upper set contains  $428+513+1115+561=2861$  cases. At both ends, this is the smallest number of unchanged bins such that the total number of cases at that extreme is at least 2237. With the KEEP option, information is retained and passed on to the transform output only for those bins which correspond to extreme values of the target. Information in the muddy middle is discarded.

Revised after keeping outer 20.0 percent...

Class	N	Mean pctile
VLM_10_4(++)	561	46.060
VLM_20_4(++)	1037	46.874
VLM_5_4(+-)	687	47.455
VLM_10_4(-+)	343	50.004
VLM_20_4(-+)	917	50.004
VLM_10_4(--)	696	50.004
VLM_3_4(+-)	529	50.004
VLM_3_4(++)	1284	50.004
VLM_5_4(-+)	336	50.004
VLM_5_4(++)	537	50.004
VLM_20_4(-+)	802	50.004
VLM_10_4(--)	597	50.004
VLM_3_4(--)	805	52.075
VLM_20_4(--)	1115	52.406
VLM_3_4(-+)	513	54.415
VLM_5_4(--)	428	54.744

## The ARMA Transform

An ARMA (autoregressive, moving average) model can be fit to a moving window in the database (an indicator) or a market, and any of several quantities can be computed to generate a new database variable. Readers unfamiliar with ARMA models may consult any of the widely available references.

ARMA models use historical values of a time series to estimate future values, often just the next value. The ARMA model does this by exploiting either or both of two tendencies that may be present in a time series: autoregressive (AR) and moving average (MA). The ARMA modeling method looks for these effects and builds an optimal model that incorporates them. The autoregressive tendency refers to the fact that the expected current value of a time series is significantly related to recent past values of the series. The moving-average tendency, as it is used in the ARMA context, refers to the fact that each actual value of a time series deviates from its expected value due to some random shock. The expected current value of the time series is significantly related to recent past shocks that impacted the series. ARMA models can use either or both of these tendencies to create a forecast: the current and recent past values of the series let us use the AR component of the model to predict the next expected value, and the current and past shocks experienced by the series let us use the MA component of the model to predict the next expected value. Both of these predictions (AR and MA) may be combined to make a pooled prediction (ARMA) of the next value of the series. *TSSB* uses the ARMA model to derive a number of different types of indicators that can serve as inputs to a prediction model. The ARMA transform is invoked as follows:

```
Transform TransformName IS ARMA [Specs] ;
```

The following specifications may be included:

**SERIES = *Source*** - This specifies the series which will be fit with an ARMA model. The following sources are available:

***VarName*** - *The name of a variable in the database. If multiple markets are present, each market will be processed independently.*

***VarName:Market*** - *The name of a variable in the database, using values in the specified market. If multiple markets are present, the specified market's computed values will be cloned onto each market.*

**@CLOSE** - *The close of the market will be used. Legal only if there is just one market.*

**@CLOSE:Market** - *The close of the named market will be used.*

*For the above market series, @OPEN, @HIGH, @LOW, and @VOLUME are also legal.*

**ARMA WINDOW = Integer** - This specifies gives the length of the data window. If the ARMA model has only one parameter, a window length of as little as 20 is acceptable, although larger windows, such as 100, are more stable. If two parameters are estimated, a window length of 100 should be considered a minimum.

**AR = Integer** - This specifies the number of AR parameters.

**MA = Integer** - This specifies the number of MA parameters.

**ARMA OUTPUT = Type** - This specifies the value for use as an indicator that will be generated for the database. The following types are available:

**PREDICTED** - *The predicted value of the last case in the window*

**SHOCK** - *The shock (deviation from the predicted value) experienced by the last case in the window*

**STANDARDIZED SHOCK** - *The shock divided by the standard deviation of shocks*

**MSE** - *The mean squared error (variance of the shocks) in the window*

**RSQUARE** - *The fraction of series variance that is predictable by the ARMA model*

**AR PARAMETER Integer** - *The value of the trained AR parameter at the specified lag*

**MA PARAMETER Integer** - *The value of the trained MA parameter at the specified lag*

**ARMA TRANSFORM = LOG** - This optional specification decrees that the source series will be transformed by taking natural logs before the ARMA model is fit. The source series must be strictly positive. This is useful when the source series is market prices.

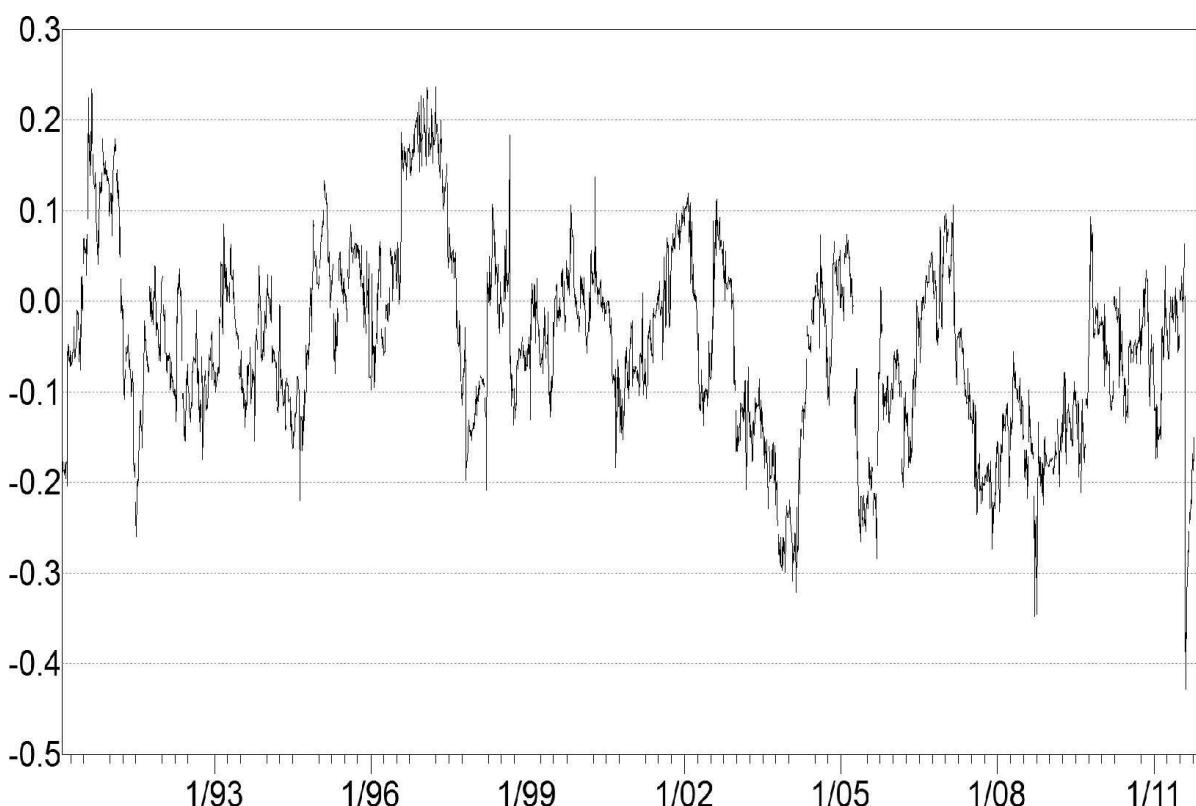
**ARMA TRANSFORM = DIFFERENCE** - This optional specification decrees that the source series will be differenced before the ARMA model is fit. Differencing will often remove or greatly decrease nonstationarity of the mean of a series.

**ARMA UNDO TRANSFORM**- This option makes sense only if at least one of the above transforms is performed and the output is either PREDICTED or a SHOCK. If this option is not used, the predicted

value (and hence the derived shock) remains in the transformed domain. If this option is used, the transform is undone for the prediction, and the shock computed accordingly. Note that if the only transform is a difference, the shock will be the same regardless of whether or not the transform is undone.

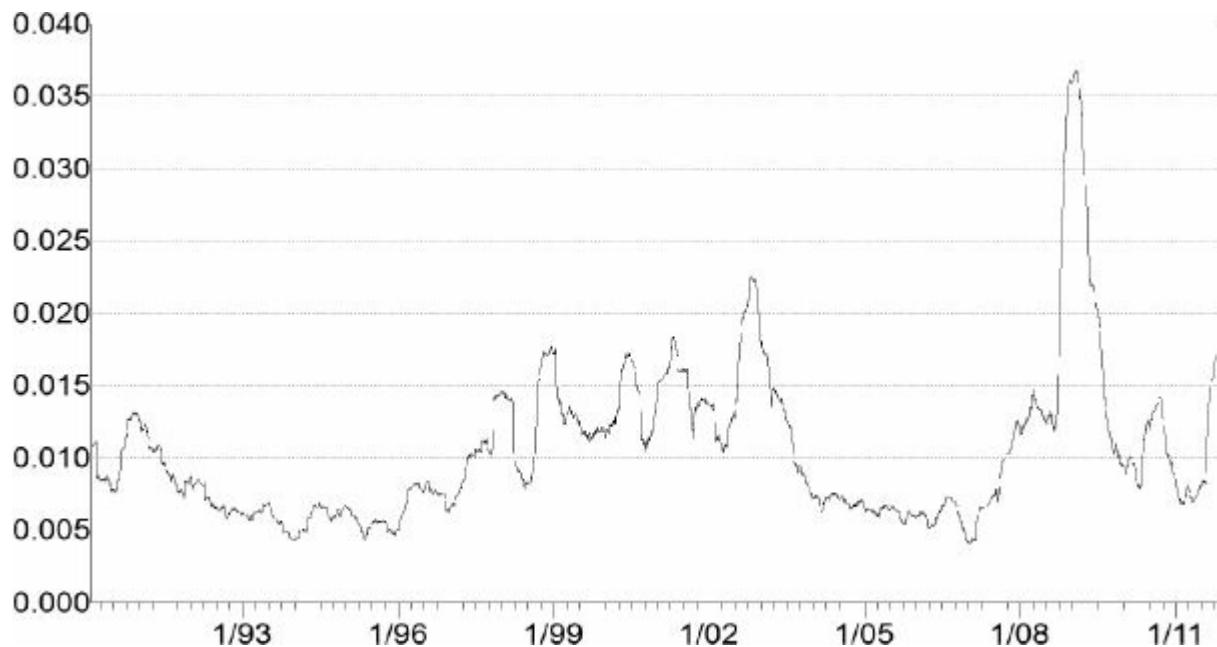
For example, the following transform will fit an ARMA model containing just one AR parameter (a common, stable, and usually effective model; an excellent first choice) and create a new indicator whose value is the AR parameter, which indicates the degree and sign of lag-one serial correlation in the change of the closing value of OEX. This may be useful predictive information because it indicates whether the market is in a trending or mean-reversion mode. Since this is a market, it makes sense to first do a log transform and then compute differences so that the ARMA model is operating on changes in the logs, not actual market prices. The resultant series is graphed at the bottom of this page in [Figure 19](#).

```
TRANSFORM OEX10_LD_AR IS ARMA [
    SERIES = @CLOSE:OEX
    ARMA WINDOW = 100
    ARMA TRANSFORM = LOG
    ARMA TRANSFORM = DIFFERENCE
    ARMA OUTPUT = AR PARAMETER 1
    ARMA AR = 1
    ARMA MA = 0
] ;
```

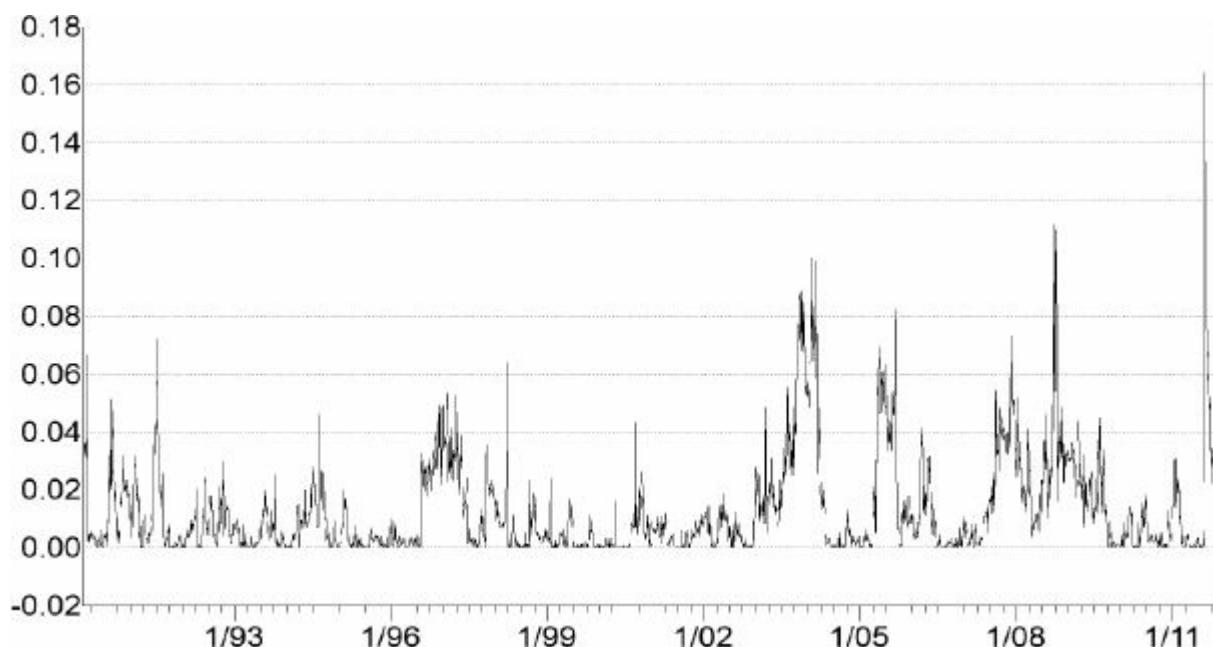


**Figure 19:** The AR parameter for changes in the close of OEX

It is interesting to examine the mean squared error (ARMA OUTPUT = MSE) as the window moves along. This is just the variance of the shocks within the window. This series, shown in [Figure 20](#) below, is obviously worthless as an indicator because it is so nonstationary. But we may find the corresponding R-square (ARMA OUTPUT = RSQUARE) to be of some value. It is shown in [Figure 21](#).

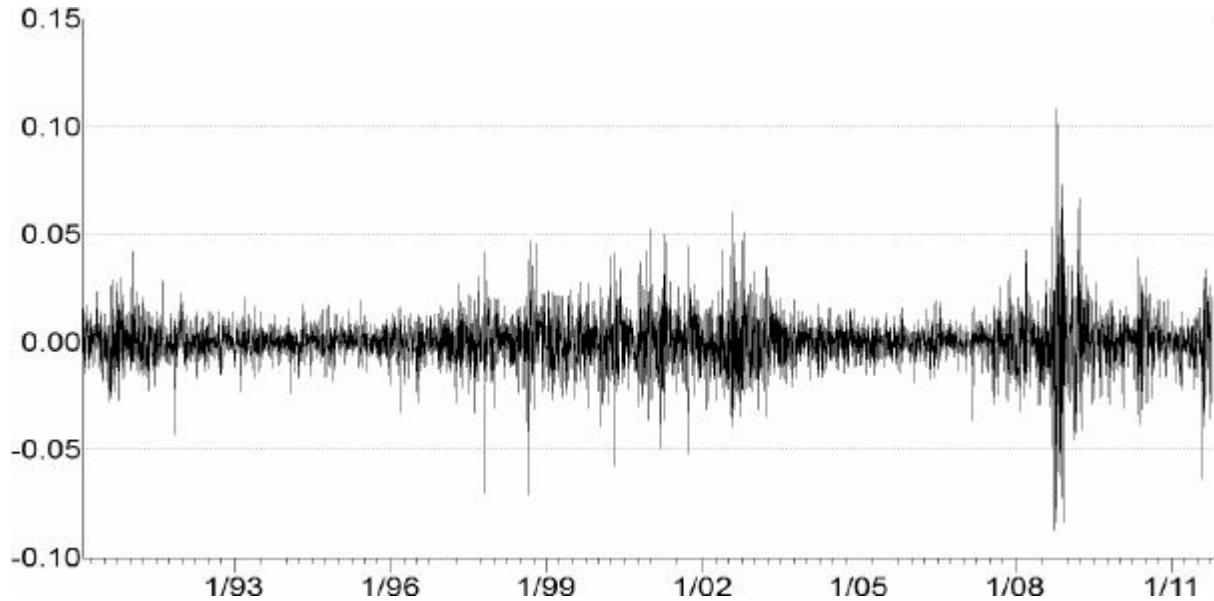


**Figure 20:** Variance of the shocks

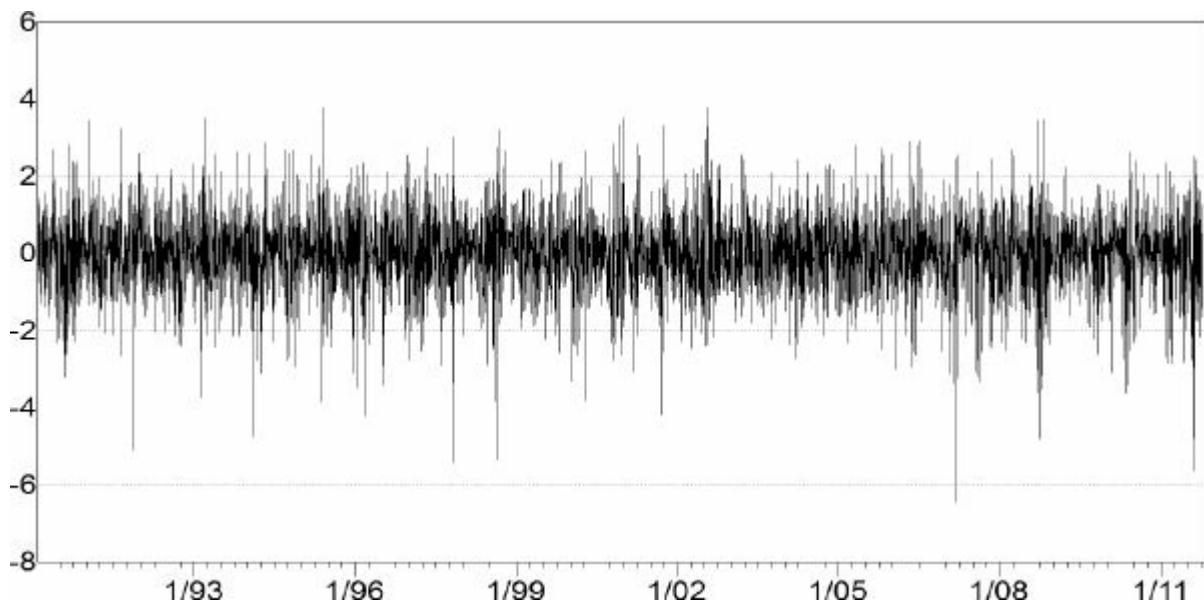


**Figure 21 :**R-square within the window

In many cases, the indicator we are most interested in is the shock series. For each day, this is the component of today's move that is not predictable by the ARMA model. The shocks for the OEX series ( $ARMA\ OUTPUT = SHOCK$ ) are plotted in [Figure 22](#) below. Clearly the indicator is too nonstationary in its variance for this series to be usable in a prediction model. However, by dividing each day's shock by the standard deviation of the shocks within the window we can standardize them and obtain the much better behaved series ( $ARMA\ OUTPUT = STANDARDIZED\ SHOCK$ ) shown in [Figure 23](#).



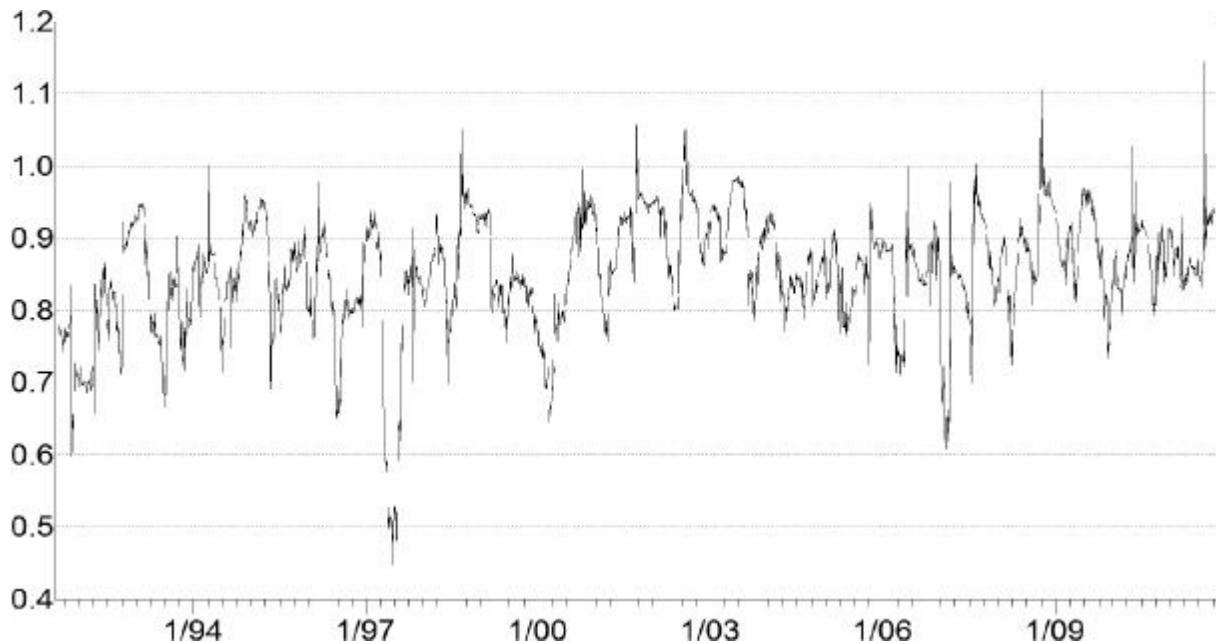
**Figure 22:** ARMA shocks as the window moves



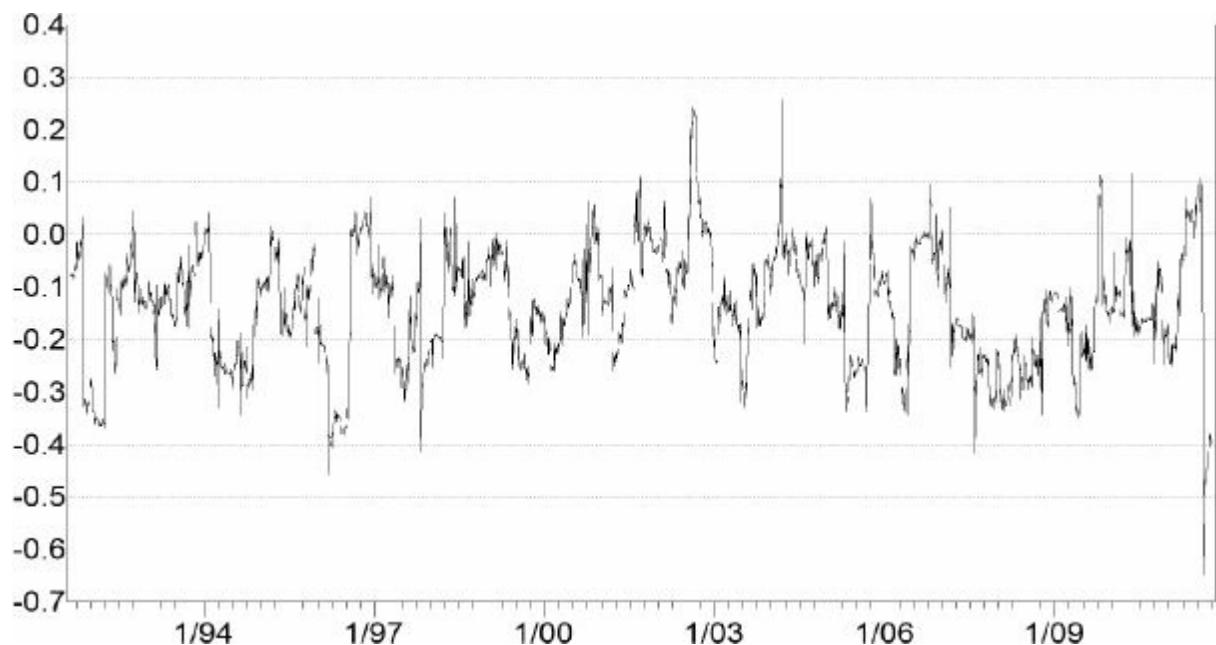
**Figure 23:** Standardized shocks (shocks divided by the window shock std dev)

The VIX series is extremely nonstationary, so one would be inclined to difference it to induce, or at least improve, stationarity. On the other hand, the

use of a 100-day moving window produces decent stationarity within each window, meaning that an ARMA model may be effective. We'll try each. [Figure 24](#) below shows the AR parameters of the raw series, while [Figure 25](#) shows that for the differenced series. It's interesting to note the strong tendency in the latter toward negative serial correlation. See [here](#) for the script declarations that produced these series.



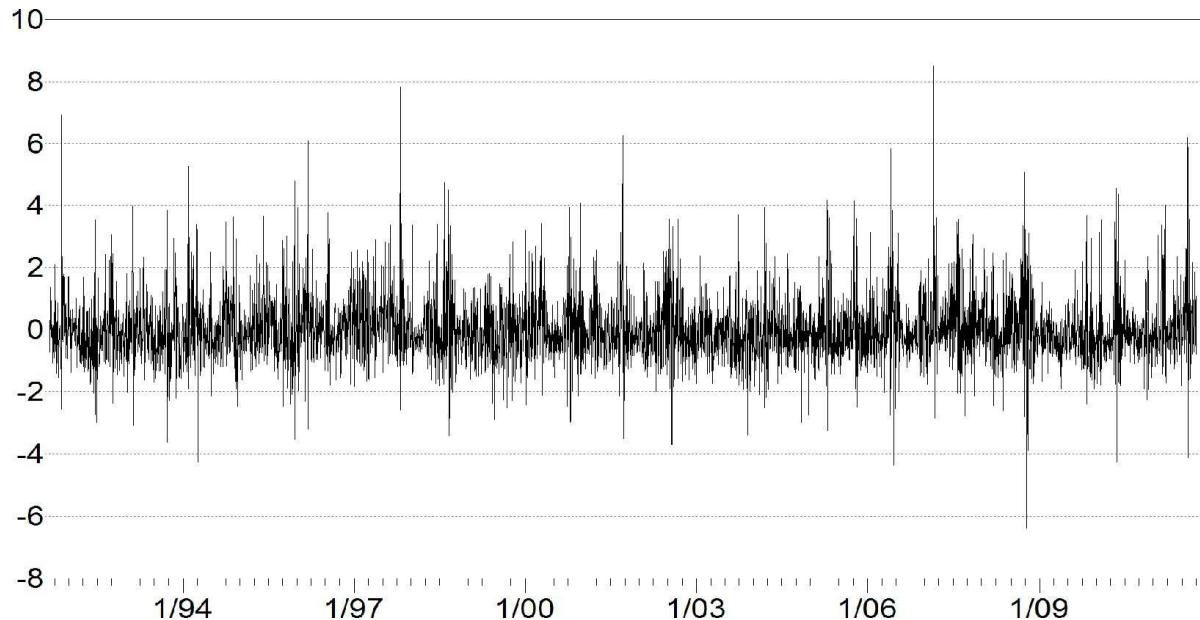
**Figure 24:** AR parameters of the original VIX series



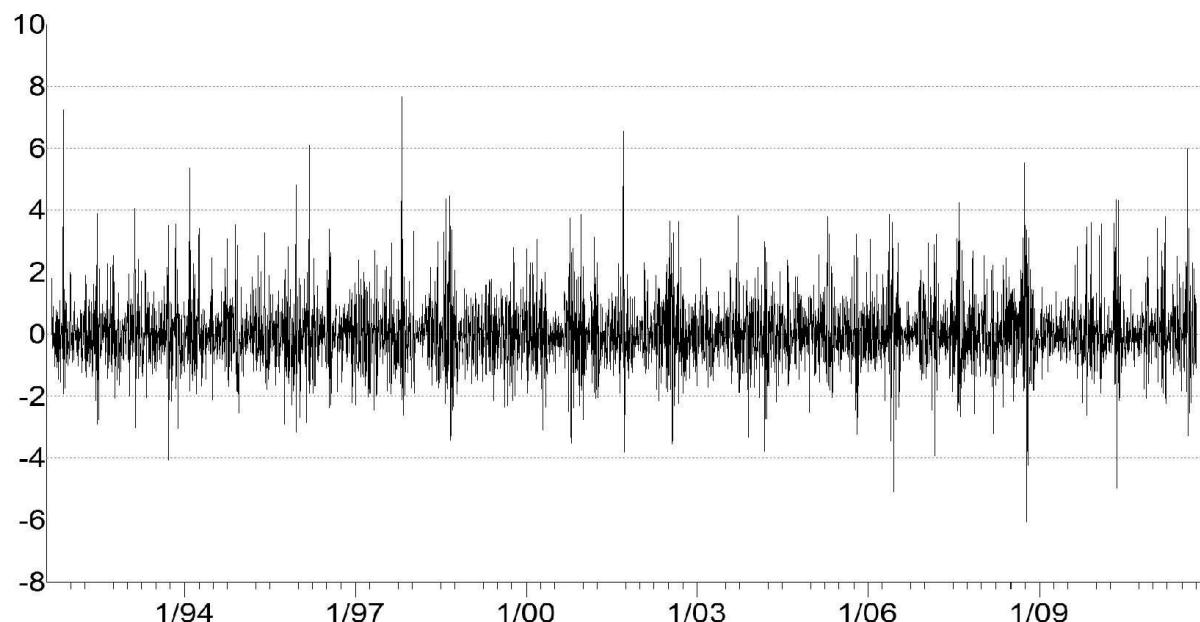
**Figure 25:** AR parameters of the differenced VIX series

As with OEX, we are probably most interested in the shock series as an indicator. [Figure 26](#) below is the standardized shocks for the original VIX

series, and [Figure 27](#) is for the differenced series. Notice how similar they are, even though the ARMA models are totally different. This makes sense, because the shock each day is the unpredictable component of the new value, and this should be about the same, whether you are predicting an actual value or a difference. However, if you look closely you will see that the shocks from the differenced series are slightly more stationary in the mean than those from the raw series.



**Figure 26:** Standardized shocks of the original VIX series



**Figure 27:** Standardized shocks of the differenced VIX series

Here are the script file declarations that produced Figures [24](#) through [27](#):

```
TRANSFORM VIX10_AR IS ARMA [
    SERIES = @CLOSE:VIX
    ARMA WINDOW = 100
    ARMA OUTPUT = AR PARAMETER 1
    ARMA AR = 1
    ARMA MA = 0
] ;
```

```
TRANSFORM VIX10_D_AR IS ARMA [
    SERIES = @CLOSE:VIX
    ARMA WINDOW = 100
    ARMA TRANSFORM = DIFFERENCE
    ARMA OUTPUT = AR PARAMETER 1
    ARMA AR = 1
    ARMA MA = 0
] ;
```

```
TRANSFORM VIX10_STD IS ARMA [
    SERIES = @CLOSE:VIX
    ARMA WINDOW = 100
    ARMA OUTPUT = STANDARDIZED SHOCK
    ARMA AR = 1
    ARMA MA = 0
] ;
```

```
TRANSFORM VIX10_D_STD IS ARMA [
    SERIES = @CLOSE:VIX
    ARMA WINDOW = 100
    ARMA TRANSFORM = DIFFERENCE
    ARMA OUTPUT = STANDARDIZED SHOCK
    ARMA AR = 1
    ARMA MA = 0
] ;
```

## The PURIFY Transform

It is a universal fact of life in predictive modeling that useful predictive information in an indicator is diluted by variability attributed to other, less relevant phenomena. Such phenomena may contribute little or no useful predictive information, and hence be essentially noise. If the developer is able to identify a potential source of the noise pollution, it may be possible to remove it and thereby purify the original indicator, increasing its predictive power.

The method for doing this is straightforward: first one must identify a series, measured concurrently with the ‘polluted’ indicator, which is believed to be a source of pollution. This series, called the *purifier*, may be prices of a market, or an indicator itself, or some other market measure such as volume or open interest. Then apply one or more moving-window functions to the purifier series. Find a model (linear is fast and easy) which does a decent job of using these functions of the purifier to predict the indicator being purified. Finally, subtract the predicted values of the indicator from the actual values of the indicator. This predicted value is, to at least some degree, the pollutant. Thus, subtracting it from the series to be purified has the effect of reducing the pollution in the indicator, leaving behind a higher concentration of useful information.

The following command is used to purify a series:

```
TRANSFORM TransformName IS PURIFY [Specs] ;
```

Some of the specifications are mandatory, and others are optional. All possible specifications are now discussed, grouped by their uses.

### Defining the Purified and Purifier Series

The user must specify two series. These are the series to be purified, and the purifier series. As with all transforms, the generated output series is given the name of the transform, *TransformName*, in the definition. The purified and purifier series may be any of the following:

- **The open, high, low, close, or volume of a market.** The purified series is invariably an indicator, not actual trading prices of a market. Nonetheless, it is often advantageous to either read these two series as TSSB markets (READ MARKET HISTORIES) or let them be ‘markets’ that have been operated on by a prior transform. Computing them from the built-in

library, or reading them in from an external database, can, in some instances, deprive the user of valuable early history information. This will be discussed in detail later in this section.

- **An indicator computed from the built-in library.** If this is done, the length of early history available to the transform will be reduced by the maximum lookback in the variable definition list (READ VARIABLE LIST). If any internally computed indicator has a longer lookback than the indicator(s) needed for the purified or purifier series, data will be wasted. Thus, use of the internal library may better be done in a separate script file if available history is limited. The separately computed indicators and the PURIFY transform values can be merged for training and testing of trading systems by means of an APPEND DATABASE command.
- **A variable read in from an external data file with the APPEND DATABASE command.** If this is done, the user should ensure that the external data file starts at as early a date as possible in order to maximize the length of history available to the transform. The appending is done *before* transforms are computed, so even if extensive historical data is available prior to appending, if the appended data file starts at a later date, the previously available data will become unavailable. Also, realize that the PURIFY transform, like all transforms, preserves market associations. Thus, the file read should contain data for any market(s) that will ultimately be traded.

Exactly one PURIFIED and one PURIFIER specification are required. The following commands are used to define the purified and purifier series:

**PURIFIED = @ close:MarketName**

*This names a market whose close will be purified. It is also legal to use the ‘open’, ‘high’, ‘low’, or ‘volume’ of a market. Because the purified series is virtually always an indicator, in practice this series will not actually be trading prices of a market. Rather, it will be an indicator that is read in as if it were a market by means of the READ MARKET HISTORIES command. Usually this is the best approach because this method provides the most history to the transform, and also because data read in as a TSSB market is automatically cloned to all markets named in the READ MARKET LIST file, which simplifies model training and testing.*

**PURIFIED = VariableName**

*This names an indicator that will be purified. This indicator may have been computed from the built-in library, or computed from a prior transform in the script file, or read in from an external*

database. One powerful technique which gives the user great versatility but does not generally cost availability of historical data is to read in a variable as a TSSB market, apply an expression transform, and then use the result of this expression transform as the purified or purifier series.

**PURIFIER = @ close:MarketName**  
**PURIFIER = VariableName**

The same considerations just discussed for the purified series apply to the purifier series, with one small exception. The only difference is that although the purified series will nearly always be an indicator, the purifier series will often be actual market price history.

**PURIFY USE LOG PURIFIER**

This optional command decrees that the natural log of the purifier series will be taken before predictor functions are computed from the series. This is usually appropriate when the purifier series is market price histories, and usually not needed when the purifier series is an indicator.

## Specifying the Predictor Functions

The user must specify at least one predictor function and at least one lookback length for the predictor function(s). The user must also specify whether one or two predictors are to be used in the linear model that uses the functions to predict the purified series. The relevant specifications are the following:

**PURIFY N PREDICTORS = Integer**

This is the number of predictors (functions of the purifier series) that the purification model will use to predict the purified series. It must be 1 or 2. If 2 is specified, all possible pairs of predictors will be tested, so execution time could be slow if the user employs numerous predictor candidates.

**PURIFY LOOKBACKS = ( Integer Integer ... )**

This list of one or more integers specifies the lookbacks (window lengths) in the purifier series to use when computing the predictor functions. Each must be at least 3. This specification must appear if any predictor function other than VALUE (defined below) is used.

**PURIFY PREDICTOR FAMILY = TREND**

*This makes available to the purification model the linear trend of the purifier series, measured over each lookback distance given in the PURIFY LOOKBACKS specification.*

**PURIFY PREDICTOR FAMILY = ACCELERATION**

*This makes available to the purification model the quadratic component (rate of change in trend) of the purifier series, measured over each lookback distance given in the PURIFY LOOKBACKS specification.*

**PURIFY PREDICTOR FAMILY = ABS VOLATILITY**

*This makes available to the purification model the volatility of the purifier series, measured over each lookback distance given in the PURIFY LOOKBACKS specification. The volatility here is the exponentially smoothed absolute change.*

**PURIFY PREDICTOR FAMILY = STD VOLATILITY**

*This makes available to the purification model the volatility of the purifier series, measured over each lookback distance given in the PURIFY LOOKBACKS specification. The volatility here is the standard deviation of the changes in the purifier series, annualized by multiplying by  $\sqrt{252}$ .*

**PURIFY PREDICTOR FAMILY = Z SCORE**

*This makes available to the purification model the moving z-score of the purifier series, measured over each lookback distance given in the PURIFY LOOKBACKS specification. The z-score is found by computing the mean and standard deviation of the purifier series across the lookback window, subtracting the mean from the current value of the series, and dividing by the standard deviation. Nonlinear compression combined with hard limiting is used to prevent extreme values due to tiny standard deviations.*

**PURIFY PREDICTOR FAMILY = SKEW**

*This makes available to the purification model an order-statistic-based measure of skewness.*

**PURIFY PREDICTOR FAMILY = VALUE**

*This predictor specification is different from all of the others. The VALUE predictor makes available to the purification model the current value of the purifier series. It is not a function, and hence lookback distance is irrelevant. The VALUE option would almost never make sense when the purifier series is market price history. However, it may be that the user has contrived a special predictor for the purification model, a predictor that is designed to be used*

*itself, as opposed to functions of it being used. In this case, the VALUE predictor would be specified. The PURIFY USE LOG PURIFIER option is ignored for the VALUE predictor; the actual value is always used.*

## Miscellaneous Specifications

This section discusses the PURIFY transform specifications that do not fit in the prior categories. These are as follows:

### **PURIFY WINDOW = Integer**

*This mandatory line specifies the number of cases (the lookback length) that are used to train the purification model. It must be at least 3, and is usually much larger, perhaps as much as several hundred.*

### **PURIFY NORMALIZE = PERCENT**

*By default, the output of the PURIFY transform is the actual value of the purified series minus the predicted value. But if the PURIFY NORMALIZE = PERCENT option appears, this difference is divided by the absolute value of the purified series and then multiplied by 100. This expresses the difference as a percent of the actual value. If, as is often the case, the local standard deviation of the purified series is proportional to its absolute value, the PERCENT normalization can stabilize the variance of the transform output. Note that purifying the log of such a series also produces variance stabilization, and usually does so better than the PERCENT option.*

### **PURIFY NORMALIZE = STDERR**

*By default, the output of the PURIFY transform is the actual value of the purified series minus the predicted value. But if the PURIFY NORMALIZE = STDERR option appears, the difference is divided by the standard error of the estimate produced by the purification model. Note that near the beginning of the available history, when the model has incomplete information, the standard error can drop to near zero, producing huge z-scores. For this reason, the output of the PURIFY transform is strongly constrained near the beginning of the data series, and moderately constrained thereafter to prevent extreme values that can hinder subsequent model training.*

### **PURIFY OUTPUT = FIRST PREDICTOR**

*This option is for diagnostic purposes only. It disables purification. The output of the PURIFY transform when this option is employed is the first PURIFY PREDICTOR FAMILY variable named in the option list, computed at the first lookback in the PURIFY LOOKBACKS list. For example, suppose the user had the following option as the first predictor: PURIFY PREDICTOR FAMILY = Z SCORE. Then the output of the transform would be the exact z-score values used as predictors by the purification model. Subsequent predictors and lookbacks other than the first in the list are ignored. This option can occasionally be a handy way to inspect the predictors in order to verify visually (or even numerically) that they look like what you expect them to look like. It's unlikely that casual users would ever employ this option.*

## Usage Considerations

The PURIFY transform is integrated into the train/test cycle like other transforms, and it mostly behaves the same as others. However, there are two aspects of the nature of purification itself that merit special consideration.

First, purification almost always has a very long total lookback length. It requires extensive history before it has the full amount of data needed to meet the user's specifications. This is because two windows are involved. At the outermost level we have the PURIFY WINDOW lookback which tells the transform how many cases will go into training the purification model. Then, for each case in this window, we have the windows for computing the predictor variables, the longest of which in the PURIFY LOOKBACKS list may be quite long. So when computing the purified value of any record (bar of data in a market), the oldest case in the training set for the purification model will be back in history by the PURIFY WINDOW distance, and in order to compute the predictors for this case we will need an additional historical window whose length is the longest lookback in the PURIFY LOOKBACKS list. It's not unusual to need a total lookback of several years, which is substantial if data is limited. Computation does begin immediately, without waiting for the full lookback to be reached. However, the first few values will be seriously in error due to massive lack of data, and it will probably be on the order of a hundred or so records before accuracy becomes even marginally usable. Full accuracy is not obtained until the entire total lookback distance is reached.

For this reason we are inspired to make as much history as possible available to the PURIFY transform. In most situations the easiest way to do this is to do the purification alone, in a single script file, and write the purified series as a

database using the WRITE DATABASE command. Read both the purified and the purifier series as TSSB markets. Have a token variable list, probably containing just one variable definition which has an extremely short lookback. (This is needed because the current version of the program requires that either a variable list or database have been read before any transform is done. This requirement may be eliminated in a future version.) It is perfectly legitimate and often useful to then apply one or more expression transforms to the ‘markets’ read, and then purify the output of the transform. This does not cause the loss of history, so it is innocuous. On the other hand, early history of a length equal to the longest lookback in the variable definition list *will* be eliminated, which is why we want to keep the lookback in the variable definition list very short.

If the trading model(s) being developed require other indicators, they can be computed in a separate script file and saved to a database. Their lookbacks will cost market history, but that’s not a problem, because the purification will have been done separately, and the history lost due to lookbacks from the internal indicator library will just be early ‘warm-up’ data in the purified series, data that is probably best discarded anyway!

Of course it’s legal for the purified and/or purifier series to be computed from the internal library, or read in from an outside source with an APPEND DATABASE command. Just be aware of the potentially long lookback requirement of purification and try hard to provide historical data that begins at as early a date as possible.

The second issue of purification is that the purified and purifier series must be precisely concurrent in history if full accuracy is to be maintained. Suppose one series has missing data on a certain date. Then even though the ending dates of the two series are aligned to the ‘current’ date, the two series will remain aligned only as far back as the missing data. After that, the measurement dates of the two series will become misaligned and accuracy can suffer.

In order to prevent this from happening, the first time a PURIFY transform is encountered in the script file the database will be preprocessed to eliminate any dates that do not have a full complement of markets. When this happens, a line similar to the following will appear in the audit log:

```
Removing database records with incomplete markets reduced  
cases from 11924 to 11918
```

In this example, the database originally contained 11924 records, but 6 of them did not have a complete set of records, leaving 11918 complete dates. This is a one-time operation.

## A Simple Example

We now examine a simple two-part example. The first script file purifies the log of a ‘market’ called VIX, and the second file uses the purified value to make trade decisions. Here is the first script file:

```
READ MARKET LIST "OEX_VIX.TXT" ;
READ MARKET HISTORIES "E:\SP100\OEX.TXT" ;
READ VARIABLE LIST "VARS_PURIFY_DEMO1.TXT" ;

TRANSFORM LOG_VIX IS EXPRESSION ( 0 ) [
    LOG_VIX = LOG ( @CLOSE:VIX )
] ;

TRANSFORM PURIF_LOG IS PURIFY [
    PURIFIED = LOG_VIX
    PURIFIER = @close:OEX
    PURIFY USE LOG PURIFIER
    PURIFY WINDOW = 300
    PURIFY LOOKBACKS = ( 11 22 44 65 130 260 )
    PURIFY N PREDICTORS = 2
    PURIFY PREDICTOR FAMILY = TREND
    PURIFY PREDICTOR FAMILY = ACCELERATION
    PURIFY PREDICTOR FAMILY = VOLATILITY
    PURIFY PREDICTOR FAMILY = Z SCORE
] ;

TRAIN ;

WRITE DATABASE "T_PURIFY_DEMO2.DAT" ;
```

The market list OEX\_VIX.TXT contains only two markets: VIX, which is the market sentiment series which will be purified, and OEX, the market index that will be the purifier series.

The variable list VARS\_PURIFY\_DEMO1.TXT contains only one token indicator: **DUMMY: LINEAR PER ATR 3 5**

VIX tends to have a standard deviation proportional to its magnitude, so taking its log before purification stabilizes its variance, which is a crucial property of stationarity. Like all transforms (except ARMA), the expression transform used to compute the log of VIX does not cost any market history.

The purifier series is the close of OEX, and its log will be taken before any predictor functions are computed.

The purification model will be trained with a moving window of 300 cases. For

each case in that window, six different lookbacks, ranging from 11 through 260, will be used to compute the predictor functions.

The model will be given the four named predictor functions, each evaluated at each of the six lookbacks, making a total of 24 predictor candidates. The two best predictors will be found.

The TRAIN command causes the purification to be performed across the entire database at once, and the results will be written to a database file.

Here is the second part of this example, in which the database just created is read, and a trading model is walked forward:

```
READ MARKET LIST "OEX.TXT" ;
READ MARKET HISTORIES "E:\SP100\OEX.TXT" ;
READ VARIABLE LIST "VARS_PURIFY_DEMO2b.TXT" ;
APPEND DATABASE "T_PURIFY_DEMO2.DAT" ;

MODEL MOD_PURIF IS LINREG [
    INPUT = [ PURIF_LOG ]
    OUTPUT = RETURN
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    RESTRAIN PREDICTED
] ;

WALK FORWARD BY YEAR 10 2000 ;
```

This time, the market list contains only one market, OEX. This is because we want to trade only OEX in this example. If VIX, as it exists in the market history file, were tradeable and the file contained the actual trading prices, we might (or might not) be interested in also including it in the market list. If this were done, all performance results would include trades in VIX as well as OEX.

The new variable list VARS\_PURIFY\_DEMO2b.TXT contains just one line:

```
RETURN: NEXT DAY ATR RETURN 252
```

This variable is needed because it is the target for training and testing the model. It would have been legal to place this line in the variable definition list for the first half of this example, in which the purification was done. If so, the RETURN variable would have been computed there, written to the database, and read in this example, saving the trouble of having this separate variable list. However, the RETURN variable has a lookback of 252 days. If it had been computed in the purification script, its entire lookback of 252 days would have been unavailable to the purification transform, a significant loss.

After this one-line variable list is read, we append the database that was computed in the first half of this example. This contains the purified VIX for both the OEX and VIX markets. However, since the existing database contains only OEX, all VIX records will be eliminated during the APPEND operation, leaving the model to be trained and tested on only OEX.

The model is trained and tested using purified VIX as the sole indicator, and RETURN as the target. The CRITERION is irrelevant because it is used for stepwise selection only, and this is a single-indicator model. Nonetheless, it must be included in any model definition.

Finally, the model is walked forward from 2000 using 10-year training sets.

# Complex Prediction Systems

Prior chapters have discussed Transforms, Models, Committees, and Oracles, and simple tutorial examples of each have been presented. Those examples are probably close to the sorts of prediction systems that most users would employ for a practical trading system. However, much more exotic systems are possible in *TSSB*. This chapter will demonstrate how to construct an extraordinarily complex prediction system using the basic building blocks available in *TSSB*.

First, we should briefly review some of the elementary principals involved. Modeling methodologies in *TSSB* are divided into two broad families: *Models* and *Committees*. (For the purposes of the discussion in this section, *Oracles* are considered to be *Committees*. Also, we will use the lower-case *model* and *committee* to refer to them in a generic sense only, and the upper-case *Model* and *Committee* to refer to them in the specific *TSSB* sense as well as the generic sense.)

Models and Committees are very similar, often identical in their usage in *TSSB*. Nonetheless, there are some important distinctions which will now be discussed. Their most basic similarity is that they both take one or more inputs and map values of the inputs to an output. In general, the inputs (often called *indicators* for Models) will be backward-looking in time (they depend only on past and current values of market history), while the output will be a prediction of a forward-looking variable often called the *target*. The overall goal of a model-based trading system is to map *indicators* to a *target* and base trading decisions on predicted values of the target. Models and Committees play a crucial role in this process.

*TSSB* executes Transforms, Models, and Committees in the order in which they appear in the script file. The user can feed the output of one of these items into the input of another item. For example, Model outputs can serve as Transform inputs. Committee outputs can serve as the inputs of other Committees. Many other sequences of data flow are possible. We will present an example of this in the next section. But first, we will explore the similarities and differences between Models and Committees.

On a philosophical level, the fundamental difference between a Model and a Committee is that a Model (usually) takes indicators as inputs, while a Committee (always) takes predictions made by Models or other Committees as inputs. As will be seen, *TSSB* allows Models to be used as committees, so this rule is not inviolate, but it is a good basis of understanding and motivation.

Here are the specific issues involving Models and Committees in *TSSB*:

- Many of the most common modeling methodologies are available for both Models and Committees. These are LINREG, GRNN, MLFN, TREE, BOOSTED TREE, and FOREST. However, some methodologies are rarely appropriate for committees and hence are available only for Models. These are QUADRATIC, OPSTRING, and SPLIT LINEAR. Similarly, some methodologies are inherently committees, not generally suitable for use as models. These are AVERAGE and CONSTRAINED.
- A Model can take anything as an input, including indicators, predictions made by Committees, and predictions made by other Models. Thus, a Model can always be used as a committee if the user wishes. However, a Committee can take as input only predictions made by Models or other Committees. *Indicators cannot be used as inputs to a Committee*. This is in keeping with the underlying philosophy that the purpose of a committee is to combine multiple predictions into a single pooled prediction.
- The most important distinction between a Committee and a Model that is being used as a committee arises if the inputs to the ‘committee’ are from Models in which the FRACTILE THRESHOLD ([here](#)) option is used. In this case, if a Model is being used to combine those FRACTILE THRESHOLD predictions, the inputs to this Model are the fractiles, the same quantities that are preserved in the database as the component Models’ predictions. But if a Committee is being used to combine the FRACTILE THRESHOLD predictions, then the inputs to the Committee are the raw outputs of the component Models, the outputs *before* they are converted to fractiles. There is some modest theoretical justification to preferring this latter approach, although the reasoning is not compelling. Using the fractiles as committee inputs, which happens when a Model is used as a committee, is certainly a viable option.

## Stacking Models and Committees

We now present an extreme example of the sort of stacking that is available in TSSB. It must be pointed out that this example is far more complex than any that most users would devise. However, it does demonstrate how multiple levels of Models and Committees can be stacked for sophisticated operation.

The first section of this script file is shown below. Model MOD\_LOOK\_5 examines four indicators (linear trend, RSI, Stochastic, and Reactivity) that have a short lookback, just five bars. Model MOD\_LOOK\_20 is similar, but its indicators look back 20 bars. It makes sense to combine these two models with an Oracle which is gated by a measure of volatility. The idea is that one volatility regime may favor a short lookback distance, while another volatility regime may favor a longer lookback. But it is not obvious how far the volatility indicator should look back. One volatility lookback may be more effective than another, or they may actually complement each other. So we employ two Oracles. One uses PVR\_5, a short lookback measure of volatility, as its gate. The other uses the same volatility family but with a longer lookback. Finally we combine the predictions of these two Oracles using a Committee.

```
MODEL MOD_LOOK_5 IS LINREG [
    INPUT = [ LIN_5 RSI_5 STO_5 REACT_5 ]
    OUTPUT = HIT_MISS_1
    MAX STEPWISE = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD_LOOK_20 IS LINREG [
    INPUT = [ LIN_20 RSI_20 STO_20 REACT_20 ]
    OUTPUT = HIT_MISS_1
    MAX STEPWISE = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

ORACLE ORAC_NO_PS_5 [
    INPUT = [ MOD_LOOK_5 MOD_LOOK_20 ]
    GATE = [ PVR_5 ]
    OUTPUT = HIT_MISS_1
    MIN CRITERION FRACTION = 0.1
] ;
```

```

ORACLE ORAC_NO_PS_20 [
  INPUT = [ MOD_LOOK_5 MOD_LOOK_20 ]
  GATE = [ PVR_20 ]
  OUTPUT = HIT_MISS_1
  MIN CRITERION FRACTION = 0.1
] ;

COMMITTEE COMM_NO_PS IS CONSTRAINED [
  INPUT = [ ORAC_NO_PS_5 ORAC_NO_PS_20 ]
  OUTPUT = HIT_MISS_1
  MAX STEPWISE = 0
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

```

Now we work toward a similar goal, but using an entirely different approach, somewhat opposite the approach just taken. In the example shown above, the user explicitly specified lookbacks, and then volatility-gated Oracles combined the predictions that came from the different lookbacks. The approach shown below reverses the technique. It uses PRESCREEN model options to create specialist models based on volatility, and provides a diverse set of indicators from which the model may choose. An Oracle with the HONOR PRESCREEN option combines the models. Because we don't know which lookback (5 or 20) will be best, we try both and then combine them with a Committee, much as we did above. Here are these commands:

```

MODEL MOD_PS_POS_5 IS LINREG [
  INPUT = [ CMMA_5 CMMA_20 LIN_5 RSI_5 STO_5 REACT_5
            LIN_20 RSI_20 STO_20 REACT_20 ]
  OUTPUT = HIT_MISS_1
  PRESCREEN PVR_5 > 0.0
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD_PS_NEG_5 IS LINREG [
  INPUT = [ CMMA_5 CMMA_20 LIN_5 RSI_5 STO_5 REACT_5
            LIN_20 RSI_20 STO_20 REACT_20 ]
  OUTPUT = HIT_MISS_1
  PRESCREEN PVR_5 <= 0.0
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

```

```

ORACLE ORAC_PS_5 [
  INPUT = [ MOD_PS_POS_5 MOD_PS_NEG_5 ]
  GATE = [ PVR_5 ]
  HONOR PRESCREEN
  OUTPUT = HIT_MISS_1
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD_PS_POS_20 IS LINREG [
  INPUT = [ CMMA_5 CMMA_20 LIN_5 RSI_5 STO_5 REACT_5
            LIN_20 RSI_20 STO_20 REACT_20 ]
  OUTPUT = HIT_MISS_1
  PRESCREEN PVR_20 > 0.0
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD_PS_NEG_20 IS LINREG [
  INPUT = [ CMMA_5 CMMA_20 LIN_5 RSI_5 STO_5 REACT_5
            LIN_20 RSI_20 STO_20 REACT_20 ]
  OUTPUT = HIT_MISS_1
  PRESCREEN PVR_20 <= 0.0
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

ORACLE ORAC_PS_20 [
  INPUT = [ MOD_PS_POS_20 MOD_PS_NEG_20 ]
  GATE = [ PVR_20 ]
  HONOR PRESCREEN
  OUTPUT = HIT_MISS_1
  MIN CRITERION FRACTION = 0.1
] ;

COMMITTEE COMM_PS IS CONSTRAINED [
  INPUT = [ ORAC_PS_5 ORAC_PS_20 ]
  OUTPUT = HIT_MISS_1
  MAX STEPWISE = 0
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

```

Now we take still a third approach, completely different from those just shown. We train two linear models using the EXCLUSION GROUP option, and then combine them using an MLFN Model. This nonlinear Model lets us pick up any nonlinearities in the relationships. Here are these commands:

```

MODEL MOD_EX1 IS LINREG [
    INPUT = [ CMMA_5 CMMA_20 LIN_5 RSI_5 STO_5 REACT_5
              LIN_20 RSI_20 STO_20 REACT_20 ]
    OUTPUT = HIT_MISS_1
    MAX STEPWISE = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    EXCLUSION GROUP = 1
] ;

MODEL MOD_EX2 IS LINREG [
    INPUT = [ CMMA_5 CMMA_20 LIN_5 RSI_5 STO_5 REACT_5
              LIN_20 RSI_20 STO_20 REACT_20 ]
    OUTPUT = HIT_MISS_1
    MAX STEPWISE = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    EXCLUSION GROUP = 1
] ;

MODEL MOD_EX IS MLFN [
    INPUT = [ MOD_EX1 MOD_EX2 ]
    OUTPUT = HIT_MISS_1
    OUTPUT LINEAR
    DOMAIN REAL
    FIRST HIDDEN = 2
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

```

At this point we have three predictions available:

**COMM\_NO\_PS** is based on the first example, in which the user explicitly specified lookbacks for the model inputs, and used volatility-gated Oracles to combine the models.

**COMM\_PS** is based on the second example, in which all indicators were supplied to the component models, which were then trained to be volatility-based specialists using the PRESCREEN command.

**MOD\_EX** is based on the third example, in which a Model is used as a committee to combine the predictions of EXCLUSION GROUP models.

We can wrap this all up into a single prediction by combining these three predictions with a final Committee:

```
COMMITTEE COMM_FINAL IS CONSTRAINED [
    INPUT = [ COMM_NO_PS COMM_PS MOD_EX ]
    OUTPUT = HIT_MISS_1
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;
```

Once again it must be stated that few users would want to create a trading system as complex as the multi-layered behemoth just shown. The complete script file is over 150 lines long! Still, this example illustrates the nearly unlimited ability of *TSSB* to stack prediction upon prediction using assorted Models, Committees, and Oracles.

# Graphics

TSSB contains a variety of methods for plotting variables. These variables can be indicators and targets in the database, whether computed internally or read in from an external file. They can also be predicted values from walkforward or cross validation runs, as long as the predicted values were saved via the PRESERVE PREDICTIONS command ([here](#)).

Plots are displayed on the screen, but they can also be printed via the *File/Print* menu selection. In order to print a plot, the user must click on the plot so that Windows highlights it. If no plot is highlighted, the *File/Print* option will be grayed out so that it cannot be selected.

The plotting options available through the menu system under the *Plot* selection include the following:

- ***Series*** - The variable is plotted as a time series, with dates labeled at the bottom of the plot.
- ***Series + Market*** - This is identical to a *Series* plot except that the log of the market close is overlaid with the variable plot. Also, the user can plot a horizontal threshold line.
- ***Histogram*** - A histogram of a variable is plotted. In a multiple-market environment, the histogram can be based on a single market or all markets pooled.
- ***Thresholded Histogram*** - This is similar to the *Histogram* plot except that subsets of the complete dataset (all markets) based on indicators or predicted values can be plotted.
- ***Density Map*** - Various density estimators are used to produce a sophisticated version of a scatterplot for showing the relationship between two variables.
- ***Bivariate Plot*** - Shows the relationship between a target and two indicators.
- ***Trivariate Plot*** - Shows the relationship between a target and two indicators, conditional on a third indicator.
- ***Prediction Map*** - For a model having two inputs, this displays a two-dimensional map showing how values of the indicators affect the predicted target, thus revealing the inner workings of the model.

- ***Indicator-Target relationship*** - Shows a historical time series plot of the relationship between an indicator and a target.

***NOTE... Many of the plots shown in this chapter are black-and-white renderings of color images, and hence they do not reproduce well. The authors will at some point include full color versions on the TSSB website. For now the reader must use his or her imagination.***

## Series Plot

The variable is plotted as a time series, with dates labeled along the bottom of the plot. The user specifies the following items:

**Variable** - The variable to be plotted. They are listed in alphabetical order.

**Market** - All markets are listed, even if there is only one. The user selects one.

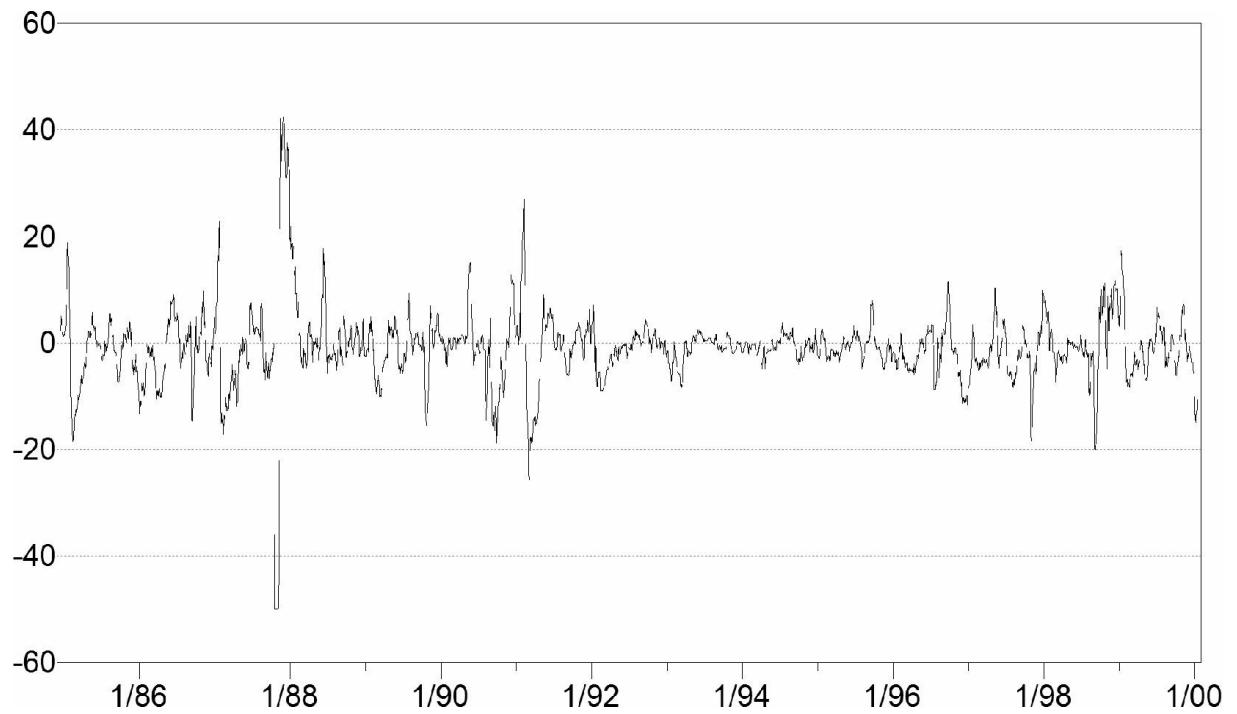
**Connect** - If this box is checked, the values will be connected with straight lines. If it is not checked, each value will be isolated and represented by a vertical line.

After the plot is displayed, an individual section can be magnified. Place the mouse cursor at the left edge of the section to be magnified. Press and hold the left mouse button while dragging the cursor to the right. The selected area will be highlighted. Release the mouse button when the desired right boundary is attained.

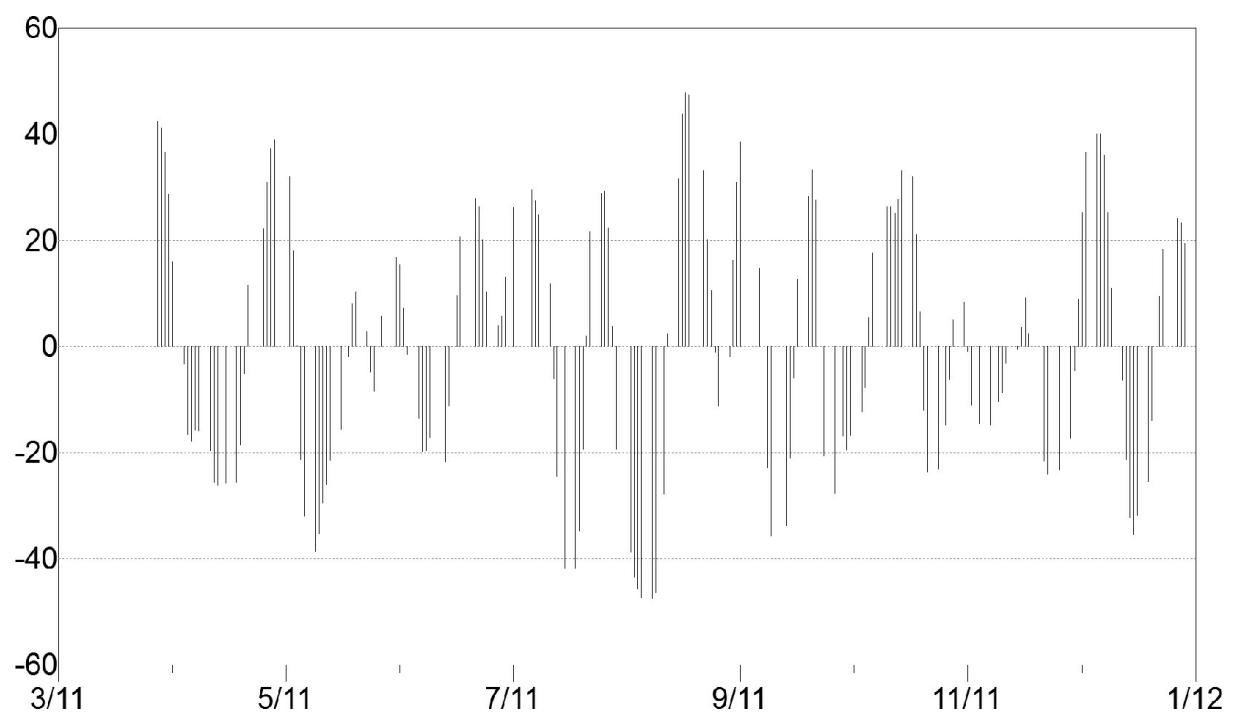
If a section of the plot has been magnified, a horizontal scroll bar will appear. The user can move the magnified section along the entire time-range of the series in three ways:

- 1) Drag the elevator button to move a large distance
- 2) Click anywhere inside the elevator bar on either side of the button to move in medium jumps.
- 3) Click on the small buttons on the extreme left or right of the bar to move in small jumps.

The following page shows series plots with and without the *connect* option.



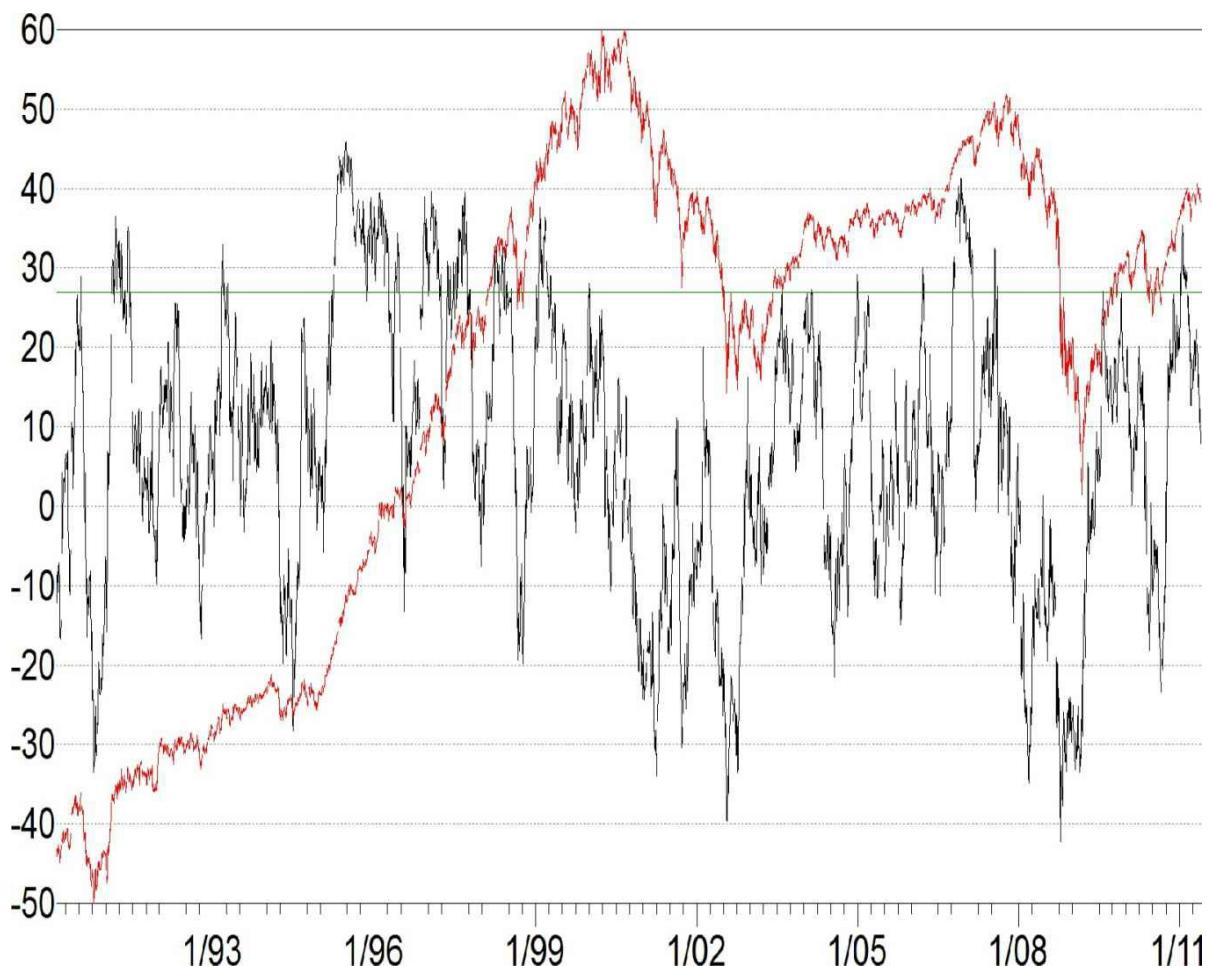
**Figure 28:** Series plot with *connect* option



**Figure 29:** Series plot without *connect* option

## Series + Market

This is identical to the *Series* option above with two exceptions. First, it overlays the log of the market close on top of the variable. This overlay is colored red and it is automatically scaled to fill the entire plot, top to bottom. Second, the user has the option of specifying a threshold for display. This threshold, if specified, is plotted as a horizontal green line. An example of this plot is shown below:



**Figure 30:** Series + Market plot, also using *threshold* option

# Histogram

A histogram of a variable is plotted. The height of each bar represents the fraction of the total number of observations that fall into that bin. The user specifies the following items:

**Number of Bins** - This many bars will be displayed, each corresponding to a bin that counts how many cases fall into a range. Making this an odd number causes the graph to be centered at zero if the upper and lower limits are equal (of opposite sign), which is visually appealing.

**Histogram Variable** - The variable to be plotted. The candidates are listed alphabetically.

**Primary Market** - This lists all markets, even if there is only one. In addition, *Pooled* appears as the option at the top of the list. The user may select an individual market, in which case only data from that market will be used to compute the histogram. If instead the user selects *Pooled*, data from all markets will be pooled to compute the histogram.

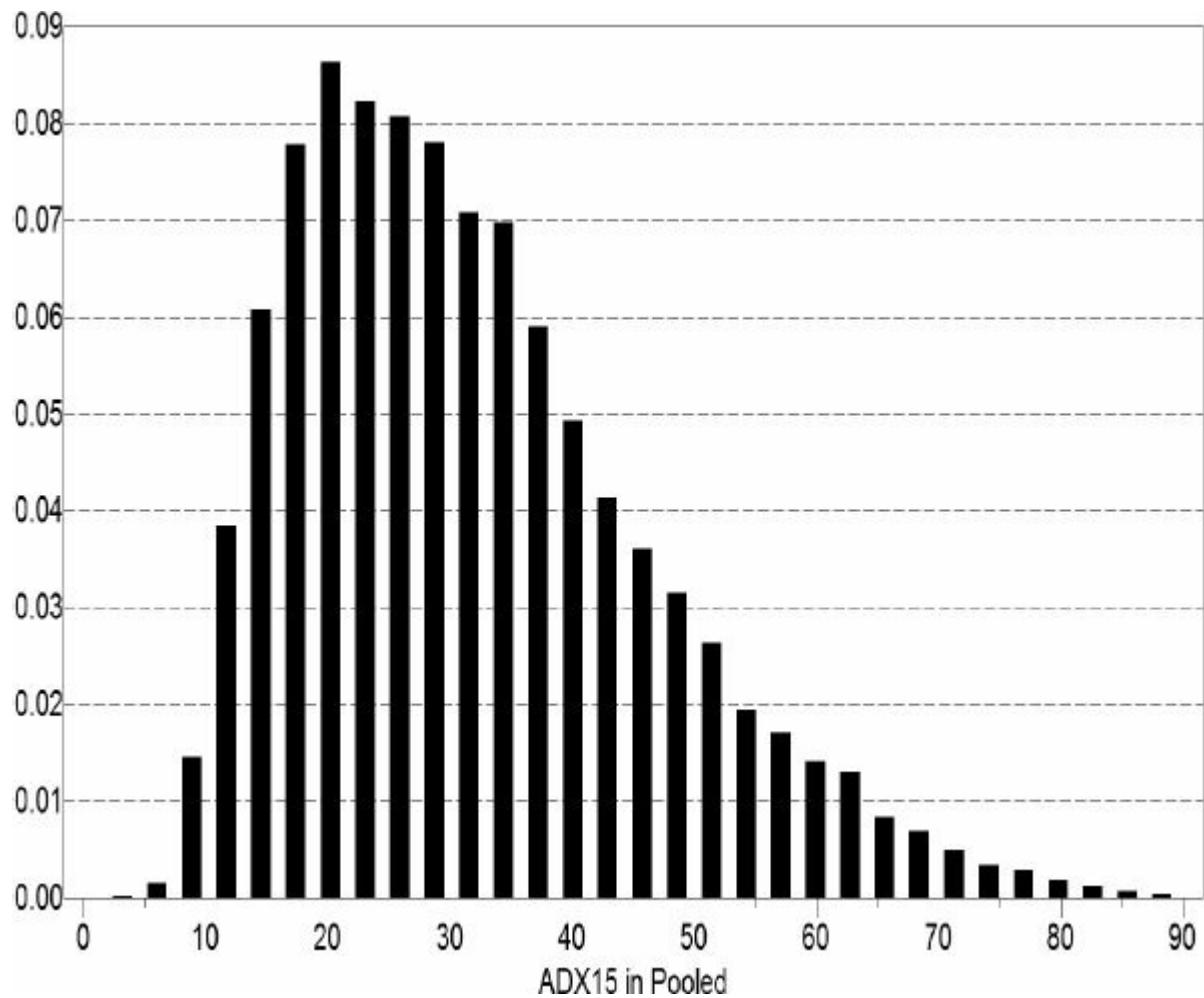
**Use Second Market** - If this box is checked, the histogram will show the variable's distribution in two markets, using bars with different appearances. The primary market is displayed as solid black bars, and the secondary market is displayed as cross-hatched bars. This enables the user to easily compare the distributions in different markets, or in a single market versus pooled data.

**Secondary Market** - If the *Use Second Market* box is checked, this list of markets will appear so that the user can select a second market or pooled data for display.

**Lower Limit** - This comprises a check box and a field for entering a numeric value. If the box is checked, the user must enter a number in the field. This will be the lower (left) limit for the plot. All values less than or equal to this number will be cumulated in the leftmost histogram bin. This is handy for variables that have one or a few extreme values that cause the graph to become unnaturally compressed.

**Upper Limit** - This is similar to the *Lower Limit* except that it specifies the upper (right) limit for the plot.

**Normalize Areas** - This is relevant only if the user checks the *Use Second Market* box to display overlaid histograms. If the *Normalize Areas* box is not checked, the height of each bar will represent the fraction of cases in each bin relative to the *total number of cases* and the vertical axis will be labeled as this fraction. If this box is checked, the height of each bar will represent the fraction of cases in each bin relative to the *number of cases in that market* and the vertical axis will be labeled zero to one. This latter option is usually preferable to the former.



**Figure 31:** A typical histogram

## Thresholded Histogram

This is similar to the ordinary histogram described on the prior page, but with a twist. What makes the thresholded histogram interesting is that the user also selects a *Thresholded Variable*. In an ordinary histogram, all cases are plotted. But in a thresholded histogram, only those cases which lie in the lower and/or upper specified percent of the thresholded variable are plotted. In other words, the plot is based on only a subset of the cases, and the particular subset is determined by the value of the thresholded variable for each case.

The user specifies the following items:

**Number of Bins** - This many bars will be displayed, each corresponding to a bin that counts how many cases fall into a range. Making this an odd number causes the graph to be centered at zero if the upper and lower limits are equal, which is visually appealing.

**Plotted Variable** - The variable whose histogram is to be plotted. The candidates are listed in alphabetical order.

**Thresholded Variable** - The variable whose thresholded values determine which cases are plotted. The candidates are listed in alphabetical order.

**Lower Limit** - This comprises a check box and a field for entering a numeric value. If the box is checked, the user must enter a number in the field. This will be the lower (left) limit for the variable being plotted. All values less than or equal to this number will be cumulated in the leftmost histogram bin. This is handy for variables that have one or a few extreme values that cause the graph to become unnaturally compressed.

**Upper Limit** - This is similar to the *Lower Limit* except that it specifies the upper (right) limit for the plotted variable.

**Lower Percent** - This comprises a check box and a field for entering a numeric value that refers to the thresholded variable. If the box is checked, the user must enter in the field a number greater than zero and less than 100. The specified percent of cases having the smallest values of the thresholded variable are included in the plot and represented as solid bars, black by default and red if *overlaid colors* is selected..

**Upper Percent** - This is similar to the *Lower Percent* except that it specifies a

high cutoff for the thresholded variable. The specified percent of cases having the largest values of the thresholded variable are included in the plot and are represented as hatched (or solid blue if *overlaid colors* is selected) bars if the user also specified a lower percent.

**Normalize Areas** - This is relevant only if the user specifies both a lower percent and an upper percent for the thresholded variable. In this case, the lower percent will be represented by solid bars and the upper percent by hatched bars. If the *Normalize Areas* box is not checked, the height of each bar will represent the fraction of cases in each bin relative to the number of cases that meet *either* threshold criterion and the vertical axis will be labeled as this fraction. If this box is checked, the height of each bar will represent the fraction of cases in each bin relative to the number of cases that meet the *individual* (lower or upper) threshold criterion and the vertical axis will be labeled zero to one.

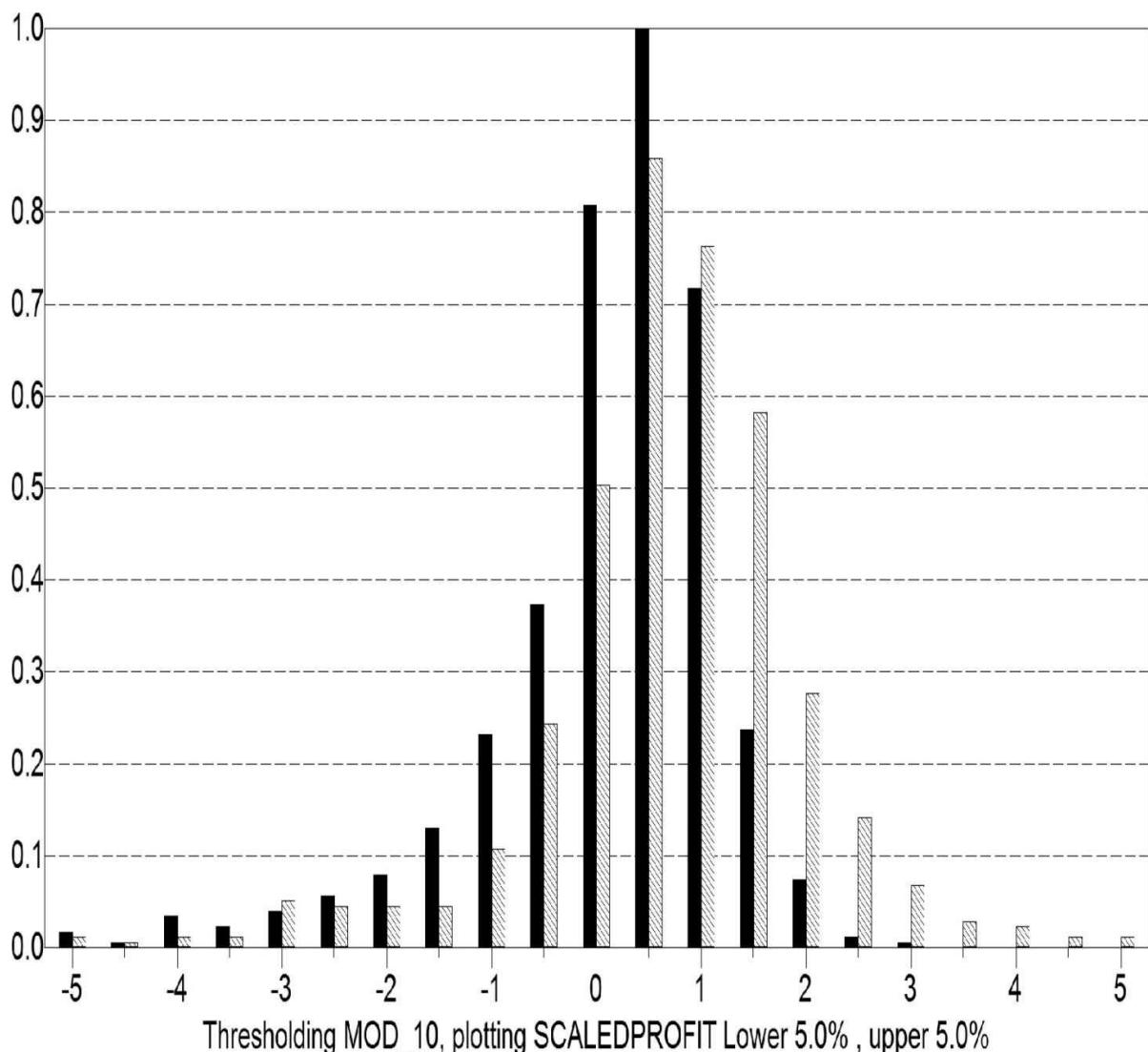
**Overlaid colors** - This is useful only if both *Lower Percent* and *Upper Percent* are employed. If this box is not checked, the lower percent bars are solid black and the upper are hatched black. If this box is checked, the lower bars are red and the upper blue.

This axis labeling when the *Normalize Areas* box is checked means that the numeric value attached to bar heights has a very different meaning from that when the areas are not normalized. As stated above, when the histogram is normalized, each bar's height represents the concentration of cases in that bar relative to the total number of cases that meet the associated threshold criterion. *The numeric value associated with that bar height is its concentration relative to the bar having the highest concentration.* So for example, suppose a normalized bar has a height of 0.25. This means that the concentration of cases in this bar is one-quarter of the concentration of the most concentrated (i.e., tallest) bar.

One particularly useful application for the thresholded histogram is when we are filtering trades with a single variable such as an indicator or the predicted profit from a model. For example, suppose that large values of the filter (thresholded) variable correspond to large expected profits, and suppose we want to keep just the largest ten percent cases. We would enter '10' in the *Upper Percent* box. This will cause the cases kept by the filter to be displayed as hatched bars. We have two reasonable choices for the *Lower Percent*. If we enter 100, the solid bars will display a complete histogram of the target variable. If we enter 90, the solid bars will show the cases that were rejected

by the filter. Of course, the utility of this display is not limited to filtering systems. The histogram can display any target.

Another approach is to display only the extremes of the thresholded variable, for example the highest and lowest five percent. The histogram shown on the [here](#) uses the output of a model (MOD\_10) as the thresholded variable to plot values of the target, *ScaledProfit*. The lower (solid bars) and upper (hatched bars) five percent of predictions are used. Observe that when the model has a very large prediction, the target also tends to be large. The converse is also true. This is good!



**Figure 32:** A thresholded histogram

# Density Map

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

The problem with a scatterplot is that if there are numerous cases, so many that limited display or visual resolution results in multiple cases being overlaid on the same spot, it is impossible to see the quantity of cases at that spot. Whether one case lies there, or a thousand, it's still a single dot.

A density plot overcomes this problem by using the gray level or color of the display to portray the *density* (or, optionally, measures related to the density) of the pair of variables at every point on the display. It does this by fitting a statistical smoothing window to the data in order to produce an estimated bivariate density. (Users interested in more details on this topic should search the internet for density estimation using Parzen windows.) Examples will appear later.

The following parameters may be set by the user:

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

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

**Lower Limit** - This comprises a check box and a field for entering a numeric value. If the box is checked, the user must enter a number in the field. This will be the lower (left or bottom) limit for the plot. Cases outside this range still take part in computations, but the plot does not extend to include them. Limits can be set separately for the horizontal and vertical variables.

**Upper Limit** - This is similar to the *Lower Limit* except that it specifies the upper (right or top) limit for the plot.

**Resolution** - This is the number of grid locations along both axes that are used for computation. The plot interpolates between the grid points.

Larger values result in more accuracy but longer run time.

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

**Tone shift** - This has no effect on computation, but it controls how the density estimates translate to grey levels or color on the display. Positive values will push the display toward yellow (or black for B&W display), and negative values toward blue (or white). Use this option to highlight desired features, experimenting as needed.

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

**Plot in color** - By default the display is black and white. Checking this box causes the display to be shades of yellow and blue.

**Histogram equalization** - If this box is checked, the assignment of tone/color to each displayed value is done such that every possible shade is used in equal amounts. This makes details more visible. However, in many cases these details are mostly noise. Use of this option has the potential to seriously clutter the display with random noise.

**Sharpen** - Checking this box causes the display to highlight detail near extremes at the expense of detail in intermediate values.

**Actual density** - This, the default, plots the smoothed bivariate density of the two specified variables. A density plot is analogous to a scatterplot. The display defines a grid, with the value of one variable defining a position along the horizontal axis, and the value of the other variable defining a position along the vertical axis. Any pair of values for these two variables defines a point in the display, and the color or tone at that point depicts the concentration of data near that

point. If the *Plot in color* box is checked, areas of high density, which are the result of numerous cases clustering in that region, are colored yellow. Areas of low density (sparsely populated with cases) are colored blue. If the *Plot in color* box is not checked, low density is displayed as white and high density as black. The *Actual density* option is the easiest to understand and is recommended for most users. Note that financial data typically contains so much noise compared to predictive information that predictive relationships are difficult to see. This display is most useful for examining relationships between indicators or visualizing indicator/target relationships on a macro level.

**Marginal density** - This option is not generally useful, but it can be an interesting adjunct to the *Actual density* when they are compared. The display is identical that of the *Actual density* option, except that the densities plotted are not the actual bivariate densities. Rather, the tone/color at each point is determined by the product of the two marginal densities. (The marginal density of a variable is its probability distribution, considered alone, ignoring other variables.) Therefore, this plot shows what the actual density would look like if the horizontal and vertical variables had their observed univariate distributions, but there was no relationship between them. By comparing the *Actual density* to the *Marginal density* one can visually assess the degree of relationship between the horizontal and vertical variables. If the two plots are identical, chances are the two variables have little relationship. But if the *Actual density* shows a pattern not observed in the *Marginal density*, there probably is a significant relationship between the horizontal and vertical variables.

**Inconsistency** - A potential problem with scatterplots and their analogous *Actual density* plots is that these plots unavoidably include marginal densities in the display. For example, if one of the variables clusters at one extreme of its range, most of the cases will be plotted in a band there. Even if the two variables are unrelated, the map will show a possibly confusing gradient that may appear to show a relationship when none is actually present. What many people are most interested in is inconsistencies in the actual bivariate density *after taking marginal distributions into account*. We want to see where the actual density differs from the product of the marginals (the density that would be expected if the variables were not related). Visually comparing an *Actual density* plot to a *Marginal density* plot, as suggested earlier, reveals only large discrepancies.

But by numerically comparing them, we can plot their inconsistency with great sensitivity. In this plot, areas which have a large value of the bivariate density compared to the product of the marginal densities are shown as black (for black-and-white plots) or yellow (for color plots). Areas containing fewer cases than the product of the marginal distributions would predict are depicted as white (for B&W plots) or blue (for colored plots).

**Mutual information** - The mutual information between two variables is a sophisticated measure of how much their values are related. It is roughly analogous to a correlation coefficient, but it is totally nonlinear. It captures any sort of relationship, not just monotonic relationships. The mutual information between two variables is the sum of the mutual information in regions across their entire domain. The *Mutual information* plot shows by tone/color the contribution of each area to the total mutual information. Areas that contribute greatly to the mutual information are displayed as black (for B&W plots) or yellow (for color plots). Areas that contribute little mutual information are shown as white (for B&W plots) or blue (for color plots). This plot is useful only if there is significant mutual information between the variables. In a high noise environment, as is typically the case when one variable is an indicator and the other is a target, the mutual information is so small that the contributions of individual regions are mostly noise

In order to provide easily understood examples of these four types of density plot, a pair of variables having high negative correlation was used. These plots appear on the [here](#).

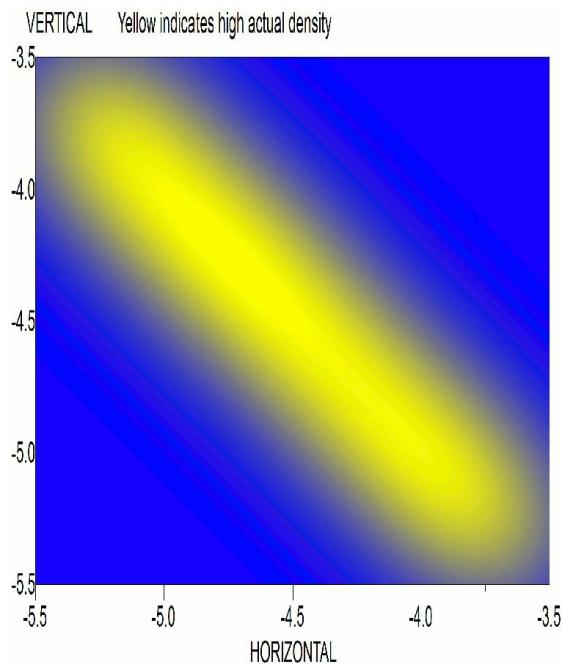
[Figure 33](#) is an *Actual density* plot of these two variables. The prominent yellow band shows that low values of the horizontal value are strongly associated with high values of the vertical variable, and vice versa.

[Figure 34](#) is a *Marginal density* plot of the same data. Notice that the relationship between the two variables is gone, although they individually cover the same territory. If one compares this figure to the *Actual density* plot, the relationship between the vertical and horizontal variables is obvious.

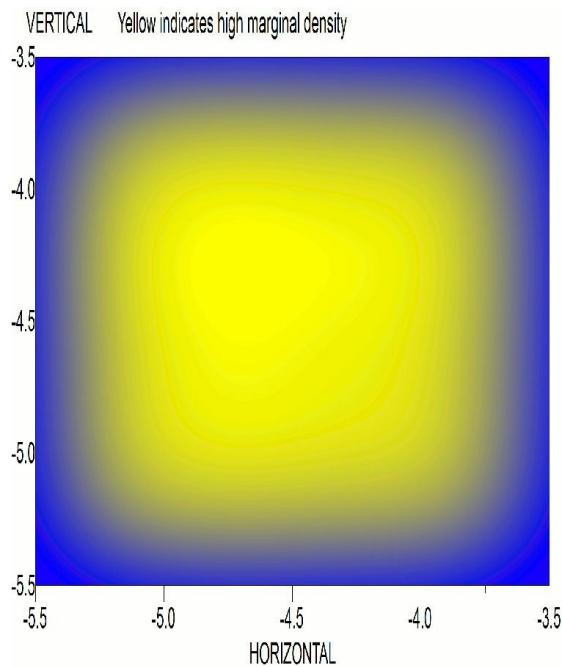
[Figure 35](#) is the corresponding *Inconsistency* plot. Naturally, the negative correlation is prominent. The brighter yellow at the corners indicates that the inconsistency is greater at the extremes than in the interior (mid-range), a common phenomenon.

[Figure 36](#) is the *Mutual information* plot. It shows that cases along the band of

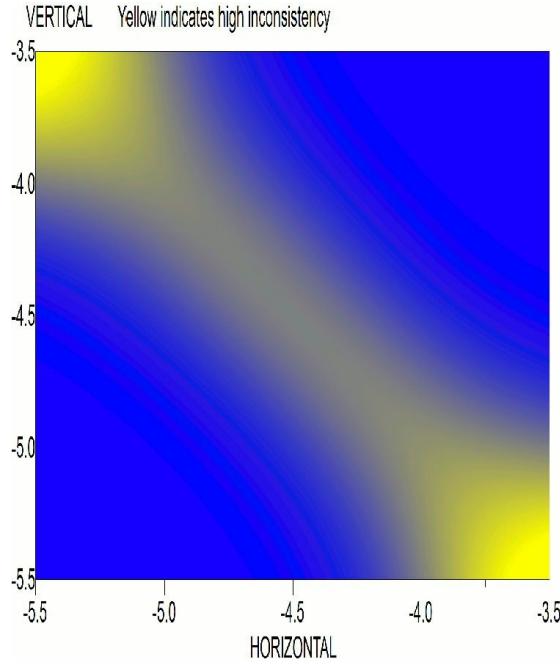
high density are the primary contributors to the total mutual information.



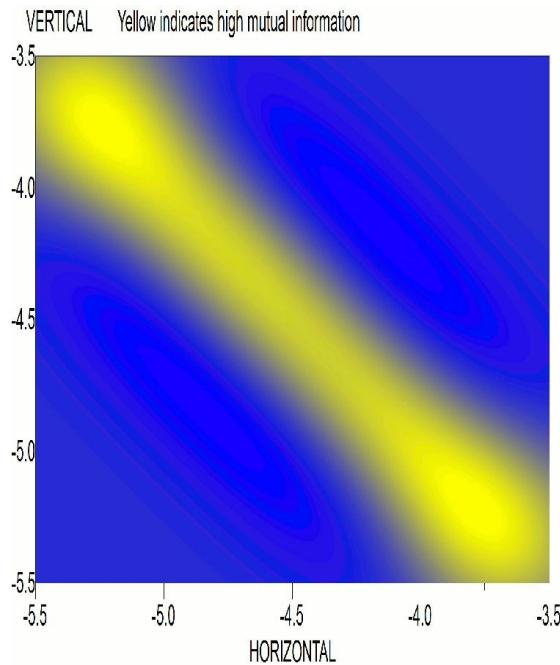
**Figure 33:** Actual density



**Figure 34:** Marginal density



**Figure 35:** Inconsistency

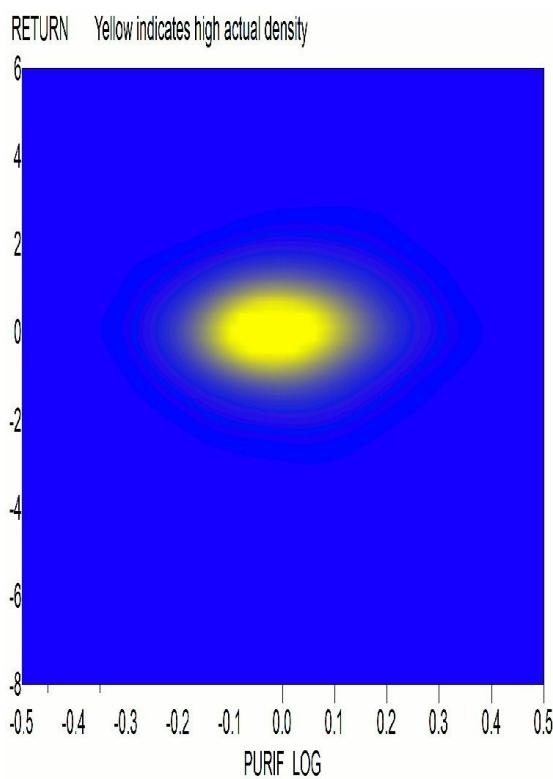


**Figure 36:** Mutual information

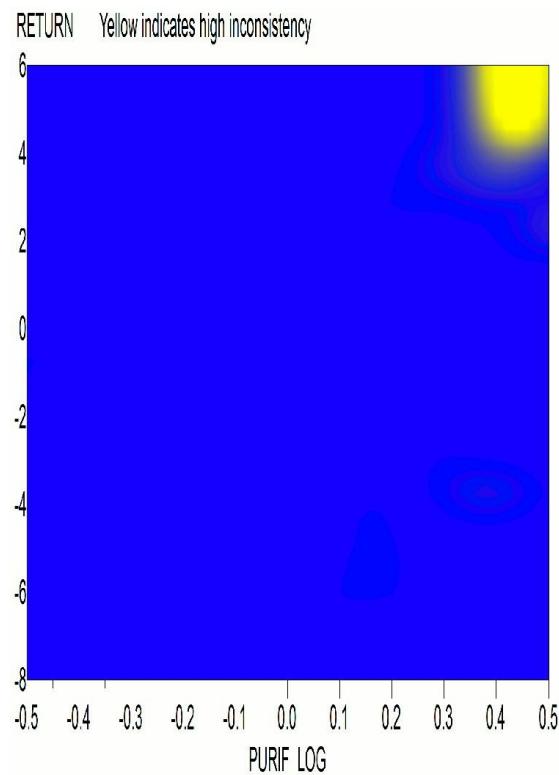
Figures 37 and 38 below employ actual data. Figure 37 is the *Actual density* of an indicator called PURIF\_LOG and a forward-looking variable called RETURN. It is a nearly flat ellipse, meaning that there is little relationship between the variables. If one looks closely, it is apparent that there is a slight upward tilt to it, so that larger values of PURIF\_LOG tend to be slightly associated with larger values of RETURN. However, the relationship is tiny. A *Marginal density* plot, not shown, looks similar, though without the slight tilt. A *Mutual Information* plot, also not shown, is totally worthless. This is because

the relationship between the two variables is so small that their mutual information across the display region is swamped out by random noise.

However, the *Inconsistency* plot, shown in [Figure 38](#) below, reveals a fascinating bit of useful information. The bright yellow area in the upper-right corner shows that there is an unexpectedly large concentration of cases that simultaneously have very large values of PURIF\_LOG and very large values of RETURN. These concentrations are all relative; this plot does not mean that such cases are common in an absolute sense, only that they are common *relative to what would be expected*. Such large values of PURIF\_LOG are quite rare, as are such large values of RETURN. Because these values are rare, their simultaneous occurrence should be even more rare if they were unrelated. So what this plot is showing is that their simultaneous occurrence is not nearly as rare as would be expected if the two quantities were unrelated. This sort of event, in which the appearance of combinations of variables is relatively common *compared to what one would expect based on their individual distributions*, cannot be seen on an ordinary *Actual density* plot or its conventional analog, a scatterplot.



**Figure 37:** Actual density



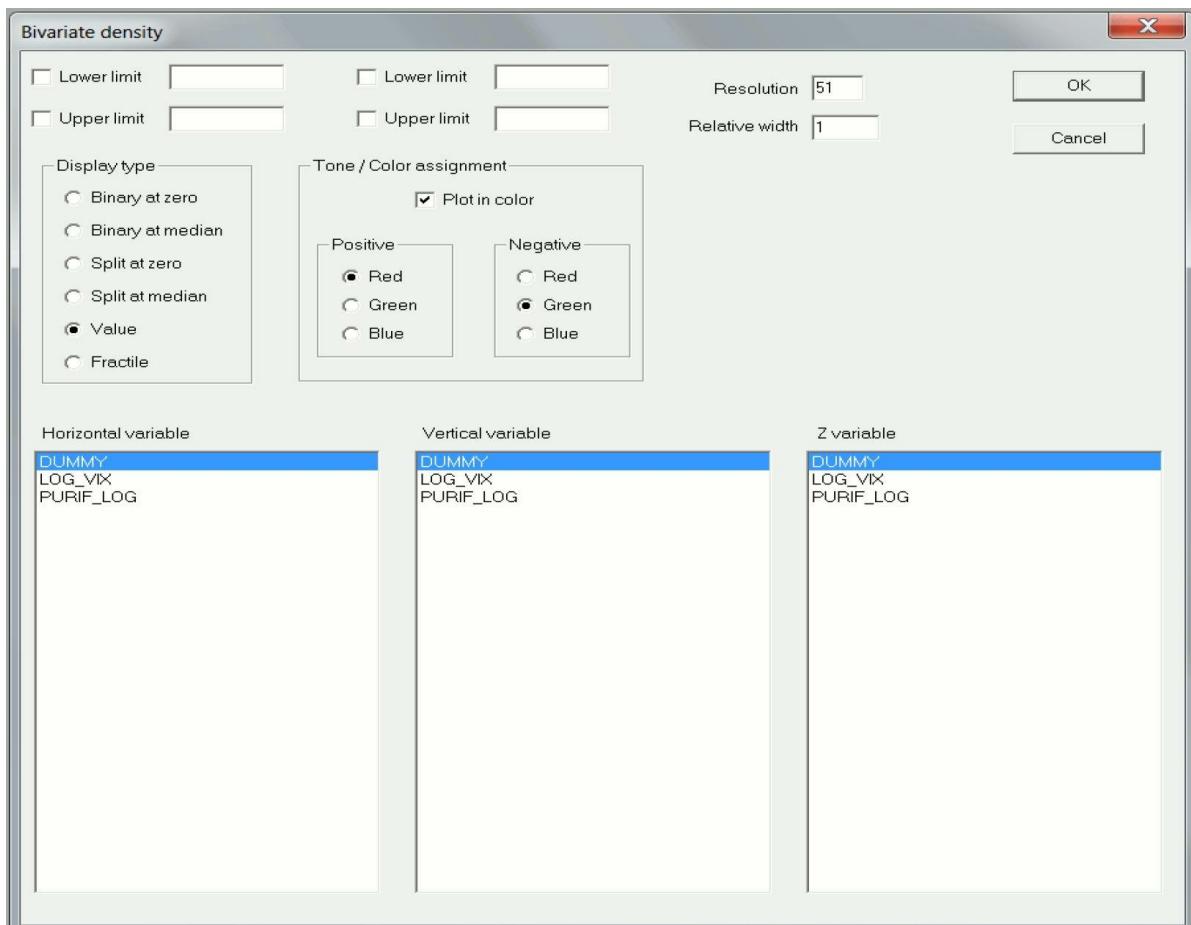
**Figure 38:** Inconsistency

## Bivariate and Trivariate Plots

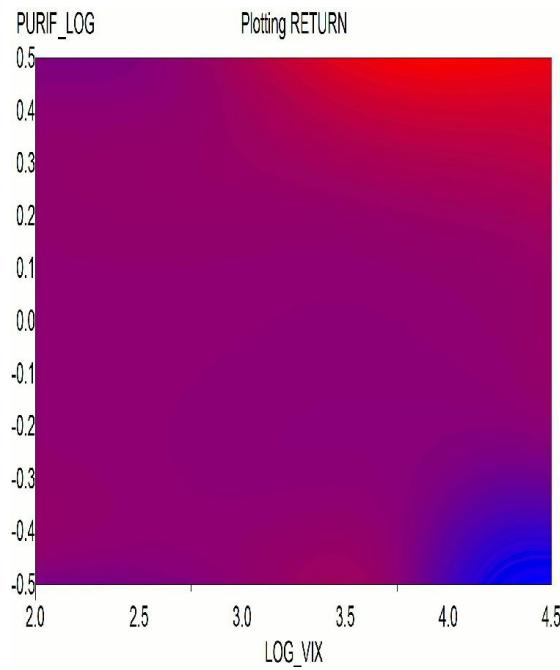
We saw in the prior section how the relationship between two variables (often an indicator and a target) could be displayed with a density map. It can also be useful to see how the value of one variable, usually a target or other measure of future market behavior, relates to *two* other variables, typically indicators. It may be that certain combinations of indicator values are associated with unusually large or small values of the target. Such patterns can be revealed with a bivariate plot.

Of course, paper and computer screens are two-dimensional, and with three variables we have a third dimension. One solution is to plot a 3-D projection (a visual representation) of the curved surface defined by the target as a function of the two indicators. But this can often obscure important details, depending on the viewpoint from which the smoothed function is observed. A more reliable method is to use black-and-white tone or color as the ‘third dimension’ of the plot.

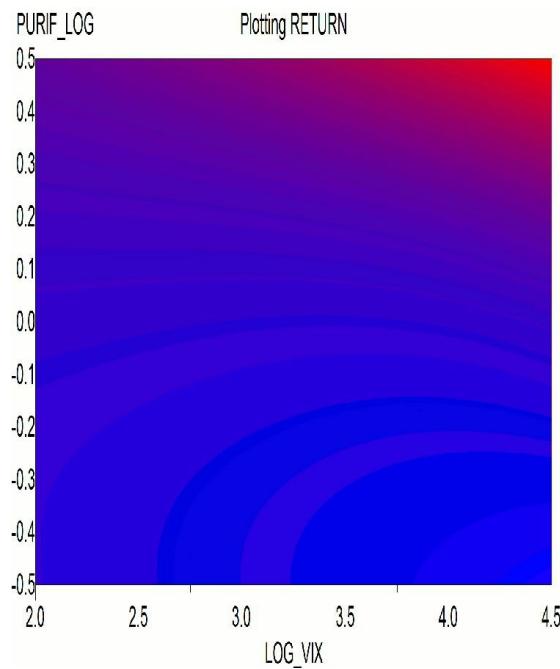
[Figure 39](#) below shows the dialog box by which the user enters the parameters for the plot. Some sample plots are shown on the [here](#), and a discussion follows.



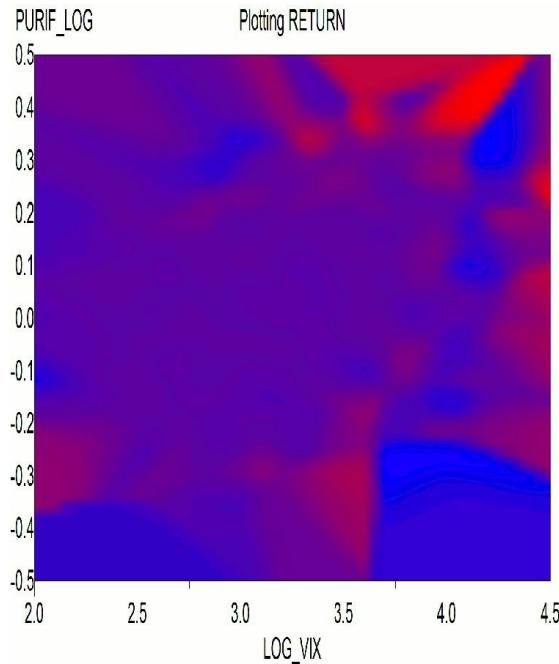
**Figure 39:** Entering parameters for a bivariate plot



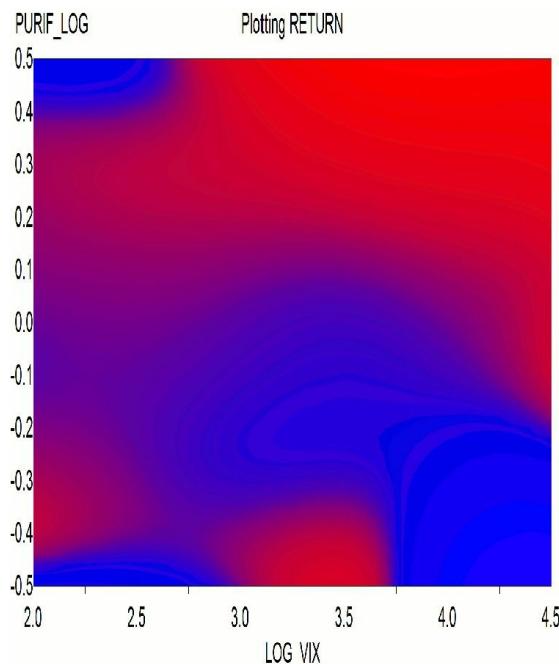
**Figure 40:** Value plot with width=1.0



**Figure 41:** Value plot with width=2.0



**Figure 42:** Value plot with width=0.2



**Figure 43:** Fractile plot with width=1.0

**Horizontal variable** is the variable (usually an indicator) whose value lies along the horizontal axis. **Vertical variable** is the variable (also usually an indicator) whose value lies along the vertical axis. **Z variable** is the variable (usually a target or other measure of future market move) whose value is displayed by gray level or color.

If the **Plot in color** box is not checked, the minimum value of the Z variable will be plotted as brightest white, and the maximum as darkest black. If this box is

checked, the minimum value of the Z variable will be the brightest selected color for ***negative*** and the maximum will be the brightest ***positive*** color, and intermediate values will be dimmer shades. Central values of the Z variable will be shades of gray, regardless of whether the color box is checked. In keeping with engineering conventions and to avoid confusion for color-blind readers, all plots shown in this section are colored with red as positive (hot, or electrical positive) and blue (cold) as negative. Users who are more comfortable with the common financial convention of red for negative and green for positive may do so; in fact, this is the default, so no change is required. However, the user should be aware that many people are color-blind to red-green, and so this color combination, though the default, should be used with caution.

***Resolution*** is the number of horizontal and vertical grid points at which the smoothed function is computed. Large values provide a better appearance but require more computer time.

Variations in the value of the Z variable are smoothed using a Gaussian window. (Users interested in more details on this topic should search the internet for *Parzen window*.) The width of this window relative to the standard deviation of the horizontal and vertical variables is specified with the ***Relative width*** parameter. The default value is 1.0, which sets the width equal to the standard deviation in each direction (indicator axis). Making the width a multiple of the standard deviation of each indicator prevents indicators with a smaller standard deviation from being overly smoothed. Larger values of the relative width result in more blurring (smoothing), which is good for diminishing the impact of random noise. Smaller values reveal more detail, which can be good in a low-noise situation but confusing if a lot of noise is present. Larger relative width values are would be analogous to a longer term moving average - more smoothing but greater loss of detail.

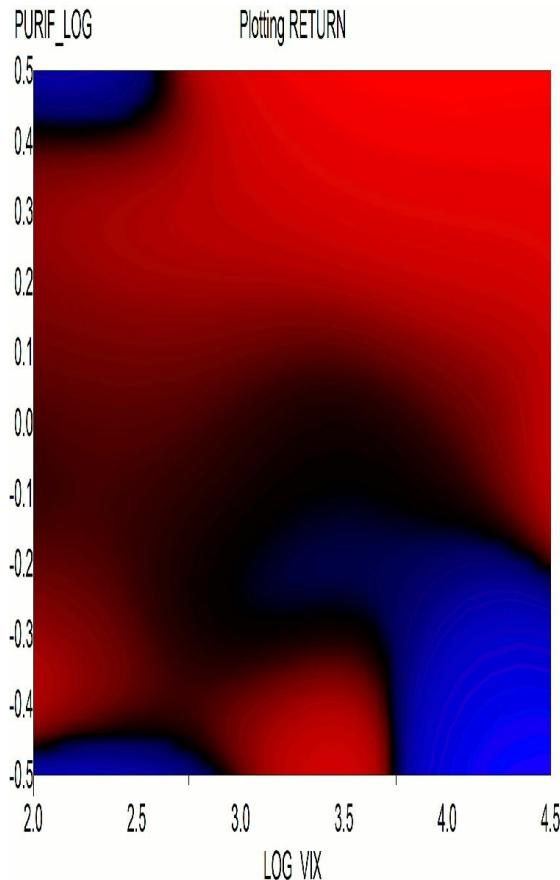
***Display Type*** controls the basic nature of the display. The following types are available:

- ***Value*** is the simplest and, in a sense, the most ‘honest’ display type. This is because the full range of the Z variable is mapped to the full range of the display tone/color, and this mapping is done in a linear fashion. Figures [here](#), [41](#), and [42](#) on the prior page demonstrate the *Value* option for three different widths. Notice that, as is commonly the case, the default relative width of 1 seems to be the best compromise between revealing detail while simultaneously blurring noise. Raising the width to 2.0 results in severe blurring, while lowering it to 0.2 fills the image with apparently random noise, a common situation in financial applications which have

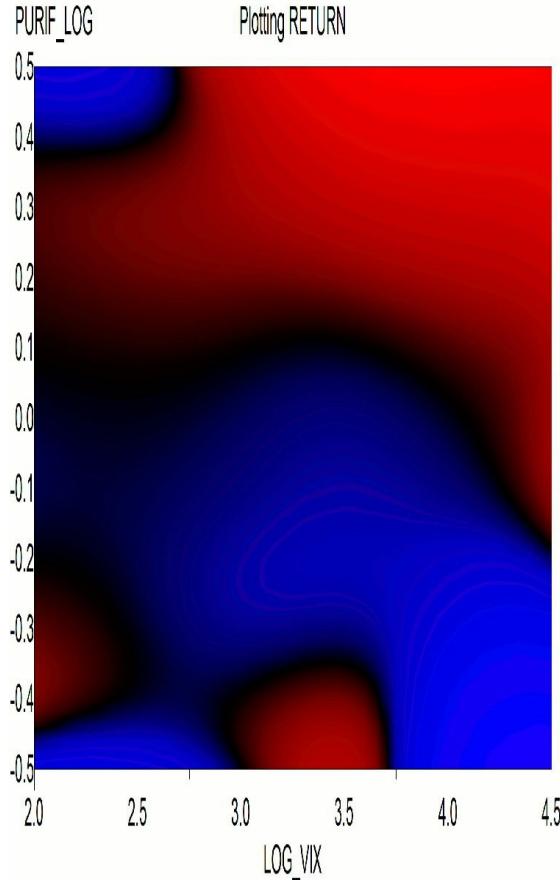
low predictivity relative to noise.

- **Fractile** is a variation on the *Value* display method. As with *Value*, the full range of the Z variable is mapped to the full range of the display tone/color. But the *Fractile* display function is nonlinear, computed in such a way as to reduce the impact of extreme values of the Z variable on the display. Readers familiar with image processing and photo-editing programs will recognize this as *histogram equalization*: every tone/color appears on the display in equal amounts. This makes full use of the range of display tones/colors, which usually reveals much more detail than the *Value* option. Suppose, for example, that most data cases have Z values that cluster around a small range, but one or a few values are unusually large. The *Value* option would display these outliers at one extreme (such as green, if green were the selected positive color), while clumping the mass of ‘usual’ values in a narrow, unrevealing range of tones/colors (poorly differentiated shades of red if red were the negative color). The *Fractile* option, which spreads out such clumps across a wide range for better display, is shown in [Figure 43](#) on the prior page. Note how this figure makes equal use of reds and blues, thus revealing more information, while the other figures on that page are dominated by one color.
- **Split at zero** is a variation on the *Fractile* option that uses the sign of the Z variable to choose the color. (This option is legal but pointless if the *Color* box is not checked.) The *Split at zero* option computes the fractiles of positive and negative values of the Z variable separately. The fractiles of negative Zs are displayed in shades of the chosen color, with the most negative values being depicted with brightest intensity. Similarly, positive values are displayed in the other selected color. For both signs, values of Z near zero are assigned such a dim tone of the appropriate color that they appear gray. In some cases it can be handy to know the sign of the Z variable in various regions of the indicator space. Neither the *Value* nor the *Fractile* display options allow this. An example of this display option is shown in [Figure 44](#) on the next page.
- **Split at median** is similar to the *Split at zero* display option, except that instead of the split point being zero, it is the median of the Z variable’s distribution. This is useful when one wants to split the fractile computation so that each color is processed separately and roughly equally, but the median of Z is not close to zero. An example of this display option is shown in [Figure 45](#) on the next page. For example suppose blue is specified for Z values less than the median and red for Z values greater than the median. Cases near the median will appear nearly black. The further from the median the brighter will be the red or blue color.

- **Binary at zero** and **Binary at median** are similar to the split versions above, except that no gradations of color are done. Values of Z below the median (or zero) are one selected color, and values above are the other, both at uniformly bright intensity. If the *Color* box is unchecked, values below are white and those above are black.



**Figure 44:** Split at 0



**Figure 45:** Split at median

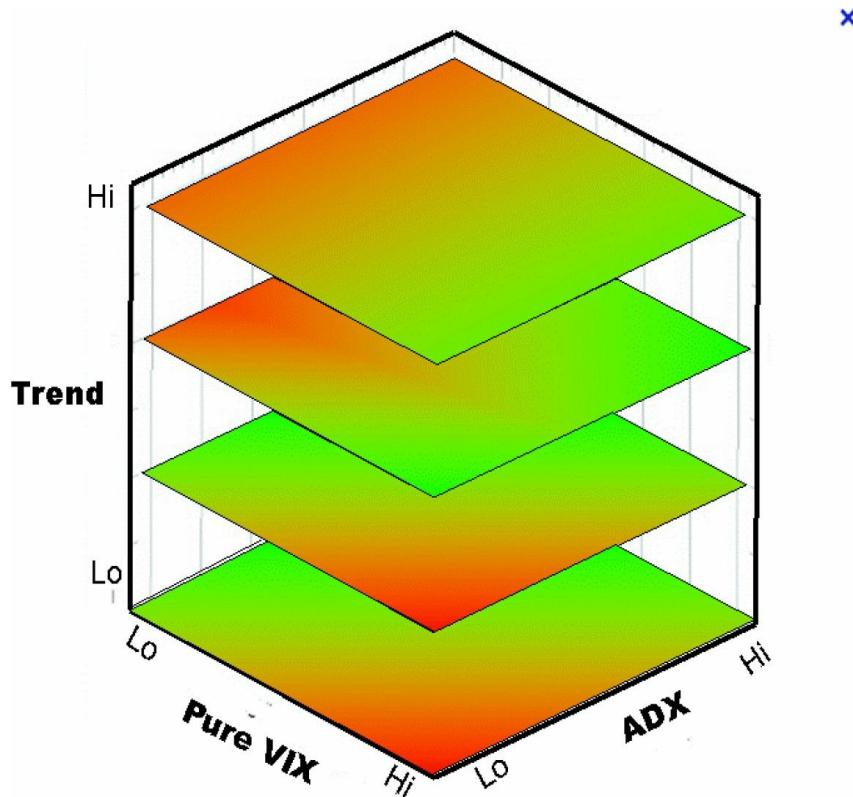
Considerable experimentation with the various display types and smoothing widths may be required in order to discover a set of parameters that portrays the information in a clear manner. Indeed, sometimes certain features will be obvious with some display parameters and invisible with others. This is especially true with the *Value* display option compared to the others. The *Value* option is ‘honest’ in the sense that the mapping from Z to tone/color is linear. Thus, an extreme value in Z will be portrayed as a similarly extreme value of tone/color. This is good if you want such extreme values to be prominent. However, the price paid for this honesty is high: the remainder of the Z variable’s distribution will have only a narrow range of tones/colors with which to be displayed. Detail in the bulk of the distribution will be obscured.

Finally, as with most plot functions in *TSSB*, the user has the ability to specify display limits for the variables. By default, the limits are computed automatically according to the actual range of the data. If the user checks any of the four *limit* boxes, the corresponding specified limit(s) will override the default choice. The left pair of limit blocks applies to the horizontal variable, and the right pair applies to the vertical variable.

## Trivariate Plots

A trivariate plot expands a bivariate plot by bringing in a third indicator. This third variable is used to slice a cross section in the relationship defined by three indicators and the target. One can think of a trivariate plot as a bivariate plot that is conditional on a third indicator. The two bivariate indicators may relate to the target in one manner when the third variable is large, and in another manner when the third variable is small. A trivariate plot lets us visualize such relationships.

An example of this cross-sectioning process is shown in [Figure 46](#) below. In this figure, each layer is a bivariate plot showing how PureVIX and ADX relate to a forward-looking target variable. However, the nature of this relationship depends on the current trend. So we see that the layers corresponding to each trend are different. A trivariate plot lets us examine a single such layer.

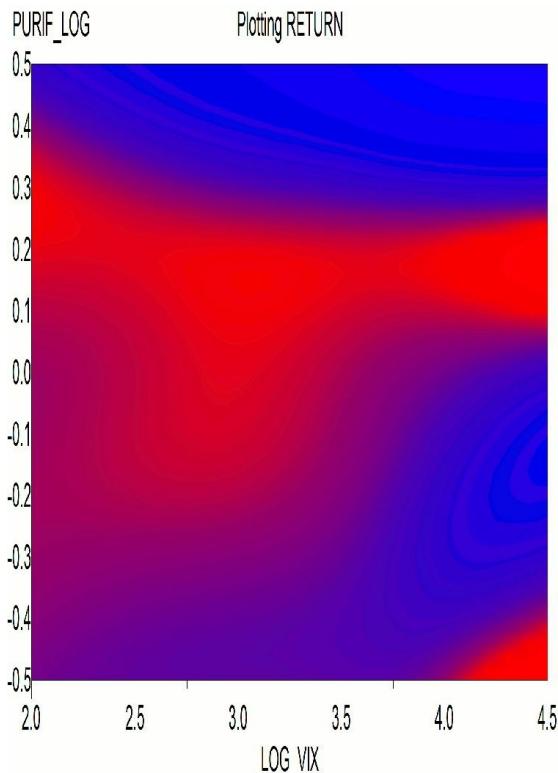


**Figure 46:** A trivariate plot in which PureVIX and ADX are cross-sectioned by Trend

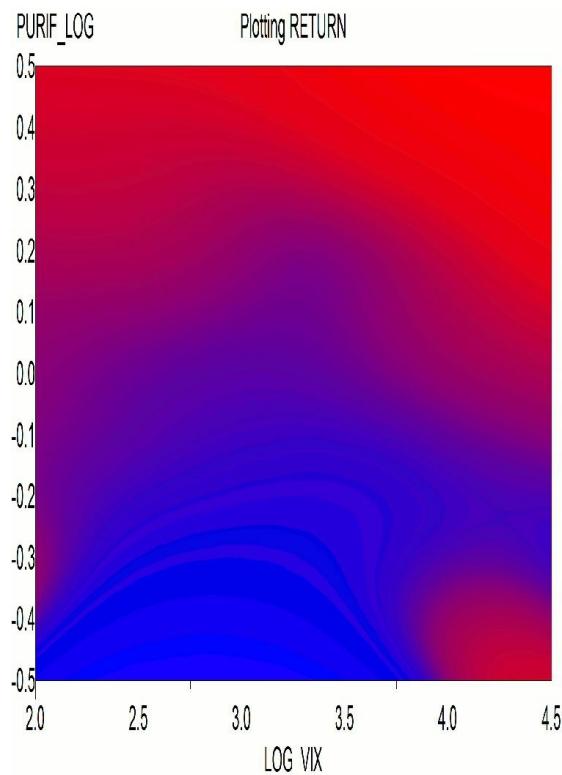
The parameters for the trivariate plot are those for a bivariate plot plus three more. The user must specify the variable that is used for cross-sectioning, the value of this cross-sectioning variable, and the relative width of the cross-sectional slice. If this cross-sectional width is very narrow (much less than 1.0) and the dataset contains few cases in the vicinity of the specified value for the

cross-sectional variable, the plot may be noisy and distorted. On the other hand, if this width is very large (much greater than 1.0), a good cross-sectional representation may not be obtained, because too much of the volume will be blurred together. Some experimentation may be required to obtain the best display.

Figures 47 and 48 below depict the same relationship, with the same parameters, as the bivariate plot in [Figure 43 here](#). However, Figure 47 shows this relationship when the indicator called DUMMY is near the value 30, while [Figure 48](#) shows the relationship when DUMMY is near -30. Notice the significant difference, especially at the top-right!



**Figure 47:** Plot with DUMMY near 30

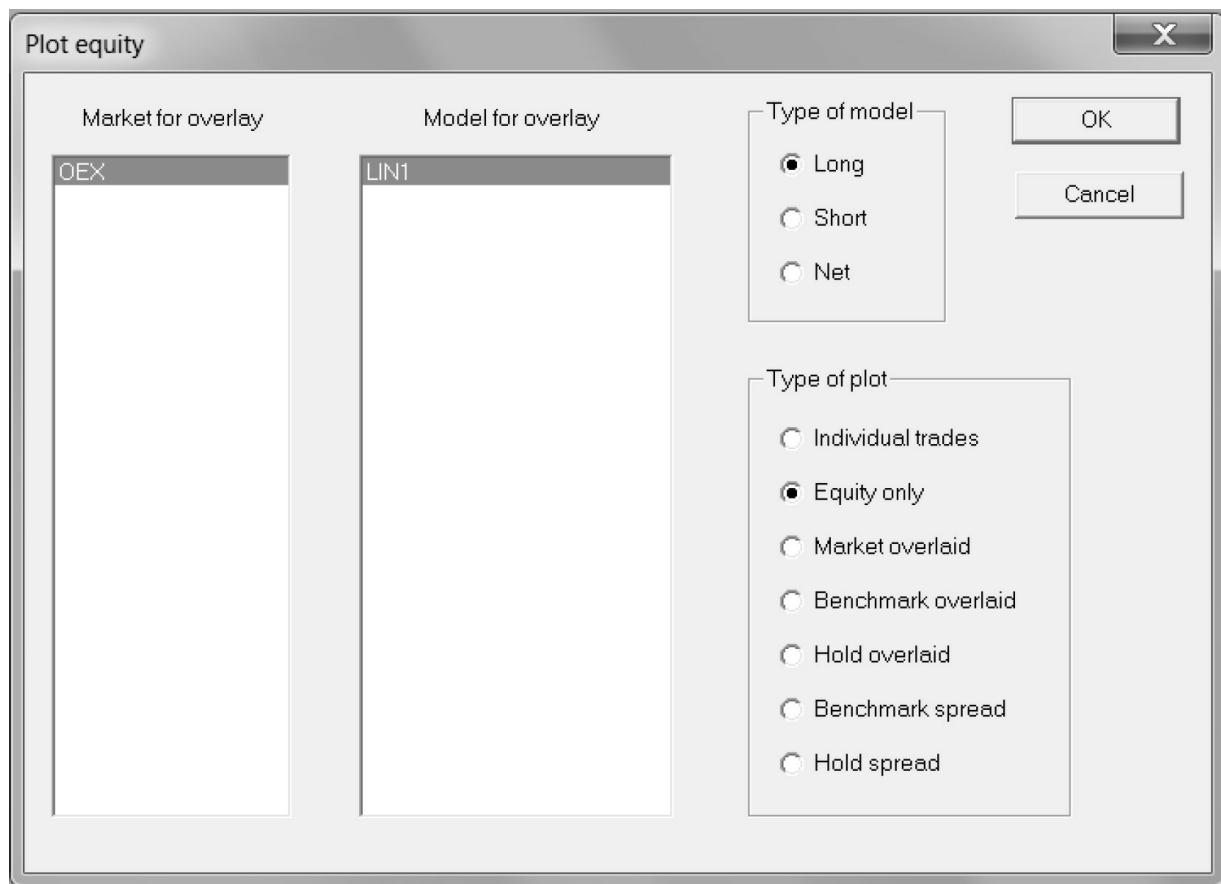


**Figure 48:** Plot with DUMMY near -30

# Equity

If the rulebase contains one or more Models, Committees, Oracles, or Portfolios, and a TRAIN, WALKFORWARD, or CROSS VALIDATE command has been executed, the equity can be graphed. If a WALK FORWARD or CROSS VALIDATE command has just been executed, the equity is the out-of-sample performance.

The user must choose whether the equity curve portrays the performance of only long trades, only short trades, or the net performance of all trades. This is done by selecting one of the three choices shown in the upper-right corner of the Plot Equity dialog shown in [Figure 49](#) below. Because Portfolios are by nature long-only, short-only, or net long+short, the choice of which to display in a plotted equity curve is ignored when a Portfolio is selected for plotting.



**Figure 49:** Dialog box for plotting equity

The user must select exactly what is plotted from among the following choices:

***Individual trades*** - The equity of each individual trade is plotted. This plot is not usually very informative, but it can be handy for locating time periods of unusually fast or slow trading.

**Equity only** - The cumulative equity is plotted.

**Market overlaid** - The cumulative equity is plotted in black, and the log of the market price is overlaid in red. If multiple markets are present, the user must select the market to be overlaid.

**Benchmark overlaid** - The cumulative equity is plotted in black, and a benchmark cumulative equity curve is overlaid in red. *Benchmark overlaid* is probably the most informative plot in the equity curve family, as the benchmark shows the average equity that would be obtained by a trading system that makes random guesses for its trade decisions, but that makes the same number of long and short trades as the system whose equity is selected. This red benchmark curve shows how much the development process has used an imbalance in long and short positions to take advantage of any long-term trend in the market. The degree to which the actual equity curve exceeds the benchmark shows the degree to which the trading system is exhibiting intelligence in its choice of trades. An example of a *Benchmark Overlaid* plot is shown in [Figure 50 here](#).

**Hold overlaid** - The cumulative equity is plotted in black, and the cumulative equity curve of a buy-and-hold system for a long-only plot, or a sell-short-and-hold system for a short-only plot, is overlaid in red. If the user selected a ‘net long+short’ plot, *Hold overlaid* makes no sense, as the red overlay will just be the equity curve of an always-neutral system, not very interesting! In fact, even if the user is plotting long-only or short-only equity, the *Hold overlaid* option is of questionable value, because the different amount of time spent in the market for the trading system and for the ‘hold’ system causes a serious difference in scaling, making visual interpretation difficult. The *Benchmark overlaid* option is far better than *Hold overlaid* in most situations.

**Benchmark spread** - The difference between the cumulative equity of the system being selected and that of the benchmark described under *Benchmark overlaid*, is plotted. This is the degree to which the performance of the trading system exceeds the average performance of a system that takes equal advantage of any trend in the market but that makes random guesses for its trade decisions. The primary advantage of this plot is to see if there are meaningful epochs when the spread expanded (a good thing) or contracted (a bad thing).

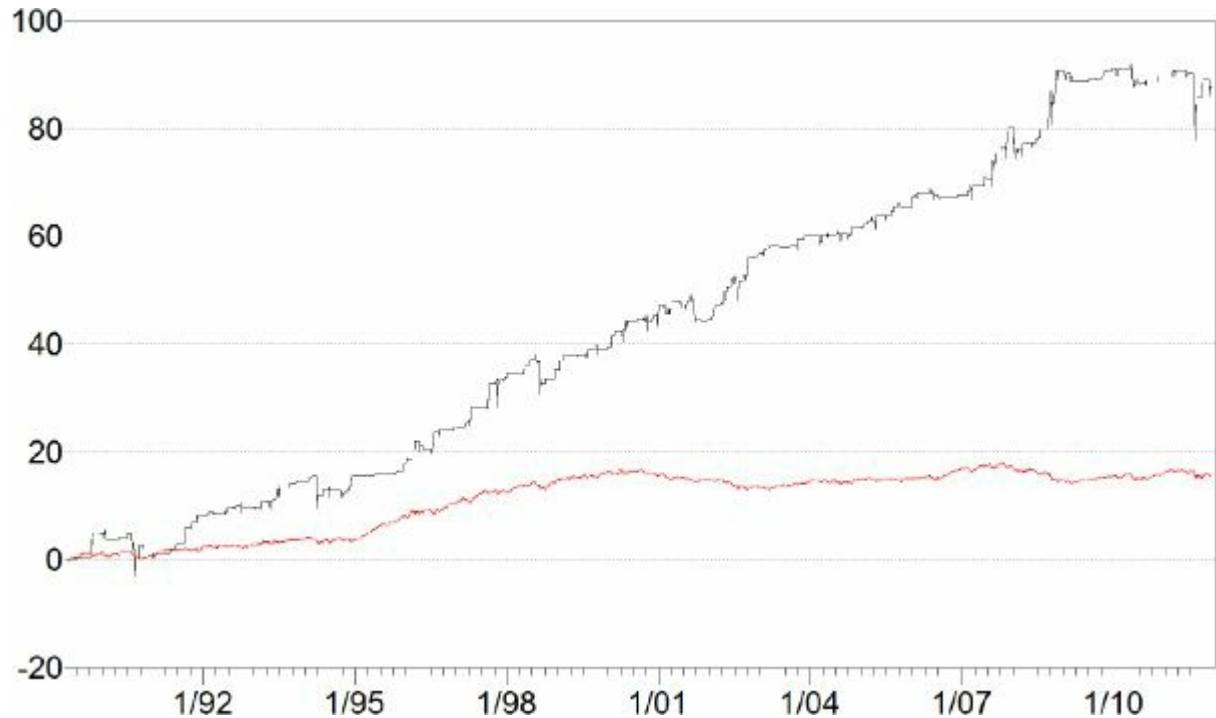
**Hold spread** - The difference between the cumulative equity of the system being

studied and that of a buy-and-hold strategy (for a long-only plot) or a sell-short-and-hold strategy (for a short-only plot) is shown. Because comparing the equity of a trading system to that of a hold strategy is not generally useful due to scaling issues, the *Hold spread* plot is not recommended.

There are several issues to consider when plotting equity curves:

- The date corresponding to each position on the plot is that of the trade decision which will cause the equity change. This is obviously prior to the actual change in equity. This may be annoying to some users, although it can also be argued that flagging an equity change when it is ‘in the box’ even if not yet realized is the better approach. In any case, it is unavoidable, because equity curves are based on targets which can have distant look-aheads. In fact, some targets (such as the TSSB library’s HIT OR MISS family) have an indefinite lookahead. Moreover, if the user imports targets from an external source, the lookahead is undefined. Thus, the only possibility is to plot an equity change at the moment that it is known to be in progress and guaranteed, rather than waiting until it has been attained.
- Benchmarks, which may optionally be overlaid with the equity curve, are computed differently for Models (including Committees and Oracles) versus Portfolios. For models, benchmark normalization is based on the number of long and short trades across the entire out-of-sample period, with all folds pooled. This is probably the best approach, as pooling across what may be unusually active and inactive years yields stability. However, such pooling is not possible for Portfolios, because in general different Models will be used as Portfolio members in each fold. These Models may have very different target variances (which impacts their contribution to the benchmark) and they may have very different trading frequencies. Thus, there is no way to pool benchmarks across all folds. The benchmark must be evaluated separately for each fold, based on the Models present and the trades they make. This is not a disaster; in fact, some might argue that this approach should be used for Models as well as Portfolios. However, it does sometimes lead to seemingly anomalous behavior. For example, suppose some Model has no out-of-sample trades in a given year. The benchmark curve for this Model will nonetheless change during that year, while the contribution of this Model to a Portfolio’s benchmark will be flat for this year. Reasonable users may argue whether this is a good or a bad thing, but the fact remains that for Portfolios there is no alternative if testing is to be honest.
- If multiple markets are present, equity curves and benchmarks may show

sudden sharp jumps up or down. This effect, which is ugly and may be mistaken for erroneous operation, is really due to markets appearing and disappearing in the market universe. Some markets (i.e. Google) are young, appearing in trading in the last few years. Other markets may disappear when a company merges with another company or ceases doing business. As a result, major contributions to equity curves and benchmarks may come and go, resulting in large discontinuities in the curves.



**Figure 50:** A *Benchmark overlaid* plot

# Prediction Map

It can be informative to examine a display of how a pair of input variables maps to a prediction in a trained model. Obviously, this helps in understanding how a model operates, the patterns that it sees in the data that lead it to make decisions.

Beyond this, it can help the user distinguish between nonlinear models that are acting on real information versus those that are modeling noise. A valid model will tend to have predictions that vary smoothly across the range of predictors, while those that model noise will tend to have predictions that clump into numerous small groups with irregular boundaries. (Of course, this applies only to nonlinear models such as neural networks that are capable of such clumping.)

The following things should be kept in mind:

- Because the map is two dimensional, it can be applied only to models that use two predictors. Only two-input models are shown in the list of available models, and if no two-input models exist, the *Prediction Map* option is grayed out.
- It makes no sense to consider a prediction map for a committee or oracle, which uses model outputs as its inputs.
- The model displayed will be the most recently trained model. Thus, it is nearly always best to use a TRAIN command as the last command in the script file if you plan on displaying a prediction map. Otherwise, the map will be based on only the most recent fold, which is unlikely to be what the user wishes.

The user can set the following options:

**Model** - This lists all models that have two inputs. The user selects the desired model.

**Horizontal Variable** - The user selects which of the two inputs will be mapped along the horizontal axis.

**Vertical Variable** - The user selects which of the two inputs will be mapped along the vertical axis.

**Lower Limit** - This comprises a check box and a field for entering a numeric value. If the box is checked, the user must enter a number in the field. This will be the lower (left or bottom) limit for the plot. Limits can be set separately for the horizontal and vertical

variables.

**Upper Limit** - This is similar to the *Lower Limit* except that it specifies the upper (right or top) limit for the plot.

**Resolution** - By default, the map will be computed as a 51 by 51 grid. This should be fine for most applications. If you wish to print a very high resolution plot for publication, you can set a higher resolution, at the price of slower display time.

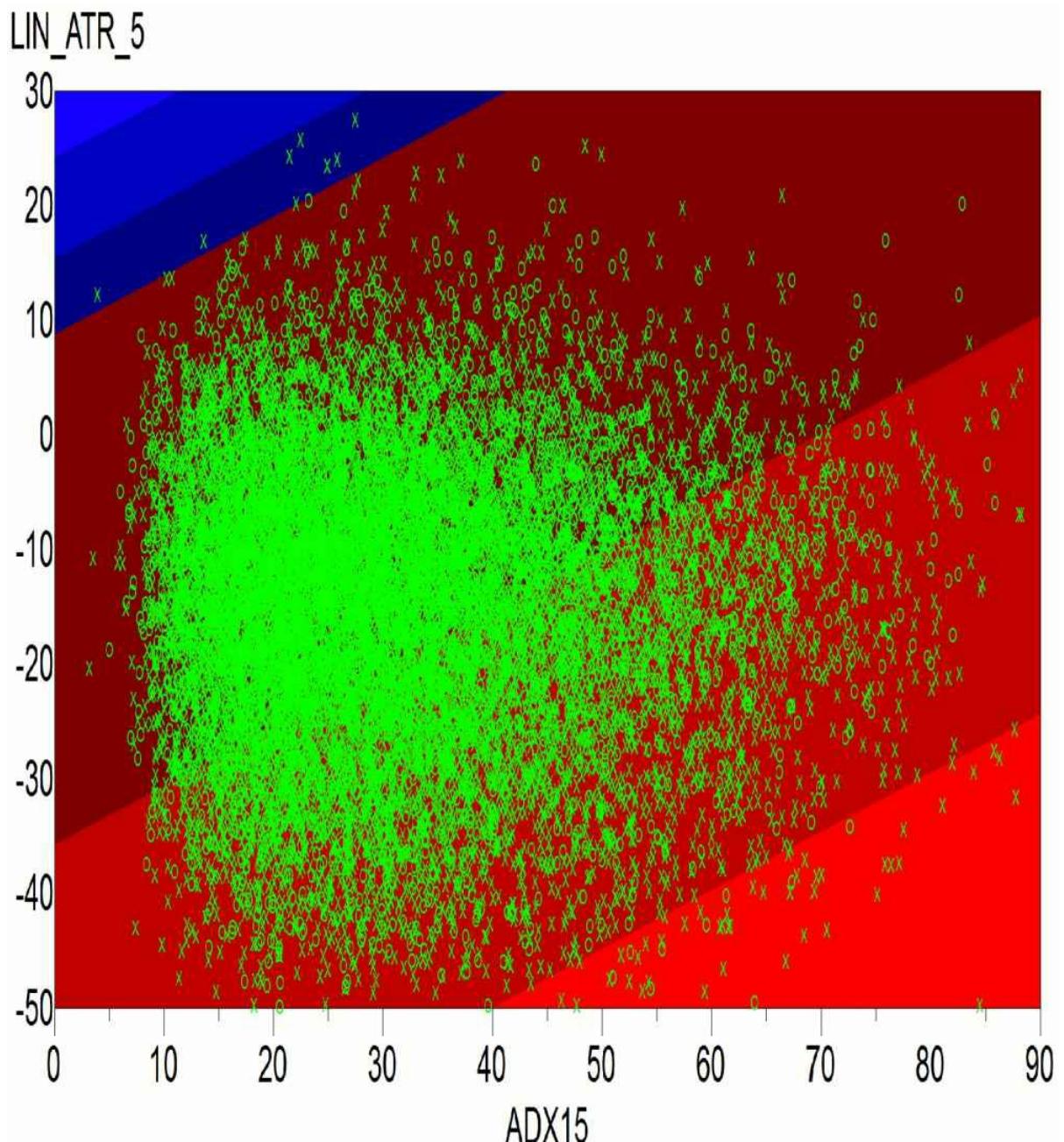
**Show Cases** - If this box is checked, training cases are displayed on the map. Those with a positive target value are printed as X, and those with a negative or zero target are printed with the letter O.

**Multiple Tones** - If this box is not checked, two colors will be used: red for positive predictions and blue for zero or negative predictions. By checking the *Multiple Tones* box, three levels of shading for each color will be used. Red will still be used for positive predictions, and blue for negative and zero. But the brightest red and blue will be reserved for the most extreme ten percent of predictions of that sign. The dimmest version of each color will be used for predictions below the median (in absolute value). Intermediate brightness will be used for the remainder (values between the median and the 90'th percentile). But see the next option, which impacts color assignment.

**Split at Median** - The default color assignment mentioned above (red for positive and blue for negative) can be overridden by checking the *Split at Median* box. This option is handy in filtering applications in which so many cases are profitable trades that most or all predictions are positive. When this box is checked, the split for color assignment will occur at the median of the target rather than at zero. If this box is not checked and all predictions have the same sign, a single color will be used even if the *Multiple Tones* box is checked, making the plot worthless. Thus, if all predictions have the same sign, the *Split at Median* box should be checked in order to have a meaningful plot.

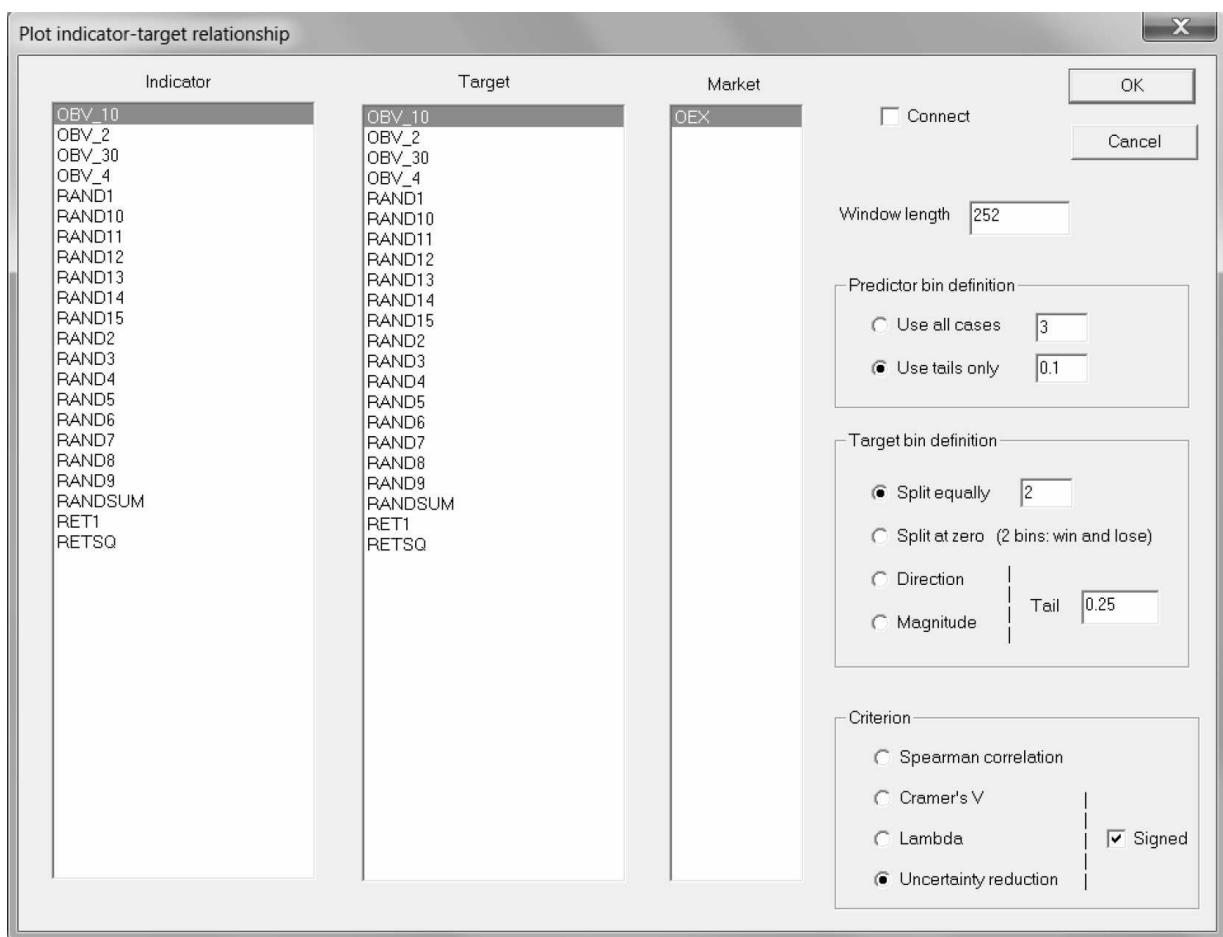
A typical prediction map is shown on the [here](#). This is for a model that filters a decent long-only trading system. Because the trading system has many more wins than losses, the model is biased toward predicting positive trades (red), so the user may wish to check the *Split at Median* box to equalize colors. (It was not checked for this display.)

This map reveals some interesting things about operation of the model. Larger values of ADX15, a 15-bar ADX measure, result in predictions of more profit. At the same time, larger values of LIN\_ATR\_5, a 5-bar trend indicator, result in predictions of less profit. Thus, we see that the filtering model is heavily based on short-term counter-trend; if the market has a strong short-term up-trend, the filtering model is likely to abort a trade recommended by the external trading system. This is especially true if ADX15 is small. Interesting.



# Indicator-Target Relationship

TSSB offers several algorithms for relatively quickly screening batches of indicators to assess their degree of relationship to a target. The chi-square test ranks indicators based on their individual relationships to the target, and the nonredundant predictor screening test ranks indicators in terms of their predictive power that is not redundant with other predictors. Both tests also offer statistical assessment of the probability that relationships are legitimate as opposed to good luck. But both of these tests benefit from a supplement, the ability to examine the consistency of a relationship across time. The *Indicator-Target Relationship Plot* allows the user to view a historical plot of the strength and nature of the relationship between an indicator and a target. The dialog for performing this test is as shown below:



**Figure 52:** Dialog box for Indicator-Target relationship

The user inputs the following information:

**Indicator** - The independent variable for relationship computations

**Target** - The dependent variable for relationship computations

**Market** - If multiple markets are present, the user must specify the market to be plotted.

**Connect** - If this box is checked, the plot will be a single line consisting of individual line segments connecting the points. If it is not checked, the plot will be a separate vertical line segment connecting each point to the horizontal axis. For this application, this latter option (box checked) is generally more effective.

**Window length** - The length of the moving window used to compute the relationships. If the user specifies a length less than 2 it will be set to 2. If the length exceeds the number of data points it will be reduced to the number of data points. The plot is technically undefined for the first *window*-1 points, so these early points are all set equal to the value at the *window* point.

**Predictor bin definition** - The user can either select ‘use all cases’ and specify the number of bins to employ, or select ‘use tails only’ and specify the tail fraction, a number greater than zero and less than 0.5. In the former case, the range of the predictor is split in such a way that all bins contain approximately the same number of data points. In the latter case, the specified fraction of extreme values of the indicator are kept in each tail. Thus, the total number of cases kept is twice the tail fraction. Choosing ‘use tails only’ is usually most appropriate because the majority of predictive information tends to be in the tails. A tail fraction of 0.05 to 0.1 is typical, as this will examine the cases having the 5 to 10 percent largest and 5 to 10 percent smallest values of the indicator.

**Target bin definition** - The user can select ‘use all cases’ and specify the number of bins to employ, select ‘split at zero’, select ‘direction’ and specify a tail fraction, or select ‘magnitude’ and specify a tail fraction. In the first case, the range of the target is split in such a way that all bins contain the same number of data points. In the second case, there are two bins, and the bin membership of each case is defined by the sign (win/lose) of the target. In many applications it is most effective to split the target into two or three equal bins. Three will correspond to a big win, a big loss, and an intermediate return that is relatively inconsequential. However, this can lead to display anomalies as discussed in the examples to follow. The last two options (Direction and Magnitude) will be discussed in the context of anomalies later in this section.

**Criterion** - The user may choose from the following:

**Spearman correlation** - A rank-based (and hence immune to outliers) measure of monotonic correlation. This ranges from -1 (perfect inverse correlation) to +1 (perfect correlation).

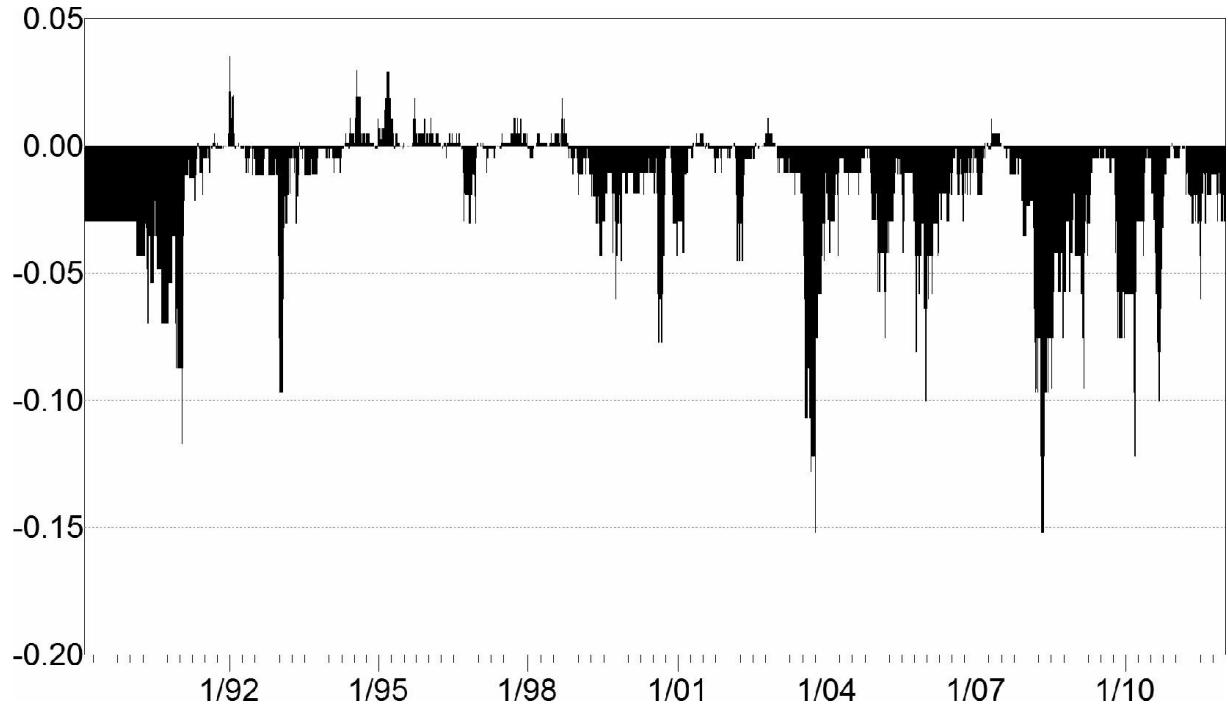
**Cramer's V** - A contingency-table-based measure of nominal (category) correlation which ranges from 0 (no correlation) to 1 (perfect correlation).

**Lambda** - A contingency-table-based measure of nominal (category) correlation which ranges from 0 (no correlation) to 1 (perfect correlation).

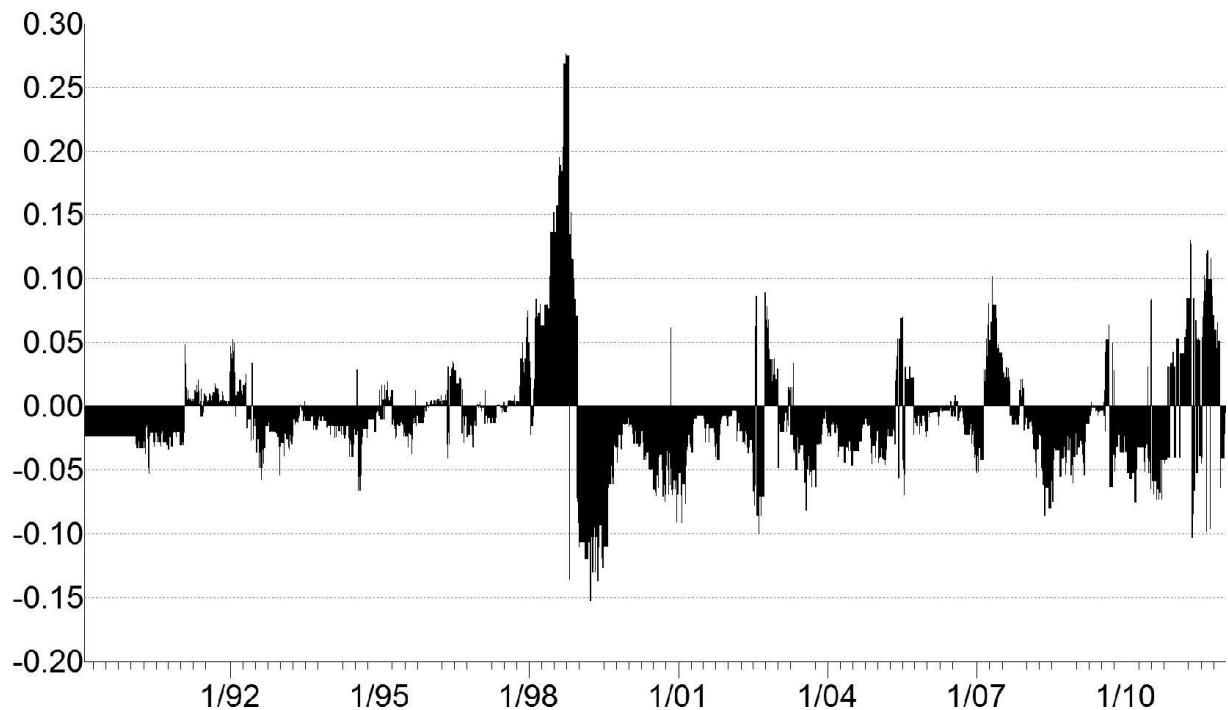
**Uncertainty reduction** - A contingency-table-based measure of nominal (category) correlation which ranges from 0 (no correlation) to 1 (perfect correlation). This, the default, is best in most applications.

**Signed** - This is valid only for Cramer's V, Lambda, and Uncertainty reduction, which are, by definition unsigned (never negative). They measure the degree of correlation, but not its nature (positive or inverse). If the 'Signed' box is checked, the nature of the relationship between the indicator and the target is assessed in each window by examining the relationship between the most extreme bins of the predictor and target. If it is determined that the relationship is primarily inverted (large values of the indicator correspond to small values of the target) then the sign of the correlation measure is made negative. This checkbox is ignored if the user chooses 'Spearman correlation' because that measure is inherently signed.

Figures [here](#) and [here](#) on the next page show examples of these plots, and they are discussed on the following page.



**Figure 53:** Signed uncertainty reduction with 2 target bins



**Figure 54:** Signed uncertainty reduction with 3 target bins

[Figure 53](#) on the prior page shows the signed uncertainty reduction based on two target bins. The degree of predictability is not very large, but probably usable because it does have a valuable trait: it almost always has the same sign. This means that the relationship between this indicator and target, though fairly small, is nicely consistent. This sort of consistency would not be seen if there were no legitimate relationship between them, and a well-designed trading system

should be able to take advantage of this relationship.

[Figure 54](#) shows the same plot with one difference: there are three target bins instead of two. The plot does not look nearly as good, because in late 1998 the indicator-target relationship spikes and then shockingly switches sign! What's going on?

The answer is that for much of 1998 (and several other epochs) the indicator loses its ability to discriminate between large wins and large losses. The indicator-target relationship becomes small and temporarily switches to positive. Look at [Figure 53](#) to see this. So why does the uncertainty reduction hit nearly 0.3 during this time, a huge value? It's because during this time the indicator becomes extremely powerful for discriminating between an upcoming flat market versus a large upcoming price change. It cannot say whether this large price change will be up or down, but it can say that it's coming. Thus, during most of 1998 the plot shows the huge predictive power of the indicator, but because its ability to determine the direction of the future move is positively correlated (though very small!), the plot's sign reverses right in the middle of this highly predictive period.

This sort of anomaly is the motivation for the two options discussed in the [next section](#).

## Isolating Predictability of Direction Versus Magnitude

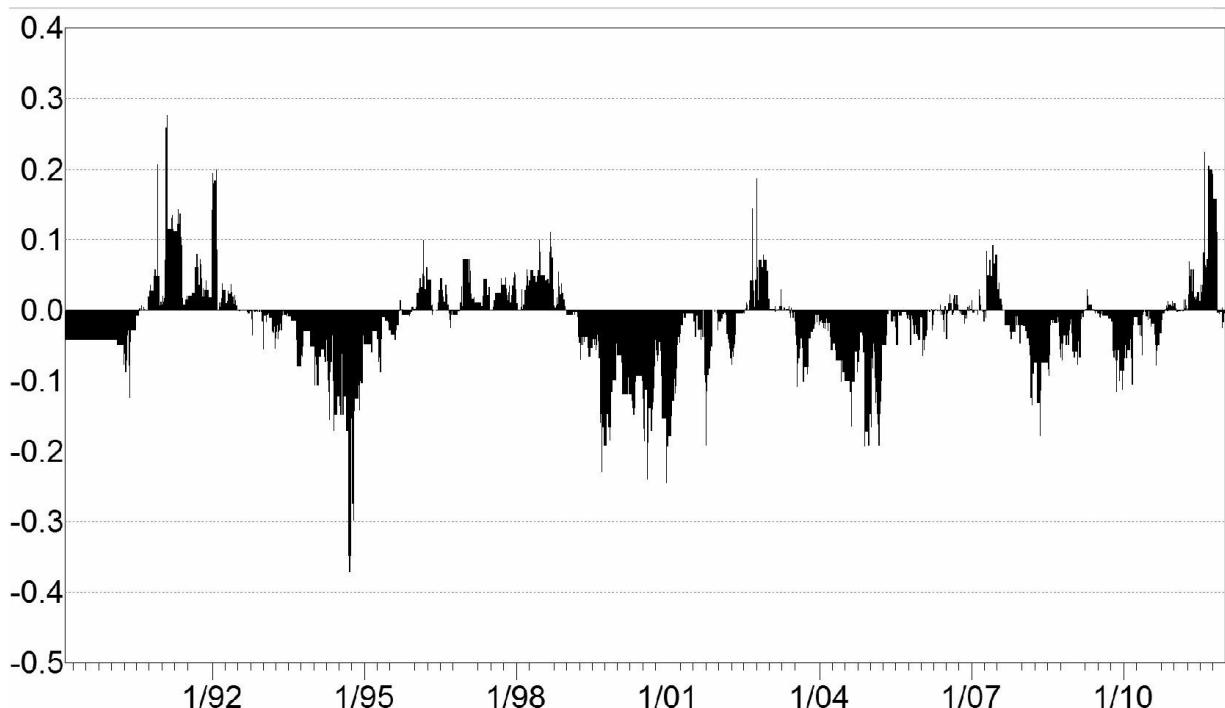
We just saw that using three target bins can produce an anomalous display when the indicator's ability to predict the magnitude (absolute value) is large but its ability to predict the direction of the move is small. One can obtain a plot that shows high predictability, but the sign of the plot flips in ugly and confusing ways, even showing narrow reversed spikes in the midst of a mountain of predictability. The cause of this phenomenon can be visualized by using the *Direction* and *Magnitude* target bin options.

Both of these options require that the user specify a tail fraction, greater than zero and less than or equal to 0.5. For the *Direction* option, the graph displayed is the indicator's ability to predict whether the target is strongly positive or strongly negative. This is computed by examining the specified upper and lower tails, ignoring the central values. This provides two target bins, and the algorithm measures how well the bin of the indicator provides predictive information about the bin (upper tail versus lower tail) of the target. If the tail fraction is specified to be 0.5, this option is equivalent to selecting 'use all cases' and setting the number of bins to two.

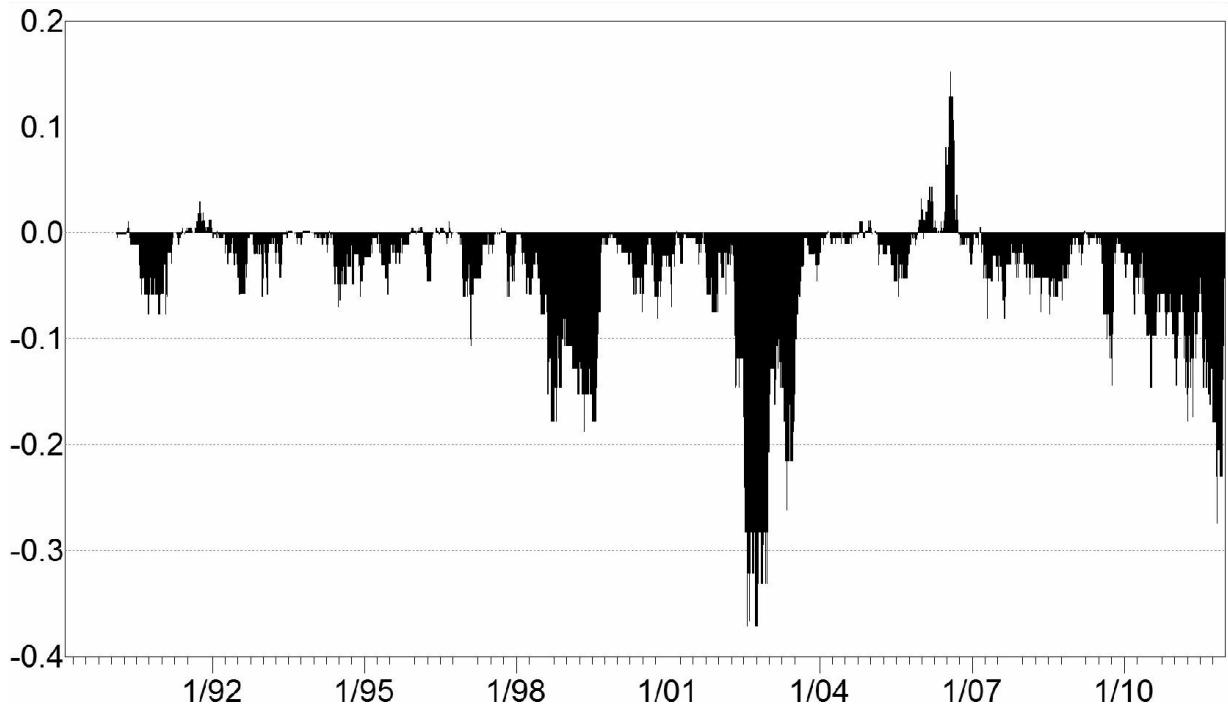
The *Magnitude* option is slightly more complex. The entire distribution is used. The two tail bins (upper and lower) are pooled into a single bin, and this is contrasted with the center bin (all cases that do not lie in a tail). Thus, we have two target bins: center and (combined) tail. The algorithm measures how well the indicator bin provides information about whether the target lies in the center bin versus the tail bin (which encompasses both the left and right tails).

The utility of this can be seen by reviewing [Figure 54](#) and then examining Figures [here](#) and [56](#) on the next page. Note from [Figure 54](#) that the anomalous area occurs late in 1998. In [Figure 55](#), which shows the ability of the indicator to predict the direction (unusually up versus unusually down) of upcoming market movement, we see that during this time period the indicator has very little power to predict the direction. But [Figure 56](#) shows that during this same time period, the indicator has strong ability to predict the magnitude of the target. This is the power that appears in [Figure 54](#) as a result of using three target bins, but the poor power to predict the direction of the market causes the sudden reversal in the plot.

Note that this indicator has strong and quite consistent power to predict the magnitude of the target. The fact that it is negative almost everywhere means that the relationship between the indicator and the magnitude of target is inverse: large values of the target imply small upcoming market movement.



**Figure 55:** Signed uncertainty reduction showing power to predict direction



**Figure 56:** Signed uncertainty reduction showing power to predict magnitude

# Finding Independent Predictors

here of the *Committees* chapter we saw many methods for creating models whose predictions will be combined using a committee. It is usually best to create these models in an intelligent, manual way. However, this may not always be possible, or even required. Sometimes we are well served by an automated approach. The FIND GROUPS command is a useful way of automating the discovery of models that can be effective committee members.

The idea is that we find small sets of predictors (typically 2 or 3) which, taken as a group, have power to predict a target variable. Moreover, the members of each predictor group are to some degree independent of the members of the other predictor groups, thus avoiding serious redundancy.

The algorithm to accomplish this is simple. The list of predictor candidates is examined and a linear model is found that optimally predicts the target variable based on a few indicators. Then, the remaining predictor candidates are tested for predictability from the prediction set just found. Any candidates that have significant predictability are considered to be redundant to the set just found and eliminated from future contention. An optimal linear model is found by testing the remaining candidates. This process is repeated a specified number of times to produce groups of predictors that are reasonably independent of one another. The syntax of this command is as follows:

```
FIND GROUPS [ Specifications ] ;
```

Most of the specifications for this command are identical to the similarly named specifications for models. Rather than repeat detailed descriptions that appear elsewhere in this document, we will just provide a brief summary of each command and then give a page reference for users who want more details.

**INPUT = [ Var1 Var2 ... ]** - Lists the input variables that are candidates for inclusion. Range and family options are also available. See [here](#).

**OUTPUT = VarName** - Names the variable that is to be predicted, the target. See [here](#).

**PROFIT = VarName** - If PROFIT is not specified here, the OUTPUT variable, which is the target being predicted, is also used for computing profit factor performance measures. However, if the target variable is not interpretable as a profit, a profit variable can be named with this optional specification. See [here](#).

**MAX STEPWISE** = *Number* - The maximum number of predictor variables that may be included in each group. This must be specified. See [here](#).

**STEPWISE RETENTION** = *Number* - Optionally specifies the number of ‘best’ candidate predictor sets kept at each stage of stepwise selection. See [here](#).

**FRACTILE THRESHOLD** - Optionally decrees that, in a multiple-market situation, predicted values are normalized fractiles across markets instead of actual predicted values. See [here](#).

**CRITERION** = *Criterion* - Names the criterion that is used to select optimal predictors. This may be any of the model optimization criteria, such as RSQUARE, PROFIT FACTOR, and so forth. See [here](#).

**MIN CRITERION CASES** = *Integer* - Specifies the minimum number of cases that must meet the optimal threshold. See [here](#).

**MIN CRITERION FRACTION** = *RealNumber* - Specifies the minimum fraction of cases that must meet the optimal threshold. See [here](#).

**RESTRAIN PREDICTED** - Causes the values of the target variable to be compressed at the extreme values before training. See [here](#).

**GROUP RSQUARE CUTOFF** = *RealNumber* - This number in the range 0-1 specifies the threshold for removing candidates from future consideration. If any candidate can be predicted from any prior group with R-square at least equal to this value, the candidate is eliminated due to redundancy. Specifying 1.0 will effectively remove all redundancy checks, which will maximize the number of groups that can be found, at the price of possibly serious redundancy. Specifying a value near zero will quickly eliminate candidates with even tiny redundancy. The result is that few groups will be possible, perhaps only one, though the groups that are able to be found will be highly independent.

**MAX GROUPS** = *Number* - The maximum number of groups that can be found. This maximum may not always be reached. Setting this to a huge value will result in all possible groups being found, though run time may be excessive.

Both AUDIT.LOG and REPORT.LOG contain detailed descriptions of each group as it is defined. In addition, AUDIT.LOG lists the R-square of each

candidate with each prior group. Since this listing can be extensive, it is not included in REPORT.LOG.

After all groups are found, a summary of the chosen variables is printed. This summary also considers the best predictors and predictor sets examined as part of the stepwise selection procedure. The printout shows which variables were selected most often. The summary contains not only values for individual predictor candidates, but it is also broken down by family, lookback, index usage, historical normalization, and cross-sectional normalization. Columns list the observed percentage selection rate, the expected rate if all variables were equally valuable, the ratio of observed to expected, and a rough estimate of the probability of this extreme ratio occurring if all variables were equally effective. The list is sorted in decreasing order of the selection ratio, because large values of this ratio imply that the variable was selected more often than would normally be expected.

When the user is employing the *FIND GROUPS* command to judge the relative importance of predictors, it is suggested that *MAX GROUPS* be set to the number of models that will later be used in committee generation. Setting the *MAX GROUPS* parameter to one will judge the candidates as if only a single model were being used, which is appropriate if that is to be the case. However, if the user will later define multiple models having mutually exclusive predictors, it is probably best to generate the same number of groups here. Of course, if the goal is to winnow down the candidates, eliminating those that are almost certainly worthless, it is probably best to set MAX GROUPS to one. Probably.

The most accurate estimation of importance will be obtained if *STEPWISE RETENTION* is set to a small fraction of the total number of candidates. Small values of this parameter will lead to identification of relatively few of the very best predictors, while large values will produce a larger, more comprehensive list of good predictors, less focused on the very best. Some experimentation may be needed to find the best compromise. In the extreme case of setting *STEPWISE RETENTION* to an effectively infinite value, all candidates will be judged to be equally important, an obviously useless judgement.

## A FIND GROUPS Demonstration

We now consider a typical use of the FIND GROUPS command. This example will use the command for two purposes: to judge the relative importance of predictors in a very large list of candidates (almost 300!), and to create reasonably independent models that will be combined via a committee. Here is the command that finds the groups:

```
FIND GROUPS [  
    INPUT = [ CMMA_5 - VMUTINF_3 ]  
    OUTPUT = DAY_RETURN_1  
    MAX STEPWISE = 3  
    STEPWISE RETENTION = 4  
    MAX GROUPS = 5  
    GROUP RSQUARE CUTOFF = 0.9  
    CRITERION = PROFIT FACTOR  
    MIN CRITERION FRACTION = 0.1  
    RESTRAIN PREDICTED  
];
```

Since one of our goals is to rank the utility of predictor candidates, we set STEPWISE RETENTION to 4, a small fraction of the total number of candidates. There is nothing special about 4; any value in that neighborhood will do. In fact, it might be useful to try several nearby values and compare results. Also, we set MAX GROUPS to 5, the actual number of models we will later use in the committee. This guarantees that the candidate rankings will reflect this exact number of models and will ignore candidates that would only appear in later groups if MAX GROUPS were larger.

The audit log file for the FIND GROUPS command is extensive, so we will here show only select portions. The first model found is as shown on the [here](#).

```
-----> Group 1 <-----  
  
LINREG Model GroupFinder predicting DAY_RETURN_1  
Regression coefficients:  
    0.001267  CMMA_5N  
    0.001775  QUODDEV_5  
    0.000842  DAU_STD_32_2  
    0.040350  CONSTANT  
  
MSE = 0.28999  R-squared = 0.00439  ROC area = 0.54209  
Buy-and-hold profit factor = 1.160  Sell-short = 0.862  
Dual-thresholded outer PF = 1.500  
    Outer long-only PF = 1.449  Improvement Ratio = 1.249  
    Outer short-only PF = 1.567  Improvement Ratio = 1.818
```

After the model is shown, the log file lists the remaining candidates in the order in which they were defined or they appeared in the database, and the R-square for each of them relative to the model just discovered. Any whose R-square exceeds the user-specified elimination threshold are flagged. Here is a small subset of this output:

```
R-square of remaining candidates with group just created...
```

CMMA_5	0.898
CMMA_10	0.686
CMMA_20	0.404
CMMA_10N	0.769
CMMA_20N	0.482
DAU_NRG_32_1	0.672
DAU_NRG_32_2	0.919
Removing DAU_NRG_32_2 with R-square = 0.919	
DAU_NRG_32_3	0.391
DAU_NRG_32_4	0.108

The remaining four models are similarly presented. Then, a selection summary is printed. If the candidates were imported from a different program via a database file, then only the variables themselves can be summarized. However, if the variables were computed internally by the *TSSB* program, more extensive summaries will be presented. The first such summary is the family, and a subset of this output is shown on the [here](#). It is sorted in descending order of the ratio of the percent of times the variable was selected to the number of times that would be expected if all candidates were equally predictive.

---

```
FIND GROUPS selection summary...
```

---

```
Total best variables = 120
```

```
Families selected...
```

Name	Percent	Expected	Ratio	Prob
QUADRATIC DEVIATION	15.00	1.08	13.9500	0.0000
LINEAR DEVIATION	12.50	1.08	11.6250	0.0000
IMAG DIFF MORLET	9.17	1.08	8.5250	0.0000
CUBIC DEVIATION	5.83	1.08	5.4250	0.0000
IMAG PRODUCT MORLET	5.83	1.08	5.4250	0.0000
ACCEL ADX	3.33	0.72	4.6500	0.0003
DAUB STD	14.17	3.23	4.3917	0.0000

In the table shown above, we see for example that a member of the QUADRATIC DEVIATION family (see [here](#)) was selected in a ‘best’

intermediate or final group 15.00 percent of the time during stepwise selection for the five groups. If all candidates had equal predictive value, a QUADRATIC DEVIATION predictor would have been selected on average just 1.08 percent of the time. So it was selected 13.95 times more often than expected. If all candidates were equally predictive, obtaining such over-representation by random chance has a probability of zero to at least four decimal places.

The next summary printed is by lookback length. Here is the beginning of this table:

#### **Lookbacks . . .**

<b>Value</b>	<b>Percent</b>	<b>Expected</b>	<b>Ratio</b>	<b>Prob</b>
5	17.50	7.17	2.4413	0.0000
32	21.67	9.68	2.2389	0.0000
16	10.83	7.17	1.5113	0.0598
20	25.83	19.00	1.3599	0.0281
10	14.17	11.11	1.2750	0.1434
15	5.00	6.81	0.7342	0.2156

We see that candidates having a lookback distance of five days were part of a ‘best’ trial group 17.5 percent of the time, when we would expect a rate of 7.17 percent. However, it is important to avoid reading too much into this table unless the candidates were carefully chosen to balance families and lookbacks. For example, it may be that many of the best families featured lookbacks of five days. If this happened to be the situation, the results are really determined by the families, and the lookbacks are just a byproduct of how the user defined the candidate variables.

Tables based on index usage ([here](#)) and historical normalization ([here](#)) are printed next. Finally, we see a table showing the individual candidate predictors. Here are the first few lines from this table. Note that even though the expected usage is the same for all variables, it is still printed for clarity and conformity with the other tables.

**Variables selected...**

Name	Percent	Expected	Prob
QUODDEV_10	11.67	0.36	0.0000
LINDEV_5	11.67	0.36	0.0000
IDMORLET_5	9.17	0.36	0.0000
DAU_STD_32_3	7.50	0.36	0.0000
CUBEDEV_20	5.83	0.36	0.0000
IPMORLET_5	5.83	0.36	0.0000
DAU_STD_32_2	4.17	0.36	0.0000
DAU_NRG_32_3	4.17	0.36	0.0000

The user may then create a script file that employs the models discovered with the FIND GROUPS command and that combines the model outputs via a committee. Here is such a script file:

```
MODEL MOD_1 IS LINREG [
  INPUT = [ CMMA_5N QUODDEV_5 DAU_STD_32_2 ]
  OUTPUT = DAY_RETURN_1
  MAX STEPWISE = 0
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
  RESTRAIN PREDICTED
] ;

MODEL MOD_2 IS LINREG [
  INPUT = [ CUBEDEV_20 DVLM_5_4_B DAU_STD_32_3 ]
  OUTPUT = DAY_RETURN_1
  MAX STEPWISE = 0
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
  RESTRAIN PREDICTED
] ;

MODEL MOD_3 IS LINREG [
  INPUT = [ LINDEV_5 DPPV_10 IPMORLET_5 ]
  OUTPUT = DAY_RETURN_1
  MAX STEPWISE = 0
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
  RESTRAIN PREDICTED
] ;
```

```
MODEL MOD_4 IS LINREG [
    INPUT = [ QUODDEV_10 DAU_MAX_32_3 VMUTINF_2 ]
    OUTPUT = DAY_RETURN_1
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    RESTRAIN PREDICTED
] ;
```

```
MODEL MOD_5 IS LINREG [
    INPUT = [ VLM_5_4 IDMORLET_5 DAU_NRG_32_3 ]
    OUTPUT = DAY_RETURN_1
    MAX STEPWISE = 0
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    RESTRAIN PREDICTED
] ;
```

```
COMMITTEE COMM IS CONSTRAINED [
    INPUT = [ MOD_1 MOD_2 MOD_3 MOD_4 MOD_5 ]
    OUTPUT = DAY_RETURN_1
    MAX STEPWISE = 99999
    STEPWISE RETENTION = 99999
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    RESTRAIN PREDICTED
] ;
```

```
WALK FORWARD BY YEAR 15 1999 ;
```

Here is an edited version of the audit log file output for this demonstration. Notice that even though this committee was generated by purely automatic means (the FIND GROUPS command), it greatly outperforms every component model in terms of out-of-sample performance.

---

```
-----  
Walkforward is complete. Summary...  
-----
```

```
Model MOD_1...
```

```
Results for the restrained values...
MSE = 0.37591   R-squared = -0.00219   ROC area = 0.52003
Buy-and-hold profit factor = 1.099   Sell-short = 0.910
Dual-thresholded outer PF = 1.164
    Outer long-only PF = 1.293   Improvement Ratio = 1.177
    Outer short-only PF = 0.972   Improvement Ratio = 1.067
```

**Model MOD\_2...**

**Results for the restrained values...**

MSE = 0.37687 R-squared = -0.00474 ROC area = 0.51942  
Buy-and-hold profit factor = 1.099 Sell-short-and-hold =  
0.910

Dual-thresholded outer PF = 1.113

Outer long-only PF = 1.090 Improvement Ratio = 0.992  
Outer short-only PF = 1.143 Improvement Ratio = 1.256

**Model MOD\_3...**

**Results for the restrained values...**

MSE = 0.37637 R-squared = -0.00342 ROC area = 0.50997  
Buy-and-hold profit factor = 1.099 Sell-short-and-hold =  
0.910

Dual-thresholded outer PF = 1.060

Outer long-only PF = 1.428 Improvement Ratio = 1.300  
Outer short-only PF = 0.978 Improvement Ratio = 1.074

**Model MOD\_4...**

**Results for the restrained values...**

MSE = 0.37710 R-squared = -0.00535 ROC area = 0.49917  
Buy-and-hold profit factor = 1.099 Sell-short-and-hold =  
0.910

Dual-thresholded outer PF = 1.029

Outer long-only PF = 1.087 Improvement Ratio = 0.990  
Outer short-only PF = 0.958 Improvement Ratio = 1.052

**Model MOD\_5...**

**Results for the restrained values...**

MSE = 0.37653 R-squared = -0.00383 ROC area = 0.51577  
Buy-and-hold profit factor = 1.099 Sell-short-and-hold =  
0.910

Dual-thresholded outer PF = 0.988

Outer long-only PF = 1.130 Improvement Ratio = 1.028  
Outer short-only PF = 0.904 Improvement Ratio = 0.994

---> Committee results <---

**Results for the restrained values...**

MSE = 0.37520 R-squared = -0.00029 ROC area = 0.52996  
Buy-and-hold profit factor = 1.099 Sell-short-and-hold =  
0.910

Dual-thresholded outer PF = 1.302

Outer long-only PF = 1.452 Improvement Ratio = 1.322  
Outer short-only PF = 1.124 Improvement Ratio = 1.234

# Market Regression Classes

When working with a large collection of markets, such as the components of the S&P500, it is often too much to expect that a single prediction model will be useful across all markets in our universe. It may be that one model performs well for interest-rate issues, while another does well for consumer product companies. If we try to find one ‘universal’ model that performs well for all markets, we will probably end up with a model that performs excellently for none. For this reason, *TSSB* has the ability to categorize a universe of markets into subgroups that have similar relationships between predictors and the target. This is done with the following command:

```
REGRESSION CLASS [ Specifications ] ;
```

Most of the specifications for this command are identical to the similarly named specifications for models. Rather than repeat detailed descriptions that appear elsewhere in this document, we will just provide a brief summary of each command and then give a page reference for users who want more details.

**METHOD = HIERARCHICAL / SEQUENTIAL / LEUNG** - Specifies the class separation algorithm to use. The HIERARCHICAL method uses straightforward hierarchical clustering to produce classes that maximize the minimum ROC area (Receiver Operating Characteristic curve, [here](#)) in component markets. This has many wonderful optimality properties but is much too slow unless the number of cases is small. The SEQUENTIAL method is a modified hierarchical method that is sub-optimal but runs at a reasonable rate. The LEUNG method uses the algorithm of Leung, Ma, and Zhang (2001), slightly modified, to find classes. Each class is guaranteed to have a ROC area greater than 0.5 and a positive Spearman Rho between the predicted and the target.

**INPUT = [ Var1 Var2 ... ]** - Lists the input variables that are candidates for inclusion. Range and family options are also available. See [here](#).

**OUTPUT = VarName** - Names the variable that is to be predicted, the target. See [here](#).

**MAX STEPWISE = Number** - The maximum number of predictor variables that may be included in each group’s prediction model. This must be specified. See [here](#).

**RESTRAIN PREDICTED** - Causes the values of the target variable to be compressed at the extreme values before training. See [here](#).

**MIN CASES = Integer** - Specifies the minimum number of cases that must be in a market in order for the market to enter into computations. Markets that have fewer cases will not join any class. Larger values encourage greater stability by preventing minor markets from exerting undue influence on results.

**MAX GROUPS = Number** - This does not affect computation, only the printing of results in the log file. The hierarchical and sequential algorithms begin by considering each market to be its own class. As time passes, the number of classes is gradually reduced by merging classes. When the number of classes drops to or below the MAX GROUPS threshold, detailed group membership information will be printed to the log files. The Leung method operates in the opposite direction: classes are generated one at a time. Class membership will be printed until more than MAX GROUPS classes are obtained. If you set MAX GROUPS to a very large number, the audit log file may be huge because it contains vast detail. In general, you should set this to the number of regression classes you are truly interested in, or a somewhat larger number, for economy in the log file.

**INITIALIZE = Integer** - (Mandatory for the SEQUENTIAL method, ignored for the other methods) This specifies the number of class pairs that are kept for merge testing (described in detail later). Execution time is a roughly linear function of this value, and larger values result in more accurate performance. Values around 1000 are probably a reasonable compromise between time and quality in most cases, although it should be set to as large a value as computer time allows.

# REGRESSION CLASS Demonstrations

We now consider typical uses of the REGRESSION CLASS command. These examples will demonstrate each of the three algorithms, and more information about the nature of each algorithm will be provided in the context of discussing audit log output. These examples use the markets in the S&P 100, demanding that a market contain data for at least 500 days in order to be considered. They employ nine predictor candidates: linear, quadratic, and cubic trends at lookbacks of 5, 15, and 50 days. In most practical applications, considerably more than nine predictor candidates will be used. However, execution time increases rapidly as the number of indicators increases, so they are limited for now. At most two predictors will be used in each model, and details are to be printed for only the ten dominant groups. These specifications appear in the declarations of all three examples.

## The Hierarchical Method

The hierarchical method for creating regression classes is the most versatile of the three methods, but it is also the slowest in most cases. If time allows and the user wants maximum control, this is the method that should be used. The Leung method is also excellent, but it is more rigid in terms of user control. This will be discussed later.

The hierarchical method examines all possible pairs of markets. For each pair (such as IBM and BOL) it pools the cases in the two markets and then fits an optimal linear model using stepwise predictor selection. Three ROC areas for this model (see [here](#)) are computed: that for each of the two markets individually, and that for the pooled pair of markets. The merging criterion for this pair of markets is defined the smallest of these three ROC areas. The motivation for this definition is that if we try to use a single model to handle two markets, then this model should be effective in each market individually as well as in the pooled data. To the degree that *any* of these three interests are not met, the pooling should be discouraged.

Once this ROC-based merging criterion is computed for every possible pair of markets (a *slow* process if there are a large number of markets!), some of the best market pairs are merged. After this initial merging is complete, the process is repeated. However, in this second round of merging, some pairs of markets will have been merged in the first round. Such merged pairs are treated as if they are a single market. Then this process is repeated over and over, until just two market sets remain. It should be apparent that in general, the ROC area at

each round of merging will steadily decrease as we force each single model to handle more markets.

Here is the REGRESSION CLASS script file declaration for this example, followed by the audit log output for the run:

```
REGRESSION CLASS [
  INPUT = [ LIN_ATR_5 - CUB_ATR_50 ]
  OUTPUT = RETURN
  MIN CASES = 500
  MAX STEPWISE = 2
  MAX GROUPS = 10
  RESTRAIN PREDICTED
  METHOD = HIERARCHICAL
] ;
```

```
-----  
REGRESSION CLASS by hierarchical merging beginning...  
-----
```

```
Starting with 74 valid markets
Iteration 1 leaves 65 groups Crit=0.5546
Iteration 2 leaves 57 groups Crit=0.5529
Iteration 3 leaves 50 groups Crit=0.5471
Iteration 4 leaves 44 groups Crit=0.5463
Iteration 5 leaves 39 groups Crit=0.5439
Iteration 6 leaves 35 groups Crit=0.5424
Iteration 7 leaves 31 groups Crit=0.5381
Iteration 8 leaves 28 groups Crit=0.5362
Iteration 9 leaves 25 groups Crit=0.5351
Iteration 10 leaves 22 groups Crit=0.5344
Iteration 11 leaves 20 groups Crit=0.5339
Iteration 12 leaves 18 groups Crit=0.5333
Iteration 13 leaves 16 groups Crit=0.5316
Iteration 14 leaves 14 groups Crit=0.5320
Iteration 15 leaves 13 groups Crit=0.5308
Iteration 16 leaves 12 groups Crit=0.5298
Iteration 17 leaves 11 groups Crit=0.5296
Iteration 18 leaves 10 groups Crit=0.5284
```

**Markets in each group...**

**Class 1**

**AA**

**AVP**

**AXP**

**BAC**

**... {Remainder of Group1, followed by Groups 2 through 8}**

**Class 9**

**MMM**

**Class 10**

**SLB**

**... {Iterations 19 through 25}**

**Iteration 26 leaves 2 groups Crit=0.5154**

**Markets in each group...**

**Class 1**

**AA**

**ABT**

**AMGN**

**AVP**

**AXP**

**BA**

**BAC**

**... {Remainder of Group 1, followed by Group 2}**

This audit log output begins by stating that there are 74 valid markets, each of which defines its own group. Actually, there were 100 markets, but the script imposed the restriction MIN CASES = 500 which removes from consideration any markets that do not contain at least 500 days. There were 26 such markets due to the changing membership of the S&P 100.

After the first pass, we are down to 65 groups because 9 pairs of markets were merged. At the end of this first pass, the ROC criterion was 0.5546. Successive passes continue merging pairs, steadily reducing the number of groups, with the price of also steadily reducing the ROC criterion. (There is one pass in which the criterion actually increases a little. This is fairly rare and due to nothing more than random behavior.)

The script specified MAX GROUPS = 10. Thus, when we reach the point of having 10 groups, the market membership of each group is printed. This output is extensive, so most of it is omitted here.

Notice that Groups 9 and 10 are single-market groups. Price behavior in these two markets (MMM and SLB) was so unique that they did not comfortably fit in with any other markets.

Iterations continue until at Iteration 26 we are down to just 2 groups. By this time the ROC criterion has dropped to 0.5154.

There is no definitive way to know the ideal stopping point. In rare cases it may be that a particular merge causes a sudden sharp dropoff in the ROC criterion. When this happens, one would be inclined to stop just before this awkward merge. Unfortunately, it is usually the case that the criterion slowly and steadily drops, with no sharp change. Thus, the user has to trade off the benefit of having a relatively large criterion with the alternative benefit of having few groups.

## The Sequential Method

The hierarchical method just described has great theoretical and practical advantages. However, the computer time needed to find the optimal model for *every possible pair* of markets can be prohibitive. The sequential method does a respectable job of imitating the hierarchical method but with considerably more speed.

The sequential method begins by computing the ROC criterion for a subset of the possible pairs. In this example, we specify INITIALIZE = 1000 to tell it to compute the criterion for 1000 randomly selected pairs of markets. The pair with the maximum ROC criterion is merged. Bookkeeping is done to account for the fact that these two markets are no longer separate entities, and then the vector of *INITIALIZE* pairs (1000 in this example) is topped off with more randomly selected pairs. Once again the pair having the best ROC criterion is merged. This process is repeated until only two classes remain, just as was the case for the hierarchical method.

In essence, the only difference between the hierarchical method and the sequential method is that the former tests all possible pairs for merging at each step, while the latter tests only *INITIALIZE* pairs. If *INITIALIZE* is much less than the number of possible pairs, the usual situation, then the time savings will be substantial.

One might worry that there is a high probability that the ‘best’ merging pair will not be selected for the randomly chosen initialization set. This is true, but it is not usually a serious problem. The reason is that even though the initialization set will almost certainly fail to contain numerous pairs whose merging criteria

are superior to all pairs in the complete set, nonetheless the best pair in the initialization set will almost certainly be very good, a pair which should be merged. It may not be the best, which the hierarchical method would immediately merge, but it is still a pair that the hierarchical method would merge early in the process. And as soon as this best pair is merged in the sequential method, the set will be topped off with more randomly selected pairs. The end result is that the sequential method merges almost the same markets as the hierarchical method, but it does so in a slightly different order. This order may not be optimal, but it still ends up with about the same classes as optimal merging would.

Of course, it is possible that a severe streak of bad luck can subvert the sequential algorithm, while the hierarchical method is deterministic and hence immune to bad luck. Still, as long as *INITIALIZE* is set reasonably large, luck should play only a minimal role in results.

Here is the script file entry for sequential-method merging, followed by the audit log output for this run:

```
REGRESSION CLASS [
    INPUT = [ LIN_ATR_5 - CUB_ATR_50 ]
    OUTPUT = RETURN
    MIN CASES = 500
    MAX STEPWISE = 2
    MAX GROUPS = 10
    RESTRAIN PREDICTED
    INITIALIZE = 1000
    METHOD = SEQUENTIAL
] ;
```

---

```
-----  
REGRESSION CLASS by sequential merging beginning...  
-----
```

```
Starting with 74 valid markets
74: Merging 26 and 54 Crit=0.5635
73: Merging 5 and 69 Crit=0.5632
72: Merging 39 and 55 Crit=0.5624
71: Merging 26 and 65 Crit=0.5612
70: Merging 17 and 68 Crit=0.5611
{ ... Iterations through 16}
15: Merging 1 and 3 Crit=0.5298
14: Merging 6 and 7 Crit=0.5297
13: Merging 2 and 6 Crit=0.5304
12: Merging 7 and 10 Crit=0.5284
11: Merging 4 and 6 Crit=0.5278
```

```
-----  
Markets in each of 10 groups...  
-----
```

```
Class 1 has 12 markets and ROC area = 0.5423
```

```
AA  
CAT  
CMCSA  
DD  
DELL
```

```
... {Remainder of Group1, followed by Groups 2 through 8}
```

```
Class 9
```

```
MMM
```

```
Class 10
```

```
SLB
```

```
... {More iterations}
```

```
3: Merging 1 and 2 Crit=0.5189
```

```
-----  
Markets in each of 2 groups...  
-----
```

```
Class 1 has 69 markets and ROC area = 0.5222
```

```
AA  
ABT  
AMGN  
AVP  
AXP
```

```
... {Remainder of Group 1, followed by Group 2}
```

As with the hierarchical method, this audit log output begins by stating that there were 74 valid markets, each of which defines its own group. Actually, there were 100 markets, but the script imposed the restriction MIN CASES = 500 which removes from consideration any markets that do not contain at least 500 days. There were 26 such markets due to the changing membership of the S&P 100.

The hierarchical method merges groups in small chunks to speed operation, a trivial sacrifice of optimality that produces great speedup relative to merging one pair at a time. But the sequential method, which is already much, much faster than the hierarchical method, can afford to merge one pair at a time. The audit log identifies the group numbers of the merged pairs, which is of little interest to most users, but can provide useful information to advanced users by

identifying patterns of merging. More importantly, the audit log lists the usually decreasing ROC criterion. If it suddenly plunges, this is an indication of the point at which merging might best be terminated.

The script specified MAX GROUPS = 10. Thus, when we reach the point of having 10 groups, the market membership of each group is printed. This output is extensive, so most of it is omitted here. Notice that Groups 9 and 10 are single-market groups, as was the case with hierarchical merging. Price behavior in these two markets (MMM and SLB) was so unique that they did not comfortably fit in with any other markets. Note that even though the group memberships will usually be similar with hierarchical and sequential merging, the associated numbering may be different. In this case, Groups 9 and 10 happened to have the same group numbers in both methods, though Group 1 was different.

Iterations continue until we are down to just 2 groups. By this time the ROC criterion has dropped to 0.5189.

As with hierarchical merging, there is no definitive way to know the ideal stopping point. In rare cases it may be that a particular merge causes a sudden sharp dropoff in the ROC criterion. When this happens, one would be inclined to stop just before this awkward merge. Unfortunately, it is usually the case that the criterion slowly and steadily drops, with no sudden change. Thus, the user has to trade off the benefit of having a relatively large criterion with the alternative benefit of having few groups.

## The Leung Method

The paper “A New Method for Mining Regression Classes in Large Data Sets” by Leung, Ma, and Zhang (IEEE Transactions on Pattern Analysis and Machine Intelligence, January 2001) presents a mathematically rigorous algorithm for discovering regression classes in large and noisy datasets. The algorithm is extremely complex, so only an overview will be presented here. The Leung method differs from the hierarchical and sequential methods in several key ways:

- The hierarchical and sequential methods begin with each market defining its own class, and it steadily merges markets and existing classes to *reduce* the number of classes as it proceeds. The Leung method begins with no classes, and it discovers new classes one at a time, thus *increasing* the number of classes as it proceeds.
- Once a class is defined by the Leung method, it remains unchanged throughout the remainder of processing. In the hierarchical and sequential methods, existing classes can be merged to create larger classes.
- The Leung method can quickly identify markets that are so unlike other markets that they can never be pooled with other markets in a single class. As such markets are encountered during processing, they will be identified and removed from further consideration. The hierarchical and sequential methods identify such markets only at the end of processing when they remain isolated, having been unable to merge with any other markets.
- Perhaps most important, the Leung method contains inherent statistical tests to determine if a group of markets is so statistically consistent that these markets can indeed be considered to be members of the same regression class. Such classes will be labeled ‘Good’ in the audit log. Classes that fail an internal consistency test will be labeled ‘Bad’ in the audit log.

It is important to understand that several different and extremely sophisticated criteria are involved in the Leung algorithm. Some criteria determine whether two or more markets are cast into the same class. Others rate the statistical uniformity of a class. Naturally, there is a high correlation between these criteria. Thus, if a class consists of two or more markets, it is likely that it receives a ‘Good’ rating. Conversely, a single isolated market is likely to be given a ‘Bad’ rating. Nonetheless, it is possible, in rare circumstances, for two or more markets to be assigned to the same ‘Bad’ group. Also, an isolated market may receive a ‘Good’ rating if it has excellent internal consistency and predictive power. The bottom line is that the user should be wary of any group, regardless of its size, which receives a ‘Bad’ rating.

Here is the script declaration for the Leung method, followed by the corresponding audit log output. Note that we set MAX GROUPS=50 because the Leung method has a tendency to find small and poor groups first. We want to make sure that all interesting groups are printed.

```
REGRESSION CLASS [
    INPUT = [ LIN_ATR_5 - CUB_ATR_50 ]
    OUTPUT = RETURN
    MIN CASES = 500
    MAX STEPWISE = 2
    MAX GROUPS = 50
    RESTRAIN PREDICTED
    METHOD = LEUNG
] ;
```

---

```
-----  
REGRESSION CLASS by Leung method beginning...  
-----
```

One recalcitrant market; making it a class

```
1 classes; 73 markets remain unclassified
Class Markets
  1      1  Bad
```

Markets in each group...

```
Class 1 (Bad)
  MSFT
```

```
2 classes; 71 markets remain unclassified
Class Markets
  1      1  Bad
  2      2  Good
```

Markets in each group...

```
Class 1 (Bad)
  MSFT
```

```
Class 2 (Good)
  AMGN
  MRK
```

Making one or more poor markets their own class

```
3 classes; 70 markets remain unclassified
```

```
Class Markets
```

1	1	Bad
2	2	Good
3	1	Bad

```
Markets in each group...
```

```
Class 1 (Bad)
```

```
MSFT
```

```
Class 2 (Good)
```

```
AMGN
```

```
MRK
```

```
Class 3 (Bad)
```

```
DELL
```

```
... {More iterations}
```

```

34 classes; 0 markets remain unclassified
  Class Markets
    1      1  Bad
    2      2  Good
    3      1  Bad
    4      2  Good
    5      1  Good
    6      1  Good
    7      1  Bad
    8      2  Good
    9      2  Good
   10      1  Bad
   11      1  Bad
   12      4  Good
   13      2  Bad
   14      1  Good
   15      2  Bad
   16      3  Good
   17      2  Good
   18      1  Good
   19      2  Bad
   20      2  Bad
   21      1  Bad
   22      3  Good
   23      1  Good
   24      1  Bad
   25      4  Good
   26      1  Bad
   27      6  Good
   28      2  Good
   29      1  Bad
   30      1  Bad
   31      5  Good
   32      2  Good
   33      1  Bad
   34     11  Good

```

The first thing the algorithm does here is to identify what the program calls a recalcitrant market. This is a market that is so unusual and internally inconsistent that it cannot merge with any other market. We soon see that this market is MSFT.

It then identifies a regression class consisting of two markets: AMGN and MRK. This class is labeled ‘Good’ because these two markets can be pooled into a single dataset for which a single prediction model is effective.

A third class consisting of the single market DELL is identified. This market is labeled ‘Bad’ because it fails one or more of the internal consistency tests.

This process is repeated until a total of 34 classes are found. As is typical of the algorithm, it tends to find small and single-market classes first. It isn’t until the

12'th class that it finds a ‘Good’ class consisting of more than 2 markets. Class 27 is ‘Good’ and contains 6 markets, and it is not until its final discovery, Class 34, that we find a class that contains 11 markets. The markets that make up each of these classes are printed in the audit log but not reproduced here because the listing is extensive.

One disadvantage of the Leung method compared to the hierarchical and sequential methods is that the user has no control over the number and size of classes. In those other two methods, merging can be terminated at any point according to the user’s choice of class properties. But the Leung method uses sophisticated statistical tests to determine the optimal classes, and the user has no alternative but to accept those decisions. On the other hand, the user does have the comfort of knowing that the choices made by the Leung algorithm have a decent degree of statistical rigor, while the choice of stopping point made by the user in the hierarchical and sequential methods is more or less arbitrary.

# Developing a Stand-Alone System

In this chapter much of the material in prior chapters comes together. We will now walk through every step in the development of a stand-alone trading system. It must be emphasized that the approach shown here is not by any means the only correct approach. An infinite number of variations are possible. And due to time and space constraints, some aspects of this development will be abbreviated. Still, the procedures described here cover the most common and important aspects of system development.

# Choosing Predictor Candidates and the Target

If there is one rule that trading system developers should remember, it is this: *predictive power should come primarily from the predictor variables, not from the model*. In fact, powerful models (those able to fit complex patterns) are more likely to harm performance by overfitting than they are to help performance by harnessing subtle information in the predictors. Thus, it is almost always in the best interest of the developer to expend massive effort in finding effective predictor variables and then present them to a relatively weak model such as linear regression.

## Choosing the Target

Before delving into the complexities of predictor selection, we'll mention target selection. In most cases, selecting a target does not involve a difficult decision because the target is intimately connected to the application, such as the desired trade duration, and hence not open to much discussion. Thus, we will not devote much attention to the target. Nonetheless, here are a few common situations:

- An excellent choice is a *one-day-ahead* trading system in which a position is taken at the next open of the market after a signal is given, and then the trade is closed at the following open. This method has the valuable advantage that reasonably valid statistical tests can be performed on out-of-sample trades to estimate the probability that results as good as those obtained could have been obtained by pure random good luck. Such tests are difficult to impossible for trading systems that look ahead more than one day. One-day-ahead systems also have at most one position open at a time, which simplifies margin and risk calculations. In this case, a target such as NEXT DAY LOG RATIO or NEXT DAY ATR RETURN is appropriate.
- Another popular choice is to open a position at the next open after a signal is given, and then hold the position until either a specified profit is made or a specified loss is suffered. This use of the HIT OR MISS target has the advantage of corresponding to closing trades with *limit* and *stop* orders, a common trading habit. It also has the enormous advantage that the distribution of returns has exceptionally light tails; extreme wins and losses are impossible because they are limited by the fixed exit points. This is a great benefit to model building, as the training algorithm will not focus undue energy on eliminating severe losses or capturing huge wins. On the other hand, as with all model-based systems other than *one-day-ahead*, multiple positions are likely. As long as buy/sell signals are given

(the prediction is more extreme than the threshold), positions are piled on, even if one or more positions are already open. This can complicate real-life trading due to margin requirements and risk wariness. Unfortunately, with any model-based development system (not just *TSSB*), this problem is impossible to avoid without introducing other problems that are usually even more serious. The moral of the story is that one should always set the stops and limits in the HIT OR MISS target to be as tight as possible so that positions are kept open for only short stretches of time. Also, one can set the cutoff bar count of the HIT OR MISS target to a low value to preclude extended time in a position.

- Perhaps the least desirable choice is to specify in advance that a trade will be kept open for a fixed length of time, such as a number of days, using a target such as SUBSEQUENT DAY ATR RETURN. This shares with HIT OR MISS targets the disadvantage that multiple positions will often be open. It also shares with one-day ahead systems the disadvantage that extreme wins and losses are possible, making effective training difficult. Finally, although the *tapered block bootstrap* and the *stationary bootstrap* can theoretically be used to provide statistical tests of the validity of the trading system, these tests are notoriously unreliable in practice, and they should not be overly trusted to give valid results. In summary, a fixed multiple-day-ahead target has all of the disadvantages of the other choices, and none of the advantages. This should be a last resort.

One final word is needed concerning the trading systems that allow multiple positions to be open. This includes all systems except one-day-ahead. In addition to the obvious practical problems of large margin requirements and high risk of ruin, there is a subtle but enormous statistical danger: the variance of net returns of such a system is huge. This is because there is high serial correlation between trades. Suppose our strategy is to hold positions for ten days. Suppose also that a trade we open today is destined to score a large win. Then if we open a new position tomorrow, it, too, will likely have a large win, because nine of their ten days overlap. The net effect is that wins and losses are exaggerated, resulting in a huge error variance (uncertainty of the estimate) of total returns. Thus, a developer may discover a system that has an enormous profit factor and conclude that wealth is right around the corner, when in fact it's just a random lucky string of highly correlated wins.

Here's another way of looking at it. Suppose you play a game in which you toss a coin 20 times. Every time it comes up heads you win a dollar, and every time it comes up tails you lose a dollar. Most of the time you will probably come out about even because wins and losses will roughly cancel each other. Now suppose you modify the game. You still win a dollar for heads and lose a dollar

for tails. But if it comes up heads, for the next toss you must use a coin that is strongly biased toward heads. If, as is very likely, that toss comes up heads, your next toss again is with a coin strongly biased towards heads. The same is true for tails: if a toss comes up tails, your next toss is with a coin strongly biased towards tails. You can see how wins and losses would not cancel well. Instead, you will experience long strings of wins or losses. In the original, fair game your final total would be the sum of a large number of small wins and losses. But in the modified game, your final total would be the sum of a few strings of extraordinarily large wins and losses. This will result in huge variance for your total gain, making it nearly impossible to decide, based on your gains, whether the coin is fair on average..

The bottom line is this:

- Your best choice will often be a *one-day-ahead* system because it is most amenable to statistical confirmation of results, although it does have the disadvantage of occasional large wins and losses which can distort performance figures and possibly training.
- Another excellent alternative is a *hit-or-miss* system with very tight limit and stop exits. This system allows multiple positions, which complicate statistical analysis and margin requirements. However, if the exit points are tight, special bootstrap tests like the tapered-block and stationary are generally applicable, and large positions are unlikely. The huge advantage of a *hit-or-miss* system is that extreme wins and losses are impossible (at least in the theoretical analysis, in which markets cannot blow through stops!). This facilitates effective training and interpretation of results.
- A *hit-or-miss* system with a distant target and/or stop still has the advantage of a light-tailed distribution, but the possibility of large positions accumulating makes statistical analysis and margin calculations difficult. This is generally not recommended.
- A system with a fixed holding period of many days is the worst possible choice. It has every disadvantage discussed above, and no real advantage. It should be used only if circumstances make it mandatory.

## Quality Does Not Equal Quantity for Predictors

Developers of model-based trading systems face a fundamental quandary. One does not want to inadvertently ignore an indicator that has powerful predictive ability. At the same time, the degree of noise inherent in market dynamics makes it likely that if you examine a sufficient number of indicators, you will stumble upon some whose pattern of noise over the historical sample happens, by pure random chance, to appear to be highly predictive of the target. Naturally, if such an indicator is included in the trading model, that deceptive pattern will soon vanish in real life, leaving the trader with a worthless indicator.

It would be unrealistic to expect the system developer to specify in advance the exact predictors (perhaps two or three) that his or her model will employ. Virtually all developers rely on intelligent automated algorithms to select effective predictors from among a set of candidates. The task of the developer is to limit as much as possible the size of the candidate list. Herein lies what many or most experts agree is the most difficult yet single most important aspect of model development. If the list of candidates is extensive, it is likely that one or more truly useful indicators will be among them. This is good. On the other hand, a large candidate list increases the likelihood that one or more of them will have a pattern of random noise that impersonates predictive power. This is bad. Thus, the developer is faced with the unenviable task of coming up with a list that is likely to contain useful predictors but unlikely to contain more than the inevitable few useless indicators. He or she will then trust an intelligent automated selection algorithm to separate the sheep from the goats, trusting that careful initial selection of candidates will minimize the possibility that the selection algorithm will make an accidental poor choice.

How does one come up with a candidate list? Here are a few thoughts in no particular order:

- As in much of life, there is no substitute for experience. A person who has designed model-based trading systems for many years in many markets will have a solid intuitive feel for which indicators work in a given environment and which do not. The new, inexperienced developer should make use of whatever experienced colleagues are available.
- Market professionals have developed many theory-based hypotheses about market behavior. For example, momentum-driven systems are based on the notion of investor under-reaction to new information; only slowly do investors become unanchored from prior beliefs. As a result, prices do not move instantaneously to a new level implied by breaking news. That delayed reaction is a gradual trend that momentum indicators can exploit.

Conversely, overbought and over-sold indicators are based on over-reaction to news, with investors taking prices beyond a level implied by the news.

- There is almost always a huge correlation between the target distance and the historical lookback period for computing useful indicators. If you are designing a one-day-ahead system, it would be unusual to find useful predictive information in an indicator that is based on more than 20 or so days of history. There are exceptions to this rule, but not many. In fact, looking back as few as five days is often ideal for a one-day-ahead system. Similarly, if you are designing a hit-or-miss system that will typically be in the market for 30-50 days, it is unlikely that an indicator based on the most recent five days will be of much value.
- When in doubt, look to price momentum indicators. Recent trend, and sudden deviation from recent trend are often the best choices. They tend to have monotonic relationships with targets, so they are particularly useful indicators for weak models such as linear regression.
- More exotic indicators based on volume, information content, and other esoteric quantities can contain valuable predictive information, but it almost always is in a highly nonlinear relationship with the target, necessitating powerful nonlinear models.
- Stationarity, at least at a reasonable visual level, is crucial for predictors. It is impossible to overemphasize the importance of this. For this reason, visual examination of a time-series plot of every predictor candidate is mandatory. If an indicator hovers around, say, 20-40 for a year or two, then drops to 5-15 for a few years, this predictor is worthless. The indicators built into *TSSB* have been designed to have decent stationarity in most markets and time periods, but it is still important to look at them before putting them to use.
- In a multiple-market situation, it is crucial that the distribution of each indicator be similar in all markets. If this is not the case, some markets will dominate training and trading while others will be ignored. This will almost always be a serious problem. As with stationarity, the indicators built into *TSSB* have been designed to have distributions that are fairly independent of the market in which they operate. Still, one should take advantage of the cross-market conformity tests built into the program in order to confirm this property. This will be discussed more later.

## Predictor and Target Selection for this Study

This section discusses the particular variables selected for this development of a stand-alone trading system.

The first choice made was to develop a one-day-ahead trading system for the longest-lived markets in the Standard and Poors 100. We will deal with the problem of extreme wins and losses by means of the RESTRAIN PREDICTED command ([here](#)), which while not perfect, is a decent way of handing the problem. The statistical tractability of one-day-ahead systems is a powerful incentive to make this choice. Hence, our target is NEXT DAY ATR RETURN 250. This target assumes that we will take a position at the open of the day following the signal and close it at the open of the following day. The size of the position will be inversely related to the *average true range* over the prior year (250 days).

The type of system developed here is different from what has been seen in any other example in this tutorial document. Here we present what is often called a *balanced* or *market-neutral* system. In order to remain relatively immune to unexpected large movement of the total market, a balanced system holds an equal number of long and short positions each day. In theory at least, if the market suddenly moves en mass, losses on one side will be compensated by gains on the other side. If the trading system can do a good job of finding a few markets that are expected to move upward (at least relative to the market as a whole), and also find a few that are expected to fall, then the system can take long positions in the former and short positions in the latter. Over the long term, a balanced system can expect to make money in any sort of market. (As an interesting side note, the supposed market neutrality of a balanced system has tempted some traders to take on absurd positions, and subsequent market unwinding in unanticipated directions has caused some monumental collapses of major equity firms. Beware.)

The variables used in this study are saved to the database and tested using the following script file. The commands shown are all basic and discussed in detail elsewhere in this tutorial, so detailed discussions are not provided here.

```
READ MARKET LIST "D:\BOOSTER\TEST\SP100_MAJOR.TXT" ;
INDEX IS OEX ;
READ MARKET HISTORIES "E:\SP100\C.TXT" ;
CLEAN RAW DATA 0.4 ;
READ VARIABLE LIST "StandAloneVars.TXT" ;
OUTLIER SCAN ;
CROSS MARKET IQ ;
WRITE DATABASE "StandAlone.dat" "StandAlone.fam" ;
```

Here is a listing of the predictor candidates chosen for this study. Note that they follow the precepts mentioned in the prior section. They are various ways of measuring trend or deviation from trend, indicators that have a long history of effective predictive power. No measures of volatility or more complex variables are included in this study because they generally require nonlinear models, which in turn require very large training sets and extended training time.

**LIN\_ATR\_5: LINEAR PER ATR 5 100**  
**LIN\_ATR\_15: LINEAR PER ATR 15 100**

**MOM\_5\_100: PRICE MOMENTUM 5 100**  
**MOM\_10\_100: PRICE MOMENTUM 10 100**

**ADX15: ADX 15**  
**ADX30: ADX 30**

**LINDEV\_10: LINEAR DEVIATION 10**  
**LINDEV\_40: LINEAR DEVIATION 40**  
**QUADDEV\_10: QUADRATIC DEVIATION 10**  
**QUADDEV\_40: QUADRATIC DEVIATION 40**

**INDDEV\_10: DEVIATION FROM INDEX FIT 10 0**  
**INDDEV\_40: DEVIATION FROM INDEX FIT 40 0**

**INT\_10: INTRADAY INTENSITY 10**  
**INT\_20: INTRADAY INTENSITY 20**

**DINT\_10: DELTA INTRADAY INTENSITY 10 10**  
**DINT\_20: DELTA INTRADAY INTENSITY 20 20**

Details concerning these predictor candidates can be found in the [Variables chapter](#). However, here is a basic summary:

*LIN\_ATR\_5* and *LIN\_ATR\_15* are price velocity (linear trend, or slope of price change) indicators at lookbacks of 5 and 15 days.

*MOM\_5\_100* and *MOM\_10\_100* are similar to the preceding two except that they are based on a simple change in price rather than a linear fit.

*ADX15* and *ADX30* are the traditional ADX trend indicators. Because they are non-directional they are likely more suitable for nonlinear models than linear. For this reason, a quadratic model will be included in the development.

*LINDEV\_10* through *QUADDEV\_40* measure the deviation of the current price from what would be predicted by a linear or quadratic fit of recent prices. On first glance it might appear that using a lookback of 40 days violates the earlier precept that one-day-ahead systems should focus on very recent history, rarely

looking back more than 20 days. However, these four indicators measure *today's* deviation, which is as recent as it gets! Here, the lookback just determines the length of time which is used to compute the prediction. This, of course, is relevant, but it is swamped out by the immediacy of using today's deviation. Also, especially for a quadratic fit, using 40 days is reasonable in order to get stable coefficients.

*INDDEV\_10* and *INDDEV\_40* compute a least-squares linear regression function for estimating the market price from the index (OEX in this example). They then subtract today's estimated value from today's actual value. This measures how much a particular market has suddenly deviated from what would have been expected based on its recent relationship to the index. As with the prior deviation indicators, even though this is a one-day-ahead system it is reasonable to try going back 40 days for the fit, because the primary information content of this indicator comes from the current price.

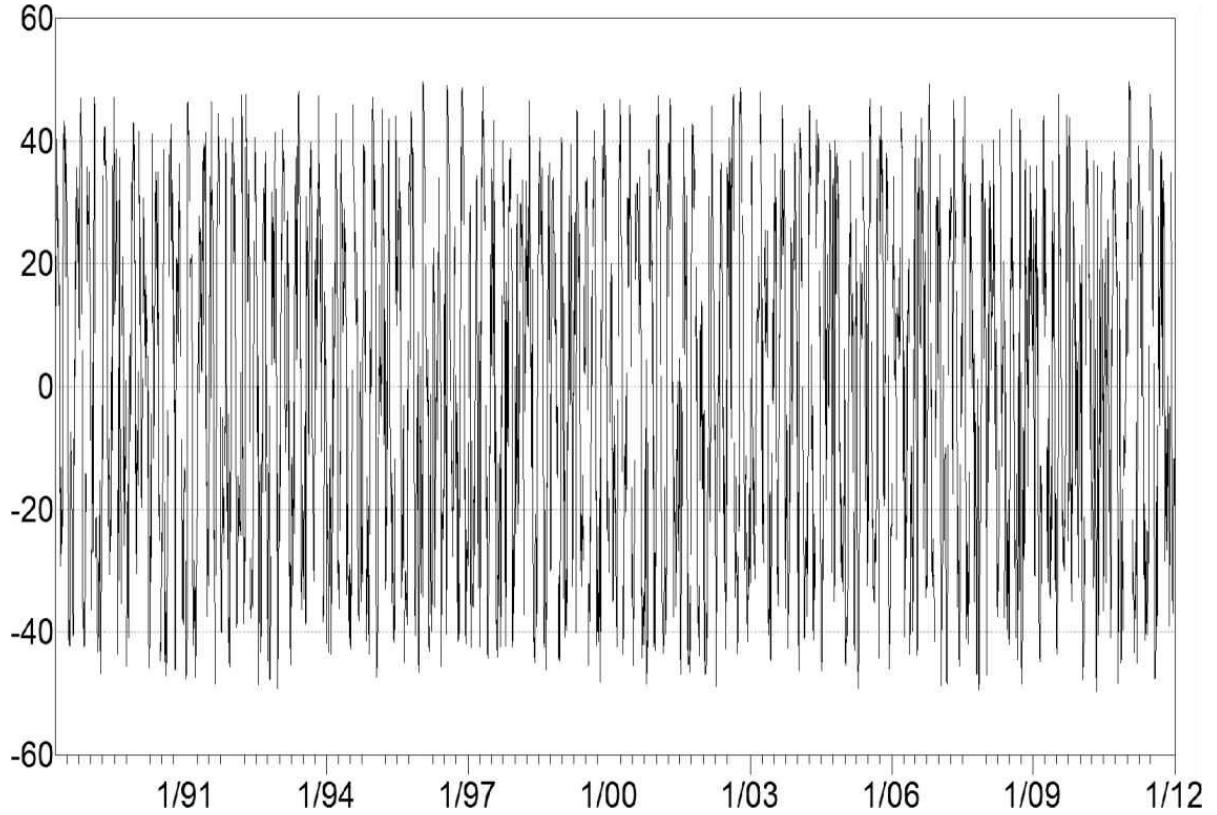
*INT\_10* and *INT\_20* are the intraday intensities at lookbacks of 10 and 20 days. These measure daily changes relative to daily true ranges.

*DINT\_10* and *DINT\_20* are the differences in intraday intensity over short lookback periods.

## Stationarity

If an indicator slowly wanders up and down, staying in one range for an extended period before moving to another range for another extended period, this indicator will usually be useless for model-based trading. This is because the predictions of a model in which this indicator is used will similarly wander, and it will be impossible for the training algorithm to find a signal threshold that produces at least reasonably uniformly distributed trades. The model would trade frequently for a long period of time, and then shut down for a long period of time. The result would be a model whose training is dependent on market behavior over particular time periods. Sometimes this is what the user wants, but usually this lack of universality is problematic. In fact, if the user want models that specialize in specific market conditions, there are better ways of doing this, such as prescreening, oracles, and triggering. These topics are all discussed elsewhere.

For an example of the sort of stationarity that is almost always ideal for an indicator, look at [Figure 57](#) below, which shows one of the indicators used in this demonstration. Imagine that you were to draw a horizontal line anywhere in that figure, but especially near the extremes of the series (around plus or minus 40). The series would cross outside that line regularly across its entire history. In nearly all applications this is exactly the sort of behavior that you want. Note that *TSSB* contains a variety of stationarity tests built into its tool set. However, most of these tests are far more strict than is needed for most applications. Simple visual confirmation of the sort shown in this plot is almost always sufficient.



**Figure 57:** An ideally stationary indicator

## The Problem of Outliers

If one or more cases have a predictor or target value that is extreme relative to the majority of cases in the training set, most models will be compromised in their training. This is because the training algorithm will go to great lengths to reduce the impact of prediction error for outlier cases, at the expense of performance for the majority of ‘normal’ cases. *TSSB* contains a model training option called *RESTRAIN PREDICTED* ([here](#)) which compresses the target as needed to reduce the effect of extreme target values. This is necessary because some targets, such as market moves over a fixed time period, are inherently at risk of extreme values; markets occasionally experience wild gyrations.

Unlike the situation with targets, the developer has no real excuse for presenting a model with training cases that have extreme values of predictors. The user has full control over the definition of indicators, and suitable compression can (and should!) be built into the definition. All of the indicators in *TSSB*’s library have been designed to avoid extreme values. Nonetheless, before embarking on a development task, the user should confirm that no outliers are present. This is done with the *OUTLIER SCAN* command described [here](#). This command was run on the indicator set used for the development task of this chapter, obtaining:

-----  
**Outlier / Entropy scan**  
-----

Variable	Min	Mkt	Max	Mkt	IQ	Rn	Ratio
<b>Entropy</b>							
LIN_ATR_5	-50.00	UNH	49.99	AAPL	25.48	3.9	0.90
LIN_ATR_15	-50.00	MRK	49.95	AVP	25.78	3.9	0.90
MOM_5_100	-49.94	AEP	49.67	AVP	23.35	4.3	0.88
MOM_10_100	-49.77	BA	48.93	AVP	23.63	4.2	0.88
ADX15	2.99	AEP	100.00	DELL	19.79	4.9	0.81
ADX30	3.40	BMY	100.00	DELL	14.34	6.7	0.71
LINDEV_10	-43.76	UNH	43.76	DELL	43.83	2.0	0.98
LINDEV_40	-49.97	MRK	49.94	MSFT	45.28	2.2	0.98
QUADDEV_10	-37.94	UNH	37.94	DELL	35.95	2.1	0.98
QUADDEV_40	-49.95	MRK	49.90	AVP	42.14	2.4	0.97
INDDEV_10	-47.92	CVS	47.93	DELL	49.10	2.0	0.99
INDDEV_40	-49.94	MRK	49.87	AVP	41.67	2.4	0.97
INT_10	-48.08	DVN	48.32	DVN	22.23	4.3	0.85
INT_20	-49.10	CMCSA	47.11	AEP	22.80	4.2	0.86
DINT_10	-49.97	DVN	49.60	DVN	29.79	3.3	0.92
DINT_20	-49.94	DVN	49.89	CMCSA	30.10	3.3	0.92
RETURN	-14.07	MRK	10.99	BAX	0.84	29.5	0.33

Sorted by ratio, worst to best

Variable	Ratio	Entropy
RETURN	29.5	0.338
ADX30	6.7	0.714
ADX15	4.9	0.815
INT_10	4.3	0.858
MOM_5_100	4.3	0.881
INT_20	4.2	0.867
MOM_10_100	4.2	0.882
LIN_ATR_5	3.9	0.907
LIN_ATR_15	3.9	0.901
DINT_10	3.3	0.926
DINT_20	3.3	0.928
INDDEV_40	2.4	0.973
QUADDEV_40	2.4	0.978
LINDEV_40	2.2	0.984
QUADDEV_10	2.1	0.985
LINDEV_10	2.0	0.986
INDDEV_10	2.0	0.993

The most important figure in this table is the ratio of the range (max minus min) divided by the interquartile range. The range (numerator) encompasses extreme values, while the interquartile range (denominator) exemplifies ‘normal’ behavior of the variable. There is no magic threshold for declaring that an indicator is good or bad, but it would not hurt to examine a histogram of any indicators that have an unusually large ratio. If the value of an indicator for one

or a very few cases are far from the mass of cases, that indicator should be considered suspicious. Here we see that ADX30 is a little suspicious, but not seriously so, and all others are fine.

## Cross-Market Compatibility

The trading system being developed in this chapter is designed to be used with the component markets in the Standard and Poors 100. Since multiple markets are involved, it is crucial that the markets behave similarly for every indicator and the target. Suppose one market has huge variance of an indicator (or indicators, or the target) used in the prediction model, much greater than the variance in other markets. Then this market will dominate training and execution. The model will be trained so as to optimize performance on this dominant market, to the near exclusion of performance in other markets. Thus, it is crucial that we always test that all indicator candidates, as well as the target, behave similarly in all markets. The indicators and targets in the built-in *TSSB* library have been designed to have good cross-market compatibility. Still, it does not hurt to verify this. The *CROSS-MARKET IQ* test was run on the dataset used in this development effort, and satisfactory results were obtained. The following is an abbreviated listing of the results:

```
Cross market IQ Range overlap test for variable LIN_ATR_5
These results are sorted worst to best
    OEX has insufficient data or statistic is undefined
    DVN    0.83250
    SO     0.86832
    CAT    0.89873
    PEP    0.90642
    XOM    0.91244
    AA     0.92112
    CPB    0.92295
    FDX    0.93014
```

The exact nature of this test is described [here](#). What we see in the partial table just shown is that for the indicator *LIN\_ATR\_5* the worst market is DVN, with an IQ range overlap of 0.83250. It is very good to see the worst being this good. Each indicator should be checked this way.

This test also produces two useful summary tables. Here is one:

```

Median IQ range overlap across markets, worst to best...
    LIN_ATR_15  0.95841
        INT_20   0.95956
        ADX30   0.96073
        DINT_20  0.96324
    LIN_ATR_5   0.96607
        INT_10   0.97242
        DINT_10  0.97477
MOM_10_100   0.97493
        RETURN  0.97560
MOM_5_100    0.97611
        ADX15   0.97978
INDDEV_40    0.98312
LINDEV_40    0.98597
QUADDEV_40   0.98663
QUADDEV_10   0.98914
LINDEV_10    0.98930
INDDEV_10    0.98947

```

The table just shown says that the worst indicator in terms of IQ range overlap is *LIN\_ATR\_15*, and even it has a median (across all markets) IQ range of 0.95841, which is fabulous cross-market compatibility.

The second summary table shows the median (across all indicators and the target) IQ range overlap for each market. The first few markets in this table are shown below. Note that OEX is excluded from the study because it is an index market, and index markets never appear in the database. This is of no consequence for this test. In this table we see that DVN is the worst market for cross-market compatibility, and even it has a median IQ range overlap of 0.95117, which is a wonderful figure.

```

Median IQ range overlap across variables, worst to best...
OEX has insufficient data or statistic is undefined
DVN   0.95117
AA    0.95153
FDX   0.96056
CPB   0.96070
CAT   0.96290
XOM   0.96357
WFC   0.96424

```

## Data Snooping: Friend or Foe?

The bane of every trading system developer is the paucity of training and testing data. In most engineering and social science fields, the availability of data is limited only by resources such as time and money. While these can be difficult hindrances, they are not impossible to overcome in most cases. But markets only go back so far in time, and their behavior changes. Behavior in a market that may have been predictable fifty years ago, if the market even existed then, is unlikely to be predictable in the same way today. And we cannot just say that the thousands of equity and commodity markets currently available provide nearly unlimited data, because these markets have (or must be assumed to have) significant correlation. Sure, you can develop a trading system for IBM, test it in T, and rejoice if it does well in that market. But your joy must be tempered by the fact that IBM and T are impacted by numerous common events and fundamental factors, so one certainly cannot consider data in a different market to be out-of-sample, a source of independent confirmation of performance.

This dilemma leads to the necessity of a difficult decision: do we use all available history, right up to the current date, give the development effort one good shot, and hope for the best if we put the system into practice? This choice makes maximum use of available history, which is good, but it requires a lot of faith in the ability of the developer and his or her development tools. Properly done, as with the data-pooling walkforward capabilities in *TSSB*, this will provide an unbiased estimate of future performance *as long as the user does not attempt multiple methodologies to develop the final trading system, and then choose the methodology having best walkforward performance*. If the developer picks the best of these, the chosen model will probably be good, but its performance figure will be optimistically biased. In other words, if you *use all available data and then experiment* to find the best system, you will not have a reliable estimate of future performance. You will just have to hope for the best when you begin trading the system with real money.

The other option is to hold out a recent chunk of history, such as the most recent year, and devote enormous research power to producing an excellent system with the data prior to the segment excluded from study. This has the advantage of giving the developer complete freedom to study, experiment, and tweak performance. In particular, if the developer is inexperienced or is using new, unstudied indicators or targets, this method may be required. After a highly effective trading system is developed with the older, excluded data, it can then be tested on the most recent, virgin data. This will provide an unbiased estimate of future performance. Of course, the price paid for this delightful accomplishment is that the experimentation phase, in which the developer laboriously constructed a beautiful trading system, did not have access to the

most recent market data. This is a high price to pay, but this is the route taken in this tutorial, because it allows demonstration of some basic experimental techniques.

# Checking Stability with Subsampling

While not strictly necessary, it's always interesting to use subsampling (or resampling for models other than a GRNN) to check on the stability of the relationship between the predictor candidates and the target. There are three essential core components of the development of a model-based trading system:

- Choose one or several predictors
- Train a model to predict a target using these predictors
- Find a threshold for making trade decisions based on the model's predictions

Ideally, these components should be stable. In other words, changing some aspect of the training and test data should cause little change in the choice of the predictors, the nature of the models, and the location of the threshold.

The obvious way to examine stability is with walkforward testing, and certainly every fold of a walkforward run should be studied. In particular, one should pay close attention to the out-of-sample performance in each fold. If it varies widely from fold to fold, especially if the folds are large (a year) and the change is massive (a bad loss amidst many wins), one should be suspicious.

However, some variation in walkforward folds is unavoidable. We may be testing a long-only system, and if a huge bear market comes along it may be that the only reasonable thing we can expect from our system is that it avoid being in the market during the worst downward moves. We should be happy if our system loses a lot less money than a buy-and-hold strategy would lose. Thus, time-slicing for stability checks has limited utility.

Examination of stability can be approached from a different direction which avoids interference from market trends. Instead of slicing by time, we take random samples from the entire extent of the dataset. This can be done with the SUBSAMPLE or RESAMPLE model option. The SUBSAMPLE option is usually the better choice because the RESAMPLE option is incompatible with GRNN models due to duplication, and also because subsampling accomplishes essentially the same thing as resampling, but with a smaller training set.

One can subsample the entire dataset and use the TRAIN command, but a key measure of the quality of a model is how well its predictions hold up out of sample. This lets us distinguish a model that is so powerful (overfitted) that it learns noise, from a model that has just enough power to learn the true patterns in the data. Hence, the ideal testing method is to embed subsampling into a

walkforward run and examine the summary. With this in mind, here is a script file that trains and tests five subsampled models:

```
RETAIN YEARS 1900 THROUGH 2010 ;
READ DATABASE "StandAlone.dat" "StandAlone.fam" ;

MODEL LINMOD1 IS LINREG [
  INPUT = [ LIN_ATR_5 - DINT_20 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  SUBSAMPLE 60 PERCENT
  MAX STEPWISE = 2
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
  SHOW SELECTION COUNT
] ;
```

... Models 2, 3, and 4

```
MODEL LINMOD5 IS LINREG [
  INPUT = [ LIN_ATR_5 - DINT_20 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  SUBSAMPLE 60 PERCENT
  MAX STEPWISE = 2
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
  SHOW SELECTION COUNT
] ;
```

```
WALK FORWARD BY YEAR 5 1999 ;
```

For economy, only two of the five models are shown above. They are all identical. Each uses the full set of predictor candidates and predicts the restrained target. Other model parameters mimic the parameters that will be used in the final prediction models:

- A maximum of two predictors are used.
- STEPWISE RETENTION = 10 is reasonable, large enough to guarantee thorough testing of various indicator combinations, yet not so large as to require excessive computation time.
- Since this is to be a balanced system, we must use the FRACTILE THRESHOLD option and a BALANCED criterion (ten percent here).

- The MIN CRITERION FRACTION should be set to match the BALANCED fraction (0.1 is 10 percent), even though this parameter plays no practical role in training a balanced system at this time. It may play a role in a future release of *TSSB*, so it's a good habit to set it to match.
- The SHOW SELECTION COUNT option lets us see which parameters were most and least popular for each model.

The key here is that all five models include the SUBSAMPLE 60 PERCENT option. This decrees that a randomly selected 60 percent of the dataset will be used to train the model. If the model is able to find little or no *true* relationship between the predictor candidates and the target, the model will just be learning random noise, and each of the five models will learn a different set of noise points. In this case, the predictors selected will be more or less random, and out-of-sample performance will vary widely, since the out-of-sample predictions will be in a random relationship with the target. But if the combination of the predictor candidates and the linear model is able to find a true, repeatable pattern, this pattern will appear in the randomly selected training sets of all five models. This will result in at least approximately the same predictors being selected for all five models, and at least similar out-of-sample performance. Here is a table showing the out-of-sample profit factor (top row) of each of the five models, and the predictors selected, ordered from most to least popular:

1.081	1.081	1.079	1.082	1.072
INT_10	INT_10	INT_10	INT_10	INT_10
LIN_ATR_5	LIN_ATR_5	LIN_ATR_5	LIN_ATR_5	LIN_ATR_5
MOM_5_100	MOM_5_100	MOM_5_100	MOM_5_100	MOM_5_100
MOM_10_100	MOM_10_100	MOM_10_100	MOM_10_100	MOM_10_100
DINT_10	DINT_10	DINT_10	DINT_10	DINT_10
INT_20	LIN_ATR_15	LIN_ATR_15	LIN_ATR_15	LIN_ATR_15
INDDEV_10	LINDEV_40	INT_20	INT_20	INT_20
LIN_ATR_15	INDDEV_10	INDDEV_10	QUADDEV_40	QUADDEV_40
LINDEV_40	INT_20	INDDEV_40	INDDEV_10	INDDEV_10
QUADDEV_40	INDDEV_40	LINDEV_10	INDDEV_40	INDDEV_40
QUADDEV_10	QUADDEV_10	QUADDEV_40	LINDEV_40	LINDEV_40
LINDEV_10	QUADDEV_40	LINDEV_40	LINDEV_10	LINDEV_10
INDDEV_40	LINDEV_10	DINT_20	QUADDEV_10	QUADDEV_10
DINT_20	DINT_20	QUADDEV_10	DINT_20	DINT_20
ADX30	ADX30	ADX30	ADX30	ADX30
ADX15	ADX15	ADX15	ADX15	ADX15

Note the remarkable uniformity among the five models, especially for the most and least popular indicators. Of course, with a 60 percent overlap, the models do have many training cases in common. Still, this degree of agreement among the models is encouraging. We don't yet know how much practical performance

we will be able to get out of this predictive system, but the news is good so far.

Another interesting observation is that the two ADX indicators are at the bottom of the list for all five models. Because ADX is nondirectional (it indicates trend strength, not direction), one would expect ADX15 and ADX30 to have no predictive power whatsoever with a linear model, and this is exactly what we are seeing. But if the models were learning only random noise instead of true patterns, ADX would have just as much ‘power’ as directional variables. Hence ADX15 and ADX30 could appear anywhere in the lists. Yet all five models are ranking all of the directional variables above ADX. Nice!

As a final note, this test was re-run with a subsample rate of just 20 percent. Obviously, this produces tremendous diversity in the training sets. Nonetheless, for the five models, out-of-sample profit factor ranged from 1.069 to 1.088, a fairly narrow range. Moreover, the most, second-most, and third-most popular variables were INT10, LIN\_ATR\_5, and MOM\_5\_100 respectively for all five models! This is remarkable conformity and is very encouraging.

# How Long Does the Model Hold Up?

Before one gets too far along in the development process, it is useful to see how far into the future a trained model holds onto its predictive power. Whenever computer time allows, this should also be the final step in development, although resource limitations often prevent this final step. But it should nearly always be feasible to test at least some of the predictive components early in the process. Information about the usable life of a model before it needs retraining is obviously important to deployment. But it also plays a key role in the development process. If one finds that a model needs frequent retraining, one should account for that in any experimentation, or else revise the plan.

*TSSB* contains a facility to get a good idea about how long a model retains its predictive power. This is done by running two or three different walkforward fold (out-of-sample) sizes. At a minimum, fold sizes of a year and a month should be tested. If at all possible, a fold size of a day should also be used to provide an optimal-performance baseline. Here is a script file to test the linear model and predictor candidate set which form the main foundation of the predictive system:

```
RETAIN YEARS 1900 THROUGH 2010 ;
READ DATABASE "StandAlone.dat" "StandAlone.fam" ;

MODEL LINMOD IS LINREG [
  INPUT = [ LIN_ATR_5 - DINT_20 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 2
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
] ;

WALK FORWARD BY YEAR 5 1999 ;
WALK FORWARD BY MONTH 60 199901 ;
WALK FORWARD BY DAY 1250 19990101 ;
```

The testing walks forward always using about five years of training data. This is 60 months, or 1250 days. Out-of-sample profit factors were as follows:

By year: 1.070  
By month: 1.085  
By day: 1.085

The first thing many people will note is that these profit factors are not terribly

impressive. However, one must remember that this is a very special trading system: it's balanced. Every day it *must* take a long position in ten percent of its markets, and it must also take a short position in another ten percent of its markets. This is a double-edged sword. It has the disadvantage of often requiring the trader to open long positions when the prediction is for a strong downward move, and similarly taking short positions when the prediction is for an upward move. Systems that only need to take a position in the direction of the predicted move have an obvious advantage! But the payoff is substantial, because a balanced system is far more immune to mass market moves than unbalanced systems. Thus, one can safely (at least most of the time!) use leverage to take on far larger positions than one can with an unbalanced system. This multiplies the potential profits without similarly multiplying the potential losses (except in exceptional circumstances).

The important information in these three numbers is that the model appears to hold up well for at least a month, but it loses considerable power when asked to survive for a year without retraining. It would be beneficial to conduct all further experiments using a monthly walkforward. Unfortunately, computer resources for this demonstration (as is often the case in real life) preclude monthly retraining.

## Finding Models for a Committee

It almost goes without saying that modern model-based trading systems use a committee of some sort for making the final trading decisions. The performance of a committee of models is almost always superior to that of a single model, often extremely so. We already discussed [here](#) some general guidelines for committee formation. This example will use a few simple linear models, always a good choice, along with a single nonlinear model, to be committee members.

If one is using the one-shot approach discussed in the prior section, then the EXCLUSION GROUP option is a good way to find committee members. However, in an experimentation environment, the FIND GROUPS command used on the entire training period may be preferable. The disadvantage of this approach is the walkforward results in the experimentation phase will be optimistically biased due to using all of the data to find the best component models. But this is not a disaster, because the final test involving the excluded data will still be unbiased, and it's nice to have the stability of knowing in advance exactly which indicators will be used in all component models. With the EXCLUSION GROUP option, different indicators will generally be used in different folds, and none of these indicator sets will benefit from having access to all training data simultaneously. Of course, this is a judgement call with no hard answer; the FIND GROUPS method seemed better for this demonstration. Thus, we use the following script file:

```
RETAIN YEARS 1900 THROUGH 2010 ;
READ DATABASE "StandAlone.dat" "StandAlone.fam" ;

FIND GROUPS [
    INPUT = [ LIN_ATR_5 - DINT_20 ]
    OUTPUT = RETURN
    MAX STEPWISE = 2
    STEPWISE RETENTION = 10
    MAX GROUPS = 3
    GROUP RSQUARE CUTOFF = 0.8
    FRACTILE THRESHOLD
    CRITERION = BALANCED_10
    MIN CRITERION FRACTION = 0.1
    RESTRAIN PREDICTED
] ;
```

Notice that this script file begins by retaining only years through 2010. The complete dataset runs through the end of 2011, which will be the test (out-of-sample) year. The data begins in 1988, but it's also safe to begin the retained period at any year prior to this, such as 1900.

The FIND GROUPS command uses all candidate indicators, setting a limit of

two predictors in each model. A STEPWISE RETENTION of 10 is reasonable, large enough to guarantee thorough testing of various indicator combinations, yet not so large as to require excessive computation time. The GROUP RSQUARE CUTOFF of 0.8 is also reasonable.

Here are the three indicator groups found. We will use these three linear models, along with a quadratic model for token nonlinearity, in a committee.

```
-----> Group 1 <-----  
Regression coefficients:  
    -0.000747  LIN_ATR_5  
    -0.000872  INT_10  
    0.023183  CONSTANT  
  
-----> Group 2 <-----  
Regression coefficients:  
    -0.000732  INT_20  
    -0.000826  DINT_10  
    0.022511  CONSTANT  
  
-----> Group 3 <-----  
Regression coefficients:  
    -0.000981  MOM_10_100  
    -0.000212  LINDEV_10  
    0.023257  CONSTANT
```

# The Trading System

Development of the actual trading system is where the majority of experimentation takes place. Typically, one would try several model types, several sets of predictors, and several committee types. Because in this demonstration we have held out the entire year 2011 for testing, we are free to experiment and tweak performance as much as we like. For reasons of economy this tutorial will limit experimentation to varying only the committee type, but in an actual application other variations should be attempted.

As has been mentioned earlier, it is nearly always best to use powerful predictors in a weak model, as opposed to weak predictors in a powerful model. For this reason, three linear models will form the foundation of this trading system. However, one should usually consider the possibility that some nonlinearity is present in the predictor set, and hence include at least one nonlinear model in the design. A quadratic model generally has an excellent combination of training speed and nonlinear predictive power, so that type is chosen here. Here is the entire trading system script file:

```
RETAIN YEARS 1900 THROUGH 2010 ;
READ DATABASE "StandAlone.dat" "StandAlone.fam" ;

MODEL LIN1 IS LINREG [
  INPUT = [ LIN_ATR_5 INT_10 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 0
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
] ;

MODEL LIN2 IS LINREG [
  INPUT = [ INT_20 DINT_10 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 0
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
] ;
```

```
MODEL LIN3 IS LINREG [
  INPUT = [ MOM_10_100 LINDEV_10 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 0
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
] ;
```

```
MODEL QUAD1 IS QUADRATIC [
  INPUT = [ LIN_ATR_5 - DINT_20 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 2
  STEPWISE RETENTION = 10
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
] ;
```

```
COMMITTEE COMM1 IS CONSTRAINED [
  INPUT = [ LIN1 LIN2 LIN3 QUAD1 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 4
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
  MCP TEST = 1000
] ;
```

```
COMMITTEE COMM2 IS CONSTRAINED [
  INPUT = [ LIN1 LIN2 LIN3 QUAD1 ]
  OUTPUT = RETURN
  RESTRAIN PREDICTED
  MAX STEPWISE = 0
  FRACTILE THRESHOLD
  CRITERION = BALANCED_10
  MIN CRITERION FRACTION = 0.1
  MCP TEST = 1000
] ;
```

```

COMMITTEE COMM3 IS AVERAGE [
    INPUT = [ LIN1 LIN2 LIN3 QUAD1 ]
    OUTPUT = RETURN
    RESTRAIN PREDICTED
    MAX STEPWISE = 0
    FRACTILE THRESHOLD
    CRITERION = BALANCED_10
    MIN CRITERION FRACTION = 0.1
    MCP TEST = 1000
] ;

```

**WALK FORWARD BY YEAR 5 1999 ;**

The first three models are the three linear models discovered with the FIND GROUPS command discussed earlier. In order to accommodate nonlinearity, a QUADRATIC model is also included. Finally, three committees are employed. The first is a CONSTRAINED committee which uses stepwise selection to choose the models that it will use. The second is also CONSTRAINED, but all four models are forced to be used. Finally, an AVERAGE committee is used to simply average the predictions of the four models. These are all tested with yearly walkforward and a five-year training period. Pooled out-of-sample profit factors are as follows:

LIN1	1.093
LIN2	1.070
LIN3	1.082
QUAD1	1.072
COMM1	1.077
COMM2	1.091
COMM3	1.092

We now have a bit of a quandary: the first linear model is the best performer, though only by a trivial amount. If the difference were larger, it would be tempting to base the trading system entirely on this one model. However, in this nearly tied situation, it is scary to base trading on just two predictors. Thus, we choose to base trade decisions on COMM3, the average of the four models' predictions. Keep in mind that these performance figures are subject to random variation, so there is no guarantee that the observed rank ordering will exactly correspond to the rank ordering of true quality. For this reason, it seems good to favor the known general superiority of committees over single models.

The ordering of committee performance is no great surprise. COMM1 is the only one of the three that uses stepwise selection to choose its component models. This additional power encourages overfitting. COMM2 has adjustable coefficients, which makes it more powerful than COMM3 which is a simple average. So we see that out-of-sample performance is inversely related to committee power.

## The Final Test

After the developer has performed numerous experiments to learn as much as possible about the predictors and target, and has tweaked performance to perfection, it is time to take a deep breath and test the complete trading system on the data that was withheld during the experimentation phase. The script file for this is identical to that used in the development and shown in the prior section, with one exception. It does a single walkforward year, 2011:

```
WALK FORWARD BY YEAR 5 2011 ;
```

Results for this test year are as follows:

LIN1	1.043
LIN2	1.045
LIN3	1.015
QUAD1	1.032
COMM1	1.041
COMM2	1.019
COMM3	1.040

This is a considerable disappointment. At least all models and committees made money in the test year, but it was not much. Also, it turns out that the decision to use COMM3 rather than LIN1, even though LIN1 was slightly superior, may have been a bad choice, because LIN1 is also better than COMM3 in this test year.

This brings up a subtle but important issue. We already know that LIN1 was superior to COMM3 in the development phase, and now we see that it is also superior in the test year. So when we begin to trade the system for real money (assuming that we choose to do so), should we trade LIN1 instead of COMM3? In fact, LIN2 was the best of all in the test year. Maybe we should choose to trade it.

There is no simple answer to this dilemma, but there are some vital issues to consider. Perhaps the most important issue is that we chose *in advance* to trade COMM3, and we just tested it on virgin data. The implication is that its performance of 1.040 is an unbiased estimate of its expected future performance. If we change our mind after seeing that LIN1 (or LIN2) is superior, we can no longer say that the observed performance of LIN1, 1.043, or that of LIN2, 1.045, is an unbiased estimate of future performance. These superior figures have been inflated by selection bias, the result of the real possibility that random good luck intervened to subvert the natural ordering of performances. By definition, good luck will not continue for long, if at all.

There is yet another aspect of this issue. Suppose we had not done any experimentation. Instead, suppose we had taken the (nearly) one-shot approach described early in this chapter. But also suppose that instead of just one shot, we had tried several models and walked them forward on all available data, and then chosen the best performer for real-money trading. Again, the performance of the best model would be optimistically inflated by selection bias. Nonetheless, under normal conditions it would be in our best interest to trade the best performer, because it is most likely to be the truly best. We should just understand that the performance we will obtain in the future will probably not be as good as what we observed, due to selection bias.

Does this mean that in this case we should trade LIN1 (or perhaps LIN2) instead of COMM3? The answer is no, because we are in a different situation here. In our current situation, we are not looking at just the relative performances in the test year. We also have experimentation-phase walkforward results pooled from 1999 through 2010, and those results showed COMM3 to be a good performer, practically tied with LIN1 and much superior to LIN2 (which was the grand *loser* then). We would be foolish to discount those results, especially since they are based on 12 years of pooled trade results, and the figures obtained with this final test are for a single year.

Does this leave you craving more rigorous decision-making rules? Indeed it should, because there are few hard-and-fast rules. Sometimes you have to fly by the seat of your pants and weigh alternatives without the benefit of rigorous theory. But we'll leave you with two definitive rules that should always be given strong consideration:

- 1) The more years that go into a study, the more reliable are the results. Rank orderings produced by 12 years of pooled data are more stable than those produced by a single test year.
- 2) The moment you examine competing models and choose the best you have introduced selection bias. You have favored not only the truly best model, but also the luckiest model. Thus, the performance of the chosen best will, on average, be optimistic.

If you keep these two rules in mind you should be able to navigate the development waters with reasonable safety.

# Trade Simulation and Portfolios

You may follow a TRAIN, WALK FORWARD, or CROSS VALIDATE command with the PRESERVE PREDICTIONS command, and follow this with a TRADE SIMULATOR command. This provides more extensive performance statistics than are provided by default, and it also allows you to execute slightly more sophisticated trading rules than those inherent in default operation. (Recall that the default trading rule in TSSB is based on a simple threshold: if the model's prediction exceeds an upper (long) or lower (short) threshold a new position is taken and the trade's realized return is either the value of the target variable or a designated profit variable if the target can not be interpreted as a gain or loss.)

The syntax of this command is as follows:

```
TRADE SIMULATOR Model Detail TradeRule PerformanceMeasure
```

*Model* names a model that is in the script file. Trades for this model will be simulated.

The *Detail* may be GLOBAL or BY MARKET. This option, although always required, is relevant only if multiple markets are present. If you choose GLOBAL, only a summary with all markets pooled (aggregate performance) will be printed. If you choose BY MARKET, not only will the global summary be printed, but results for each individual market will also be printed. Note that this will produce a voluminous printout if a large number of markets are present. This option must be GLOBAL if an equity curve will be written or displayed.

The *TradeRule* must be one of the following:

```
LONG ( Enter Maintain )
LONG ( TRAINED Maintain )
SHORT ( Enter Maintain )
SHORT ( TRAINED Maintain )
DUAL ( LongEnter LongMaintain ShortEnter ShortMaintain )
DUAL ( TRAINED LongMaintain TRAINED ShortMaintain )
```

The rules that begin with LONG will take only long positions. The position will be opened when the model's prediction equals or exceeds the *Enter* threshold value, and it will be closed when the prediction drops below the *Maintain* threshold value. In other words, a position is entered if the prediction is equal to or more extreme than the *Enter* value and it is held so long as the forecast value remains at or beyond the *Maintain* value. Obviously, the *Maintain* value must

not exceed the *Enter* value. Negative values are permitted, in which case the long position will be maintained even though the prediction is negative, as long as the prediction equals or exceeds the *Maintain* value.

If instead of specifying a numeric *Enter* value, you specify the keyword TRAINED, the position will be opened when the predicted value equals or exceeds the optimal in-sample value determined during training. This is legal for cross validation and walkforward as well as ordinary training, because the program keeps track of the optimal value for each test fold.

If this TRAINED option is used, the *Maintain* specification is not an actual prediction threshold. Rather, it is a multiplier for the optimal open value found during training. If it is specified as its legal maximum, 1.0, the maintain threshold will equal the open threshold. Specifying it as, say, 0.5 means that the long position will be held as long as the prediction is at least half of the open threshold. Negative values are legal.

Note that most of the time, the optimal trained prediction threshold for a long position will be positive. In the unusual but occasional instance that the threshold is negative, the specified *Maintain* value will be ignored, and the maintain threshold will be set equal to the open threshold.

The rules that begin with SHORT will take only short positions. The parameters associated with these two commands are similar to those for LONG commands, with rules appropriately flipped.

The rules that begin with DUAL take both long and short positions. *Open* and *Maintain* thresholds must be separately specified for the long and short trades.

The *PerformanceMeasure* defines the statistic that will be used to measure performance. In all cases, we assume that Day 0 has closed and a decision has been rendered. The position is taken at the open of Day 1 and closed at the open of a subsequent day, at which point the return for the transaction is logged. This option may be the following:

PERCENT - This is the percent change in the market.

ATR(Distance) - This is the change in the market expressed as a multiple of ATR measured back in time over the specified distance, which must be at least 2. This option is recommended if multiple markets are present, because it provides better cross-market conformity.

POINTS - This is the actual point change in the market.

Note that the TRADE SIMULATOR option obviously requires that market histories be present! If you are using a READ DATABASE command to read precomputed variables, you must *precede* the READ DATABASE command with READ MARKET LIST and READ MARKET HISTORIES commands.

## Writing Equity Curves

After the trade simulator has been run, you may write the equity curve to a comma-delimited text file readable by Excel and most standard statistics packages. The units written are the units selected in the trade simulator: *Percent*, *ATR*, or *Points*.

To write the equity to a comma-delimited text file, use the following command:

```
WRITE EQUITY "FileName" ;
```

As with all file names in *TSSB*, the name must contain only letters, numbers, and the underscore character (\_). The full path name may be specified. If no path is supplied, the file will be written to the current directory.

The file will begin with the following header:

**Date, Long, Short, Change, Cumulative**

These quantities are defined as follows:

**Date** - The date as YYYYMMDD.

**Long** - The number of long positions open on the specified date (for multiple markets).

**Short** - The number of short positions open on the specified date (for multiple markets).

**Change** - The change in equity as of the open of the specified date relative to the open of the prior day's date.

**Cumulative** - The cumulative change in equity as of the open of the specified date.

If the data is intraday, the time as HHMM or HHMMSS will follow the date.

If there is only one market, it will obviously be impossible (except for extremely rare pathological situations involving crossed thresholds) to simultaneously have a long and short position. However, if several markets are present, some markets may be long on a given day while others are short.

It is vital to understand how the date, position, and change are related. The idea is that the market closes on what we will call *Day 0*. At this time we have all information needed to make a decision as to the position to take at the next

trading opportunity (bar or day). This is the open of the next day (or bar for intraday data). So a position will be taken on *Day 1* and this position will appear in the equity file for the date of *Day 1*. Equity will not change on that day, even though one might mark equity to the market throughout the day. At some point, after the close of the market on some day, the decision will be made to close the position at the next opportunity. This will be the open of the market on the day following the decision. Here is an actual example of an equity file:

Date	Long	Short	Change	Cumulative
20000104	0	1	0.00000	0.00000
20000105	1	0	3.45964	3.45964
20000106	0	1	0.02766	3.48730
20000107	0	1	-0.45047	3.03683

As of the close of the day *prior* to 20000104 the system decided to take a short position. Thus, it takes this position at the open of 20000104. This short position is recorded in the file. The equity does not change because we need a full 24 hours to track the market change. Thus, the equity for 20000104 is still zero.

At the close of 20000104 the system decides it wants to turn long. At the open of 20000105 it closes the short position and opens a long position. This new position is recorded in the file. It made a 24-hour profit of 3.45964 as a result of closing the short position. The new long position does not enter into this figure at all yet.

At the close of 20000105 the system decides to go short again. At the open of 20000106 it closes the long position and opens a short position. This change is recorded for this date. The profit for the 24-hour long position is 0.02766, making a cumulative profit for the original short position and then the long position of 3.48730.

At the close of 20000106 it decides to continue holding the short position. This is reflected in the position field of 20000107. The short position that it held from the open of 20000106 to the open of 20000107 resulted in a loss of -0.45047, which drops the cumulative equity as of the open of 20000107 to 3.03683.

In summary, the position shown for each record is the position held on the specified date, including the overnight session that follows the specified date. The equity and cumulative equity for a specified date is the value as of the open of that date.

We conclude with a two-part example of computing equity and writing it to a text file. It is certainly legal to perform all operations in a single script file.

However, it is more efficient to split the operations into a preliminary file that computes and saves all predictors and targets, and a subsequent script file that reads this dataset, fits the model, and writes the equity. Here is the initial file, with the variable definition file shown after:

```
READ MARKET LIST "D:\BOOSTER\TEST\OEX_VIX.TXT" ;
INDEX IS VIX ;
READ MARKET HISTORIES "E:\SP100\VIX.TXT" ;
READ VARIABLE LIST "D:\BOOSTER\TEST\VIX_STUDY_VARS.TXT" ;

TRANSFORM VIX10_NODIFF IS ARMA [
SERIES = @CLOSE:VIX
ARMA WINDOW = 50
ARMA OUTPUT = STANDARDIZED SHOCK
ARMA AR = 1
ARMA MA = 0
] ;

WRITE DATABASE "ARMA_EQUIITY.DAT" "ARMA_EQUIITY.FAM" ;
```

Here is the variable definition file, VIX\_STUDY\_VARS.TXT:

```
PURIF_DAVE2: PURIFIED INDEX 2 300 2 3 6 11 22 44 65 130 260
RETURN: NEXT DAY ATR RETURN 250
HITMISS16: HIT OR MISS 16.0 16.0 1 0
```

This variable definition file computes a purified index, which we may choose to use in a subsequent study. It also computes two types of return.

There is an extremely important and subtle aspect to this variable definition file. Whenever possible, *TSSB* assumes that all markets in the market list are to be traded. However, in some situations it is clear that an index is not to be traded. This is the case when the MINUS INDEX modifier is used for an indicator, or when a PURIFIED INDEX is computed. Thus, in this example the VIX market will be excluded from trading, which is good. (In the unusual situation that you want to trade VIX along with OEX you could copy the VIX market to another market file and include this copied file in the market list.) In this example, if you knew in advance that you would have no use for a purified index indicator and did not wish to compute it, you should include only OEX in the market list. If you also included VIX in the market list, the program would assume that VIX is just another tradeable market and do so!

Once the database of indicator and target candidates is written by this preliminary script file, this database can be used in a model, and the model's equity curve can be written. The script file to do this is as follows:

```

READ MARKET LIST "D:\BOOSTER\TEST\SYM_OEX.TXT" ;
READ MARKET HISTORIES "E:\SP100\OEX.TXT" ;
READ DATABASE "ARMA_EQUIITY.DAT" "ARMA_EQUIITY.FAM" ;

MODEL NODIFF IS LINREG [
  INPUT = [ VIX10_NODIFF ]
  OUTPUT = HITMISS16
  MAX STEPWISE = 1
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

WALK FORWARD BY YEAR 5 2000 ;
PRESERVE PREDICTIONS ;
TRADE SIMULATOR NODIFF GLOBAL DUAL (TRAINED 1.0 TRAINED
  1.0) POINTS ;
WRITE EQUITY "EQTY.TXT" ;

```

The first thing to notice is that in addition to reading the database created by the prior script file, we also need to read the OEX market. If we were just training and testing models this would not be necessary. But since we are invoking the TRADE SIMULATOR, it needs market history. The SYM\_OEX.TXT file names just one market, OEX. We do not want to get VIX into the mix at this point. It served its purpose in computing indicators, and now we are done with it. We do not want to trade it!

The model is walked forward, and the PRESERVE PREDICTIONS command preserves its out-of-sample predictions in the database as just another variable. The TRADE SIMULATOR is invoked for this model. The GLOBAL option is used even though there is just one market, because this is required for equity computations. The DUAL option specifies that we will count both long and short positions. The TRAINED option for both long and short positions says that we will use the trained threshold for each fold, and 1.0 is the factor for holding a position. This implies that we will open and close positions according to the prediction relative to the trained threshold. All equities will be in terms of raw point values.

Finally, we write the equity file.

## Performance Measures

For individual markets (printed if the BY MARKET option is specified), the first line for each market shows the total number of bars in the history, the number of those in which a long or short position is taken, and the percent of bars spent long or short. For the global summary, these numbers are combined across all markets. In addition, the global summary prints the unpooled but combined profit factor. In other words, this profit factor considers every trade in every market as a separate event rather than pooling them at the end of every time slice (bar).

The next line shows the total return across the entire historic time period, measured in whatever unit (PERCENT or ATR) was requested.

The next line shows the profit factor. For the global summary in a multiple-market situation, the trades are pooled across all markets at the end of each time slice (bar). This will generally produce a different profit factor than would be obtained by treating each market's trades as individuals. This latter quantity is printed on the first line, as mentioned above.

The last line shows the maximum drawdown over the historical time period, and the number of bars it covered. This drawdown is measured in the same units (PERCENT or ATR) as the other figures.

Note that a seeming anomaly can occur in some conditions. Intuitively, it would seem as if the long and short counts in a single DUAL run should equal the corresponding long and short counts in individual LONG and SHORT runs. This usually happens. However, suppose the following two conditions are true:

- 1) The enter threshold for a short position is positive. (This may be explicitly specified, or happen through internal automation if the TRAINED option is used.)
- 2) The long maintain threshold is less than the short enter threshold

Then the following sequence of actions may occur:

- 1) A prediction exceeds the long entry threshold, so a long position is taken.
- 2) The subsequent prediction exceeds the long maintain threshold but is less than the short enter threshold. Thus, under the LONG scenario, the long position would be maintained. But under the DUAL scenario, the short entry will take precedence over the long maintain status, resulting in a reversal of position. This is an unavoidable philosophical problem.

Several work-arounds are possible, but they would cause more problems than they solve. This is fairly rare, and happens only under conditions that could be considered pathological.

## Portfolios (File-Based Version)

A portfolio is a set of trading systems that are combined into a single trading system that often has better net performance (better risk/reward ratio) than any individual trading system in the portfolio. *TSSB* contains two versions of portfolios: *File Type* and *Integrated*. They do almost the same thing (find and test optimal combinations of trading systems), but they do so in very different ways. The most important difference is that the *File Portfolio* reads equity files in order to obtain its component trading systems. As a result, *File Portfolios* can be used to process trading systems produced by other programs, such as *TradeStation™*. On the other hand, in order to build Portfolios based on *TSSB* models, the user must execute the *Trade Simulator* and write equity files. This is not only a nuisance, but it limits some capabilities. Therefore, *TSSB* also implements *Integrated Portfolios*. These are much more convenient as they can directly process trading results produced by *Models*, *Committees*, and *Oracles*. This section describes *File Portfolios*. *Integrated Portfolios*, which will be the Portfolio of choice for the vast majority of users, are presented in a separate chapter of their own ([here](#)).

After the trade simulator has been run for two or more models, and corresponding equity files have been written, the user may be interested in computing net performance figures for a combination of equity curves. This can be done with the FILE PORTFOLIO command, which merges two or more files of equity curves to produce a single combined equity curve. The resulting curve can then be displayed or written to a file exactly like the curve produced by a TRADE SIMULATOR command.

Note that the FILE PORTFOLIO command does not need to be in the same script file as the TRADE SIMULATOR commands that produced the equity curves. This is because all necessary information is contained in the equity curve files that the FILE PORTFOLIO command reads. The user can create a short, simple script file that contains nothing but one or more FILE PORTFOLIO commands, thus separating the model training/testing operations from the study of possible portfolios. In fact, the equity curve files read by this command need not have been created from *TSSB*, as long as they are in the file format described in the section on writing equity curves, [here](#).

The syntax for the FILE PORTFOLIO command is as follows:

```
FILE PORTFOLIO  PortfolioName  [  PortfolioSpecs  ] ;
```

The *PortfolioName* may be up to 15 characters long, and it may not contain spaces or any special characters other than the underscore (\_). At this time the

name has no use, but it has been included as a mandatory part of the syntax due to the fact that planned enhancements to the program will require that portfolios be named.

There is only one *PortfolioSpec* that is required:

```
EQUITY FILE = [ File1 File2 ... ]
```

This specification names two or more equity files that will go into the portfolio. These files must be in the file format described in the section on writing equity curves, [here](#). Each file name must be enclosed in double quotes ("...") and may not contain spaces or special characters other than the underscore (\_).

Here is an example of a simple PORTFOLIO command:

```
PORTFOLIO DEMO_PORT [
    EQUITY FILE = [ "EQTY1.TXT" "EQTY2.TXT" "EQTY3.TXT" ]
] ;
```

The example command just shown defines a portfolio called *DEMO\_PORT* that merges three equity curve files. There are several optional specifications available. These are:

***EQUALIZE*** - By default, all returns, drawdowns, et cetera, are computed by summing the equity curves. Thus, the net return of a portfolio is by default the sum of the returns of all of its components. Naturally, this inflates returns and drawdowns relative to those of individual components, making comparisons difficult. By specifying the *EQUALIZE* option, the portfolio performance figures are based on the mean of the components instead of their sum, which can make visual inspection of performance tables easier. Of course, this is only a scaling issue. Returns and drawdowns are scaled equally if this option is employed. Ratio-based performance figures such as the profit factor and Sharpe ratio are not affected at all by this option.

***OPTIMIZE = NumberInPort Trials*** - By default, all of the named equity files are included in the portfolio. As an alternative, the *OPTIMIZE* command automatically selects an optimal subset of the named files. Currently, optimization is performed by finding the subset that maximizes the Sharpe ratio. Alternative optimization criteria may be included in a future release. The user specifies the number of files to be included in the portfolio, as well as the number of random trials that will be tested in order to find the best. This should be

very large if the number of equity files is large. If  $n$  is the number of equity files and  $m$  is the portfolio size, the number of combinations is  $n! / ((n-m)! m!)$ , which blows up quickly as  $n$  increases. Ideally, *Trials* should be at least somewhat larger than this quantity, although this may not always be possible. However, it is not usually a serious problem if the number of trials is small relative to the number of possible combinations. This is because although the optimal portfolio found will probably not be the true optimum, it will likely be close to the best. A future release of the program may include an advanced genetic algorithm for optimization.

**WALK FORWARD FROM Year** - The *OPTIMIZE* command produces a large selection bias. In other words, random luck plays a significant role in selecting the best portfolio over a given time period. It is likely that in a different time period this optimal portfolio will not perform as well as it did in the time period in which it was selected. The *WALK FORWARD FROM Year* option lets the user evaluate the degree to which this effect occurs. The entire available time period prior to the specified year is used to find the optimal portfolio, and then this portfolio's performance is evaluated in the specified year. Next, the specified year is appended to the time period used for finding the optimal portfolio, and optimization is performed again. This new optimal portfolio is tested on the following year. This operation of walking forward one year at a time is repeated until the end of the supplied historical data. This command may be used only when the *OPTIMIZE* command is also used.

Here is an example of a PORTFOLIO command that uses 100 trials to find an optimal portfolio of two systems (out of three candidates) and walks it forward from 2005 to evaluate the degree to which optimality holds up. Also, the *EQUALIZE* option is used to make visual interpretation of results easier.

```
PORTFOLIO DEMO_PORT [
    EQUITY FILE = [ "EQTY1.TXT" "EQTY2.TXT" "EQTY3.TXT" ]
    EQUALIZE
    OPTIMIZE = 2 100
    WALK FORWARD FROM 2005
];
```

Numerous statistics are printed in the AUDIT.LOG file. These quantities are based on the daily values present in the equity file. See [here](#) for a discussion of the options available in creating an equity curve.

**Minimum** - The minimum daily profit, negative for a loss.

**Maximum** - The maximum daily profit.

**Mean** - The mean daily profit.

**StdDev** - The standard deviation of daily profits.

**Rng/Std** - The range of daily profits (maximum minus minimum) divided by their standard deviation.

**Sharpe** - The Sharpe ratio of the equity curve. This is the mean daily profit, divided by the standard deviation, and multiplied by the square root of 252, which approximately annualizes the value.

**PF** - The profit factor. This is the sum of positive profits divided by the sum of negative profits (i.e. losses).

**Drawdown** - The maximum net equity loss (peak cumulative equity dropping to subsequent trough).

**Recovery** - The number of days required for the worst drawdown to recover to the peak from which the drawdown began.

These statistics will be printed for each individual equity file, as well as for the portfolio obtained by combining all of the equity files. If the *OPTIMIZE* option is used, these statistics will also be printed for the optimal portfolio. The files selected for the optimal portfolio, as well as those not selected, will be listed. These are identified by the numeric order in which they were specified in the *PORTFOLIO* command. The first file named in the *EQUITYFILE* list is number 1, the second is number 2, and so forth.

If the *WALK FORWARD FROM Year* option appears, results for each out-of-sample year will be printed. This includes three lines:

**Trn** - The best portfolio's in-sample (portfolio selection) performance. This is the performance of the optimal portfolio in the time period in which it was selected, the data prior to the current walkforward year. This quantity will be optimistically biased on average.

**OOS** - The performance of this portfolio for the current walkforward year, the year immediately following the training (portfolio optimization) period. This out-of-sample quantity is an unbiased estimate of the true performance of the optimal portfolio (assuming that the individual portfolio components themselves are unbiased). On average this will be less than that in the *Trn* training period,

although natural market variation will often cause this to exceed the training performance.

**All** - The pooled performance of trading all systems in the current walkforward year. Comparing this to the *OOS* performance shows the difference between simply trading all systems versus trading just the previously selected optimal portfolio.

After walkforward is complete, all out-of-sample data is pooled into a single collection of trades, and two more lines of performance statistics are printed. These are *OPT*, which is the performance of the optimal portfolio (unbiased, since the portfolio was selected before this performance is computed), and *All*, which is the net performance of all equity files. Comparing these two items lets the user evaluate the effect of finding an optimal portfolio versus just trading all systems.

## A Portfolio Example

We conclude the discussion of file portfolios with a complete example. This example employs five predictive models:

**MOD1LONG** - Linear model with indicators selected based on long profit factor

**MOD1SHORT** - Linear model with indicators selected based on short profit factor

**MOD1DUAL** - Linear model with indicators selected based on combined profit factor

**MOD2DUAL** - Linear model with indicators selected based on combined profit factor, looking back a short distance

**MOD3DUAL** - Linear model with indicators selected based on combined profit factor, looking back a longer distance

Here is the script file through the model definitions. We will not bother reproducing the variable definition file EQUITY.TXT here, as files of this type have been treated extensively already.

```

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" ;

MODEL MOD1LONG IS LINREG [
  INPUT = [ LIN_ATR_5 QUA_ATR_5 CUB_ATR_5 LIN_ATR_15
            QUA_ATR_15 CUB_ATR_15 ]
  OUTPUT = RETURN
  MAX STEPWISE = 2
  CRITERION = LONG PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD1SHORT IS LINREG [
  INPUT = [ LIN_ATR_5 QUA_ATR_5 CUB_ATR_5 LIN_ATR_15
            QUA_ATR_15 CUB_ATR_15 ]
  OUTPUT = RETURN
  MAX STEPWISE = 2
  CRITERION = SHORT PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD1DUAL IS LINREG [
  INPUT = [ LIN_ATR_5 QUA_ATR_5 CUB_ATR_5 LIN_ATR_15
            QUA_ATR_15 CUB_ATR_15 ]
  OUTPUT = RETURN
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD2 IS LINREG [
  INPUT = [ LIN_ATR_5 QUA_ATR_5 CUB_ATR_5 ]
  OUTPUT = RETURN
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

MODEL MOD3 IS LINREG [
  INPUT = [ LIN_ATR_15 QUA_ATR_15 CUB_ATR_15 ]
  OUTPUT = RETURN
  MAX STEPWISE = 2
  CRITERION = PROFIT FACTOR
  MIN CRITERION FRACTION = 0.1
] ;

```

Next we have the commands that walk the models forward, preserve the predictions, and execute the trade simulator for each model. The data which

will later be used for portfolio building and testing will begin at the year 2000, the first model walkforward year. We could, of course, TRAIN the models instead of walking them forward, which would let portfolio building examine the entire dataset. However, building and testing portfolios on in-sample data makes little sense, because the returns are optimistically biased. By building portfolios based on the models' out-of-sample trades we are using honest, unbiased trade results to find and test optimal portfolios. Here are these commands, followed by the AUDIT.LOG output produced by them:

```
WALK FORWARD BY YEAR 5 2000 ;
PRESERVE PREDICTIONS ;

TRADE SIMULATOR MOD1LONG GLOBAL LONG (TRAINED 1.0) ATR 250
;
WRITE EQUITY "LONG1.TXT" ;

TRADE SIMULATOR MOD1SHORT GLOBAL SHORT (TRAINED 1.0) ATR
250;
WRITE EQUITY "SHORT1.TXT" ;

TRADE SIMULATOR MOD1DUAL GLOBAL DUAL (TRAINED 1.0 TRAINED
1.0) ATR 250 ;
WRITE EQUITY "DUAL1.TXT" ;

TRADE SIMULATOR MOD2 GLOBAL DUAL (TRAINED 1.0 TRAINED 1.0)
ATR 250 ;
WRITE EQUITY "DUAL2.TXT" ;

TRADE SIMULATOR MOD3 GLOBAL DUAL (TRAINED 1.0 TRAINED 1.0)
ATR 250 ;
WRITE EQUITY "DUAL3.TXT" ;

MODEL MOD1LONG
Bars=6025 Long=347 (5.76%) Short=0 (0.00%)
Total return = 20.026 ATR units
Profit factor = 1.1766
Maximum drawdown = 11.649 on 20080804
289 bars to bottom and 736 bars to recovery

MODEL MOD1SHORT
Bars=6025 Long=0 (0.00%) Short=396 (6.57%)
Total return = -5.276 ATR units
Profit factor = 0.9605
Maximum drawdown = 23.633 on 20070802
1903 bars to bottom and never recovered
```

```

MODEL MOD1DUAL
Bars=6025 Long=382 (6.34%) Short=362 (6.01%)
Total return = 16.444 ATR units
Profit factor = 1.0673
Maximum drawdown = 23.485 on 20070816
    1907 bars to bottom and 2203 bars to recovery

```

```

MODEL MOD2
Bars=6025 Long=457 (7.59%) Short=506 (8.40%)
Total return = -7.200 ATR units
Profit factor = 0.9769
Maximum drawdown = 28.953 on 20070801
    1898 bars to bottom and 2236 bars to recovery

```

```

MODEL MOD3
Bars=6025 Long=462 (7.67%) Short=465 (7.72%)
Total return = 28.977 ATR units
Profit factor = 1.0969
Maximum drawdown = 15.197 on 20060921
    1023 bars to bottom and 1223 bars to recovery

```

We now consider two different portfolios. The first is a simple combination of the *MODILONG* and *MODISHORT* models. This would be a common operation for many users. It is solidly established that trying to find a single model that performs well for both long and short trades is not wise. Performance is almost always improved by using separate models for long and short trades. We can then use the *PORTFOLIO* command to compute the net performance obtained by trading both models simultaneously. Here is the command to do so, followed by the AUDIT.LOG file output produced by this command:

```

PORTFOLIO MOD1_NET [
    EQUITY FILE = [ "LONG1.TXT" "SHORT1.TXT" ]
] ;

```

---

```
Processing Portfolio MOD1_NET
```

---

Basic profit statistics for 3004 records									
System	Minimum	Maximum	Mean	StdDev	Rng/Std	Sharpe	PF	Drawdown	Recovery
1	-2.861	3.588	0.0067	0.319	20.248	0.332	1.177	11.649	736
2	-2.842	3.531	-0.0018	0.329	19.382	-0.085	0.961	23.633	Never
All	-2.861	3.588	0.0049	0.418	15.412	0.186	1.069	16.683	433

In this example we see that the long system (System 1 because LONG1.TXT is named first in the file list) is a decent winner, while the short system actually loses money. Thus, it's not surprising that the net performance of the two systems is inferior to that of the long-only system. We should not judge this example too harshly, because the indicators provided to the models are limited, just a few simple trends, to keep things simple. More exotic indicators improve

performance a lot.

Note, by the way, that the trade simulator reported 6025 bars, while this portfolio reports 3004 records (one equity record per bar) in the equity files. This is because the market file being traded, OEX, contains prices for 6025 days. But the portfolio study is the out-of-sample walkforward data, which begins about halfway through the market history. When the equity files are created, no records are written until the first trade is signaled. This eliminates having an equity file inflated by a large number of leading records whose equity is zero.

The second PORTFOLIO example is more complex because it demonstrates optimization and walkforward testing of the optimal portfolios. Here are the script file commands that read the equity files that were previously written (see [here](#)) and do the optimization and testing:

```
PORTFOLIO PORT_OF_TWO [
    EQUITY FILE = [ "DUAL1.TXT" "DUAL2.TXT" "DUAL3.TXT" ]
    EQUALIZE
    OPTIMIZE = 2 100
    WALK FORWARD FROM 2005
];
```

This defines a portfolio called *PORT\_OF\_TWO*. It reads the equity curve files for the three long/short models. The *EQUALIZE* option is invoked to make reading the table of results easier. The *OPTIMIZE* option specifies that our portfolio will consist of two of these three models. It also says that 100 trials will be used to find the best portfolio, a value that is ridiculously excessive. There are only three ways that one can choose two models out of three! Still, the selection process is very fast, so making the number of trials huge is cheap insurance against randomness playing a cruel trick and causing the program to miss the optimum. Finally, the walkforward process begins with 2005 as the first out-of-sample year.

It's worth exploring in a little more detail exactly what happens when we combine the walkforward model training/testing with the walkforward portfolio training/testing. This is the best, most honest approach, but it is a little complicated. Here is a summary of the steps that are happening behind the scenes:

Model training ends with 1999; the models are evaluated in 2000.  
Model training ends with 2000; the models are evaluated in 2001.  
Model training ends with 2001; the models are evaluated in 2002.  
Model training ends with 2002; the models are evaluated in 2003.  
Model training ends with 2003; the models are evaluated in 2004.

Portfolio optimization ends with 2004; the optimal portfolio is evaluated in 2005.

Model training ends with 2004; the models are evaluated in 2005.

Portfolio optimization ends with 2005; the optimal portfolio is evaluated in 2006.

Model training ends with 2005; the models are evaluated in 2006.

Portfolio optimization ends with 2006; the optimal portfolio is evaluated in 2007.

Model training ends with 2006; the models are evaluated in 2007.

Portfolio optimization ends with 2007; the optimal portfolio is evaluated in 2008.

Et cetera, until the available market history is exhausted.

Notice that portfolio out-of-sample testing always stays one year ahead of the model out-of-sample testing. This ensures that the optimal portfolio is always being based on out-of-sample performance of the models.

The following output is generated by this example. We will break it up into sections for easier presentation.

```
-----  
Processing Portfolio PORT_OF_TWO  
-----  
Keeping 2 systems in portfolio  
Using 100 replications  
Equalizing portfolio to one contract total  
First walkforward year is 2005  
  
Basic profit statistics for 3011 records  
System Minimum Maximum Mean StdDev Rng/Std Sharpe PF Drawdown Recovery  
1 -5.301 3.588 0.0055 0.468 19.001 0.185 1.067 23.485 2203  
2 -5.301 3.588 -0.0024 0.502 17.720 -0.076 0.977 28.953 2236  
3 -5.301 4.891 0.0096 0.526 19.376 0.290 1.097 15.197 1223  
All -5.301 3.588 0.0042 0.354 25.099 0.190 1.054 16.312 2202  
Opt -5.301 3.588 0.0075 0.393 22.610 0.305 1.097 14.732 2026  
  
Selected:  
1  
3  
  
Not selected:  
2
```

The first three lines report the performance statistics for the three component systems. System 2 is a money-loser. If we trade all three systems we obtain poor results due to the unfavorable presence of System 2. By trading just the best pair of systems, we obtain a portfolio Sharpe ratio of 0.305, which is better than any individual component and also better than trading all three. Note that the *days to recovery* reported here for the individual systems can sometimes be a day or so different from that reported by the trade simulator. This is due to alignment issues related to matching trade dates among multiple systems. It should never be anything more than a trivial difference.

The table just shown concerns the optimal portfolio derived from the entire available model walkforward period, which begins in the year 2000. The next step is to walk the optimization process forward. Each year is printed in the AUDIT.LOG file, but for economy we will show only the first year here.

For walkforward OOS year 2005, trained best portfolio is: 2 3										
Data	Minimum	Maximum	Mean	StdDev	Rng/Std	Sharpe	PF	Drawdown	Recovery	
Trn	-2.431	2.416	-0.0044	0.317	15.292	-0.221	0.941	12.670	Never	
OOS	-0.891	0.934	-0.0050	0.229	7.951	-0.346	0.919	4.106	Never	
All	-0.989	1.245	-0.0015	0.223	9.999	-0.107	0.974	4.020	Never	

In the table above we see that based on model out-of-sample data ending in 2004, the best portfolio, consisting of Systems 2 and 3, had a net loss. It's Sharpe ratio is -0.221. When this two-system portfolio is tested in 2005 it obtains an even worse Sharpe ratio of -0.346. In this year (2005) we would actually have been best off trading all three systems, because this portfolio has a Sharpe ratio of -0.107.

After all walkforward years have been tested, a summary table is computed by pooling all of the OOS data. Here is this table:

Data	Minimum	Maximum	Mean	StdDev	Rng/Std	Sharpe	PF	Drawdown	Recovery
Opt	-5.301	3.588	0.0138	0.431	20.621	0.509	1.155	10.086	36
All	-5.301	3.588	0.0124	0.388	22.931	0.509	1.148	9.451	Never

In this case, the Sharpe ratio happens (purely by coincidence) to be the same whether we traded the optimal portfolio or all systems in each test year. We do see that the optimal portfolio had a slightly higher mean return per day, as well as a slightly higher profit factor.

It must be emphasized that this optimization example has been kept deliberately simple for the purposes of this tutorial. In a real application, the component models would use far more sophisticated indicators and thereby have improved performance. Also, we might have a dozen candidate models, or perhaps even hundreds, and choose a substantial fraction of them for the optimal portfolio. In such a situation, performance will likely be much improved over that obtained in this example.

***Important note on WALK FORWARD folds:*** The folds in the PORTFOLIO command are based on the dates in the equity file, which are the dates of actual change in equity. However, the folds in the ordinary WALK FORWARD command are based on the date on which the trade decision is made, which most users would consider more sensible. Thus, the folds do not exactly correspond. In general they are shifted with respect to each other by two days, and hence results may not exactly agree with each other. This is not incorrect, it is just an unavoidable difference of fold definition.

# Integrated Portfolios

The prior section discussed file-based Portfolios. Those are useful in situations in which the developer wishes to import equity curves from other programs such as TradeStation™ and make use of *TSSB*'s Portfolio building and testing algorithms. However, inmost situations the developer wants to build Portfolios from models, committees, and oracles already implemented in a *TSSB* script. In such cases, using the *Trade Simulator* to produce equity files, and then reading them with the FILE PORTFOLIO command, is a nuisance. For this reason, *TSSB* also includes a Portfolio building and testing algorithm fully integrated into the train/test cycle. This *Integrated Portfolio* is the subject of this chapter.

There are two subtle differences between the *File Portfolio* and the *Integrated Portfolio* that should be mentioned. First, the *File Portfolio* deletes from the beginning and end of the equity files all records that have no equity change. This reduces the number of cases included in the equity history. If a user employs the *Trade Simulator* to produce equity files, processes these files with the *File Portfolio*, and compares the results to those obtained strictly internally with the *Integrated Portfolio*, means and standard deviations may be slightly different. This is because the different number of cases results in dividing sums by different numbers of bars. This is of no practical consequence.

Second, the *File Portfolio* bases walkforward folds on the dates in the equity files. The *Trade Simulator* assigns the dates according to when equity changes are recorded. But the *Integrated Portfolio* bases walkforward folds on the dates on which trade decisions are made, which is a more sensible and practical approach. Since trade decisions obviously precede resulting changes in equity, walkforward folds will be slightly misaligned between the two versions, producing slightly different fold performance statistics.

A Portfolio is a collection of two or more Models, Committees, or Oracles. (Actually, it is legal for a Portfolio to contain only one component, but that would be pointless.) A Portfolio may contain a mixture of Models, Committees, and Oracles. The Portfolio must not be defined until all of its components have been defined. Portfolios are ignored for CROSS VALIDATE commands. They are evaluated only during TRAIN and WALK FORWARD commands.

An *Integrated Portfolio* is defined by means of the following command:

**PORFTOLIO PortName [ Specifications ];**

The following specifications are available:

**MODELS = [ Model\_1 Model\_2 Model\_3 ... ]**

*This mandatory specification lists the component Models (or Committees or Oracles) of the portfolio. Depending on other specifications, all of these Models may make up the Portfolio, or only a subset of the list.*

**LONG ONLY**

**SHORT ONLY**

*By default, the Portfolio includes all trades, long and short, of the component models. Including one of these optional specifications causes the Portfolio to consider only long or only short trades of the components.*

**TYPE = FIXED**

*The TYPE of the Portfolio must be specified. The FIXED Portfolio uses all of the components listed in the MODELS specification.*

**TYPE = OPTIMIZE Nused IS**

*The TYPE of the Portfolio must be specified. The IS (In-Sample) type uses ‘Nused’ components of those listed in the MODELS specification. The subset is chosen by maximizing the Sharpe Ratio of the Portfolio. The NREPS specification, described below, must appear if the IS type of Portfolio is defined. When a TRAIN command is executed, the optimal subset of components is found by examining the entire dataset, just as is done to train the components. When a WALK FORWARD command is executed, the optimal subset is found by examining the in-sample fold data, the same data that is used to train the components. The performance of the Portfolio will be based on the out-of-sample data in each fold, the same data that is used to assess the components.*

**TYPE = OPTIMIZE Nused OOS Nfolds FOLDS**

*The TYPE of the Portfolio must be specified. The OOS (Out-Of-Sample) type uses ‘Nused’ components of those listed in the MODELS specification. The subset is chosen by maximizing the Sharpe Ratio of the Portfolio. The NREPS specification, described below, must appear if the OOS type of Portfolio is defined. This type of Portfolio is evaluated only when a WALK FORWARD command is executed. Double folds are implemented: the components are trained on the training fold and evaluated on the out-of-sample fold. Then the Portfolio’s optimal subset is chosen by examining the ‘Nfolds’ most recent folds of out-of-sample data, and the Portfolio’s performance is evaluated based on the data in the next out-of-sample fold. A detailed example appears later in*

*this chapter.*

**EQUALIZE PORTFOLIO STATS**

*This option has no effect on the selection of the optimal subset of components, or on any other aspect of computation. It affects only printing of performance results. If this option does not appear in the Portfolio definition, printed results are based on the sum of the components' trades. If this option is used, printed results are based on the mean of the components' profits and losses.*

**OVERLAP = Integer**

*This option has the same use as in training the components. If the target has a lookahead greater than one bar, it is best (though not critical) to set the OVERLAP to one less than the target's lookahead.*

**NREPS = Integer**

*This option is required if the IS or OOS type is employed. It specifies the number of randomly selected subsets to try in order to find the subset that maximizes the Portfolio optimization criterion, currently the Sharpe Ratio.*

## A FIXED Portfolio Example

One useful application of a FIXED Portfolio is to facilitate training separate Models (or entire trading systems) to handle long and short trades independently. It is well known that it is difficult to find a single trading system (a Model or set of Models combined with a Committee or Oracle) that works well for both long and short trades. Specialization in one or the other almost always provides the best performance. By making use of a Portfolio we can train long and short systems separately, and then obtain net performance estimates. Consider the following simple example:

```
MODEL LIN_LONG 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 = 2
    LONG ONLY
    CRITERION = LONG PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

MODEL LIN_SHORT 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 = 2
    SHORT ONLY
    CRITERION = SHORT PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
] ;

PORTFOLIO Port [
    MODELS = [ LIN_LONG LIN_SHORT ]
    TYPE = FIXED
] ;
```

This example employs two models that are almost identical. Their only difference is that the first chooses its predictors to maximize the long profit factor, and it executes only long trades. The second maximizes the short profit factor and executes only short trades. The simple Portfolio combines these two sets of trades and prints net performance results.

## An OOS Portfolio Example

The prior example showed how a fixed Portfolio could be used to pool long and short specialists in order to obtain net results from trading both sides of the market. Another use of Portfolios is to examine the performance of several (perhaps a great many) Models or complete trading systems that also contain Committees and Oracles. The Portfolio will then automatically select an optimal subset of the candidates.

At this time the only optimization criterion available is the Sharpe Ratio. The popular wisdom is that the Sharpe Ratio is a poor measure of financial performance, the leading argument being that it penalizes large upward moves in equity as much as it penalizes large downward moves. This is not the forum to examine the pros and cons of the Sharpe ratio, but for now please understand that this argument contains significant technical flaws. In fact, the Sharpe Ratio is an excellent way of comparing portfolios, primarily because it is sensitive to the desirable effect of losses in one component being offset by gains in another component. A laudable goal of finding a good portfolio is making its equity curve as smooth as possible. The Sharpe Ratio does an admirable job of achieving this goal.

Another minor weakness of the current Portfolio optimization algorithm is that it simply tries a large number (specified with the NREPS parameter) of randomly selected subsets and chooses the best. A future release of *TSSB* may employ genetic optimization, although unless the number of candidates is huge, the difference in execution speed between naive search and intelligent search is insignificant.

It should be obvious that choosing the best subset from among competing candidates introduces a large optimistic bias. The important question is not how well the portfolio performs in the time period in which it was selected. Rather, we need to know whether a portfolio's performance holds up in the future. This inspires us to use walkforward testing in evaluating a Portfolio.

The choice to use walkforward testing immediately leads to another question: what data do we use for finding the optimal subset? We could use the same time period as was used to train the candidate models. This is often reasonable, and it is the method employed with the TYPE=...IS option. However, there is a significant risk with this method. Suppose some component is based on an excessively powerful model, meaning that the model overfits the data and hence has superb in-sample performance. This component will likely be selected for the optimal portfolio, even though its out-of-sample performance may be dismal due to the overfitting. Thus, we are inspired to select the optimal portfolio

based on the components' out-of-sample performance, which is much more honest than in-sample performance. This calls for double walkforward, in which multiple out-of-sample folds are needed. The example script file shown on the [here](#), and the explanation that follows, make this operation clear.

```
MODEL LIN_1 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 = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    EXCLUSION GROUP = 1
] ;

MODEL LIN_2 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 = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    EXCLUSION GROUP = 1
] ;

MODEL LIN_3 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 = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    EXCLUSION GROUP = 1
] ;

MODEL LIN_4 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 = 2
    CRITERION = PROFIT FACTOR
    MIN CRITERION FRACTION = 0.1
    EXCLUSION GROUP = 1
] ;
```

```

PORTFOLIO OptPort [
  MODELS = [ LIN_1 LIN_2 LIN_3 LIN_4 ]
  TYPE = OPTIMIZE 2 OOS 3 FOLDS
  NREPS = 1000
] ;

WALK FORWARD BY YEAR 5 2006 ;

```

This example defines four candidate Models using the EXCLUSION GROUP option. This is an excellent method for creating Committee components, but it is not necessarily the best method for building a Portfolio. Most users would want to be more explicit in defining the components. But it's quick and easy, so this is the method used for this example.

The TYPE option in this Portfolio specifies that two of the four candidates will be chosen for the Portfolio, and this selection will be based on three out-of-sample training folds. Let us look in detail at the sequence of operations that will be performed.

- Train the Models using 2001-2005
- Test the Models using 2006
- Train the Models using 2002-2006
- Test the Models using 2007
- Train the Models using 2003-2007
- Test the Models using 2008
- Train the Models using 2004-2008
- Test the Models using 2009
- Train the Portfolio using 2006-2008 Model OOS results
- Test the Portfolio using 2009 Model OOS results
- Train the Models using 2005-2009
- Test the Models using 2010
- Train the Portfolio using 2007-2009 Model OOS results
- Test the Portfolio using 2010 Model OOS results
- Et cetera

The Portfolio's performance will be reported in two ways in the AUDIT.LOG file. It will be reported for each fold in which the Portfolio is evaluated, and then reported one last time in the walkforward summary section. The following statistics are printed:

**Minimum** - The minimum daily profit, negative for a loss.

**Maximum** - The maximum daily profit.

**Mean** - The mean daily (bar) profit across all days.

**StdDev** - The standard deviation of daily (bar) profits.

**Rng/Std** - The range of daily profits (maximum minus minimum) divided by their standard deviation.

**Sharpe** - The Sharpe ratio of the equity curve. This is the mean daily profit, divided by the standard deviation, and multiplied by the square root of 252, which approximately annualizes the value.

**PF** - The profit factor. This is the sum of positive profits divided by the sum of negative profits (i.e. losses).

**Drawdown** - The maximum net loss (peak cumulative equity dropping to subsequent trough).

**Recovery** - The number of days required for the worst drawdown to recover to the peak from which the drawdown began.

Note that when these results are reported for the complete Portfolio, by default they will be based on the sum of component results. If the EQUALIZE PORTFOLIO STATS option is used in the Portfolio definition, these results reflect the mean of the components rather than their sum.

Here is an example of fold results for the example just show:

```
Portfolio OPTPORT statistics for 755 training dates
(Mean and StdDev are across all dates, not just all trades.)
```

System	Minimum	Maximum	Mean	StdDev	Rng/Std	Sharpe	PF	Drawdown	Recovery
LIN_1	-3.423	3.588	0.0404	0.407	17.234	1.575	1.861	6.035	225
LIN_2	-3.423	3.588	0.0298	0.499	14.048	0.947	1.423	8.389	245
LIN_3	-3.423	3.588	0.0032	0.483	14.525	0.104	1.033	14.031	Never
LIN_4	-3.423	3.588	0.0364	0.479	14.644	1.208	1.575	6.959	145
Pooled	-13.694	14.352	0.1097	1.591	17.630	1.095	1.396	29.642	234
Opt sub	-6.847	7.176	0.0768	0.802	17.488	1.521	1.697	11.447	213
Opt OOS	-1.805	1.725	-0.0079	0.354	9.968	-0.356	0.904	5.302	Never

Optimization kept the following candidates:

```
LIN_1
LIN_4
```

Optimization rejected the following candidates:

```
LIN_2
LIN_3
```

The table above first shows the individual performance of the four candidates in the Portfolio's training set (the recent out-of-sample Model data). Then it pools all candidates. Next it shows the performance for the optimal subset. Finally, it shows the performance of this optimal subset in the Portfolio's out-of-sample period.

The walkforward summary provides the same information, but in a different

format:

Portfolio OPTPORT pooled OOS performance with 746 dates...

Min = -4.02348 Max = 5.94531

Mean = 0.00652 StdDev = 0.62720

Profit factor = 1.05098 Annualized Sharpe ratio= 0.16498

Max drawdown = 12.80320 320 days to recovery