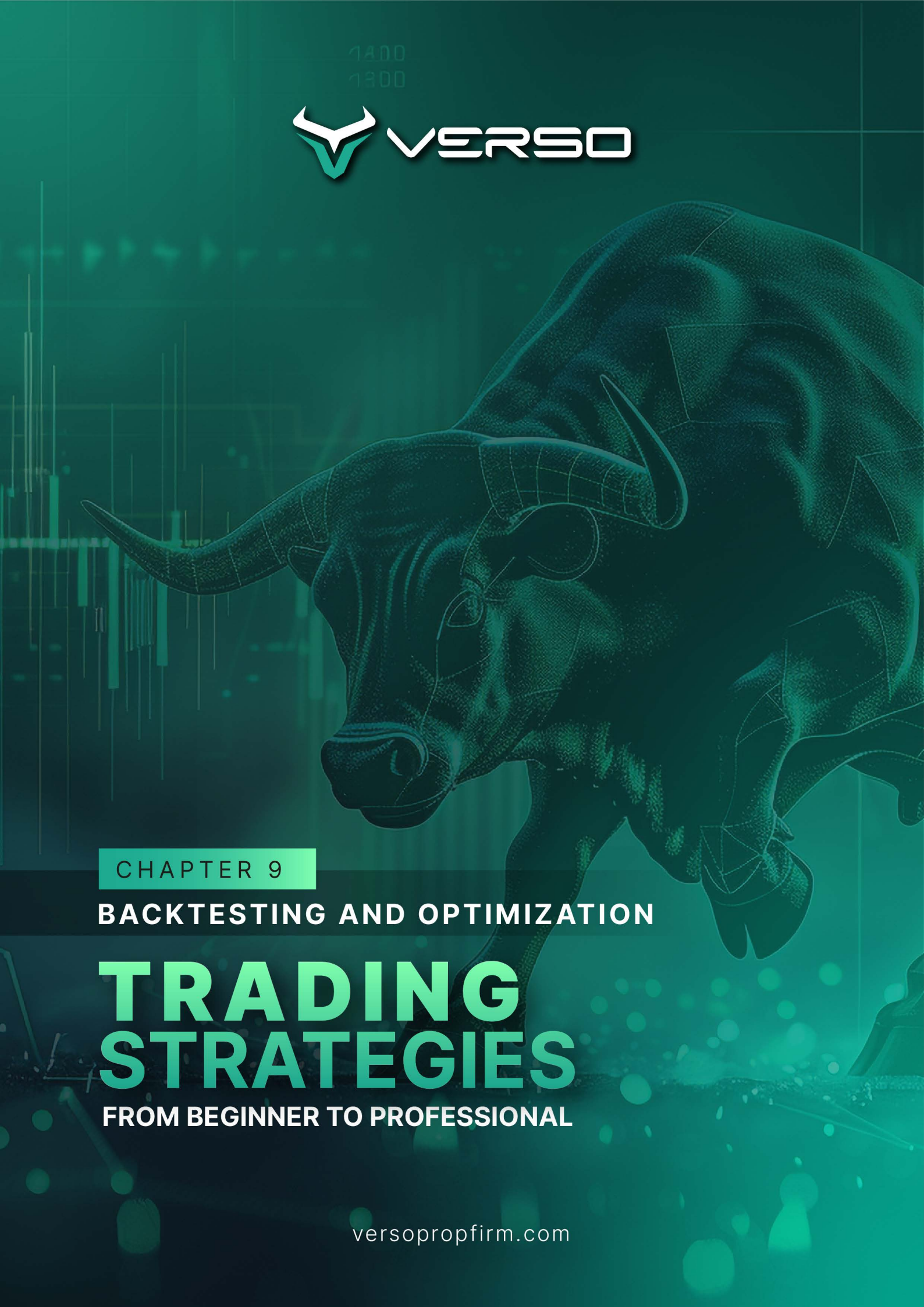

CHAPTER 9

BACKTESTING AND OPTIMIZATION

TRADING STRATEGIES

FROM BEGINNER TO PROFESSIONAL

versopropfirm.com

Backtesting and Optimization

Backtesting is like having a time machine that allows you to test your strategies in the past before risking real money. But beware: poorly done backtesting is worse than no backtesting at all, as it gives you a false sense of security.

The Deadly Sins of Backtesting

1. **Look-Ahead Bias** Using information that wasn't available at the time of the decision.
2. **Survivorship Bias** Only testing on assets that "survived" until today, ignoring those that disappeared.
3. **Data Snooping (Overfitting)** Testing so many variations until you find one that "works" by chance.
4. **Ignoring Transaction Costs** Not including realistic spreads, commissions, and slippage.

Professional Backtester

python

```
class ProfessionalBacktester:
    def __init__(self, initial_capital=100000, commission=0.001, slippage=0.0005):
        self.initial_capital = initial_capital
        self.commission = commission
        self.slippage = slippage
        self.trades = []
        self.equity_curve = []
        self.positions = {}

    def add_realistic_costs(self, entry_price, exit_price, quantity, trade_type):
        """
        Adds realistic trading costs
        """
        # Commission
        entry_commission = abs(entry_price * quantity * self.commission)
        exit_commission = abs(exit_price * quantity * self.commission)

        # Slippage (worse price due to market impact)
        if trade_type == 'buy':
```

```
            entry_price_adjusted = entry_price * (1 + self.slippage)
            exit_price_adjusted = exit_price * (1 - self.slippage)
        else:
            entry_price_adjusted = entry_price * (1 - self.slippage)
            exit_price_adjusted = exit_price * (1 + self.slippage)

        # Bid-ask spread impact
        spread_cost = abs(entry_price * quantity * 0.0002)  # Typical 2 bps
        total_costs = entry_commission + exit_commission + spread_cost
        return {
            'entry_price_adjusted': entry_price_adjusted,
            'exit_price_adjusted': exit_price_adjusted,
            'total_costs': total_costs
        }

    def execute_strategy(self, data, strategy_function, start_date=None, end_date=None):
        """
        Executes a strategy with rigorous controls
        """
        if start_date:
            data = data[data.index >= start_date]
        if end_date:
            data = data[data.index <= end_date]
        current_capital = self.initial_capital
        current_positions = {}
        for i, (timestamp, row) in enumerate(data.iterrows()):
            # Data available up to this point (avoids look-ahead bias)
            available_data = data.iloc[:i+1]

            # Execute strategy
            signals = strategy_function(available_data)

            # Process signals
            for signal in signals:
                if signal['action'] == 'BUY':
                    self._execute_buy(signal, row, timestamp, current_capital)
                elif signal['action'] == 'SELL':
```

```
            self._execute_sell(signal, row, timestamp)
        elif signal['action'] == 'CLOSE':
            self._close_position(signal, row, timestamp)

    # Calculate equity
    portfolio_value = self._calculate_portfolio_value(current_positions, row)
    self.equity_curve.append({
        'timestamp': timestamp,
        'portfolio_value': portfolio_value,
        'cash': current_capital,
        'positions_value': portfolio_value - current_capital
    })
```

python

```
def _execute_buy(self, signal, row, timestamp, available_capital):
    """
    Executes buy order with realistic controls
    """
    symbol = signal['symbol']
    entry_price = row['Open']  # Assumes execution at next open

    # Calculate position size
    if signal.get('position_size_pct'):
        position_value = available_capital * signal['position_size_pct']
        quantity = position_value / entry_price
    else:
        quantity = signal.get('quantity', 100)

    # Apply realistic costs
    costs = self.add_realistic_costs(entry_price, entry_price, quantity, 'buy')
    adjusted_entry = costs['entry_price_adjusted']

    # Verify sufficient capital
    total_cost = adjusted_entry * quantity + costs['total_costs']
    if total_cost > available_capital:
        return False  # Order rejected
```

```
        # Execute trade
        trade = {
            'symbol': symbol,
            'entry_timestamp': timestamp,
            'entry_price': adjusted_entry,
            'quantity': quantity,
            'trade_type': 'LONG',
            'status': 'OPEN',
            'costs': costs['total_costs']
        }
        self.trades.append(trade)
        self.positions[symbol] = trade
        return True

    def generate_performance_report(self):
        """
        Generates performance report
        """
        if not self.equity_curve:
            return "No data available for analysis"

        # Convert to DataFrame for analysis
        equity_df = pd.DataFrame(self.equity_curve)

        # Basic metrics
        total_return = (equity_df['portfolio_value'].iloc[-1] / self.initial_capital) - 1
        trading_days = len(equity_df)
        annualized_return = ((1 + total_return) ** (252 / trading_days)) - 1

        # Volatility
        daily_returns = equity_df['portfolio_value'].pct_change().dropna()
        volatility = daily_returns.std() * np.sqrt(252)

        # Sharpe Ratio (assumes risk-free rate of 2%)
        risk_free_rate = 0.02
        sharpe_ratio = (annualized_return - risk_free_rate) / volatility
```

```
        # Maximum Drawdown
        rolling_max = equity_df['portfolio_value'].cummax()
        drawdown = (equity_df['portfolio_value'] - rolling_max) / rolling_max
        max_drawdown = drawdown.min()

        # Win Rate and Profit Factor
        completed_trades = [t for t in self.trades if t['status'] == 'CLOSED']
        if completed_trades:
            winning_trades = [t for t in completed_trades if t.get('pnl', 0) > 0]
            losing_trades = [t for t in completed_trades if t.get('pnl', 0) < 0]
            win_rate = len(winning_trades) / len(completed_trades)
            total_wins = sum(t.get('pnl', 0) for t in winning_trades)
            total_losses = abs(sum(t.get('pnl', 0) for t in losing_trades))
            profit_factor = total_wins / total_losses if total_losses > 0 else float('inf')
        else:
            win_rate = 0
            profit_factor = 0
```

python

```
# Calmar Ratio
        calmar_ratio = annualized_return / abs(max_drawdown) if max_drawdown != 0 else
float('inf')
        return {
            'total_return': total_return,
            'annualized_return': annualized_return,
            'volatility': volatility,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': max_drawdown,
            'calmar_ratio': calmar_ratio,
            'win_rate': win_rate,
            'profit_factor': profit_factor,
            'total_trades': len(completed_trades),
            'trading_days': trading_days
        }
```

Parameter Optimization Without Overfitting Walk-Forward Analysis: The Professional Optimization

```
python
class WalkForwardOptimizer:
    def __init__(self, strategy_function, parameter_ranges, optimization_window=252,
forward_window=63):
        self.strategy_function = strategy_function
        self.parameter_ranges = parameter_ranges
        self.optimization_window = optimization_window  # 1 year for optimization
        self.forward_window = forward_window  # 3 months for validation

    def generate_parameter_combinations(self):
        """
        Generates all parameter combinations to test
        """
        from itertools import product
        param_names = list(self.parameter_ranges.keys())
        param_values = list(self.parameter_ranges.values())
        combinations = []
        for combo in product(*param_values):
            param_dict = dict(zip(param_names, combo))
            combinations.append(param_dict)
        return combinations

    def walk_forward_analysis(self, data):
        """
        Executes complete walk-forward analysis
        """
        results = []
        start_idx = self.optimization_window
        while start_idx + self.forward_window < len(data):


# Optimization period
        opt_start = start_idx - self.optimization_window
        opt_end = start_idx
        optimization_data = data.iloc[opt_start:opt_end]
```

```
        # Validation period (forward)
        val_start = start_idx
        val_end = start_idx + self.forward_window
        validation_data = data.iloc[val_start:val_end]

        # Optimize parameters in optimization period
        best_params = self.optimize_parameters(optimization_data)

        # Validate in forward period
        forward_performance = self.validate_parameters(validation_data, best_params)
        results.append({
            'optimization_period': (opt_start, opt_end),
            'validation_period': (val_start, val_end),
            'best_parameters': best_params,
            'forward_performance': forward_performance
        })

        # Move window forward
        start_idx += self.forward_window
        return results

    def optimize_parameters(self, data):
        """
        Optimizes parameters in a specific period
        """
        parameter_combinations = self.generate_parameter_combinations()

        best_sharpe = -float('inf')
        best_params = None
        for params in parameter_combinations:
            # Execute strategy with these parameters
            backtester = ProfessionalBacktester()

            # Modify strategy_function to use specific params
            def parameterized_strategy(data):
                return self.strategy_function(data, **params)
```

```
            backtester.execute_strategy(data, parameterized_strategy)
            performance = backtester.generate_performance_report()

    # Use Sharpe as optimization criterion (avoids overfitting)
            sharpe = performance.get('sharpe_ratio', -float('inf'))
            if sharpe > best_sharpe:
                best_sharpe = sharpe
                best_params = params.copy()
        return best_params

    def validate_parameters(self, data, parameters):
        """
        Validates parameters in forward period
        """
        backtester = ProfessionalBacktester()
        def parameterized_strategy(data):
            return self.strategy_function(data, **parameters)

        backtester.execute_strategy(data, parameterized_strategy)
        return backtester.generate_performance_report()

    def analyze_stability(self, walk_forward_results):
        """
        Analyzes parameter stability over time
        """
        all_params = {}

        # Collect all optimal parameters
        for result in walk_forward_results:
            for param, value in result['best_parameters'].items():
                if param not in all_params:
                    all_params[param] = []
                all_params[param].append(value)

        # Calculate stability (lower variation = more stable)
        stability_metrics = {}
```

```
        for param, values in all_params.items():
            stability_metrics[param] = {
                'mean': np.mean(values),
                'std': np.std(values),
                'cv': np.std(values) / np.mean(values) if np.mean(values) != 0 else float('inf'),
                'range': max(values) - min(values)
            }
        return stability_metrics
```

Monte Carlo Analysis: Testing Robustness

python

```
class MonteCarloAnalyzer:
    def __init__(self, num_simulations=1000):
        self.num_simulations = num_simulations

    def bootstrap_returns(self, historical_returns):
        """
        Bootstraps historical returns to generate random paths
        """
        simulated_paths = []
        for _ in range(self.num_simulations):
            # Sample with replacement
            simulated_returns = np.random.choice(historical_returns,
size=len(historical_returns), replace=True)

            # Build equity path
            equity_path = [1.0]  # Start with $1
            for ret in simulated_returns:
                equity_path.append(equity_path[-1] * (1 + ret))
            simulated_paths.append(equity_path)

        return simulated_paths

    def analyze_monte_carlo_results(self, simulated_paths):
        """
```

```
    Analyzes Monte Carlo results
    """
    # Convert to numpy array for analysis
    paths_array = np.array(simulated_paths)

    # Final returns of each simulation
    final_returns = (paths_array[:, -1] - 1)

    # Maximum drawdowns of each path
    max_drawdowns = []
    for path in simulated_paths:
        path_array = np.array(path)
        rolling_max = np.maximum.accumulate(path_array)
        drawdowns = (path_array - rolling_max) / rolling_max
        max_drawdowns.append(np.min(drawdowns))

    # Statistics
    results = {
        'final_return_stats': {
            'mean': np.mean(final_returns),
            'median': np.median(final_returns),
            'std': np.std(final_returns),
            'percentile_5': np.percentile(final_returns, 5),
            'percentile_95': np.percentile(final_returns, 95),
            'probability_positive': np.mean(final_returns > 0)
        },
        'max_drawdown_stats': {
            'mean': np.mean(max_drawdowns),
            'median': np.median(max_drawdowns),
            'worst_case': np.min(max_drawdowns),
            'percentile_5': np.percentile(max_drawdowns, 5),
            'percentile_95': np.percentile(max_drawdowns, 95)
        }
    }
    return results
```

```
    def value_at_risk(self, simulated_paths, confidence_levels=[0.05, 0.01]):
        """
        Calculates Value at Risk for different confidence levels
        """
        final_values = [path[-1] for path in simulated_paths]
        final_returns = [(value - 1) for value in final_values]
        var_results = {}
        for confidence in confidence_levels:
            var_results[f'VaR_{int(confidence * 100)}%'] = np.percentile(final_returns,
confidence * 100)
        return var_results
```

versopropfirm.com