Comprehensive AI Instructions for Algorithmic Trading Systems

Generated from BuildAlpha.com content extraction - September 3, 2025

Table of Contents

Core Principles and Philosophy

Fundamental Approach to Algorithmic Trading

When providing algorithmic trading guidance, always emphasize these core principles:

1. Edge-First Methodology
- Every trading strategy must demonstrate a genuine statistical edge over random chance
- Implement baseline comparisons against random signal generation
- Use edge ratio calculations: (Strategy Performance) / (Random Strategy Performance)
- Require strategies to outperform both random signals and buy-and-hold benchmarks

2. Robustness Over Optimization
- Prioritize strategies that perform consistently across different market conditions
- Avoid over-optimization and curve-fitting at all costs
- Implement multiple robustness tests before deployment
- Focus on out-of-sample performance validation

3. Scientific Rigor
- Apply statistical significance testing to all results
- Use proper sample sizes for meaningful conclusions
- Implement Monte Carlo simulations for performance validation
- Maintain detailed documentation of all assumptions and limitations

4. Risk-Adjusted Returns
- Always evaluate strategies using risk-adjusted metrics (Sharpe ratio, Sortino ratio, Calmar ratio)

- Consider maximum drawdown as a primary constraint
- Implement position sizing based on risk parity principles
- Account for transaction costs, slippage, and market impact

AI System Behavior Guidelines

For Strategy Development:
- Always start with hypothesis formation before data mining
- Provide both theoretical justification and empirical validation
- Include failure analysis when strategies don't work
- Emphasize the importance of out-of-sample testing

For Code Implementation:
- Write modular, testable code with clear documentation
- Include error handling and edge case management
- Provide performance monitoring and logging capabilities
- Implement proper data validation and cleaning procedures

For Risk Management:
- Never recommend strategies without explicit risk controls
- Always include position sizing calculations
- Provide multiple exit strategies and stop-loss mechanisms
- Consider correlation effects in portfolio construction

Strategy Development Framework

Systematic Strategy Creation Process

Phase 1: Market Hypothesis Formation

When helping users develop trading strategies, guide them through this systematic process:

1. **Market Inefficiency Identification**
   ```python
   # Example framework for hypothesis testing
   def test_market_hypothesis(data, hypothesis_params):
   """
   Test a specific market inefficiency hypothesis

   Parameters:
   - data: Historical price/volume data
   - hypothesis_params: Dictionary containing hypothesis parameters

   Returns:
   - Statistical significance of the hypothesis
   - Effect size and practical significance
   """
   # Implementation details below
   ```

2. **Signal Generation Logic**
   - Technical indicators (moving averages, RSI, MACD, Bollinger Bands)

- Price action patterns (breakouts, reversals, momentum)
- Volume-based signals (volume spikes, volume-price divergence)
- Fundamental factors (earnings, economic indicators)
- Sentiment indicators (VIX, put/call ratios)

3. **Entry and Exit Criteria**
   - Multiple confirmation signals to reduce false positives
   - Time-based exits to prevent indefinite holding
   - Profit targets based on historical volatility
   - Stop-loss levels using ATR or percentage-based methods

Phase 2: Signal Universe Construction

Provide users with comprehensive signal libraries:

Technical Indicators (5000+ combinations)

```
# Core technical indicators with parameter ranges
TECHNICAL_SIGNALS = {
    'moving_averages': {
        'sma': [5, 10, 20, 50, 100, 200],
        'ema': [5, 10, 20, 50, 100, 200],
        'wma': [5, 10, 20, 50, 100, 200]
    },
    'momentum': {
        'rsi': [14, 21, 30],
        'stochastic': [14, 21],
        'williams_r': [14, 21]
    },
    'volatility': {
        'bollinger_bands': [20, 50],
        'atr': [14, 21],
        'volatility_ratio': [10, 20, 30]
    },
    'volume': {
        'volume_sma': [10, 20, 50],
        'on_balance_volume': [10, 20],
        'volume_price_trend': [10, 20]
    }
}
```

Price Action Patterns

```
PRICE_PATTERNS = {
    'breakouts': [
        'resistance_breakout',
        'support_breakdown',
        'range_breakout',
        'consolidation_breakout'
    ],
    'reversals': [
        'double_top',
        'double_bottom',
        'head_shoulders',
        'inverse_head_shoulders'
    ],
    'continuation': [
        'flag_pattern',
        'pennant_pattern',
        'triangle_pattern'
    ]
}
```

Phase 3: Strategy Combination and Testing

Guide users to combine signals systematically:

```python
def create_strategy_combinations(signal_universe, max_signals=3):
    """
    Generate all possible strategy combinations up to max_signals

    This prevents overfitting while ensuring comprehensive coverage
    """
    from itertools import combinations

    strategies = []
    for r in range(1, max_signals + 1):
        for combo in combinations(signal_universe, r):
            strategies.append({
                'signals': combo,
                'logic': 'AND',  # All signals must be true
                'weight': 'equal'  # Equal weighting initially
            })

    return strategies
```

Entry Signal Categories

1. Momentum Signals
- Price above/below moving averages
- RSI overbought/oversold conditions
- MACD crossovers and divergences
- Rate of change indicators

2. Mean Reversion Signals
- Bollinger Band extremes
- RSI divergences
- Price deviation from moving averages
- Volatility contraction patterns

3. Breakout Signals
- New highs/lows over N periods
- Volume-confirmed breakouts
- Range expansion patterns
- Volatility breakouts

4. Pattern Recognition Signals
- Candlestick patterns
- Chart patterns (triangles, flags, wedges)
- Support/resistance level breaks
- Trend line violations

Exit Strategy Framework

1. Profit Taking Methods

```python
def calculate_profit_targets(entry_price, volatility, method='atr'):
    """
    Calculate dynamic profit targets based on market volatility
    """
    if method == 'atr':
        return entry_price * (1 + 2 * volatility)  # 2x ATR profit target
    elif method == 'percentage':
        return entry_price * 1.02  # 2% profit target
    elif method == 'resistance':
        return find_nearest_resistance(entry_price)
```

2. Stop Loss Mechanisms

```python
def calculate_stop_loss(entry_price, volatility, method='atr'):
    """
    Calculate dynamic stop losses based on market conditions
    """
    if method == 'atr':
        return entry_price * (1 - volatility)  # 1x ATR stop loss
    elif method == 'percentage':
        return entry_price * 0.98  # 2% stop loss
    elif method == 'support':
        return find_nearest_support(entry_price)
```

3. Time-Based Exits
- Maximum holding period to prevent indefinite positions
- Intraday strategies with end-of-day exits
- Weekly/monthly rebalancing schedules

Backtesting and Validation

Comprehensive Backtesting Framework

1. Data Quality and Preparation

Always emphasize proper data handling:

```
def prepare_trading_data(raw_data):
    """
    Comprehensive data preparation for backtesting
    """
    # Remove outliers (beyond 3 standard deviations)
    data = remove_outliers(raw_data, method='zscore', threshold=3)

    # Handle missing data
    data = handle_missing_data(data, method='forward_fill')

    # Adjust for splits and dividends
    data = adjust_for_corporate_actions(data)

    # Add technical indicators
    data = add_technical_indicators(data)

    # Validate data integrity
    validate_data_quality(data)

    return data
```

2. Realistic Trading Simulation

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

    def execute_trade(self, symbol, quantity, price, timestamp):
        """
        Execute trade with realistic costs and slippage
        """
        # Apply slippage
        if quantity > 0:  # Buy order
            execution_price = price * (1 + self.slippage)
        else:  # Sell order
            execution_price = price * (1 - self.slippage)

        # Calculate commission
        commission_cost = abs(quantity * execution_price * self.commission)

        # Update positions and record trade
        self.update_position(symbol, quantity, execution_price, commission_cost)
        self.record_trade(symbol, quantity, execution_price, commission_cost,
timestamp)
```

3. Out-of-Sample Testing Protocol

Implement rigorous out-of-sample testing:

```python
def walk_forward_analysis(data, strategy, window_size=252, step_size=21):
    """
    Perform walk-forward analysis to validate strategy robustness

    Parameters:
    - data: Historical data
    - strategy: Trading strategy function
    - window_size: Training window size (252 = 1 year)
    - step_size: Step size for rolling window (21 = 1 month)
    """
    results = []

    for start in range(0, len(data) - window_size, step_size):
        # Training period
        train_data = data[start:start + window_size]

        # Out-of-sample period
        test_data = data[start + window_size:start + window_size + step_size]

        # Optimize strategy on training data
        optimized_params = optimize_strategy(strategy, train_data)

        # Test on out-of-sample data
        oos_performance = backtest_strategy(strategy, test_data, optimized_params)

        results.append({
            'train_period': (start, start + window_size),
            'test_period': (start + window_size, start + window_size + step_size),
            'performance': oos_performance
        })

    return results
```

Performance Metrics and Evaluation

1. Core Performance Metrics

```python
def calculate_performance_metrics(returns, benchmark_returns=None):
    """
    Calculate comprehensive performance metrics
    """
    metrics = {}

    # Basic metrics
    metrics['total_return'] = (1 + returns).prod() - 1
    metrics['annualized_return'] = (1 + returns.mean()) ** 252 - 1
    metrics['volatility'] = returns.std() * np.sqrt(252)

    # Risk-adjusted metrics
    metrics['sharpe_ratio'] = metrics['annualized_return'] / metrics['volatility']
    metrics['sortino_ratio'] = metrics['annualized_return'] / (returns[returns < 0].std
() * np.sqrt(252))

    # Drawdown metrics
    cumulative = (1 + returns).cumprod()
    running_max = cumulative.expanding().max()
    drawdown = (cumulative - running_max) / running_max

    metrics['max_drawdown'] = drawdown.min()
    metrics['calmar_ratio'] = metrics['annualized_return'] /
abs(metrics['max_drawdown'])

    # Win/Loss metrics
    winning_trades = returns[returns > 0]
    losing_trades = returns[returns < 0]

    metrics['win_rate'] = len(winning_trades) / len(returns)
    metrics['avg_win'] = winning_trades.mean() if len(winning_trades) > 0 else 0
    metrics['avg_loss'] = losing_trades.mean() if len(losing_trades) > 0 else 0
    metrics['profit_factor'] = abs(winning_trades.sum() / losing_trades.sum()) if los-
ing_trades.sum() != 0 else np.inf

    # Benchmark comparison
    if benchmark_returns is not None:
        metrics['alpha'] = metrics['annualized_return'] - ((1 + benchmark_returns.mean(
)) ** 252 - 1)
        metrics['beta'] = np.cov(returns, benchmark_returns)[0, 1] / np.var(benchmark_r
eturns)
        metrics['information_ratio'] = metrics['alpha'] / (returns - bench-
mark_returns).std() * np.sqrt(252)

    return metrics
```

2. Statistical Significance Testing

```
def test_statistical_significance(strategy_returns, benchmark_returns, confid-
ence_level=0.95):
    """
    Test if strategy outperformance is statistically significant
    """
    from scipy import stats

    excess_returns = strategy_returns - benchmark_returns

    # T-test for mean excess return
    t_stat, p_value = stats.ttest_1samp(excess_returns, 0)

    # Bootstrap confidence interval
    bootstrap_means = []
    n_bootstrap = 10000

    for _ in range(n_bootstrap):
        sample = np.random.choice(excess_returns, size=len(excess_returns), replace=Tru
e)
        bootstrap_means.append(sample.mean())

    alpha = 1 - confidence_level
    ci_lower = np.percentile(bootstrap_means, alpha/2 * 100)
    ci_upper = np.percentile(bootstrap_means, (1 - alpha/2) * 100)

    return {
        't_statistic': t_stat,
        'p_value': p_value,
        'is_significant': p_value < alpha,
        'confidence_interval': (ci_lower, ci_upper)
    }
```

Risk Management Systems

Position Sizing and Capital Allocation

1. Kelly Criterion Implementation

```python
def kelly_position_size(win_rate, avg_win, avg_loss, capital):
    """
    Calculate optimal position size using Kelly Criterion

    Kelly % = (bp - q) / b
    where:
    b = avg_win / avg_loss (odds received on the wager)
    p = win_rate (probability of winning)
    q = 1 - p (probability of losing)
    """
    if avg_loss == 0:
        return 0

    b = avg_win / abs(avg_loss)
    p = win_rate
    q = 1 - p

    kelly_fraction = (b * p - q) / b

    # Apply fractional Kelly to reduce risk
    conservative_kelly = kelly_fraction * 0.25  # Use 25% of full Kelly

    # Cap position size at 10% of capital
    max_position = 0.10
    position_fraction = min(conservative_kelly, max_position)

    return max(0, position_fraction * capital)
```

2. Risk Parity Position Sizing

```python
def risk_parity_weights(returns_matrix, target_volatility=0.15):
    """
    Calculate risk parity weights for portfolio construction
    """
    # Calculate covariance matrix
    cov_matrix = returns_matrix.cov() * 252  # Annualized

    # Calculate individual asset volatilities
    volatilities = np.sqrt(np.diag(cov_matrix))

    # Risk parity weights (inverse volatility)
    weights = (1 / volatilities) / (1 / volatilities).sum()

    # Scale to target volatility
    portfolio_vol = np.sqrt(weights.T @ cov_matrix @ weights)
    scaling_factor = target_volatility / portfolio_vol

    return weights * scaling_factor
```

3. Dynamic Position Sizing Based on Market Conditions

```
def dynamic_position_size(base_size, market_volatility, current_vol,
max_vol_ratio=2.0):
    """
    Adjust position size based on current market volatility
    """
    vol_ratio = current_vol / market_volatility

    # Reduce position size when volatility is high
    if vol_ratio > max_vol_ratio:
        adjustment_factor = max_vol_ratio / vol_ratio
    else:
        adjustment_factor = 1.0

    return base_size * adjustment_factor
```

Stop Loss and Risk Control Mechanisms

1. Adaptive Stop Losses

```
def calculate_adaptive_stop_loss(entry_price, atr, volatility_regime):
    """
    Calculate stop loss based on market volatility regime
    """
    base_stop_multiple = 1.5

    if volatility_regime == 'low':
        stop_multiple = base_stop_multiple * 0.8
    elif volatility_regime == 'high':
        stop_multiple = base_stop_multiple * 1.5
    else:  # normal
        stop_multiple = base_stop_multiple

    stop_distance = atr * stop_multiple
    return entry_price - stop_distance  # For long positions
```

2. Portfolio-Level Risk Controls

```python
class PortfolioRiskManager:
    def __init__(self, max_portfolio_risk=0.02, max_sector_exposure=0.20):
        self.max_portfolio_risk = max_portfolio_risk
        self.max_sector_exposure = max_sector_exposure
        self.positions = {}

    def check_risk_limits(self, new_position):
        """
        Check if new position violates risk limits
        """
        # Calculate portfolio risk with new position
        total_risk = self.calculate_portfolio_risk(new_position)

        if total_risk > self.max_portfolio_risk:
            return False, "Portfolio risk limit exceeded"

        # Check sector concentration
        sector_exposure = self.calculate_sector_exposure(new_position)
        if sector_exposure > self.max_sector_exposure:
            return False, "Sector exposure limit exceeded"

        return True, "Position approved"

    def calculate_portfolio_risk(self, new_position=None):
        """
        Calculate total portfolio risk (VaR)
        """
        # Implementation of portfolio VaR calculation
        pass
```

Correlation and Diversification Management

1. Correlation Monitoring

```python
def monitor_portfolio_correlations(returns_matrix, max_correlation=0.7):
    """
    Monitor correlations between portfolio positions
    """
    correlation_matrix = returns_matrix.corr()

    # Find highly correlated pairs
    high_correlations = []
    for i in range(len(correlation_matrix.columns)):
        for j in range(i+1, len(correlation_matrix.columns)):
            corr = correlation_matrix.iloc[i, j]
            if abs(corr) > max_correlation:
                high_correlations.append({
                    'asset1': correlation_matrix.columns[i],
                    'asset2': correlation_matrix.columns[j],
                    'correlation': corr
                })

    return high_correlations
```

2. Diversification Metrics

```
def calculate_diversification_ratio(weights, cov_matrix):
    """
    Calculate portfolio diversification ratio

    DR = (sum of weighted individual volatilities) / (portfolio volatility)
    """
    individual_vols = np.sqrt(np.diag(cov_matrix))
    weighted_avg_vol = np.sum(weights * individual_vols)
    portfolio_vol = np.sqrt(weights.T @ cov_matrix @ weights)

    return weighted_avg_vol / portfolio_vol
```

Market Analysis and Signal Generation

Technical Analysis Implementation

1. Comprehensive Technical Indicator Library

```python
class TechnicalIndicators:
    @staticmethod
    def sma(prices, period):
        """Simple Moving Average"""
        return prices.rolling(window=period).mean()

    @staticmethod
    def ema(prices, period):
        """Exponential Moving Average"""
        return prices.ewm(span=period).mean()

    @staticmethod
    def rsi(prices, period=14):
        """Relative Strength Index"""
        delta = prices.diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
        rs = gain / loss
        return 100 - (100 / (1 + rs))

    @staticmethod
    def bollinger_bands(prices, period=20, std_dev=2):
        """Bollinger Bands"""
        sma = prices.rolling(window=period).mean()
        std = prices.rolling(window=period).std()
        upper_band = sma + (std * std_dev)
        lower_band = sma - (std * std_dev)
        return upper_band, sma, lower_band

    @staticmethod
    def macd(prices, fast=12, slow=26, signal=9):
        """MACD Indicator"""
        ema_fast = prices.ewm(span=fast).mean()
        ema_slow = prices.ewm(span=slow).mean()
        macd_line = ema_fast - ema_slow
        signal_line = macd_line.ewm(span=signal).mean()
        histogram = macd_line - signal_line
        return macd_line, signal_line, histogram

    @staticmethod
    def atr(high, low, close, period=14):
        """Average True Range"""
        tr1 = high - low
        tr2 = abs(high - close.shift())
        tr3 = abs(low - close.shift())
        true_range = pd.concat([tr1, tr2, tr3], axis=1).max(axis=1)
        return true_range.rolling(window=period).mean()
```

2. Pattern Recognition System

```python
class PatternRecognition:
    @staticmethod
    def detect_breakout(prices, volume, lookback=20, volume_threshold=1.5):
        """
        Detect breakout patterns with volume confirmation
        """
        # Calculate resistance and support levels
        resistance = prices.rolling(window=lookback).max()
        support = prices.rolling(window=lookback).min()

        # Average volume
        avg_volume = volume.rolling(window=lookback).mean()

        # Breakout conditions
        resistance_breakout = (prices > resistance.shift(1)) & (volume > avg_volume *
volume_threshold)
        support_breakdown = (prices < support.shift(1)) & (volume > avg_volume *
volume_threshold)

        return resistance_breakout, support_breakdown

    @staticmethod
    def detect_reversal_patterns(high, low, close, period=5):
        """
        Detect common reversal patterns
        """
        patterns = {}

        # Double top pattern
        patterns['double_top'] = PatternRecognition._double_top(high, period)

        # Double bottom pattern
        patterns['double_bottom'] = PatternRecognition._double_bottom(low, period)

        # Head and shoulders
        patterns['head_shoulders'] = PatternRecognition._head_shoulders(high, period)

        return patterns

    @staticmethod
    def _double_top(high, period):
        """Detect double top pattern"""
        # Implementation of double top detection logic
        pass
```

3. Multi-Timeframe Analysis

```
def multi_timeframe_analysis(data, timeframes=['1D', '1W', '1M']):
    """
    Analyze signals across multiple timeframes
    """
    signals = {}

    for tf in timeframes:
        # Resample data to timeframe
        tf_data = data.resample(tf).agg({
            'open': 'first',
            'high': 'max',
            'low': 'min',
            'close': 'last',
            'volume': 'sum'
        })

        # Calculate signals for this timeframe
        signals[tf] = {
            'trend': calculate_trend_signal(tf_data),
            'momentum': calculate_momentum_signal(tf_data),
            'mean_reversion': calculate_mean_reversion_signal(tf_data)
        }

    # Combine signals across timeframes
    combined_signal = combine_timeframe_signals(signals)

    return combined_signal
```

Fundamental Analysis Integration

1. Economic Indicator Processing

```
def process_economic_indicators(indicators_data):
    """
    Process and normalize economic indicators for trading signals
    """
    processed = {}

    for indicator, data in indicators_data.items():
        # Normalize to z-scores
        processed[indicator] = {
            'raw': data,
            'zscore': (data - data.mean()) / data.std(),
            'percentile': data.rank(pct=True),
            'trend': calculate_trend(data)
        }

    return processed
```

2. Earnings and Corporate Events

```
def incorporate_earnings_data(price_data, earnings_dates, earnings_surprises):
    """
    Incorporate earnings data into trading signals
    """
    # Create earnings event indicators
    earnings_events = pd.Series(0, index=price_data.index)
    earnings_events.loc[earnings_dates] = 1

    # Pre/post earnings periods
    pre_earnings = earnings_events.shift(-5).fillna(0)  # 5 days before
    post_earnings = earnings_events.shift(1).fillna(0)  # 1 day after

    # Earnings surprise impact
    surprise_impact = pd.Series(0, index=price_data.index)
    for date, surprise in earnings_surprises.items():
        if date in surprise_impact.index:
            surprise_impact.loc[date] = surprise

    return {
        'earnings_events': earnings_events,
        'pre_earnings': pre_earnings,
        'post_earnings': post_earnings,
        'surprise_impact': surprise_impact
    }
```

Sentiment Analysis and Alternative Data

1. Market Sentiment Indicators

```
def calculate_sentiment_indicators(market_data):
    """
    Calculate various market sentiment indicators
    """
    sentiment = {}

    # VIX-based fear/greed indicator
    sentiment['vix_sentiment'] = calculate_vix_sentiment(market_data['vix'])

    # Put/Call ratio
    sentiment['put_call_ratio'] = market_data['put_volume'] / mar-
ket_data['call_volume']

    # Advance/Decline ratio
    sentiment['advance_decline'] = market_data['advancing'] / market_data['declining']

    # High-Low index
    sentiment['high_low_index'] = market_data['new_highs'] / (market_data['new_highs']
+ market_data['new_lows'])

    return sentiment
```

2. News Sentiment Analysis

```
def analyze_news_sentiment(news_data, symbol):
    """
    Analyze news sentiment for trading signals
    """
    from textblob import TextBlob

    sentiment_scores = []

    for article in news_data:
        if symbol.lower() in article['title'].lower() or symbol.lower() in article['con
tent'].lower():
            # Calculate sentiment polarity
            blob = TextBlob(article['content'])
            sentiment_scores.append({
                'date': article['date'],
                'polarity': blob.sentiment.polarity,
                'subjectivity': blob.sentiment.subjectivity,
                'source': article['source']
            })

    # Aggregate sentiment by date
    daily_sentiment = aggregate_sentiment_by_date(sentiment_scores)

    return daily_sentiment
```

Execution and Implementation

Order Management System

1. Smart Order Routing

```
class SmartOrderRouter:
    def __init__(self, brokers, execution_algorithms):
        self.brokers = brokers
        self.execution_algorithms = execution_algorithms

    def route_order(self, order):
        """
        Route order to best execution venue
        """
        # Analyze market conditions
        market_conditions = self.analyze_market_conditions(order.symbol)

        # Select optimal execution algorithm
        algo = self.select_execution_algorithm(order, market_conditions)

        # Choose best broker/venue
        broker = self.select_broker(order, market_conditions)

        # Execute order
        return self.execute_order(order, broker, algo)

    def select_execution_algorithm(self, order, market_conditions):
        """
        Select optimal execution algorithm based on order and market conditions
        """
        if order.size > market_conditions['avg_volume'] * 0.1:
            return 'TWAP'  # Time-Weighted Average Price
        elif market_conditions['volatility'] > market_conditions['avg_volatility'] * 1.
5:
            return 'Implementation_Shortfall'
        else:
            return 'Market_Order'
```

2. Execution Algorithms

```
class ExecutionAlgorithms:
    @staticmethod
    def twap_execution(order, time_horizon_minutes=60):
        """
        Time-Weighted Average Price execution
        """
        slice_size = order.quantity / (time_horizon_minutes / 5)  # 5-minute slices
        execution_schedule = []

        for i in range(0, time_horizon_minutes, 5):
            execution_schedule.append({
                'time': i,
                'quantity': slice_size,
                'order_type': 'limit',
                'price_offset': 0  # At mid-price
            })

        return execution_schedule

    @staticmethod
    def vwap_execution(order, historical_volume_profile):
        """
        Volume-Weighted Average Price execution
        """
        total_volume = historical_volume_profile.sum()
        execution_schedule = []

        for time_slot, volume in historical_volume_profile.items():
            participation_rate = 0.1  # 10% of historical volume
            slice_quantity = (volume / total_volume) * order.quantity * participa-
tion_rate

            execution_schedule.append({
                'time_slot': time_slot,
                'quantity': slice_quantity,
                'participation_rate': participation_rate
            })

        return execution_schedule
```

3. Slippage and Market Impact Modeling

```python
def estimate_market_impact(order_size, avg_daily_volume, volatility, spread):
    """
    Estimate market impact of order execution
    """
    # Participation rate
    participation_rate = order_size / avg_daily_volume

    # Temporary impact (recovers after execution)
    temporary_impact = 0.5 * spread * (participation_rate ** 0.5)

    # Permanent impact (price discovery)
    permanent_impact = volatility * 0.1 * (participation_rate ** 0.6)

    # Total impact
    total_impact = temporary_impact + permanent_impact

    return {
        'temporary_impact': temporary_impact,
        'permanent_impact': permanent_impact,
        'total_impact': total_impact,
        'participation_rate': participation_rate
    }
```

Real-Time Monitoring and Alerts

1. Performance Monitoring System

```python
class PerformanceMonitor:
    def __init__(self, strategies):
        self.strategies = strategies
        self.alerts = []
        self.performance_metrics = {}

    def monitor_real_time(self):
        """
        Monitor strategy performance in real-time
        """
        for strategy in self.strategies:
            current_metrics = self.calculate_current_metrics(strategy)

            # Check for performance degradation
            if self.detect_performance_degradation(strategy, current_metrics):
                self.trigger_alert(strategy, 'performance_degradation')

            # Check for risk limit breaches
            if self.check_risk_limits(strategy, current_metrics):
                self.trigger_alert(strategy, 'risk_limit_breach')

            # Update metrics
            self.performance_metrics[strategy.name] = current_metrics

    def detect_performance_degradation(self, strategy, current_metrics):
        """
        Detect if strategy performance is degrading
        """
        historical_sharpe = strategy.historical_metrics['sharpe_ratio']
        current_sharpe = current_metrics['sharpe_ratio']

        # Alert if Sharpe ratio drops by more than 50%
        return current_sharpe < historical_sharpe * 0.5
```

2. Risk Alert System

```
class RiskAlertSystem:
    def __init__(self, risk_limits):
        self.risk_limits = risk_limits
        self.alert_history = []

    def check_alerts(self, portfolio_state):
        """
        Check for risk limit violations
        """
        alerts = []

        # Portfolio-level alerts
        if portfolio_state['total_risk'] > self.risk_limits['max_portfolio_risk']:
            alerts.append({
                'type': 'portfolio_risk',
                'severity': 'high',
                'message': f"Portfolio risk {portfolio_state['total_risk']:.
2%} exceeds limit {self.risk_limits['max_portfolio_risk']:.2%}"
            })

        # Position-level alerts
        for position in portfolio_state['positions']:
            if position['unrealized_pnl'] < -self.risk_limits['max_position_loss']:
                alerts.append({
                    'type': 'position_loss',
                    'severity': 'medium',
                    'symbol': position['symbol'],
                    'message': f"Position loss {position['unrealized_pnl']:.
2%} exceeds limit"
                })

        return alerts
```

Infrastructure and Technology Stack

1. Data Pipeline Architecture

```python
class DataPipeline:
    def __init__(self, data_sources, storage_backend):
        self.data_sources = data_sources
        self.storage = storage_backend
        self.processors = []

    def ingest_data(self):
        """
        Ingest data from multiple sources
        """
        for source in self.data_sources:
            raw_data = source.fetch_data()

            # Data validation
            validated_data = self.validate_data(raw_data)

            # Data cleaning and preprocessing
            clean_data = self.clean_data(validated_data)

            # Store processed data
            self.storage.store(clean_data)

    def validate_data(self, data):
        """
        Validate data quality and completeness
        """
        # Check for missing values
        if data.isnull().sum().sum() > 0:
            self.handle_missing_data(data)

        # Check for outliers
        outliers = self.detect_outliers(data)
        if len(outliers) > 0:
            self.handle_outliers(data, outliers)

        # Check data freshness
        if self.is_data_stale(data):
            self.alert_stale_data(data)

        return data
```

2. Backtesting Infrastructure

```python
class DistributedBacktester:
    def __init__(self, compute_cluster):
        self.cluster = compute_cluster
        self.task_queue = []

    def run_parallel_backtests(self, strategies, data, parameters):
        """
        Run multiple backtests in parallel
        """
        # Create tasks for each strategy/parameter combination
        tasks = []
        for strategy in strategies:
            for param_set in parameters:
                task = {
                    'strategy': strategy,
                    'parameters': param_set,
                    'data': data,
                    'id': f"{strategy.name}_{hash(str(param_set))}"
                }
                tasks.append(task)

        # Distribute tasks across cluster
        results = self.cluster.map(self.run_single_backtest, tasks)

        # Aggregate results
        return self.aggregate_results(results)

    def run_single_backtest(self, task):
        """
        Run a single backtest task
        """
        strategy = task['strategy']
        parameters = task['parameters']
        data = task['data']

        # Initialize strategy with parameters
        strategy.set_parameters(parameters)

        # Run backtest
        results = strategy.backtest(data)

        return {
            'task_id': task['id'],
            'parameters': parameters,
            'results': results
        }
```

Python Implementation Guidelines

Code Structure and Best Practices

1. Modular Architecture

```python
# File: trading_system/__init__.py
"""
Algorithmic Trading System
A comprehensive framework for developing, testing, and deploying trading strategies
"""

from .data import DataManager, MarketDataFeed
from .strategies import BaseStrategy, TechnicalStrategy, FundamentalStrategy
from .backtesting import BacktestEngine, PerformanceAnalyzer
from .risk import RiskManager, PositionSizer
from .execution import OrderManager, ExecutionEngine
from .portfolio import Portfolio, PortfolioOptimizer

__version__ = "1.0.0"
__author__ = "Your Name"

# File: trading_system/strategies/base.py
from abc import ABC, abstractmethod
import pandas as pd
import numpy as np

class BaseStrategy(ABC):
    """
    Abstract base class for all trading strategies
    """

    def __init__(self, name, parameters=None):
        self.name = name
        self.parameters = parameters or {}
        self.positions = {}
        self.trades = []
        self.signals = pd.DataFrame()

    @abstractmethod
    def generate_signals(self, data):
        """
        Generate trading signals based on market data

        Parameters:
        -----------
        data : pd.DataFrame
            Market data with OHLCV columns

        Returns:
        --------
        pd.DataFrame
            DataFrame with signal columns (entry, exit, position_size)
        """
        pass

    @abstractmethod
    def calculate_position_size(self, signal, current_price, portfolio_value):
        """
        Calculate position size for a given signal
        """
        pass

    def backtest(self, data, initial_capital=100000):
        """
        Backtest the strategy on historical data
        """
        # Generate signals
```

```
        signals = self.generate_signals(data)

        # Initialize backtest engine
        engine = BacktestEngine(initial_capital)

        # Execute trades based on signals
        for date, signal in signals.iterrows():
            if signal['entry']:
                position_size = self.calculate_position_size(
                    signal, data.loc[date, 'close'], engine.portfolio_value
                )
                engine.enter_position(date, signal['symbol'], position_size, data.loc[d
ate, 'close'])

            elif signal['exit']:
                engine.exit_position(date, signal['symbol'], data.loc[date, 'close'])

        return engine.get_results()
```

2. Data Management System

```
# File: trading_system/data/manager.py
import pandas as pd
import numpy as np
from typing import Dict, List, Optional
import sqlite3
import yfinance as yf

class DataManager:
    """
    Centralized data management system
    """

    def __init__(self, database_path="trading_data.db"):
        self.db_path = database_path
        self.connection = sqlite3.connect(database_path)
        self.cache = {}
        self._initialize_database()

    def _initialize_database(self):
        """Initialize database tables"""
        cursor = self.connection.cursor()

        # Price data table
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS price_data (
                symbol TEXT,
                date DATE,
                open REAL,
                high REAL,
                low REAL,
                close REAL,
                volume INTEGER,
                adjusted_close REAL,
                PRIMARY KEY (symbol, date)
            )
        """)

        # Economic indicators table
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS economic_indicators (
                indicator TEXT,
                date DATE,
                value REAL,
                PRIMARY KEY (indicator, date)
            )
        """)

        self.connection.commit()

    def get_price_data(self, symbol: str, start_date: str, end_date: str) -> pd.Data-
Frame:
        """
        Retrieve price data for a symbol
        """
        cache_key = f"{symbol}_{start_date}_{end_date}"

        if cache_key in self.cache:
            return self.cache[cache_key]

        # Try to get from database first
        query = """
            SELECT * FROM price_data
```

```
            WHERE symbol = ? AND date BETWEEN ? AND ?
            ORDER BY date
        """

        df = pd.read_sql_query(query, self.connection, params=(symbol, start_date,
end_date))

        if df.empty:
            # Fetch from external source
            df = self._fetch_external_data(symbol, start_date, end_date)
            self._store_price_data(df)

        # Cache the result
        self.cache[cache_key] = df

        return df

    def _fetch_external_data(self, symbol: str, start_date: str, end_date: str) -> pd.D
ataFrame:
        """
        Fetch data from external source (Yahoo Finance)
        """
        try:
            ticker = yf.Ticker(symbol)
            data = ticker.history(start=start_date, end=end_date)

            # Standardize column names
            data.columns = [col.lower().replace(' ', '_') for col in data.columns]
            data['symbol'] = symbol
            data.reset_index(inplace=True)
            data['date'] = data['date'].dt.date

            return data

        except Exception as e:
            print(f"Error fetching data for {symbol}: {e}")
            return pd.DataFrame()

    def _store_price_data(self, data: pd.DataFrame):
        """
        Store price data in database
        """
        data.to_sql('price_data', self.connection, if_exists='append', index=False)
        self.connection.commit()

    def add_technical_indicators(self, data: pd.DataFrame) -> pd.DataFrame:
        """
        Add technical indicators to price data
        """
        # Simple Moving Averages
        for period in [5, 10, 20, 50, 200]:
            data[f'sma_{period}'] = data['close'].rolling(window=period).mean()

        # Exponential Moving Averages
        for period in [12, 26]:
            data[f'ema_{period}'] = data['close'].ewm(span=period).mean()

        # RSI
        data['rsi'] = self._calculate_rsi(data['close'])

        # MACD
        data['macd'], data['macd_signal'], data['macd_histogram'] = self._calcu-
late_macd(data['close'])
```

```
        # Bollinger Bands
        data['bb_upper'], data['bb_middle'], data['bb_lower'] = self._calcu-
late_bollinger_bands(data['close'])

        # Average True Range
        data['atr'] = self._calculate_atr(data['high'], data['low'], data['close'])

        return data

    def _calculate_rsi(self, prices: pd.Series, period: int = 14) -> pd.Series:
        """Calculate RSI"""
        delta = prices.diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
        rs = gain / loss
        return 100 - (100 / (1 + rs))

    def _calculate_macd(self, prices: pd.Series, fast: int = 12, slow: int = 26, sig-
nal: int = 9):
        """Calculate MACD"""
        ema_fast = prices.ewm(span=fast).mean()
        ema_slow = prices.ewm(span=slow).mean()
        macd_line = ema_fast - ema_slow
        signal_line = macd_line.ewm(span=signal).mean()
        histogram = macd_line - signal_line
        return macd_line, signal_line, histogram

    def _calculate_bollinger_bands(self, prices: pd.Series, period: int = 20, std_dev:
int = 2):
        """Calculate Bollinger Bands"""
        sma = prices.rolling(window=period).mean()
        std = prices.rolling(window=period).std()
        upper_band = sma + (std * std_dev)
        lower_band = sma - (std * std_dev)
        return upper_band, sma, lower_band

    def _calculate_atr(self, high: pd.Series, low: pd.Series, close: pd.Series, period:
int = 14):
        """Calculate Average True Range"""
        tr1 = high - low
        tr2 = abs(high - close.shift())
        tr3 = abs(low - close.shift())
        true_range = pd.concat([tr1, tr2, tr3], axis=1).max(axis=1)
        return true_range.rolling(window=period).mean()
```

3. Strategy Implementation Examples

```python
# File: trading_system/strategies/technical.py
from .base import BaseStrategy
import pandas as pd
import numpy as np

class MovingAverageCrossover(BaseStrategy):
    """
    Simple moving average crossover strategy
    """

    def __init__(self, fast_period=10, slow_period=30, **kwargs):
        super().__init__("MA_Crossover", **kwargs)
        self.fast_period = fast_period
        self.slow_period = slow_period

    def generate_signals(self, data):
        """
        Generate signals based on moving average crossover
        """
        # Calculate moving averages
        data[f'ma_fast'] = data['close'].rolling(window=self.fast_period).mean()
        data[f'ma_slow'] = data['close'].rolling(window=self.slow_period).mean()

        # Generate signals
        signals = pd.DataFrame(index=data.index)
        signals['symbol'] = data.get('symbol', 'UNKNOWN')
        signals['price'] = data['close']

        # Entry signal: fast MA crosses above slow MA
        signals['entry'] = (data[f'ma_fast'] > data[f'ma_slow']) & \
                           (data[f'ma_fast'].shift(1) <= data[f'ma_slow'].shift(1))

        # Exit signal: fast MA crosses below slow MA
        signals['exit'] = (data[f'ma_fast'] < data[f'ma_slow']) & \
                          (data[f'ma_fast'].shift(1) >= data[f'ma_slow'].shift(1))

        # Position direction
        signals['position'] = np.where(data[f'ma_fast'] > data[f'ma_slow'], 1, 0)

        return signals

    def calculate_position_size(self, signal, current_price, portfolio_value):
        """
        Calculate position size (simple fixed percentage)
        """
        risk_per_trade = 0.02  # 2% risk per trade
        return (portfolio_value * risk_per_trade) / current_price

class MeanReversionStrategy(BaseStrategy):
    """
    Mean reversion strategy using Bollinger Bands
    """

    def __init__(self, bb_period=20, bb_std=2, rsi_period=14, **kwargs):
        super().__init__("Mean_Reversion", **kwargs)
        self.bb_period = bb_period
        self.bb_std = bb_std
        self.rsi_period = rsi_period

    def generate_signals(self, data):
        """
        Generate mean reversion signals
```

```
        """
        # Calculate indicators
        data['bb_upper'], data['bb_middle'], data['bb_lower'] = self._calcu-
late_bollinger_bands(
            data['close'], self.bb_period, self.bb_std
        )
        data['rsi'] = self._calculate_rsi(data['close'], self.rsi_period)

        signals = pd.DataFrame(index=data.index)
        signals['symbol'] = data.get('symbol', 'UNKNOWN')
        signals['price'] = data['close']

        # Entry conditions
        # Long entry: price touches lower BB and RSI < 30
        long_entry = (data['close'] <= data['bb_lower']) & (data['rsi'] < 30)

        # Short entry: price touches upper BB and RSI > 70
        short_entry = (data['close'] >= data['bb_upper']) & (data['rsi'] > 70)

        # Exit conditions
        # Long exit: price reaches middle BB or RSI > 70
        long_exit = (data['close'] >= data['bb_middle']) | (data['rsi'] > 70)

        # Short exit: price reaches middle BB or RSI < 30
        short_exit = (data['close'] <= data['bb_middle']) | (data['rsi'] < 30)

        # Combine signals
        signals['long_entry'] = long_entry
        signals['short_entry'] = short_entry
        signals['long_exit'] = long_exit
        signals['short_exit'] = short_exit

        # Overall entry/exit signals
        signals['entry'] = long_entry | short_entry
        signals['exit'] = long_exit | short_exit

        # Position direction
        signals['position'] = np.where(long_entry, 1, np.where(short_entry, -1, 0))

        return signals
```

4. Backtesting Engine

```
# File: trading_system/backtesting/engine.py
import pandas as pd
import numpy as np
from typing import Dict, List, Tuple

class BacktestEngine:
    """
    Comprehensive backtesting engine
    """

    def __init__(self, initial_capital=100000, commission=0.001, slippage=0.0005):
        self.initial_capital = initial_capital
        self.commission = commission
        self.slippage = slippage

        # Portfolio state
        self.cash = initial_capital
        self.positions = {}
        self.portfolio_value = initial_capital

        # Trade tracking
        self.trades = []
        self.equity_curve = []
        self.daily_returns = []

        # Performance metrics
        self.metrics = {}

    def enter_position(self, date, symbol, quantity, price):
        """
        Enter a new position
        """
        # Apply slippage
        if quantity > 0:  # Long position
            execution_price = price * (1 + self.slippage)
        else:  # Short position
            execution_price = price * (1 - self.slippage)

        # Calculate costs
        trade_value = abs(quantity * execution_price)
        commission_cost = trade_value * self.commission
        total_cost = trade_value + commission_cost

        # Check if we have enough cash
        if total_cost > self.cash:
            return False  # Insufficient funds

        # Update cash
        self.cash -= total_cost

        # Update position
        if symbol in self.positions:
            self.positions[symbol]['quantity'] += quantity
            self.positions[symbol]['avg_price'] = (
                (self.positions[symbol]['avg_price'] * self.positions[symbol]['quant-
ity'] +
                 execution_price * quantity) / (self.positions[symbol]['quantity'] +
quantity)
            )
        else:
            self.positions[symbol] = {
                'quantity': quantity,
```

```
                'avg_price': execution_price,
                'entry_date': date
            }

        # Record trade
        self.trades.append({
            'date': date,
            'symbol': symbol,
            'action': 'BUY' if quantity > 0 else 'SELL',
            'quantity': abs(quantity),
            'price': execution_price,
            'commission': commission_cost,
            'total_cost': total_cost
        })

        return True

    def exit_position(self, date, symbol, price):
        """
        Exit a position
        """
        if symbol not in self.positions:
            return False  # No position to exit

        position = self.positions[symbol]
        quantity = position['quantity']

        # Apply slippage
        if quantity > 0:  # Closing long position
            execution_price = price * (1 - self.slippage)
        else:  # Closing short position
            execution_price = price * (1 + self.slippage)

        # Calculate proceeds
        trade_value = abs(quantity * execution_price)
        commission_cost = trade_value * self.commission
        net_proceeds = trade_value - commission_cost

        # Update cash
        if quantity > 0:  # Closing long
            self.cash += net_proceeds
        else:  # Closing short
            self.cash += net_proceeds

        # Calculate P&L
        if quantity > 0:  # Long position
            pnl = (execution_price - position['avg_price']) * quantity - commis-
sion_cost
        else:  # Short position
            pnl = (position['avg_price'] - execution_price) * abs(quantity) - commis-
sion_cost

        # Record trade
        self.trades.append({
            'date': date,
            'symbol': symbol,
            'action': 'SELL' if quantity > 0 else 'COVER',
            'quantity': abs(quantity),
            'price': execution_price,
            'commission': commission_cost,
            'pnl': pnl,
            'entry_date': position['entry_date'],
            'holding_period': (date - position['entry_date']).days
```

```
        })

        # Remove position
        del self.positions[symbol]

        return True

    def update_portfolio_value(self, date, market_prices):
        """
        Update portfolio value based on current market prices
        """
        position_value = 0

        for symbol, position in self.positions.items():
            if symbol in market_prices:
                current_price = market_prices[symbol]
                position_value += position['quantity'] * current_price

        self.portfolio_value = self.cash + position_value

        # Record equity curve
        self.equity_curve.append({
            'date': date,
            'portfolio_value': self.portfolio_value,
            'cash': self.cash,
            'position_value': position_value
        })

        # Calculate daily return
        if len(self.equity_curve) > 1:
            prev_value = self.equity_curve[-2]['portfolio_value']
            daily_return = (self.portfolio_value - prev_value) / prev_value
            self.daily_returns.append(daily_return)

    def get_results(self):
        """
        Calculate and return backtest results
        """
        if not self.trades:
            return {}

        # Convert to DataFrames
        trades_df = pd.DataFrame(self.trades)
        equity_df = pd.DataFrame(self.equity_curve)
        returns_series = pd.Series(self.daily_returns)

        # Calculate performance metrics
        total_return = (self.portfolio_value - self.initial_capital) / self.ini-
tial_capital

        if len(returns_series) > 0:
            annualized_return = (1 + returns_series.mean()) ** 252 - 1
            volatility = returns_series.std() * np.sqrt(252)
            sharpe_ratio = annualized_return / volatility if volatility > 0 else 0

            # Drawdown calculation
            cumulative_returns = (1 + returns_series).cumprod()
            running_max = cumulative_returns.expanding().max()
            drawdown = (cumulative_returns - running_max) / running_max
            max_drawdown = drawdown.min()

            # Win/Loss statistics
            winning_trades = trades_df[trades_df['pnl'] > 0] if 'pnl' in trades_df.colu
```

```
mns else pd.DataFrame()
            losing_trades = trades_df[trades_df['pnl'] < 0] if 'pnl' in trades_df.colum
ns else pd.DataFrame()

            win_rate = len(winning_trades) / len(trades_df) if len(trades_df) > 0 else
0
            avg_win = winning_trades['pnl'].mean() if len(winning_trades) > 0 else 0
            avg_loss = losing_trades['pnl'].mean() if len(losing_trades) > 0 else 0
            profit_factor = abs(winning_trades['pnl'].sum() / losing_trades['pnl'].sum(
)) if len(losing_trades) > 0 and losing_trades['pnl'].sum() != 0 else np.inf
        else:
            annualized_return = volatility = sharpe_ratio = max_drawdown = 0
            win_rate = avg_win = avg_loss = profit_factor = 0

        results = {
            'total_return': total_return,
            'annualized_return': annualized_return,
            'volatility': volatility,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': max_drawdown,
            'win_rate': win_rate,
            'avg_win': avg_win,
            'avg_loss': avg_loss,
            'profit_factor': profit_factor,
            'total_trades': len(trades_df),
            'final_portfolio_value': self.portfolio_value,
            'trades': trades_df,
            'equity_curve': equity_df,
            'daily_returns': returns_series
        }

        return results
```

Error Handling and Logging

1. Comprehensive Error Handling

```python
# File: trading_system/utils/exceptions.py
class TradingSystemException(Exception):
    """Base exception for trading system"""
    pass

class DataException(TradingSystemException):
    """Exception for data-related errors"""
    pass

class StrategyException(TradingSystemException):
    """Exception for strategy-related errors"""
    pass

class ExecutionException(TradingSystemException):
    """Exception for execution-related errors"""
    pass

class RiskException(TradingSystemException):
    """Exception for risk management errors"""
    pass

# File: trading_system/utils/logging.py
import logging
import sys
from datetime import datetime

def setup_logging(log_level=logging.INFO, log_file=None):
    """
    Setup comprehensive logging for the trading system
    """
    # Create formatter
    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # Create console handler
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setFormatter(formatter)

    # Create file handler if specified
    handlers = [console_handler]
    if log_file:
        file_handler = logging.FileHandler(log_file)
        file_handler.setFormatter(formatter)
        handlers.append(file_handler)

    # Configure root logger
    logging.basicConfig(
        level=log_level,
        handlers=handlers
    )

    return logging.getLogger(__name__)

# Usage in strategy classes
import logging
from .utils.exceptions import StrategyException

class SafeStrategy(BaseStrategy):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.logger = logging.getLogger(f"{__name__}.{self.__class__.__name__}")
```

```
    def generate_signals(self, data):
        try:
            self.logger.info(f"Generating signals for {len(data)} data points")

            # Validate input data
            if data.empty:
                raise StrategyException("Empty data provided to strategy")

            required_columns = ['open', 'high', 'low', 'close', 'volume']
            missing_columns = [col for col in required_columns if col not in data.colum
ns]
            if missing_columns:
                raise StrategyException(f"Missing required columns: {missing_columns}")

            # Generate signals with error handling
            signals = self._safe_signal_generation(data)

            self.logger.info(f"Generated {signals['entry'].sum()} entry signals")
            return signals

        except Exception as e:
            self.logger.error(f"Error generating signals: {str(e)}")
            raise StrategyException(f"Signal generation failed: {str(e)}")

    def _safe_signal_generation(self, data):
        """
        Signal generation with comprehensive error handling
        """
        try:
            # Your signal generation logic here
            signals = pd.DataFrame(index=data.index)
            # ... implementation
            return signals

        except ZeroDivisionError as e:
            self.logger.error(f"Division by zero in signal calculation: {e}")
            raise
        except KeyError as e:
            self.logger.error(f"Missing data column: {e}")
            raise
        except Exception as e:
            self.logger.error(f"Unexpected error in signal generation: {e}")
            raise
```

Testing Framework

1. Unit Testing for Strategies

```
# File: tests/test_strategies.py
import unittest
import pandas as pd
import numpy as np
from trading_system.strategies.technical import MovingAverageCrossover, MeanReversion-
Strategy

class TestMovingAverageCrossover(unittest.TestCase):

    def setUp(self):
        """Set up test data"""
        dates = pd.date_range('2020-01-01', periods=100, freq='D')
        np.random.seed(42)

        # Create synthetic price data
        prices = 100 + np.cumsum(np.random.randn(100) * 0.01)

        self.test_data = pd.DataFrame({
            'date': dates,
            'open': prices * (1 + np.random.randn(100) * 0.001),
            'high': prices * (1 + abs(np.random.randn(100)) * 0.002),
            'low': prices * (1 - abs(np.random.randn(100)) * 0.002),
            'close': prices,
            'volume': np.random.randint(1000, 10000, 100)
        }).set_index('date')

        self.strategy = MovingAverageCrossover(fast_period=5, slow_period=20)

    def test_signal_generation(self):
        """Test that signals are generated correctly"""
        signals = self.strategy.generate_signals(self.test_data)

        # Check that signals DataFrame has correct structure
        self.assertIsInstance(signals, pd.DataFrame)
        self.assertIn('entry', signals.columns)
        self.assertIn('exit', signals.columns)
        self.assertIn('position', signals.columns)

        # Check that signals are boolean/numeric
        self.assertTrue(signals['entry'].dtype == bool)
        self.assertTrue(signals['exit'].dtype == bool)

        # Check that we have some signals
        self.assertGreater(signals['entry'].sum() + signals['exit'].sum(), 0)

    def test_position_sizing(self):
        """Test position sizing calculation"""
        portfolio_value = 100000
        current_price = 100

        position_size = self.strategy.calculate_position_size(
            None, current_price, portfolio_value
        )

        # Check that position size is reasonable
        self.assertGreater(position_size, 0)
        self.assertLess(position_size * current_price, portfolio_value * 0.1)  # Max
10% position

    def test_empty_data_handling(self):
        """Test handling of empty data"""
        empty_data = pd.DataFrame()
```

```
        with self.assertRaises(Exception):
            self.strategy.generate_signals(empty_data)

    def test_insufficient_data(self):
        """Test handling of insufficient data"""
        # Data with fewer points than slow MA period
        short_data = self.test_data.head(10)

        signals = self.strategy.generate_signals(short_data)

        # Should handle gracefully (may have NaN values)
        self.assertIsInstance(signals, pd.DataFrame)

class TestBacktestEngine(unittest.TestCase):

    def setUp(self):
        """Set up test environment"""
        from trading_system.backtesting.engine import BacktestEngine
        self.engine = BacktestEngine(initial_capital=100000)

    def test_position_entry(self):
        """Test entering positions"""
        success = self.engine.enter_position('2020-01-01', 'TEST', 100, 50.0)

        self.assertTrue(success)
        self.assertIn('TEST', self.engine.positions)
        self.assertEqual(self.engine.positions['TEST']['quantity'], 100)
        self.assertLess(self.engine.cash, 100000)  # Cash should be reduced

    def test_position_exit(self):
        """Test exiting positions"""
        # First enter a position
        self.engine.enter_position('2020-01-01', 'TEST', 100, 50.0)

        # Then exit it
        success = self.engine.exit_position('2020-01-02', 'TEST', 55.0)

        self.assertTrue(success)
        self.assertNotIn('TEST', self.engine.positions)

        # Check that we made a profit
        self.assertGreater(self.engine.cash, 100000 - 100 * 50.0 * (1 + self.engine.com
mission))

    def test_insufficient_funds(self):
        """Test handling of insufficient funds"""
        # Try to buy more than we can afford
        success = self.engine.enter_position('2020-01-01', 'TEST', 10000, 50.0)

        self.assertFalse(success)
        self.assertNotIn('TEST', self.engine.positions)

if __name__ == '__main__':
    unittest.main()
```

2. Integration Testing

```python
# File: tests/test_integration.py
import unittest
import pandas as pd
import numpy as np
from trading_system import DataManager, MovingAverageCrossover, BacktestEngine

class TestIntegration(unittest.TestCase):

    def setUp(self):
        """Set up integration test environment"""
        self.data_manager = DataManager(":memory:")  # In-memory database for testing
        self.strategy = MovingAverageCrossover(fast_period=5, slow_period=20)

        # Create test data
        dates = pd.date_range('2020-01-01', periods=252, freq='D')
        np.random.seed(42)
        prices = 100 + np.cumsum(np.random.randn(252) * 0.01)

        self.test_data = pd.DataFrame({
            'symbol': 'TEST',
            'date': dates,
            'open': prices * (1 + np.random.randn(252) * 0.001),
            'high': prices * (1 + abs(np.random.randn(252)) * 0.002),
            'low': prices * (1 - abs(np.random.randn(252)) * 0.002),
            'close': prices,
            'volume': np.random.randint(1000, 10000, 252),
            'adjusted_close': prices
        })

    def test_full_backtest_pipeline(self):
        """Test complete backtest pipeline"""
        # Store test data
        self.data_manager._store_price_data(self.test_data)

        # Retrieve data with technical indicators
        data = self.data_manager.get_price_data('TEST', '2020-01-01', '2020-12-31')
        data = self.data_manager.add_technical_indicators(data)

        # Run backtest
        results = self.strategy.backtest(data, initial_capital=100000)

        # Verify results structure
        self.assertIsInstance(results, dict)
        self.assertIn('total_return', results)
        self.assertIn('sharpe_ratio', results)
        self.assertIn('max_drawdown', results)
        self.assertIn('trades', results)

        # Verify reasonable results
        self.assertIsInstance(results['total_return'], (int, float))
        self.assertIsInstance(results['sharpe_ratio'], (int, float))

    def test_strategy_robustness(self):
        """Test strategy with various market conditions"""
        # Test with trending market
        trending_data = self.test_data.copy()
        trending_data['close'] = 100 + np.cumsum(np.ones(252) * 0.001)  # Uptrend

        results_trend = self.strategy.backtest(trending_data)

        # Test with sideways market
        sideways_data = self.test_data.copy()
```

```
        sideways_data['close'] = 100 + np.random.randn(252) * 0.01  # Sideways

        results_sideways = self.strategy.backtest(sideways_data)

        # Both should complete without errors
        self.assertIsInstance(results_trend, dict)
        self.assertIsInstance(results_sideways, dict)

if __name__ == '__main__':
    unittest.main()
```

Robustness Testing and Validation

Monte Carlo Analysis Framework

1. Monte Carlo Simulation Implementation

```
def monte_carlo_analysis(strategy_returns, num_simulations=10000, confid-
ence_levels=[0.05, 0.95]):
    """
    Perform Monte Carlo analysis on strategy returns

    Parameters:
    -----------
    strategy_returns : pd.Series
        Historical strategy returns
    num_simulations : int
        Number of Monte Carlo simulations
    confidence_levels : list
        Confidence levels for VaR calculation

    Returns:
    --------
    dict : Monte Carlo analysis results
    """
    results = {
        'simulations': [],
        'statistics': {},
        'var_estimates': {},
        'drawdown_distribution': []
    }

    # Parameters for simulation
    mean_return = strategy_returns.mean()
    std_return = strategy_returns.std()
    num_periods = len(strategy_returns)

    for sim in range(num_simulations):
        # Generate random returns
        simulated_returns = np.random.normal(mean_return, std_return, num_periods)

        # Calculate cumulative performance
        cumulative_returns = (1 + pd.Series(simulated_returns)).cumprod()

        # Calculate metrics for this simulation
        total_return = cumulative_returns.iloc[-1] - 1
        volatility = pd.Series(simulated_returns).std() * np.sqrt(252)
        sharpe_ratio = (pd.Series(simulated_returns).mean() * 252) / volatility if
volatility > 0 else 0

        # Calculate maximum drawdown
        running_max = cumulative_returns.expanding().max()
        drawdown = (cumulative_returns - running_max) / running_max
        max_drawdown = drawdown.min()

        results['simulations'].append({
            'total_return': total_return,
            'volatility': volatility,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': max_drawdown
        })

        results['drawdown_distribution'].append(max_drawdown)

    # Calculate statistics
    sim_df = pd.DataFrame(results['simulations'])

    results['statistics'] = {
        'mean_return': sim_df['total_return'].mean(),
```

```
        'std_return': sim_df['total_return'].std(),
        'mean_sharpe': sim_df['sharpe_ratio'].mean(),
        'std_sharpe': sim_df['sharpe_ratio'].std(),
        'mean_max_dd': sim_df['max_drawdown'].mean(),
        'std_max_dd': sim_df['max_drawdown'].std()
    }

    # Calculate VaR estimates
    for confidence_level in confidence_levels:
        results['var_estimates'][f'var_{confidence_level}'] = np.percentile(
            sim_df['total_return'], confidence_level * 100
        )

    return results

def bootstrap_analysis(strategy_returns, num_bootstrap=10000, block_size=21):
    """
    Perform bootstrap analysis to estimate parameter uncertainty

    Parameters:
    -----------
    strategy_returns : pd.Series
        Historical strategy returns
    num_bootstrap : int
        Number of bootstrap samples
    block_size : int
        Block size for block bootstrap (to preserve autocorrelation)

    Returns:
    --------
    dict : Bootstrap analysis results
    """
    bootstrap_results = []

    for _ in range(num_bootstrap):
        # Block bootstrap to preserve time series structure
        bootstrap_sample = block_bootstrap_sample(strategy_returns, block_size)

        # Calculate metrics for bootstrap sample
        total_return = (1 + bootstrap_sample).prod() - 1
        annualized_return = (1 + bootstrap_sample.mean()) ** 252 - 1
        volatility = bootstrap_sample.std() * np.sqrt(252)
        sharpe_ratio = annualized_return / volatility if volatility > 0 else 0

        # Calculate maximum drawdown
        cumulative = (1 + bootstrap_sample).cumprod()
        running_max = cumulative.expanding().max()
        drawdown = (cumulative - running_max) / running_max
        max_drawdown = drawdown.min()

        bootstrap_results.append({
            'total_return': total_return,
            'annualized_return': annualized_return,
            'volatility': volatility,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': max_drawdown
        })

    # Convert to DataFrame for analysis
    bootstrap_df = pd.DataFrame(bootstrap_results)

    # Calculate confidence intervals
    confidence_intervals = {}
```

```
    for metric in bootstrap_df.columns:
        confidence_intervals[metric] = {
            'mean': bootstrap_df[metric].mean(),
            'std': bootstrap_df[metric].std(),
            'ci_5': np.percentile(bootstrap_df[metric], 5),
            'ci_95': np.percentile(bootstrap_df[metric], 95)
        }

    return {
        'bootstrap_samples': bootstrap_df,
        'confidence_intervals': confidence_intervals
    }

def block_bootstrap_sample(data, block_size):
    """
    Generate block bootstrap sample to preserve autocorrelation
    """
    n = len(data)
    num_blocks = int(np.ceil(n / block_size))

    # Generate random starting points for blocks
    start_indices = np.random.randint(0, n - block_size + 1, num_blocks)

    # Create bootstrap sample
    bootstrap_sample = []
    for start_idx in start_indices:
        block = data.iloc[start_idx:start_idx + block_size]
        bootstrap_sample.extend(block.values)

    # Trim to original length
    return pd.Series(bootstrap_sample[:n])
```

2. Noise Testing Framework

```
def noise_test_analysis(strategy, data, noise_levels=[0.001, 0.005, 0.01, 0.02],
num_tests=100):
    """
    Test strategy robustness by adding noise to price data

    Parameters:
    -----------
    strategy : BaseStrategy
        Trading strategy to test
    data : pd.DataFrame
        Historical price data
    noise_levels : list
        Levels of noise to add (as fraction of price)
    num_tests : int
        Number of noise tests per level

    Returns:
    --------
    dict : Noise test results
    """
    results = {}

    # Baseline performance (no noise)
    baseline_results = strategy.backtest(data)
    baseline_sharpe = baseline_results['sharpe_ratio']
    baseline_return = baseline_results['total_return']

    results['baseline'] = baseline_results
    results['noise_tests'] = {}

    for noise_level in noise_levels:
        noise_results = []

        for test_num in range(num_tests):
            # Add noise to price data
            noisy_data = add_price_noise(data, noise_level)

            # Run backtest on noisy data
            test_results = strategy.backtest(noisy_data)

            noise_results.append({
                'test_num': test_num,
                'total_return': test_results['total_return'],
                'sharpe_ratio': test_results['sharpe_ratio'],
                'max_drawdown': test_results['max_drawdown'],
                'return_degradation': (test_results['total_return'] -
baseline_return) / baseline_return,
                'sharpe_degradation': (test_results['sharpe_ratio'] -
baseline_sharpe) / baseline_sharpe if baseline_sharpe != 0 else 0
            })

        # Analyze results for this noise level
        noise_df = pd.DataFrame(noise_results)

        results['noise_tests'][noise_level] = {
            'individual_results': noise_df,
            'summary_stats': {
                'mean_return_degradation': noise_df['return_degradation'].mean(),
                'std_return_degradation': noise_df['return_degradation'].std(),
                'mean_sharpe_degradation': noise_df['sharpe_degradation'].mean(),
                'std_sharpe_degradation': noise_df['sharpe_degradation'].std(),
                'pct_positive_returns': (noise_df['total_return'] > 0).mean(),
```

```
                'pct_better_than_baseline': (noise_df['total_return'] >
baseline_return).mean()
            }
        }

    return results

def add_price_noise(data, noise_level, seed=None):
    """
    Add random noise to price data

    Parameters:
    -----------
    data : pd.DataFrame
        Original price data
    noise_level : float
        Noise level as fraction of price
    seed : int, optional
        Random seed for reproducibility

    Returns:
    --------
    pd.DataFrame : Noisy price data
    """
    if seed is not None:
        np.random.seed(seed)

    noisy_data = data.copy()

    # Add noise to OHLC prices
    for col in ['open', 'high', 'low', 'close']:
        if col in data.columns:
            noise = np.random.normal(0, noise_level, len(data))
            noisy_data[col] = data[col] * (1 + noise)

    # Ensure high >= low and other price relationships
    noisy_data['high'] = np.maximum(noisy_data['high'], noisy_data[['open', 'close']].m
ax(axis=1))
    noisy_data['low'] = np.minimum(noisy_data['low'], noisy_data[['open', 'close']].min
(axis=1))

    return noisy_data
```

3. Walk-Forward Analysis

```
def walk_forward_analysis(strategy, data, train_periods=252, test_periods=21,
step_size=21):
    """
    Perform walk-forward analysis to test strategy robustness over time

    Parameters:
    -----------
    strategy : BaseStrategy
        Trading strategy to test
    data : pd.DataFrame
        Historical data
    train_periods : int
        Number of periods for training window
    test_periods : int
        Number of periods for testing window
    step_size : int
        Step size for rolling window

    Returns:
    --------
    dict : Walk-forward analysis results
    """
    results = []

    for start_idx in range(0, len(data) - train_periods - test_periods, step_size):
        # Define training and testing periods
        train_start = start_idx
        train_end = start_idx + train_periods
        test_start = train_end
        test_end = test_start + test_periods

        # Extract training and testing data
        train_data = data.iloc[train_start:train_end]
        test_data = data.iloc[test_start:test_end]

        # Optimize strategy on training data (if applicable)
        optimized_strategy = optimize_strategy_parameters(strategy, train_data)

        # Test on out-of-sample data
        test_results = optimized_strategy.backtest(test_data)

        results.append({
            'train_period': (train_data.index[0], train_data.index[-1]),
            'test_period': (test_data.index[0], test_data.index[-1]),
            'train_performance': optimized_strategy.backtest(train_data),
            'test_performance': test_results,
            'parameters': optimized_strategy.parameters
        })

    # Analyze walk-forward results
    analysis = analyze_walk_forward_results(results)

    return {
        'individual_results': results,
        'analysis': analysis
    }

def optimize_strategy_parameters(strategy, data, parameter_ranges=None):
    """
    Optimize strategy parameters using training data

    Parameters:
```

```
    ----------
    strategy : BaseStrategy
        Strategy to optimize
    data : pd.DataFrame
        Training data
    parameter_ranges : dict
        Ranges for parameter optimization

    Returns:
    --------
    BaseStrategy : Optimized strategy
    """
    if parameter_ranges is None:
        # Default parameter ranges for common strategies
        parameter_ranges = {
            'fast_period': range(5, 21, 2),
            'slow_period': range(20, 51, 5),
            'rsi_period': range(10, 31, 2),
            'bb_period': range(15, 31, 2)
        }

    best_performance = -np.inf
    best_parameters = {}

    # Grid search over parameter combinations
    from itertools import product

    param_names = list(parameter_ranges.keys())
    param_values = list(parameter_ranges.values())

    for param_combination in product(*param_values):
        # Create parameter dictionary
        params = dict(zip(param_names, param_combination))

        # Create strategy with these parameters
        test_strategy = strategy.__class__(**params)

        try:
            # Backtest with these parameters
            results = test_strategy.backtest(data)

            # Use Sharpe ratio as optimization criterion
            performance = results['sharpe_ratio']

            if performance > best_performance:
                best_performance = performance
                best_parameters = params

        except Exception as e:
            # Skip parameter combinations that cause errors
            continue

    # Return strategy with best parameters
    return strategy.__class__(**best_parameters)

def analyze_walk_forward_results(results):
    """
    Analyze walk-forward test results
    """
    # Extract performance metrics
    in_sample_returns = [r['train_performance']['total_return'] for r in results]
    out_sample_returns = [r['test_performance']['total_return'] for r in results]
```

```
    in_sample_sharpe = [r['train_performance']['sharpe_ratio'] for r in results]
    out_sample_sharpe = [r['test_performance']['sharpe_ratio'] for r in results]

    # Calculate degradation metrics
    return_degradation = [(oos - ins) / ins if ins != 0 else 0
                          for ins, oos in zip(in_sample_returns, out_sample_returns)]

    sharpe_degradation = [(oos - ins) / ins if ins != 0 else 0
                          for ins, oos in zip(in_sample_sharpe, out_sample_sharpe)]

    analysis = {
        'in_sample_stats': {
            'mean_return': np.mean(in_sample_returns),
            'std_return': np.std(in_sample_returns),
            'mean_sharpe': np.mean(in_sample_sharpe),
            'std_sharpe': np.std(in_sample_sharpe)
        },
        'out_sample_stats': {
            'mean_return': np.mean(out_sample_returns),
            'std_return': np.std(out_sample_returns),
            'mean_sharpe': np.mean(out_sample_sharpe),
            'std_sharpe': np.std(out_sample_sharpe)
        },
        'degradation_stats': {
            'mean_return_degradation': np.mean(return_degradation),
            'std_return_degradation': np.std(return_degradation),
            'mean_sharpe_degradation': np.mean(sharpe_degradation),
            'std_sharpe_degradation': np.std(sharpe_degradation)
        },
        'consistency_metrics': {
            'pct_positive_oos_returns': np.mean([r > 0 for r in out_sample_returns]),
            'pct_positive_oos_sharpe': np.mean([s > 0 for s in out_sample_sharpe]),
            'correlation_in_out': np.corrcoef(in_sample_returns, out_sample_returns)[0,
1]
        }
    }

    return analysis
```

Statistical Significance Testing

1. Hypothesis Testing Framework

```
def statistical_significance_tests(strategy_returns, benchmark_returns=None, al-
pha=0.05):
    """
    Perform comprehensive statistical significance tests

    Parameters:
    -----------
    strategy_returns : pd.Series
        Strategy returns
    benchmark_returns : pd.Series, optional
        Benchmark returns for comparison
    alpha : float
        Significance level

    Returns:
    --------
    dict : Statistical test results
    """
    from scipy import stats

    results = {}

    # Test 1: T-test for mean return different from zero
    t_stat, p_value = stats.ttest_1samp(strategy_returns, 0)
    results['mean_return_test'] = {
        't_statistic': t_stat,
        'p_value': p_value,
        'is_significant': p_value < alpha,
        'interpretation': 'Mean return significantly different from zero' if p_value <
alpha else 'Mean return not significantly different from zero'
    }

    # Test 2: Normality test (Jarque-Bera)
    jb_stat, jb_p_value = stats.jarque_bera(strategy_returns)
    results['normality_test'] = {
        'jarque_bera_stat': jb_stat,
        'p_value': jb_p_value,
        'is_normal': jb_p_value > alpha,
        'interpretation': 'Returns are normally distributed' if jb_p_value > alpha else
'Returns are not normally distributed'
    }

    # Test 3: Autocorrelation test (Ljung-Box)
    from statsmodels.stats.diagnostic import acorr_ljungbox
    lb_result = acorr_ljungbox(strategy_returns, lags=10, return_df=True)
    results['autocorrelation_test'] = {
        'ljung_box_stats': lb_result,
        'has_autocorrelation': (lb_result['lb_pvalue'] < alpha).any(),
        'interpretation': 'Significant autocorrelation detected' if (lb_result['lb_pval
ue'] < alpha).any() else 'No significant autocorrelation'
    }

    # Test 4: Stationarity test (Augmented Dickey-Fuller)
    from statsmodels.tsa.stattools import adfuller
    adf_result = adfuller(strategy_returns)
    results['stationarity_test'] = {
        'adf_statistic': adf_result[0],
        'p_value': adf_result[1],
        'is_stationary': adf_result[1] < alpha,
        'critical_values': adf_result[4],
        'interpretation': 'Returns are stationary' if adf_result[1] < alpha else 'Re-
turns are non-stationary'
```

```
    }

    # Test 5: Heteroscedasticity test (ARCH test)
    from statsmodels.stats.diagnostic import het_arch
    arch_stat, arch_p_value, _, _ = het_arch(strategy_returns)
    results['heteroscedasticity_test'] = {
        'arch_statistic': arch_stat,
        'p_value': arch_p_value,
        'has_arch_effects': arch_p_value < alpha,
        'interpretation': 'ARCH effects detected (heteroscedasticity)' if arch_p_value
< alpha else 'No ARCH effects (homoscedasticity)'
    }

    # Benchmark comparison tests (if benchmark provided)
    if benchmark_returns is not None:
        # Test 6: Paired t-test for outperformance
        excess_returns = strategy_returns - benchmark_returns
        t_stat_excess, p_value_excess = stats.ttest_1samp(excess_returns, 0)

        results['outperformance_test'] = {
            't_statistic': t_stat_excess,
            'p_value': p_value_excess,
            'outperforms_benchmark': (t_stat_excess > 0) and (p_value_excess < alpha),
            'interpretation': 'Strategy significantly outperforms benchmark' if
(t_stat_excess > 0) and (p_value_excess < alpha) else 'No significant outperformance'
        }

        # Test 7: Variance ratio test
        var_ratio = strategy_returns.var() / benchmark_returns.var()
        f_stat = var_ratio
        f_p_value = 2 * min(stats.f.cdf(f_stat, len(strategy_returns)-1, len(bench-
mark_returns)-1),
                            1 - stats.f.cdf(f_stat, len(strategy_returns)-1, len(bench-
mark_returns)-1))

        results['variance_ratio_test'] = {
            'variance_ratio': var_ratio,
            'f_statistic': f_stat,
            'p_value': f_p_value,
            'equal_variance': f_p_value > alpha,
            'interpretation': 'Equal variance' if f_p_value > alpha else 'Unequal vari-
ance'
        }

    return results

def multiple_testing_correction(p_values, method='bonferroni'):
    """
    Apply multiple testing correction to p-values

    Parameters:
    -----------
    p_values : list
        List of p-values
    method : str
        Correction method ('bonferroni', 'holm', 'fdr_bh')

    Returns:
    --------
    list : Corrected p-values
    """
    from statsmodels.stats.multitest import multipletests
```

```
    rejected, corrected_p_values, _, _ = multipletests(p_values, method=method)

    return {
        'original_p_values': p_values,
        'corrected_p_values': corrected_p_values,
        'rejected_hypotheses': rejected,
        'method': method
    }
```

2. Performance Stability Analysis

```
def performance_stability_analysis(strategy_returns, window_size=63):
    """
    Analyze performance stability over time

    Parameters:
    -----------
    strategy_returns : pd.Series
        Strategy returns with datetime index
    window_size : int
        Rolling window size for stability analysis

    Returns:
    --------
    dict : Stability analysis results
    """
    # Rolling performance metrics
    rolling_returns = strategy_returns.rolling(window=window_size).mean() * 252
    rolling_volatility = strategy_returns.rolling(window=window_size).std() * np.sqrt(2
52)
    rolling_sharpe = rolling_returns / rolling_volatility

    # Calculate stability metrics
    stability_metrics = {
        'return_stability': {
            'mean': rolling_returns.mean(),
            'std': rolling_returns.std(),
            'coefficient_of_variation': rolling_returns.std() / rolling_returns.mean()
if rolling_returns.mean() != 0 else np.inf,
            'min': rolling_returns.min(),
            'max': rolling_returns.max()
        },
        'volatility_stability': {
            'mean': rolling_volatility.mean(),
            'std': rolling_volatility.std(),
            'coefficient_of_variation': rolling_volatility.std() / rolling_volatility.m
ean() if rolling_volatility.mean() != 0 else np.inf,
            'min': rolling_volatility.min(),
            'max': rolling_volatility.max()
        },
        'sharpe_stability': {
            'mean': rolling_sharpe.mean(),
            'std': rolling_sharpe.std(),
            'coefficient_of_variation': rolling_sharpe.std() / rolling_sharpe.mean() if
rolling_sharpe.mean() != 0 else np.inf,
            'min': rolling_sharpe.min(),
            'max': rolling_sharpe.max(),
            'pct_positive': (rolling_sharpe > 0).mean()
        }
    }

    # Regime analysis
    regime_analysis = analyze_performance_regimes(strategy_returns, rolling_sharpe)

    # Drawdown analysis
    drawdown_analysis = analyze_drawdown_patterns(strategy_returns)

    return {
        'rolling_metrics': {
            'returns': rolling_returns,
            'volatility': rolling_volatility,
            'sharpe': rolling_sharpe
        },
```

```
        'stability_metrics': stability_metrics,
        'regime_analysis': regime_analysis,
        'drawdown_analysis': drawdown_analysis
    }

def analyze_performance_regimes(returns, rolling_sharpe, threshold=0.5):
    """
    Identify and analyze performance regimes
    """
    # Define regimes based on rolling Sharpe ratio
    good_regime = rolling_sharpe > threshold
    bad_regime = rolling_sharpe < -threshold
    neutral_regime = (~good_regime) & (~bad_regime)

    regime_stats = {
        'good_regime': {
            'frequency': good_regime.mean(),
            'avg_duration': calculate_regime_duration(good_regime),
            'avg_return': returns[good_regime].mean() * 252,
            'avg_volatility': returns[good_regime].std() * np.sqrt(252)
        },
        'bad_regime': {
            'frequency': bad_regime.mean(),
            'avg_duration': calculate_regime_duration(bad_regime),
            'avg_return': returns[bad_regime].mean() * 252,
            'avg_volatility': returns[bad_regime].std() * np.sqrt(252)
        },
        'neutral_regime': {
            'frequency': neutral_regime.mean(),
            'avg_duration': calculate_regime_duration(neutral_regime),
            'avg_return': returns[neutral_regime].mean() * 252,
            'avg_volatility': returns[neutral_regime].std() * np.sqrt(252)
        }
    }

    return regime_stats

def calculate_regime_duration(regime_indicator):
    """
    Calculate average duration of regime periods
    """
    durations = []
    current_duration = 0

    for is_regime in regime_indicator:
        if is_regime:
            current_duration += 1
        else:
            if current_duration > 0:
                durations.append(current_duration)
                current_duration = 0

    # Add final duration if regime continues to end
    if current_duration > 0:
        durations.append(current_duration)

    return np.mean(durations) if durations else 0

def analyze_drawdown_patterns(returns):
    """
    Analyze drawdown patterns and recovery characteristics
    """
    # Calculate drawdowns
```

```
    cumulative_returns = (1 + returns).cumprod()
    running_max = cumulative_returns.expanding().max()
    drawdown = (cumulative_returns - running_max) / running_max

    # Identify drawdown periods
    in_drawdown = drawdown < 0
    drawdown_periods = []

    start_idx = None
    for i, is_dd in enumerate(in_drawdown):
        if is_dd and start_idx is None:
            start_idx = i
        elif not is_dd and start_idx is not None:
            drawdown_periods.append({
                'start': start_idx,
                'end': i - 1,
                'duration': i - start_idx,
                'magnitude': drawdown.iloc[start_idx:i].min(),
                'recovery_time': None  # Will be calculated separately
            })
            start_idx = None

    # Calculate recovery times
    for period in drawdown_periods:
        recovery_start = period['end'] + 1
        if recovery_start < len(cumulative_returns):
            target_value = cumulative_returns.iloc[period['start']]
            recovery_idx = None

            for i in range(recovery_start, len(cumulative_returns)):
                if cumulative_returns.iloc[i] >= target_value:
                    recovery_idx = i
                    break

            if recovery_idx is not None:
                period['recovery_time'] = recovery_idx - period['end']

    # Calculate summary statistics
    if drawdown_periods:
        drawdown_stats = {
            'num_drawdowns': len(drawdown_periods),
            'avg_duration': np.mean([p['duration'] for p in drawdown_periods]),
            'avg_magnitude': np.mean([p['magnitude'] for p in drawdown_periods]),
            'max_drawdown': min([p['magnitude'] for p in drawdown_periods]),
            'avg_recovery_time': np.mean([p['recovery_time'] for p in drawdown_periods
if p['recovery_time'] is not None]),
            'drawdown_frequency': len(drawdown_periods) / len(returns) * 252  # Annual-
ized
        }
    else:
        drawdown_stats = {
            'num_drawdowns': 0,
            'avg_duration': 0,
            'avg_magnitude': 0,
            'max_drawdown': 0,
            'avg_recovery_time': 0,
            'drawdown_frequency': 0
        }

    return {
        'drawdown_periods': drawdown_periods,
        'summary_stats': drawdown_stats,
```

```
        'drawdown_series': drawdown
    }
```

Portfolio Construction and Management

Modern Portfolio Theory Implementation

1. Mean-Variance Optimization

```
import numpy as np
import pandas as pd
from scipy.optimize import minimize
import cvxpy as cp

class PortfolioOptimizer:
    """
    Modern Portfolio Theory implementation with various optimization methods
    """

    def __init__(self, returns_data, risk_free_rate=0.02):
        """
        Initialize portfolio optimizer

        Parameters:
        -----------
        returns_data : pd.DataFrame
            Historical returns data with assets as columns
        risk_free_rate : float
            Risk-free rate for Sharpe ratio calculation
        """
        self.returns = returns_data
        self.risk_free_rate = risk_free_rate
        self.mean_returns = returns_data.mean() * 252  # Annualized
        self.cov_matrix = returns_data.cov() * 252  # Annualized
        self.num_assets = len(returns_data.columns)

    def mean_variance_optimization(self, target_return=None, risk_aversion=1.0):
        """
        Perform mean-variance optimization

        Parameters:
        -----------
        target_return : float, optional
            Target portfolio return
        risk_aversion : float
            Risk aversion parameter (higher = more risk averse)

        Returns:
        --------
        dict : Optimization results
        """
        # Define optimization variables
        weights = cp.Variable(self.num_assets)

        # Portfolio return and risk
        portfolio_return = self.mean_returns.values @ weights
        portfolio_risk = cp.quad_form(weights, self.cov_matrix.values)

        # Constraints
        constraints = [
            cp.sum(weights) == 1,  # Weights sum to 1
            weights >= 0  # Long-only constraint
        ]

        if target_return is not None:
            # Minimize risk for target return
            objective = cp.Minimize(portfolio_risk)
            constraints.append(portfolio_return >= target_return)
        else:
            # Maximize utility (return - risk_aversion * risk)
            objective = cp.Maximize(portfolio_return - risk_aversion * portfolio_risk)
```

```
        # Solve optimization problem
        problem = cp.Problem(objective, constraints)
        problem.solve()

        if problem.status == 'optimal':
            optimal_weights = pd.Series(weights.value, index=self.returns.columns)

            # Calculate portfolio metrics
            portfolio_return_val = (optimal_weights * self.mean_returns).sum()
            portfolio_risk_val = np.sqrt(optimal_weights.T @ self.cov_matrix @ optim-
al_weights)
            sharpe_ratio = (portfolio_return_val - self.risk_free_rate) / portfo-
lio_risk_val

            return {
                'weights': optimal_weights,
                'expected_return': portfolio_return_val,
                'volatility': portfolio_risk_val,
                'sharpe_ratio': sharpe_ratio,
                'status': 'optimal'
            }
        else:
            return {'status': 'failed', 'message': problem.status}

    def efficient_frontier(self, num_points=50):
        """
        Generate efficient frontier

        Parameters:
        -----------
        num_points : int
            Number of points on the efficient frontier

        Returns:
        --------
        pd.DataFrame : Efficient frontier data
        """
        # Range of target returns
        min_return = self.mean_returns.min()
        max_return = self.mean_returns.max()
        target_returns = np.linspace(min_return, max_return, num_points)

        efficient_portfolios = []

        for target_return in target_returns:
            result = self.mean_variance_optimization(target_return=target_return)

            if result['status'] == 'optimal':
                efficient_portfolios.append({
                    'target_return': target_return,
                    'expected_return': result['expected_return'],
                    'volatility': result['volatility'],
                    'sharpe_ratio': result['sharpe_ratio'],
                    'weights': result['weights']
                })

        return pd.DataFrame(efficient_portfolios)

    def maximum_sharpe_portfolio(self):
        """
        Find portfolio with maximum Sharpe ratio
        """
```

```
        # Define optimization variables
        weights = cp.Variable(self.num_assets)

        # Portfolio metrics
        portfolio_return = self.mean_returns.values @ weights
        portfolio_risk = cp.quad_form(weights, self.cov_matrix.values)

        # Maximize Sharpe ratio (equivalent to maximizing excess return / risk)
        excess_return = portfolio_return - self.risk_free_rate

        # Use the fact that max(excess_return / sqrt(risk)) is equivalent to
        # max(excess_return^2 / risk) when excess_return > 0
        objective = cp.Maximize(cp.square(excess_return) / portfolio_risk)

        constraints = [
            cp.sum(weights) == 1,
            weights >= 0,
            excess_return >= 0.001  # Ensure positive excess return
        ]

        problem = cp.Problem(objective, constraints)
        problem.solve()

        if problem.status == 'optimal':
            optimal_weights = pd.Series(weights.value, index=self.returns.columns)

            # Calculate portfolio metrics
            portfolio_return_val = (optimal_weights * self.mean_returns).sum()
            portfolio_risk_val = np.sqrt(optimal_weights.T @ self.cov_matrix @ optim-
al_weights)
            sharpe_ratio = (portfolio_return_val - self.risk_free_rate) / portfo-
lio_risk_val

            return {
                'weights': optimal_weights,
                'expected_return': portfolio_return_val,
                'volatility': portfolio_risk_val,
                'sharpe_ratio': sharpe_ratio,
                'status': 'optimal'
            }
        else:
            return {'status': 'failed', 'message': problem.status}

    def minimum_variance_portfolio(self):
        """
        Find minimum variance portfolio
        """
        # Define optimization variables
        weights = cp.Variable(self.num_assets)

        # Minimize portfolio variance
        portfolio_risk = cp.quad_form(weights, self.cov_matrix.values)
        objective = cp.Minimize(portfolio_risk)

        constraints = [
            cp.sum(weights) == 1,
            weights >= 0
        ]

        problem = cp.Problem(objective, constraints)
        problem.solve()

        if problem.status == 'optimal':
```

```
            optimal_weights = pd.Series(weights.value, index=self.returns.columns)

            # Calculate portfolio metrics
            portfolio_return_val = (optimal_weights * self.mean_returns).sum()
            portfolio_risk_val = np.sqrt(optimal_weights.T @ self.cov_matrix @ optim-
al_weights)
            sharpe_ratio = (portfolio_return_val - self.risk_free_rate) / portfo-
lio_risk_val

            return {
                'weights': optimal_weights,
                'expected_return': portfolio_return_val,
                'volatility': portfolio_risk_val,
                'sharpe_ratio': sharpe_ratio,
                'status': 'optimal'
            }
        else:
            return {'status': 'failed', 'message': problem.status}
```

2. Risk Parity and Alternative Approaches

```python
class AlternativePortfolioMethods:
    """
    Alternative portfolio construction methods beyond mean-variance
    """

    def __init__(self, returns_data):
        self.returns = returns_data
        self.cov_matrix = returns_data.cov() * 252
        self.num_assets = len(returns_data.columns)

    def risk_parity_portfolio(self, method='equal_risk_contribution'):
        """
        Construct risk parity portfolio

        Parameters:
        -----------
        method : str
            'equal_risk_contribution' or 'inverse_volatility'

        Returns:
        --------
        dict : Risk parity portfolio results
        """
        if method == 'inverse_volatility':
            # Simple inverse volatility weighting
            volatilities = np.sqrt(np.diag(self.cov_matrix))
            weights = (1 / volatilities) / (1 / volatilities).sum()
            weights = pd.Series(weights, index=self.returns.columns)

        elif method == 'equal_risk_contribution':
            # Equal risk contribution optimization
            weights = self._optimize_equal_risk_contribution()

        # Calculate portfolio metrics
        portfolio_return = (weights * self.returns.mean() * 252).sum()
        portfolio_risk = np.sqrt(weights.T @ self.cov_matrix @ weights)

        # Calculate risk contributions
        risk_contributions = self._calculate_risk_contributions(weights)

        return {
            'weights': weights,
            'expected_return': portfolio_return,
            'volatility': portfolio_risk,
            'risk_contributions': risk_contributions,
            'method': method
        }

    def _optimize_equal_risk_contribution(self):
        """
        Optimize for equal risk contribution using numerical optimization
        """
        def risk_budget_objective(weights):
            """
            Objective function for equal risk contribution
            """
            weights = np.array(weights)
            portfolio_vol = np.sqrt(weights.T @ self.cov_matrix.values @ weights)

            # Risk contributions
            marginal_contrib = self.cov_matrix.values @ weights
            risk_contrib = weights * marginal_contrib / portfolio_vol
```

```
            # Target equal risk contribution
            target_risk_contrib = np.ones(self.num_assets) / self.num_assets

            # Sum of squared deviations from target
            return np.sum((risk_contrib - target_risk_contrib) ** 2)

        # Constraints
        constraints = [
            {'type': 'eq', 'fun': lambda x: np.sum(x) - 1},  # Weights sum to 1
        ]

        # Bounds (long-only)
        bounds = [(0, 1) for _ in range(self.num_assets)]

        # Initial guess (equal weights)
        x0 = np.ones(self.num_assets) / self.num_assets

        # Optimize
        result = minimize(
            risk_budget_objective,
            x0,
            method='SLSQP',
            bounds=bounds,
            constraints=constraints
        )

        if result.success:
            return pd.Series(result.x, index=self.returns.columns)
        else:
            # Fallback to inverse volatility if optimization fails
            volatilities = np.sqrt(np.diag(self.cov_matrix))
            weights = (1 / volatilities) / (1 / volatilities).sum()
            return pd.Series(weights, index=self.returns.columns)

    def _calculate_risk_contributions(self, weights):
        """
        Calculate risk contributions for each asset
        """
        portfolio_vol = np.sqrt(weights.T @ self.cov_matrix @ weights)
        marginal_contrib = self.cov_matrix @ weights
        risk_contrib = weights * marginal_contrib / portfolio_vol

        return pd.Series(risk_contrib, index=self.returns.columns)

    def hierarchical_risk_parity(self):
        """
        Implement Hierarchical Risk Parity (HRP) portfolio
        """
        from scipy.cluster.hierarchy import linkage, dendrogram, cut_tree
        from scipy.spatial.distance import squareform

        # Calculate distance matrix from correlation
        corr_matrix = self.returns.corr()
        distance_matrix = np.sqrt((1 - corr_matrix) / 2)

        # Hierarchical clustering
        linkage_matrix = linkage(squareform(distance_matrix), method='ward')

        # Get cluster structure
        clusters = cut_tree(linkage_matrix, n_clusters=min(5, self.num_assets))

        # Calculate weights using HRP algorithm
```

```
        weights = self._hrp_weights(distance_matrix, clusters)

        # Calculate portfolio metrics
        portfolio_return = (weights * self.returns.mean() * 252).sum()
        portfolio_risk = np.sqrt(weights.T @ self.cov_matrix @ weights)

        return {
            'weights': weights,
            'expected_return': portfolio_return,
            'volatility': portfolio_risk,
            'clusters': clusters,
            'linkage_matrix': linkage_matrix
        }

    def _hrp_weights(self, distance_matrix, clusters):
        """
        Calculate HRP weights based on hierarchical clustering
        """
        # Initialize weights
        weights = pd.Series(1.0, index=self.returns.columns)

        # Recursive bisection
        def _recursive_bisection(items):
            if len(items) == 1:
                return

            # Split items into two groups
            mid = len(items) // 2
            left_items = items[:mid]
            right_items = items[mid:]

            # Calculate cluster variances
            left_var = self._calculate_cluster_variance(left_items)
            right_var = self._calculate_cluster_variance(right_items)

            # Allocate weights inversely proportional to variance
            total_var = left_var + right_var
            left_weight = right_var / total_var
            right_weight = left_var / total_var

            # Update weights
            weights[left_items] *= left_weight
            weights[right_items] *= right_weight

            # Recurse
            _recursive_bisection(left_items)
            _recursive_bisection(right_items)

        # Start recursive bisection
        items = list(self.returns.columns)
        _recursive_bisection(items)

        return weights

    def _calculate_cluster_variance(self, items):
        """
        Calculate variance of a cluster of assets
        """
        if len(items) == 1:
            return self.cov_matrix.loc[items[0], items[0]]

        cluster_cov = self.cov_matrix.loc[items, items]
        # Equal weights within cluster
```

```
        equal_weights = np.ones(len(items)) / len(items)
        cluster_var = equal_weights.T @ cluster_cov.values @ equal_weights

        return cluster_var

    def black_litterman_portfolio(self, views_dict, confidence_dict, tau=0.05):
        """
        Implement Black-Litterman model

        Parameters:
        -----------
        views_dict : dict
            Dictionary of views {asset: expected_return}
        confidence_dict : dict
            Dictionary of confidence levels {asset: confidence}
        tau : float
            Scaling factor for uncertainty of prior

        Returns:
        --------
        dict : Black-Litterman portfolio results
        """
        # Market capitalization weights (proxy using equal weights)
        market_weights = np.ones(self.num_assets) / self.num_assets

        # Implied equilibrium returns
        risk_aversion = 3.0  # Typical value
        pi = risk_aversion * self.cov_matrix.values @ market_weights

        # Views matrix P and views vector Q
        P = np.zeros((len(views_dict), self.num_assets))
        Q = np.zeros(len(views_dict))

        for i, (asset, view) in enumerate(views_dict.items()):
            asset_idx = list(self.returns.columns).index(asset)
            P[i, asset_idx] = 1
            Q[i] = view

        # Confidence matrix Omega
        Omega = np.diag([1 / confidence_dict[asset] for asset in views_dict.keys()])

        # Black-Litterman formula
        tau_sigma = tau * self.cov_matrix.values

        # New expected returns
        M1 = np.linalg.inv(tau_sigma)
        M2 = P.T @ np.linalg.inv(Omega) @ P
        M3 = np.linalg.inv(tau_sigma) @ pi + P.T @ np.linalg.inv(Omega) @ Q

        mu_bl = np.linalg.inv(M1 + M2) @ M3

        # New covariance matrix
        sigma_bl = np.linalg.inv(M1 + M2)

        # Optimize portfolio with Black-Litterman inputs
        weights = cp.Variable(self.num_assets)
        portfolio_return = mu_bl @ weights
        portfolio_risk = cp.quad_form(weights, sigma_bl)

        # Maximize utility
        objective = cp.Maximize(portfolio_return - 0.5 * risk_aversion * portfo-
lio_risk)
        constraints = [cp.sum(weights) == 1, weights >= 0]
```

```
        problem = cp.Problem(objective, constraints)
        problem.solve()

        if problem.status == 'optimal':
            optimal_weights = pd.Series(weights.value, index=self.returns.columns)

            return {
                'weights': optimal_weights,
                'bl_returns': pd.Series(mu_bl, index=self.returns.columns),
                'bl_covariance': pd.DataFrame(sigma_bl,
                                              index=self.returns.columns,
                                              columns=self.returns.columns),
                'views': views_dict,
                'confidence': confidence_dict
            }
        else:
            return {'status': 'failed', 'message': problem.status}
```

Dynamic Portfolio Management

1. Rebalancing Strategies

```
class DynamicPortfolioManager:
    """
    Dynamic portfolio management with various rebalancing strategies
    """

    def __init__(self, returns_data, transaction_costs=0.001):
        self.returns = returns_data
        self.transaction_costs = transaction_costs
        self.portfolio_history = []
        self.rebalancing_dates = []

    def calendar_rebalancing(self, target_weights, rebalance_frequency='monthly'):
        """
        Calendar-based rebalancing strategy

        Parameters:
        -----------
        target_weights : pd.Series
            Target portfolio weights
        rebalance_frequency : str
            'daily', 'weekly', 'monthly', 'quarterly'

        Returns:
        --------
        dict : Rebalancing results
        """
        # Generate rebalancing dates
        if rebalance_frequency == 'monthly':
            rebalance_dates = pd.date_range(
                start=self.returns.index[0],
                end=self.returns.index[-1],
                freq='M'
            )
        elif rebalance_frequency == 'quarterly':
            rebalance_dates = pd.date_range(
                start=self.returns.index[0],
                end=self.returns.index[-1],
                freq='Q'
            )
        elif rebalance_frequency == 'weekly':
            rebalance_dates = pd.date_range(
                start=self.returns.index[0],
                end=self.returns.index[-1],
                freq='W'
            )
        else:  # daily
            rebalance_dates = self.returns.index

        # Simulate portfolio performance with rebalancing
        portfolio_values = []
        current_weights = target_weights.copy()
        portfolio_value = 100000  # Initial value

        for date in self.returns.index:
            # Check if rebalancing date
            if date in rebalance_dates:
                # Calculate transaction costs
                weight_changes = abs(current_weights - target_weights).sum()
                transaction_cost = weight_changes * self.transaction_costs * portfo-
lio_value
                portfolio_value -= transaction_cost
```

```
                # Rebalance to target weights
                current_weights = target_weights.copy()
                self.rebalancing_dates.append(date)

            # Calculate daily return
            daily_return = (current_weights * self.returns.loc[date]).sum()
            portfolio_value *= (1 + daily_return)

            # Update weights based on performance (drift)
            asset_returns = self.returns.loc[date]
            current_weights = current_weights * (1 + asset_returns)
            current_weights = current_weights / current_weights.sum()

            portfolio_values.append(portfolio_value)
            self.portfolio_history.append({
                'date': date,
                'portfolio_value': portfolio_value,
                'weights': current_weights.copy()
            })

        # Calculate performance metrics
        portfolio_returns = pd.Series(portfolio_values).pct_change().dropna()

        return {
            'portfolio_values': pd.Series(portfolio_values, index=self.returns.index),
            'portfolio_returns': portfolio_returns,
            'rebalancing_dates': self.rebalancing_dates,
            'num_rebalances': len(self.rebalancing_dates),
            'total_transaction_costs': len(self.rebalancing_dates) * self.transac-
tion_costs
        }

    def threshold_rebalancing(self, target_weights, threshold=0.05):
        """
        Threshold-based rebalancing strategy

        Parameters:
        -----------
        target_weights : pd.Series
            Target portfolio weights
        threshold : float
            Rebalancing threshold (e.g., 0.05 = 5%)

        Returns:
        --------
        dict : Rebalancing results
        """
        portfolio_values = []
        current_weights = target_weights.copy()
        portfolio_value = 100000
        rebalance_dates = []

        for date in self.returns.index:
            # Check if rebalancing is needed
            weight_deviations = abs(current_weights - target_weights)
            max_deviation = weight_deviations.max()

            if max_deviation > threshold:
                # Calculate transaction costs
                weight_changes = abs(current_weights - target_weights).sum()
                transaction_cost = weight_changes * self.transaction_costs * portfo-
lio_value
                portfolio_value -= transaction_cost
```

```
                # Rebalance to target weights
                current_weights = target_weights.copy()
                rebalance_dates.append(date)

            # Calculate daily return
            daily_return = (current_weights * self.returns.loc[date]).sum()
            portfolio_value *= (1 + daily_return)

            # Update weights based on performance
            asset_returns = self.returns.loc[date]
            current_weights = current_weights * (1 + asset_returns)
            current_weights = current_weights / current_weights.sum()

            portfolio_values.append(portfolio_value)

        portfolio_returns = pd.Series(portfolio_values).pct_change().dropna()

        return {
            'portfolio_values': pd.Series(portfolio_values, index=self.returns.index),
            'portfolio_returns': portfolio_returns,
            'rebalancing_dates': rebalance_dates,
            'num_rebalances': len(rebalance_dates),
            'threshold': threshold
        }

    def volatility_targeting(self, target_volatility=0.15, lookback_window=63):
        """
        Volatility targeting strategy

        Parameters:
        -----------
        target_volatility : float
            Target portfolio volatility (annualized)
        lookback_window : int
            Lookback window for volatility estimation

        Returns:
        --------
        dict : Volatility targeting results
        """
        # Equal-weighted base portfolio
        base_weights = pd.Series(1/len(self.returns.columns), index=self.returns.column
s)

        portfolio_values = []
        leverage_history = []
        volatility_history = []
        portfolio_value = 100000

        for i, date in enumerate(self.returns.index):
            if i < lookback_window:
                # Use base weights for initial period
                leverage = 1.0
                current_weights = base_weights
            else:
                # Calculate realized volatility
                lookback_returns = []
                for j in range(max(0, i - lookback_window), i):
                    past_date = self.returns.index[j]
                    past_return = (base_weights * self.returns.loc[past_date]).sum()
                    lookback_returns.append(past_return)
```

```
                realized_vol = np.std(lookback_returns) * np.sqrt(252)
                volatility_history.append(realized_vol)

                # Calculate leverage to achieve target volatility
                if realized_vol > 0:
                    leverage = target_volatility / realized_vol
                    leverage = max(0.1, min(leverage, 3.0))  # Cap leverage between
0.1x and 3x
                else:
                    leverage = 1.0

                current_weights = base_weights * leverage

            leverage_history.append(leverage)

            # Calculate daily return
            daily_return = (current_weights * self.returns.loc[date]).sum()
            portfolio_value *= (1 + daily_return)
            portfolio_values.append(portfolio_value)

        portfolio_returns = pd.Series(portfolio_values).pct_change().dropna()

        return {
            'portfolio_values': pd.Series(portfolio_values, index=self.returns.index),
            'portfolio_returns': portfolio_returns,
            'leverage_history': pd.Series(leverage_history, index=self.returns.index),
            'volatility_history': pd.Series(volatility_history, index=self.returns.in-
dex[lookback_window:]),
            'target_volatility': target_volatility,
            'realized_volatility': portfolio_returns.std() * np.sqrt(252)
        }
```

2. Multi-Asset Strategy Allocation

```
class MultiAssetStrategyAllocator:
    """
    Allocate capital across multiple trading strategies
    """

    def __init__(self, strategy_returns, correlation_threshold=0.7):
        """
        Initialize multi-asset strategy allocator

        Parameters:
        -----------
        strategy_returns : pd.DataFrame
            Returns of individual strategies
        correlation_threshold : float
            Maximum correlation threshold for strategy selection
        """
        self.strategy_returns = strategy_returns
        self.correlation_threshold = correlation_threshold
        self.correlation_matrix = strategy_returns.corr()

    def select_uncorrelated_strategies(self):
        """
        Select strategies with low correlation
        """
        selected_strategies = []
        remaining_strategies = list(self.strategy_returns.columns)

        # Start with strategy with highest Sharpe ratio
        sharpe_ratios = {}
        for strategy in remaining_strategies:
            returns = self.strategy_returns[strategy].dropna()
            if len(returns) > 0 and returns.std() > 0:
                sharpe_ratios[strategy] = returns.mean() / returns.std() * np.sqrt(252)
            else:
                sharpe_ratios[strategy] = 0

        # Select first strategy (highest Sharpe)
        first_strategy = max(sharpe_ratios, key=sharpe_ratios.get)
        selected_strategies.append(first_strategy)
        remaining_strategies.remove(first_strategy)

        # Iteratively add strategies with low correlation
        while remaining_strategies:
            best_candidate = None
            min_max_correlation = float('inf')

            for candidate in remaining_strategies:
                # Calculate maximum correlation with selected strategies
                max_correlation = 0
                for selected in selected_strategies:
                    correlation = abs(self.correlation_matrix.loc[candidate, selected])
                    max_correlation = max(max_correlation, correlation)

                # Select candidate with lowest maximum correlation
                if max_correlation < min_max_correlation and max_correlation < self.cor
relation_threshold:
                    min_max_correlation = max_correlation
                    best_candidate = candidate

            if best_candidate is not None:
                selected_strategies.append(best_candidate)
                remaining_strategies.remove(best_candidate)
```

```
            else:
                break  # No more strategies meet correlation criteria

        return selected_strategies

    def equal_weight_allocation(self, selected_strategies=None):
        """
        Equal weight allocation across selected strategies
        """
        if selected_strategies is None:
            selected_strategies = self.select_uncorrelated_strategies()

        weights = pd.Series(1/len(selected_strategies), index=selected_strategies)

        # Calculate portfolio performance
        portfolio_returns = (self.strategy_returns[selected_strategies] * weights).sum(
axis=1)

        return {
            'weights': weights,
            'selected_strategies': selected_strategies,
            'portfolio_returns': portfolio_returns,
            'correlation_matrix': self.correlation_matrix.loc[selected_strategies, se-
lected_strategies]
        }

    def risk_parity_allocation(self, selected_strategies=None):
        """
        Risk parity allocation across strategies
        """
        if selected_strategies is None:
            selected_strategies = self.select_uncorrelated_strategies()

        strategy_data = self.strategy_returns[selected_strategies]

        # Calculate strategy volatilities
        volatilities = strategy_data.std() * np.sqrt(252)

        # Inverse volatility weights
        inv_vol_weights = (1 / volatilities) / (1 / volatilities).sum()

        # Calculate portfolio performance
        portfolio_returns = (strategy_data * inv_vol_weights).sum(axis=1)

        return {
            'weights': inv_vol_weights,
            'selected_strategies': selected_strategies,
            'portfolio_returns': portfolio_returns,
            'strategy_volatilities': volatilities
        }

    def momentum_allocation(self, lookback_period=63, selected_strategies=None):
        """
        Momentum-based allocation (overweight recent outperformers)
        """
        if selected_strategies is None:
            selected_strategies = self.select_uncorrelated_strategies()

        strategy_data = self.strategy_returns[selected_strategies]

        # Calculate rolling momentum scores
        momentum_scores = strategy_data.rolling(window=lookback_period).mean() * 252
```

```
        # Dynamic weights based on momentum (softmax transformation)
        def softmax_weights(scores):
            exp_scores = np.exp(scores - scores.max())  # Numerical stability
            return exp_scores / exp_scores.sum()

        dynamic_weights = momentum_scores.apply(softmax_weights, axis=1)

        # Calculate portfolio returns
        portfolio_returns = (strategy_data * dynamic_weights).sum(axis=1)

        return {
            'dynamic_weights': dynamic_weights,
            'selected_strategies': selected_strategies,
            'portfolio_returns': portfolio_returns,
            'momentum_scores': momentum_scores
        }

    def mean_reversion_allocation(self, lookback_period=252, selected_strategies=None):
        """
        Mean reversion allocation (overweight recent underperformers)
        """
        if selected_strategies is None:
            selected_strategies = self.select_uncorrelated_strategies()

        strategy_data = self.strategy_returns[selected_strategies]

        # Calculate long-term average performance
        long_term_avg = strategy_data.rolling(window=lookback_period).mean()

        # Calculate recent performance
        recent_performance = strategy_data.rolling(window=21).mean()  # 1 month

        # Mean reversion signal (negative of relative performance)
        mean_reversion_signal = -(recent_performance - long_term_avg)

        # Convert to weights (higher signal = higher weight)
        def signal_to_weights(signals):
            # Shift signals to be positive
            shifted_signals = signals - signals.min() + 0.01
            return shifted_signals / shifted_signals.sum()

        dynamic_weights = mean_reversion_signal.apply(signal_to_weights, axis=1)

        # Calculate portfolio returns
        portfolio_returns = (strategy_data * dynamic_weights).sum(axis=1)

        return {
            'dynamic_weights': dynamic_weights,
            'selected_strategies': selected_strategies,
            'portfolio_returns': portfolio_returns,
            'mean_reversion_signals': mean_reversion_signal
        }

    def adaptive_allocation(self, regime_indicator, bull_weights, bear_weights, selec-
ted_strategies=None):
        """
        Adaptive allocation based on market regime

        Parameters:
        -----------
        regime_indicator : pd.Series
            Binary indicator (1 = bull market, 0 = bear market)
        bull_weights : pd.Series
```

```
            Strategy weights for bull market
        bear_weights : pd.Series
            Strategy weights for bear market
        """
        if selected_strategies is None:
            selected_strategies = self.select_uncorrelated_strategies()

        strategy_data = self.strategy_returns[selected_strategies]

        # Dynamic weights based on regime
        dynamic_weights = pd.DataFrame(index=strategy_data.index, columns=selec-
ted_strategies)

        for date in strategy_data.index:
            if date in regime_indicator.index:
                if regime_indicator[date] == 1:  # Bull market
                    dynamic_weights.loc[date] = bull_weights[selected_strategies]
                else:  # Bear market
                    dynamic_weights.loc[date] = bear_weights[selected_strategies]
            else:
                # Default to equal weights if regime not available
                dynamic_weights.loc[date] = 1 / len(selected_strategies)

        # Calculate portfolio returns
        portfolio_returns = (strategy_data * dynamic_weights).sum(axis=1)

        return {
            'dynamic_weights': dynamic_weights,
            'selected_strategies': selected_strategies,
            'portfolio_returns': portfolio_returns,
            'regime_indicator': regime_indicator,
            'bull_weights': bull_weights,
            'bear_weights': bear_weights
        }
```

Advanced Techniques and Methodologies

Machine Learning Integration

1. Feature Engineering for Trading

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, RobustScaler
from sklearn.feature_selection import SelectKBest, f_regression, mutual_info_regression

class TradingFeatureEngineer:
    """
    Advanced feature engineering for trading strategies
    """

    def __init__(self, price_data, volume_data=None):
        self.price_data = price_data
        self.volume_data = volume_data
        self.features = pd.DataFrame(index=price_data.index)

    def create_technical_features(self, periods=[5, 10, 20, 50]):
        """
        Create comprehensive technical analysis features
        """
        # Price-based features
        for period in periods:
            # Moving averages
            self.features[f'sma_{period}'] = self.price_data.rolling(period).mean()
            self.features[f'ema_{period}'] = self.price_data.ewm(span=period).mean()

            # Price ratios
            self.features[f'price_sma_ratio_{period}'] = self.price_data / self.fea-
tures[f'sma_{period}']
            self.features[f'price_ema_ratio_{period}'] = self.price_data / self.fea-
tures[f'ema_{period}']

            # Volatility measures
            self.features[f'volatility_{period}'] = self.price_data.pct_change().rollin
g(period).std()
            self.features[f'realized_vol_{period}'] = self.price_data.pct_change().roll
ing(period).std() * np.sqrt(252)

            # Momentum indicators
            self.features[f'momentum_{period}'] = self.price_data.pct_change(period)
            self.features[f'roc_{period}'] = (self.price_data - self.price_data.shift(p
eriod)) / self.price_data.shift(period)

            # Mean reversion indicators
            self.features[f'zscore_{period}'] = (self.price_data -
self.features[f'sma_{period}']) / self.features[f'volatility_{period}']

        # RSI
        self.features['rsi_14'] = self._calculate_rsi(self.price_data, 14)
        self.features['rsi_21'] = self._calculate_rsi(self.price_data, 21)

        # MACD
        macd, signal, histogram = self._calculate_macd(self.price_data)
        self.features['macd'] = macd
        self.features['macd_signal'] = signal
        self.features['macd_histogram'] = histogram

        # Bollinger Bands
        bb_upper, bb_middle, bb_lower = self._calculate_bollinger_bands(self.price_data
)
        self.features['bb_upper'] = bb_upper
        self.features['bb_middle'] = bb_middle
        self.features['bb_lower'] = bb_lower
```

```
        self.features['bb_width'] = (bb_upper - bb_lower) / bb_middle
        self.features['bb_position'] = (self.price_data - bb_lower) / (bb_upper -
bb_lower)

        return self.features

    def create_volume_features(self, periods=[5, 10, 20]):
        """
        Create volume-based features
        """
        if self.volume_data is None:
            return self.features

        for period in periods:
            # Volume moving averages
            self.features[f'volume_sma_{period}'] = self.volume_data.rolling(period).me
an()

            # Volume ratios
            self.features[f'volume_ratio_{period}'] = self.volume_data /
self.features[f'volume_sma_{period}']

            # Price-volume features
            returns = self.price_data.pct_change()
            self.features[f'volume_price_trend_{period}'] = (returns * self.volume_data
).rolling(period).sum()

        # On-Balance Volume
        self.features['obv'] = self._calculate_obv(self.price_data, self.volume_data)

        # Volume-Weighted Average Price
        self.features['vwap'] = self._calculate_vwap(self.price_data, self.volume_data)

        return self.features

    def create_market_microstructure_features(self, high_data, low_data):
        """
        Create market microstructure features
        """
        # True Range and ATR
        tr = self._calculate_true_range(high_data, low_data, self.price_data)
        self.features['atr_14'] = tr.rolling(14).mean()
        self.features['atr_21'] = tr.rolling(21).mean()

        # High-Low spread
        self.features['hl_spread'] = (high_data - low_data) / self.price_data

        # Intraday returns
        self.features['intraday_return'] = (self.price_data -
self.price_data.shift(1)) / self.price_data.shift(1)

        # Gap analysis
        self.features['gap'] = (self.price_data - self.price_data.shift(1)) / self.pric
e_data.shift(1)
        self.features['gap_filled'] = np.where(
            (self.features['gap'] > 0) & (low_data < self.price_data.shift(1)), 1,
            np.where((self.features['gap'] < 0) & (high_data > self.price_data.shift(1)
), 1, 0)
        )

        return self.features

    def create_regime_features(self, periods=[20, 50, 200]):
```

```
        """
        Create market regime features
        """
        returns = self.price_data.pct_change()

        for period in periods:
            # Trend strength
            self.features[f'trend_strength_{period}'] = re-
turns.rolling(period).mean() / returns.rolling(period).std()

            # Volatility regime
            vol_ma = returns.rolling(period).std()
            self.features[f'vol_regime_{period}'] = vol_ma / vol_ma.rolling(period*2).m
ean()

            # Market state (trending vs. ranging)
            price_range = self.price_data.rolling(period).max() - self.price_data.rolli
ng(period).min()
            price_movement = abs(self.price_data - self.price_data.shift(period))
            self.features[f'market_efficiency_{period}'] = price_movement / price_range

        # VIX-like volatility measure
        self.features['volatility_of_volatility'] = returns.rolling(20).std().rolling(2
0).std()

        return self.features

    def create_cross_asset_features(self, other_assets_data):
        """
        Create cross-asset features
        """
        for asset_name, asset_data in other_assets_data.items():
            # Correlation features
            returns = self.price_data.pct_change()
            other_returns = asset_data.pct_change()

            self.features[f'corr_{asset_name}_20'] = returns.rolling(20).corr(other_ret
urns)
            self.features[f'corr_{asset_name}_60'] = returns.rolling(60).corr(other_ret
urns)

            # Relative performance
            self.features[f'relative_perf_{asset_name}_20'] = (
                returns.rolling(20).mean() - other_returns.rolling(20).mean()
            )

            # Beta
            covariance = returns.rolling(60).cov(other_returns)
            other_variance = other_returns.rolling(60).var()
            self.features[f'beta_{asset_name}_60'] = covariance / other_variance

        return self.features

    def create_calendar_features(self):
        """
        Create calendar-based features
        """
        # Day of week
        self.features['day_of_week'] = self.features.index.dayofweek

        # Month
        self.features['month'] = self.features.index.month
```

```
        # Quarter
        self.features['quarter'] = self.features.index.quarter

        # Day of month
        self.features['day_of_month'] = self.features.index.day

        # Week of year
        self.features['week_of_year'] = self.features.index.isocalendar().week

        # Holiday effects (simplified)
        self.features['is_month_end'] = (self.features.index.day > 25).astype(int)
        self.features['is_month_start'] = (self.features.index.day <= 5).astype(int)

        return self.features

    def select_features(self, target, method='mutual_info', k=20):
        """
        Feature selection using various methods
        """
        # Remove NaN values
        feature_data = self.features.dropna()
        target_data = target.loc[feature_data.index]

        if method == 'mutual_info':
            selector = SelectKBest(score_func=mutual_info_regression, k=k)
        elif method == 'f_regression':
            selector = SelectKBest(score_func=f_regression, k=k)
        else:
            raise ValueError("Method must be 'mutual_info' or 'f_regression'")

        selected_features = selector.fit_transform(feature_data, target_data)
        selected_feature_names = feature_data.columns[selector.get_support()]

        return {
            'selected_features': pd.DataFrame(selected_features,
                                              index=feature_data.index,
                                              columns=selected_feature_names),
            'feature_scores': pd.Series(selector.scores_, index=feature_data.columns),
            'selected_feature_names': selected_feature_names
        }

    # Helper methods
    def _calculate_rsi(self, prices, period):
        delta = prices.diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
        rs = gain / loss
        return 100 - (100 / (1 + rs))

    def _calculate_macd(self, prices, fast=12, slow=26, signal=9):
        ema_fast = prices.ewm(span=fast).mean()
        ema_slow = prices.ewm(span=slow).mean()
        macd_line = ema_fast - ema_slow
        signal_line = macd_line.ewm(span=signal).mean()
        histogram = macd_line - signal_line
        return macd_line, signal_line, histogram

    def _calculate_bollinger_bands(self, prices, period=20, std_dev=2):
        sma = prices.rolling(window=period).mean()
        std = prices.rolling(window=period).std()
        upper_band = sma + (std * std_dev)
        lower_band = sma - (std * std_dev)
        return upper_band, sma, lower_band
```

```
    def _calculate_obv(self, prices, volume):
        returns = prices.pct_change()
        obv = (np.sign(returns) * volume).cumsum()
        return obv

    def _calculate_vwap(self, prices, volume, period=20):
        return (prices * volume).rolling(period).sum() / volume.rolling(period).sum()

    def _calculate_true_range(self, high, low, close):
        tr1 = high - low
        tr2 = abs(high - close.shift())
        tr3 = abs(low - close.shift())
        return pd.concat([tr1, tr2, tr3], axis=1).max(axis=1)
```

2. Machine Learning Models for Trading

```
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.svm import SVR
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import xgboost as xgb
import lightgbm as lgb

class MLTradingModel:
    """
    Machine learning models for trading signal generation
    """

    def __init__(self, features, target, test_size=0.2):
        """
        Initialize ML trading model

        Parameters:
        -----------
        features : pd.DataFrame
            Feature matrix
        target : pd.Series
            Target variable (future returns)
        test_size : float
            Proportion of data for testing
        """
        self.features = features.dropna()
        self.target = target.loc[self.features.index]

        # Time series split
        split_point = int(len(self.features) * (1 - test_size))
        self.X_train = self.features.iloc[:split_point]
        self.X_test = self.features.iloc[split_point:]
        self.y_train = self.target.iloc[:split_point]
        self.y_test = self.target.iloc[split_point:]

        # Scale features
        self.scaler = StandardScaler()
        self.X_train_scaled = pd.DataFrame(
            self.scaler.fit_transform(self.X_train),
            index=self.X_train.index,
            columns=self.X_train.columns
        )
        self.X_test_scaled = pd.DataFrame(
            self.scaler.transform(self.X_test),
            index=self.X_test.index,
            columns=self.X_test.columns
        )

        self.models = {}
        self.predictions = {}
        self.model_scores = {}

    def train_linear_models(self):
        """
        Train linear regression models
        """
        models = {
            'linear_regression': LinearRegression(),
            'ridge': Ridge(alpha=1.0),
            'lasso': Lasso(alpha=0.01)
```

```
        }

        for name, model in models.items():
            # Train model
            model.fit(self.X_train_scaled, self.y_train)

            # Make predictions
            train_pred = model.predict(self.X_train_scaled)
            test_pred = model.predict(self.X_test_scaled)

            # Store results
            self.models[name] = model
            self.predictions[name] = {
                'train': pd.Series(train_pred, index=self.X_train.index),
                'test': pd.Series(test_pred, index=self.X_test.index)
            }

            # Calculate scores
            self.model_scores[name] = {
                'train_r2': r2_score(self.y_train, train_pred),
                'test_r2': r2_score(self.y_test, test_pred),
                'train_mse': mean_squared_error(self.y_train, train_pred),
                'test_mse': mean_squared_error(self.y_test, test_pred)
            }

    def train_tree_models(self):
        """
        Train tree-based models
        """
        models = {
            'random_forest': RandomForestRegressor(
                n_estimators=100,
                max_depth=10,
                min_samples_split=5,
                random_state=42
            ),
            'gradient_boosting': GradientBoostingRegressor(
                n_estimators=100,
                max_depth=6,
                learning_rate=0.1,
                random_state=42
            ),
            'xgboost': xgb.XGBRegressor(
                n_estimators=100,
                max_depth=6,
                learning_rate=0.1,
                random_state=42
            ),
            'lightgbm': lgb.LGBMRegressor(
                n_estimators=100,
                max_depth=6,
                learning_rate=0.1,
                random_state=42
            )
        }

        for name, model in models.items():
            # Train model (use original features, not scaled)
            model.fit(self.X_train, self.y_train)

            # Make predictions
            train_pred = model.predict(self.X_train)
            test_pred = model.predict(self.X_test)
```

```
            # Store results
            self.models[name] = model
            self.predictions[name] = {
                'train': pd.Series(train_pred, index=self.X_train.index),
                'test': pd.Series(test_pred, index=self.X_test.index)
            }

            # Calculate scores
            self.model_scores[name] = {
                'train_r2': r2_score(self.y_train, train_pred),
                'test_r2': r2_score(self.y_test, test_pred),
                'train_mse': mean_squared_error(self.y_train, train_pred),
                'test_mse': mean_squared_error(self.y_test, test_pred)
            }

            # Feature importance (for tree models)
            if hasattr(model, 'feature_importances_'):
                self.model_scores[name]['feature_importance'] = pd.Series(
                    model.feature_importances_,
                    index=self.X_train.columns
                ).sort_values(ascending=False)

    def train_neural_network(self):
        """
        Train neural network model
        """
        model = MLPRegressor(
            hidden_layer_sizes=(100, 50),
            activation='relu',
            solver='adam',
            alpha=0.001,
            learning_rate='adaptive',
            max_iter=500,
            random_state=42
        )

        # Train model
        model.fit(self.X_train_scaled, self.y_train)

        # Make predictions
        train_pred = model.predict(self.X_train_scaled)
        test_pred = model.predict(self.X_test_scaled)

        # Store results
        self.models['neural_network'] = model
        self.predictions['neural_network'] = {
            'train': pd.Series(train_pred, index=self.X_train.index),
            'test': pd.Series(test_pred, index=self.X_test.index)
        }

        # Calculate scores
        self.model_scores['neural_network'] = {
            'train_r2': r2_score(self.y_train, train_pred),
            'test_r2': r2_score(self.y_test, test_pred),
            'train_mse': mean_squared_error(self.y_train, train_pred),
            'test_mse': mean_squared_error(self.y_test, test_pred)
        }

    def hyperparameter_optimization(self, model_name='random_forest'):
        """
        Perform hyperparameter optimization using time series cross-validation
        """
```

```
        if model_name == 'random_forest':
            model = RandomForestRegressor(random_state=42)
            param_grid = {
                'n_estimators': [50, 100, 200],
                'max_depth': [5, 10, 15, None],
                'min_samples_split': [2, 5, 10],
                'min_samples_leaf': [1, 2, 4]
            }
        elif model_name == 'xgboost':
            model = xgb.XGBRegressor(random_state=42)
            param_grid = {
                'n_estimators': [50, 100, 200],
                'max_depth': [3, 6, 9],
                'learning_rate': [0.01, 0.1, 0.2],
                'subsample': [0.8, 0.9, 1.0]
            }
        else:
            raise ValueError("Model not supported for hyperparameter optimization")

        # Time series cross-validation
        tscv = TimeSeriesSplit(n_splits=5)

        # Grid search
        grid_search = GridSearchCV(
            model,
            param_grid,
            cv=tscv,
            scoring='neg_mean_squared_error',
            n_jobs=-1
        )

        grid_search.fit(self.X_train, self.y_train)

        # Best model
        best_model = grid_search.best_estimator_

        # Make predictions
        train_pred = best_model.predict(self.X_train)
        test_pred = best_model.predict(self.X_test)

        # Store results
        optimized_name = f'{model_name}_optimized'
        self.models[optimized_name] = best_model
        self.predictions[optimized_name] = {
            'train': pd.Series(train_pred, index=self.X_train.index),
            'test': pd.Series(test_pred, index=self.X_test.index)
        }

        # Calculate scores
        self.model_scores[optimized_name] = {
            'train_r2': r2_score(self.y_train, train_pred),
            'test_r2': r2_score(self.y_test, test_pred),
            'train_mse': mean_squared_error(self.y_train, train_pred),
            'test_mse': mean_squared_error(self.y_test, test_pred),
            'best_params': grid_search.best_params_,
            'cv_score': grid_search.best_score_
        }

        return grid_search.best_params_

    def ensemble_prediction(self, method='average'):
        """
        Create ensemble predictions from multiple models
```

```
        """
        if method == 'average':
            # Simple average
            test_predictions = pd.DataFrame({
                name: pred['test'] for name, pred in self.predictions.items()
            })
            ensemble_pred = test_predictions.mean(axis=1)

        elif method == 'weighted_average':
            # Weighted by test R2 score
            weights = {}
            total_weight = 0

            for name, scores in self.model_scores.items():
                weight = max(0, scores['test_r2'])  # Only positive R2 scores
                weights[name] = weight
                total_weight += weight

            # Normalize weights
            if total_weight > 0:
                weights = {name: w/total_weight for name, w in weights.items()}
            else:
                # Equal weights if all R2 scores are negative
                weights = {name: 1/len(self.model_scores) for name in
self.model_scores.keys()}

            # Weighted ensemble
            ensemble_pred = pd.Series(0, index=self.X_test.index)
            for name, pred in self.predictions.items():
                ensemble_pred += weights[name] * pred['test']

        # Store ensemble results
        self.predictions['ensemble'] = {'test': ensemble_pred}
        self.model_scores['ensemble'] = {
            'test_r2': r2_score(self.y_test, ensemble_pred),
            'test_mse': mean_squared_error(self.y_test, ensemble_pred),
            'method': method
        }

        if method == 'weighted_average':
            self.model_scores['ensemble']['weights'] = weights

        return ensemble_pred

    def generate_trading_signals(self, model_name='ensemble', threshold=0.001):
        """
        Generate trading signals from model predictions
        """
        if model_name not in self.predictions:
            raise ValueError(f"Model {model_name} not found")

        predictions = self.predictions[model_name]['test']

        # Generate signals based on prediction threshold
        signals = pd.Series(0, index=predictions.index)
        signals[predictions > threshold] = 1  # Buy signal
        signals[predictions < -threshold] = -1  # Sell signal

        return {
            'predictions': predictions,
            'signals': signals,
            'threshold': threshold,
            'signal_distribution': signals.value_counts()
```

```
        }

    def model_comparison(self):
        """
        Compare all trained models
        """
        comparison_df = pd.DataFrame(self.model_scores).T

        # Sort by test R2 score
        comparison_df = comparison_df.sort_values('test_r2', ascending=False)

        return comparison_df
```

Alternative Data Integration

1. Sentiment Analysis Framework

```
import requests
import pandas as pd
import numpy as np
from textblob import TextBlob
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
import yfinance as yf

class SentimentAnalyzer:
    """
    Sentiment analysis for trading signals
    """

    def __init__(self):
        self.vader_analyzer = SentimentIntensityAnalyzer()
        self.sentiment_history = []

    def analyze_news_sentiment(self, news_data, symbol):
        """
        Analyze sentiment from news articles

        Parameters:
        -----------
        news_data : list
            List of news articles with 'title', 'content', 'date', 'source'
        symbol : str
            Stock symbol to filter relevant news

        Returns:
        --------
        pd.DataFrame : Sentiment scores by date
        """
        sentiment_scores = []

        for article in news_data:
            # Check if article is relevant to symbol
            relevance_score = self._calculate_relevance(article, symbol)

            if relevance_score > 0.5:  # Only analyze relevant articles
                # TextBlob sentiment
                blob = TextBlob(article['content'])
                textblob_sentiment = blob.sentiment.polarity

                # VADER sentiment
                vader_scores = self.vader_analyzer.polarity_scores(article['content'])
                vader_sentiment = vader_scores['compound']

                # Combine sentiments
                combined_sentiment = (textblob_sentiment + vader_sentiment) / 2

                sentiment_scores.append({
                    'date': pd.to_datetime(article['date']),
                    'source': article['source'],
                    'textblob_sentiment': textblob_sentiment,
                    'vader_sentiment': vader_sentiment,
                    'combined_sentiment': combined_sentiment,
                    'relevance_score': relevance_score,
                    'title': article['title'][:100]  # First 100 chars
                })

        # Convert to DataFrame and aggregate by date
        sentiment_df = pd.DataFrame(sentiment_scores)
```

```
        if not sentiment_df.empty:
            daily_sentiment = sentiment_df.groupby('date').agg({
                'textblob_sentiment': 'mean',
                'vader_sentiment': 'mean',
                'combined_sentiment': 'mean',
                'relevance_score': 'mean'
            }).round(4)

            # Add sentiment momentum and volatility
            daily_sentiment['sentiment_momentum'] = daily_sentiment['com-
bined_sentiment'].diff()
            daily_sentiment['sentiment_volatility'] = daily_sentiment['com-
bined_sentiment'].rolling(5).std()

            return daily_sentiment
        else:
            return pd.DataFrame()

    def _calculate_relevance(self, article, symbol):
        """
        Calculate relevance of article to symbol
        """
        text = (article['title'] + ' ' + article['content']).lower()
        symbol_lower = symbol.lower()

        # Simple keyword matching (can be enhanced with NLP)
        relevance_score = 0

        # Direct symbol mention
        if symbol_lower in text:
            relevance_score += 0.5

        # Company name matching (simplified)
        company_keywords = self._get_company_keywords(symbol)
        for keyword in company_keywords:
            if keyword.lower() in text:
                relevance_score += 0.3

        # Industry keywords
        industry_keywords = self._get_industry_keywords(symbol)
        for keyword in industry_keywords:
            if keyword.lower() in text:
                relevance_score += 0.1

        return min(relevance_score, 1.0)

    def _get_company_keywords(self, symbol):
        """
        Get company-specific keywords (simplified implementation)
        """
        # In practice, this would use a comprehensive database
        company_map = {
            'AAPL': ['apple', 'iphone', 'ipad', 'mac', 'tim cook'],
            'GOOGL': ['google', 'alphabet', 'android', 'youtube', 'sundar pichai'],
            'MSFT': ['microsoft', 'windows', 'office', 'azure', 'satya nadella'],
            'TSLA': ['tesla', 'elon musk', 'electric vehicle', 'model 3', 'model y'],
            'AMZN': ['amazon', 'aws', 'prime', 'jeff bezos', 'andy jassy']
        }

        return company_map.get(symbol, [])

    def _get_industry_keywords(self, symbol):
        """
```

```
        Get industry-specific keywords
        """
        # Simplified industry mapping
        industry_map = {
            'AAPL': ['technology', 'smartphone', 'consumer electronics'],
            'GOOGL': ['technology', 'internet', 'advertising', 'cloud computing'],
            'MSFT': ['technology', 'software', 'cloud computing'],
            'TSLA': ['automotive', 'electric vehicle', 'clean energy'],
            'AMZN': ['e-commerce', 'cloud computing', 'retail']
        }

        return industry_map.get(symbol, [])

    def analyze_social_media_sentiment(self, social_media_data, symbol):
        """
        Analyze sentiment from social media posts
        """
        sentiment_scores = []

        for post in social_media_data:
            # Filter posts mentioning the symbol
            if symbol.lower() in post['text'].lower():
                # Analyze sentiment
                vader_scores = self.vader_analyzer.polarity_scores(post['text'])

                # Weight by engagement (likes, shares, etc.)
                engagement_weight = self._calculate_engagement_weight(post)

                sentiment_scores.append({
                    'date': pd.to_datetime(post['date']),
                    'platform': post['platform'],
                    'sentiment': vader_scores['compound'],
                    'engagement_weight': engagement_weight,
                    'weighted_sentiment': vader_scores['compound'] * engagement_weight
                })

        # Aggregate by date
        if sentiment_scores:
            sentiment_df = pd.DataFrame(sentiment_scores)

            daily_sentiment = sentiment_df.groupby('date').agg({
                'sentiment': 'mean',
                'weighted_sentiment': 'mean',
                'engagement_weight': 'sum'
            })

            return daily_sentiment
        else:
            return pd.DataFrame()

    def _calculate_engagement_weight(self, post):
        """
        Calculate engagement weight for social media post
        """
        likes = post.get('likes', 0)
        shares = post.get('shares', 0)
        comments = post.get('comments', 0)

        # Simple engagement score
        engagement_score = likes + shares * 2 + comments * 3

        # Normalize to 0-1 range (log transformation to handle outliers)
        normalized_score = np.log1p(engagement_score) / 10
```

```
        return min(normalized_score, 1.0)

    def create_sentiment_features(self, sentiment_data, periods=[1, 3, 5, 10]):
        """
        Create sentiment-based features for trading
        """
        features = pd.DataFrame(index=sentiment_data.index)

        # Raw sentiment features
        features['sentiment'] = sentiment_data['combined_sentiment']
        features['sentiment_momentum'] = sentiment_data['sentiment_momentum']
        features['sentiment_volatility'] = sentiment_data['sentiment_volatility']

        # Rolling sentiment features
        for period in periods:
            features[f'sentiment_ma_{period}'] = sentiment_data['combined_sentiment'].r
olling(period).mean()
            features[f'sentiment_std_{period}'] = senti-
ment_data['combined_sentiment'].rolling(period).std()
            features[f'sentiment_zscore_{period}'] = (
                (sentiment_data['combined_sentiment'] - fea-
tures[f'sentiment_ma_{period}']) /
                features[f'sentiment_std_{period}']
            )

        # Sentiment regime indicators
        features['sentiment_regime'] = np.where(
            sentiment_data['combined_sentiment'] > 0.1, 1,  # Positive
            np.where(sentiment_data['combined_sentiment'] < -0.1, -1, 0)
# Negative or Neutral
        )

        # Sentiment divergence (sentiment vs price)
        # This would require price data to be passed in

        return features

class EconomicDataIntegrator:
    """
    Integration of economic indicators and alternative data
    """

    def __init__(self):
        self.economic_indicators = {}
        self.alternative_data = {}

    def fetch_economic_indicators(self, indicators=['GDP', 'CPI', 'UNEMPLOYMENT', 'IN-
TEREST_RATES']):
        """
        Fetch economic indicators (using FRED API or similar)
        """
        # This is a simplified implementation
        # In practice, you would use APIs like FRED, Bloomberg, etc.

        economic_data = {}

        for indicator in indicators:
            # Simulate fetching data
            dates = pd.date_range('2020-01-01', periods=100, freq='M')

            if indicator == 'GDP':
                # Simulate GDP growth data
```

```
                values = np.random.normal(2.5, 1.0, 100)  # 2.5% average growth
            elif indicator == 'CPI':
                # Simulate inflation data
                values = np.random.normal(2.0, 0.5, 100)  # 2% average inflation
            elif indicator == 'UNEMPLOYMENT':
                # Simulate unemployment rate
                values = np.random.normal(5.0, 1.5, 100)  # 5% average unemployment
            elif indicator == 'INTEREST_RATES':
                # Simulate interest rates
                values = np.random.normal(2.0, 1.0, 100)  # 2% average rate

            economic_data[indicator] = pd.Series(values, index=dates)

        self.economic_indicators = economic_data
        return economic_data

    def create_economic_features(self, target_frequency='D'):
        """
        Create economic features for trading models
        """
        features = pd.DataFrame()

        for indicator, data in self.economic_indicators.items():
            # Resample to target frequency
            if target_frequency == 'D':
                resampled_data = data.resample('D').ffill()  # Forward fill for daily
            else:
                resampled_data = data

            # Raw values
            features[f'{indicator.lower()}'] = resampled_data

            # Changes and momentum
            features[f'{indicator.lower()}_change'] = resampled_data.diff()
            features[f'{indicator.lower()}_pct_change'] = resampled_data.pct_change()
            features[f'{indicator.lower()}_momentum_3m'] = resampled_data.diff(3)
            features[f'{indicator.lower()}_momentum_6m'] = resampled_data.diff(6)

            # Z-scores (relative to historical)
            features[f'{indicator.lower()}_zscore_1y'] = (
                (resampled_data - resampled_data.rolling(252).mean()) /
                resampled_data.rolling(252).std()
            )

            # Regime indicators
            if indicator in ['CPI', 'INTEREST_RATES']:
                # High/low regime for inflation and rates
                features[f'{indicator.lower()}_regime'] = np.where(
                    resampled_data > resampled_data.rolling(252).quantile(0.75), 1,
                    np.where(resampled_data < res-
ampled_data.rolling(252).quantile(0.25), -1, 0)
                )

        return features

    def integrate_alternative_data(self, alt_data_sources):
        """
        Integrate various alternative data sources
        """
        integrated_features = pd.DataFrame()

        for source_name, source_data in alt_data_sources.items():
            if source_name == 'satellite_data':
```

```
                # Economic activity from satellite data
                integrated_features['economic_activity_satellite'] = source_data['activ
ity_index']

            elif source_name == 'credit_card_spending':
                # Consumer spending data
                integrated_features['consumer_spending'] =
source_data['spending_index']
                integrated_features['consumer_spending_yoy'] = source_data['spend-
ing_index'].pct_change(252)

            elif source_name == 'web_search_trends':
                # Google Trends or similar
                for keyword, trend_data in source_data.items():
                    integrated_features[f'search_trend_{keyword}'] = trend_data
                    integrated_features[f'search_trend_{keyword}_momentum'] =
trend_data.diff()

            elif source_name == 'supply_chain_data':
                # Supply chain indicators
                integrated_features['supply_chain_pressure'] = source_data['pres-
sure_index']
                integrated_features['shipping_costs'] = source_data['ship-
ping_cost_index']

            elif source_name == 'weather_data':
                # Weather impact on commodities/agriculture
                integrated_features['temperature_anomaly'] =
source_data['temp_anomaly']
                integrated_features['precipitation_anomaly'] = source_data['pre-
cip_anomaly']

        return integrated_features

    def create_macro_regime_features(self, price_data):
        """
        Create macro regime features combining multiple data sources
        """
        features = pd.DataFrame(index=price_data.index)

        # Growth regime (based on economic indicators)
        if 'GDP' in self.economic_indicators:
            gdp_growth = self.economic_indicators['GDP'].pct_change(4)  # YoY growth
            features['growth_regime'] = np.where(
                gdp_growth > gdp_growth.quantile(0.6), 1,  # High growth
                np.where(gdp_growth < gdp_growth.quantile(0.4), -1, 0)  # Low growth
            )

        # Inflation regime
        if 'CPI' in self.economic_indicators:
            inflation = self.economic_indicators['CPI'].pct_change(12)  # YoY inflation
            features['inflation_regime'] = np.where(
                inflation > 3.0, 1,  # High inflation
                np.where(inflation < 1.0, -1, 0)  # Low inflation
            )

        # Monetary policy regime
        if 'INTEREST_RATES' in self.economic_indicators:
            rates = self.economic_indicators['INTEREST_RATES']
            rate_changes = rates.diff(3)  # 3-month change
            features['monetary_regime'] = np.where(
                rate_changes > 0.25, 1,  # Tightening
                np.where(rate_changes < -0.25, -1, 0)  # Easing
```

```
            )

        # Market volatility regime
        returns = price_data.pct_change()
        volatility = returns.rolling(21).std() * np.sqrt(252)
        vol_ma = volatility.rolling(63).mean()
        features['volatility_regime'] = np.where(
            volatility > vol_ma * 1.5, 1,  # High vol
            np.where(volatility < vol_ma * 0.5, -1, 0)  # Low vol
        )

        # Combined macro regime
        regime_components = ['growth_regime', 'inflation_regime', 'monetary_regime']
        available_components = [col for col in regime_components if col in features.col
umns]

        if available_components:
            features['macro_regime_score'] = features[available_components].sum(axis=1)
            features['macro_regime'] = np.where(
                features['macro_regime_score'] > 1, 1,  # Risk-on
                np.where(features['macro_regime_score'] < -1, -1, 0)  # Risk-off
            )

        return features
```

Asset Class Specific Considerations

Equity Trading Strategies

1. Sector Rotation and Style Factors

```
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

class EquityFactorModel:
    """
    Equity factor model for style and sector analysis
    """

    def __init__(self, stock_returns, market_data=None):
        """
        Initialize equity factor model

        Parameters:
        -----------
        stock_returns : pd.DataFrame
            Individual stock returns
        market_data : dict
            Market data including sector classifications, fundamentals
        """
        self.stock_returns = stock_returns
        self.market_data = market_data or {}
        self.factors = pd.DataFrame(index=stock_returns.index)
        self.factor_loadings = {}
        self.factor_returns = {}

    def create_style_factors(self, fundamental_data):
        """
        Create style factors (Value, Growth, Quality, etc.)

        Parameters:
        -----------
        fundamental_data : pd.DataFrame
            Fundamental data with columns like P/E, P/B, ROE, etc.
        """
        # Value factor
        if 'PE_RATIO' in fundamental_data.columns and 'PB_RATIO' in fundamental_data.co
lumns:
            # Inverse of valuation ratios (higher = more value)
            value_score = (1 / fundamental_data['PE_RATIO'].replace([np.inf, -np.inf],
np.nan)) + \
                          (1 / fundamental_data['PB_RATIO'].replace([np.inf, -np.inf],
np.nan))
            self.factors['value_factor'] = value_score.rank(pct=True)  # Percentile
ranking

        # Growth factor
        if 'EARNINGS_GROWTH' in fundamental_data.columns and 'REVENUE_GROWTH' in funda-
mental_data.columns:
            growth_score = fundamental_data['EARNINGS_GROWTH'] + fundamental_data['REV-
ENUE_GROWTH']
            self.factors['growth_factor'] = growth_score.rank(pct=True)

        # Quality factor
        if 'ROE' in fundamental_data.columns and 'DEBT_TO_EQUITY' in fundamental_data.c
olumns:
            quality_score = fundamental_data['ROE'] - fundament-
al_data['DEBT_TO_EQUITY'] / 100
            self.factors['quality_factor'] = quality_score.rank(pct=True)

        # Momentum factor (price-based)
```

```
        momentum_periods = [21, 63, 252]  # 1M, 3M, 1Y
        momentum_scores = []

        for period in momentum_periods:
            momentum = self.stock_returns.rolling(period).mean() * 252  # Annualized
            momentum_scores.append(momentum.rank(pct=True, axis=1))

        # Average momentum across periods
        self.factors['momentum_factor'] = pd.concat(momentum_scores).groupby(level=0).m
ean()

        # Size factor (market cap)
        if 'MARKET_CAP' in fundamental_data.columns:
            # Log market cap, then rank (smaller = higher rank for size factor)
            log_mcap = np.log(fundamental_data['MARKET_CAP'])
            self.factors['size_factor'] = (1 - log_mcap.rank(pct=True))  # Invert for
small-cap bias

        # Volatility factor
        volatility = self.stock_returns.rolling(63).std() * np.sqrt(252)
        self.factors['low_vol_factor'] = (1 - volatility.rank(pct=True, axis=1))
# Low vol = high rank

        return self.factors

    def create_sector_factors(self, sector_classifications):
        """
        Create sector-based factors

        Parameters:
        -----------
        sector_classifications : dict
            {stock_symbol: sector_name}
        """
        sectors = list(set(sector_classifications.values()))

        for sector in sectors:
            # Get stocks in this sector
            sector_stocks = [stock for stock, sec in sector_classifications.items()
                             if sec == sector and stock in self.stock_returns.columns]

            if len(sector_stocks) > 0:
                # Equal-weighted sector return
                sector_return = self.stock_returns[sector_stocks].mean(axis=1)
                self.factors[f'sector_{sector.lower().replace(" ", "_")}'] = sec-
tor_return

        return self.factors

    def fama_french_factors(self, market_return, risk_free_rate):
        """
        Create Fama-French style factors

        Parameters:
        -----------
        market_return : pd.Series
            Market return (e.g., S&P 500)
        risk_free_rate : pd.Series
            Risk-free rate
        """
        # Market factor (excess return)
        self.factors['market_factor'] = market_return - risk_free_rate
```

```
        # SMB (Small Minus Big) - requires market cap data
        if 'size_factor' in self.factors.columns:
            # Create size-based portfolios
            size_factor = self.factors['size_factor']

            # Top and bottom terciles
            small_cap_threshold = size_factor.quantile(0.67, axis=1)
            big_cap_threshold = size_factor.quantile(0.33, axis=1)

            smb_returns = []
            for date in self.stock_returns.index:
                if date in size_factor.index:
                    small_stocks = size_factor.loc[date] >= small_cap_threshold.loc[dat
e]
                    big_stocks = size_factor.loc[date] <= big_cap_threshold.loc[date]

                    small_return = self.stock_returns.loc[date, small_stocks].mean()
                    big_return = self.stock_returns.loc[date, big_stocks].mean()

                    smb_returns.append(small_return - big_return)
                else:
                    smb_returns.append(np.nan)

            self.factors['SMB'] = pd.Series(smb_returns, index=self.stock_returns.in-
dex)

        # HML (High Minus Low) - requires value factor
        if 'value_factor' in self.factors.columns:
            value_factor = self.factors['value_factor']

            # Top and bottom terciles
            high_value_threshold = value_factor.quantile(0.67, axis=1)
            low_value_threshold = value_factor.quantile(0.33, axis=1)

            hml_returns = []
            for date in self.stock_returns.index:
                if date in value_factor.index:
                    high_value_stocks = value_factor.loc[date] >=
high_value_threshold.loc[date]
                    low_value_stocks = value_factor.loc[date] <= low_value_threshold.lo
c[date]

                    high_value_return = self.stock_returns.loc[date,
high_value_stocks].mean()
                    low_value_return = self.stock_returns.loc[date, low_value_stocks].m
ean()

                    hml_returns.append(high_value_return - low_value_return)
                else:
                    hml_returns.append(np.nan)

            self.factors['HML'] = pd.Series(hml_returns, index=self.stock_returns.in-
dex)

        return self.factors

    def factor_regression_analysis(self, stock_symbol):
        """
        Perform factor regression for individual stock

        Parameters:
        -----------
        stock_symbol : str
```

```
            Stock symbol to analyze

        Returns:
        --------
        dict : Factor loadings and statistics
        """
        from sklearn.linear_model import LinearRegression
        from sklearn.metrics import r2_score

        if stock_symbol not in self.stock_returns.columns:
            raise ValueError(f"Stock {stock_symbol} not found in returns data")

        # Get stock returns
        stock_ret = self.stock_returns[stock_symbol].dropna()

        # Align factor data
        common_dates = stock_ret.index.intersection(self.factors.index)
        stock_ret_aligned = stock_ret.loc[common_dates]
        factors_aligned = self.factors.loc[common_dates].dropna()

        # Further align
        common_dates = stock_ret_aligned.index.intersection(factors_aligned.index)
        stock_ret_final = stock_ret_aligned.loc[common_dates]
        factors_final = factors_aligned.loc[common_dates]

        if len(common_dates) < 50:  # Need sufficient data
            return {'error': 'Insufficient data for regression'}

        # Regression
        reg = LinearRegression()
        reg.fit(factors_final, stock_ret_final)

        # Predictions and residuals
        predictions = reg.predict(factors_final)
        residuals = stock_ret_final - predictions

        # Calculate statistics
        r_squared = r2_score(stock_ret_final, predictions)

        # Factor loadings
        loadings = pd.Series(reg.coef_, index=factors_final.columns)

        # T-statistics (simplified)
        residual_std = residuals.std()
        factor_std_errors = residual_std / np.sqrt(len(factors_final))  # Simplified
        t_stats = loadings / factor_std_errors

        return {
            'loadings': loadings,
            't_statistics': t_stats,
            'r_squared': r_squared,
            'alpha': reg.intercept_,
            'residual_volatility': residual_std * np.sqrt(252),  # Annualized
            'factor_contributions': loadings * factors_final.mean() * 252  # Annualized
        }

    def sector_rotation_strategy(self, economic_indicators, lookback_period=63):
        """
        Create sector rotation strategy based on economic indicators

        Parameters:
        -----------
        economic_indicators : pd.DataFrame
```

```
            Economic indicators (GDP growth, inflation, etc.)
        lookback_period : int
            Lookback period for momentum calculation
        """
        # Get sector factors
        sector_factors = [col for col in self.factors.columns if col.startswith('sec-
tor_')]

        if not sector_factors:
            return {'error': 'No sector factors available'}

        sector_returns = self.factors[sector_factors]

        # Economic regime classification
        regimes = self._classify_economic_regimes(economic_indicators)

        # Sector performance in different regimes
        regime_performance = {}

        for regime in regimes['regime'].unique():
            if pd.notna(regime):
                regime_dates = regimes[regimes['regime'] == regime].index
                regime_sector_returns = sector_returns.loc[regime_dates]

                regime_performance[regime] = {
                    'mean_returns': regime_sector_returns.mean() * 252,  # Annualized
                    'volatility': regime_sector_returns.std() * np.sqrt(252),
                    'sharpe_ratio': (regime_sector_returns.mean() * 252) / (re-
gime_sector_returns.std() * np.sqrt(252))
                }

        # Current regime-based allocation
        current_regime = regimes['regime'].iloc[-1]

        if pd.notna(current_regime) and current_regime in regime_performance:
            # Rank sectors by Sharpe ratio in current regime
            current_performance = regime_performance[current_regime]['sharpe_ratio']
            sector_rankings = current_performance.sort_values(ascending=False)

            # Top 3 sectors get equal weight
            top_sectors = sector_rankings.head(3)
            allocation = pd.Series(0, index=sector_factors)
            allocation[top_sectors.index] = 1 / len(top_sectors)

            return {
                'allocation': allocation,
                'current_regime': current_regime,
                'regime_performance': regime_performance,
                'sector_rankings': sector_rankings
            }
        else:
            # Equal weight if regime unclear
            allocation = pd.Series(1/len(sector_factors), index=sector_factors)
            return {
                'allocation': allocation,
                'current_regime': 'unclear',
                'regime_performance': regime_performance
            }

    def _classify_economic_regimes(self, economic_indicators):
        """
        Classify economic regimes based on indicators
        """
```

```
        regimes = pd.DataFrame(index=economic_indicators.index)

        # Simple regime classification
        if 'GDP_GROWTH' in economic_indicators.columns:
            gdp_growth = economic_indicators['GDP_GROWTH']
            gdp_ma = gdp_growth.rolling(4).mean()  # 4-quarter MA

            if 'INFLATION' in economic_indicators.columns:
                inflation = economic_indicators['INFLATION']
                inflation_ma = inflation.rolling(4).mean()

                # Four regimes: Growth/Inflation combinations
                regimes['regime'] = np.where(
                    (gdp_growth > gdp_ma) & (inflation > inflation_ma), 'high_growth_hi
gh_inflation',
                    np.where(
                        (gdp_growth > gdp_ma) & (inflation <= inflation_ma), 'high_grow
th_low_inflation',
                        np.where(
                            (gdp_growth <= gdp_ma) & (inflation > inflation_ma), 'low_g
rowth_high_inflation',
                            'low_growth_low_inflation'
                        )
                    )
                )
            else:
                # Simple growth regimes
                regimes['regime'] = np.where(gdp_growth > gdp_ma, 'expansion', 'con-
traction')
        else:
            regimes['regime'] = 'unknown'

        return regimes

class PairsTradingStrategy:
    """
    Statistical arbitrage through pairs trading
    """

    def __init__(self, stock_returns, min_correlation=0.7, cointegra-
tion_threshold=0.05):
        """
        Initialize pairs trading strategy

        Parameters:
        -----------
        stock_returns : pd.DataFrame
            Stock returns data
        min_correlation : float
            Minimum correlation for pair selection
        cointegration_threshold : float
            P-value threshold for cointegration test
        """
        self.stock_returns = stock_returns
        self.stock_prices = (1 + stock_returns).cumprod()  # Convert to price levels
        self.min_correlation = min_correlation
        self.cointegration_threshold = cointegration_threshold
        self.pairs = []
        self.pair_signals = {}

    def find_cointegrated_pairs(self, sector_filter=None):
        """
        Find cointegrated pairs using Engle-Granger test
```

```
        Parameters:
        -----------
        sector_filter : dict, optional
            {stock: sector} mapping to filter pairs within sectors
        """
        from statsmodels.tsa.stattools import coint

        stocks = list(self.stock_prices.columns)
        pairs = []

        for i in range(len(stocks)):
            for j in range(i+1, len(stocks)):
                stock1, stock2 = stocks[i], stocks[j]

                # Sector filter
                if sector_filter:
                    if (stock1 in sector_filter and stock2 in sector_filter and
                        sector_filter[stock1] != sector_filter[stock2]):
                        continue  # Skip cross-sector pairs if filtering

                # Get price series
                price1 = self.stock_prices[stock1].dropna()
                price2 = self.stock_prices[stock2].dropna()

                # Align series
                common_dates = price1.index.intersection(price2.index)
                if len(common_dates) < 100:  # Need sufficient data
                    continue

                price1_aligned = price1.loc[common_dates]
                price2_aligned = price2.loc[common_dates]

                # Check correlation
                correlation = price1_aligned.corr(price2_aligned)
                if abs(correlation) < self.min_correlation:
                    continue

                # Cointegration test
                try:
                    coint_stat, p_value, critical_values = coint(price1_aligned,
price2_aligned)

                    if p_value < self.cointegration_threshold:
                        pairs.append({
                            'stock1': stock1,
                            'stock2': stock2,
                            'correlation': correlation,
                            'coint_pvalue': p_value,
                            'coint_stat': coint_stat,
                            'critical_value_5pct': critical_values[1]
                        })
                except:
                    continue  # Skip if cointegration test fails

        # Sort by cointegration strength (lower p-value = stronger)
        self.pairs = sorted(pairs, key=lambda x: x['coint_pvalue'])

        return self.pairs

    def generate_pair_signals(self, pair_info, lookback_window=60, entry_threshold=2.0,
exit_threshold=0.5):
        """
```

```
        Generate trading signals for a specific pair

        Parameters:
        -----------
        pair_info : dict
            Pair information from find_cointegrated_pairs
        lookback_window : int
            Window for calculating spread statistics
        entry_threshold : float
            Z-score threshold for entry
        exit_threshold : float
            Z-score threshold for exit
        """
        stock1, stock2 = pair_info['stock1'], pair_info['stock2']

        # Get price series
        price1 = self.stock_prices[stock1].dropna()
        price2 = self.stock_prices[stock2].dropna()

        # Align series
        common_dates = price1.index.intersection(price2.index)
        price1_aligned = price1.loc[common_dates]
        price2_aligned = price2.loc[common_dates]

        # Calculate hedge ratio using rolling regression
        from sklearn.linear_model import LinearRegression

        hedge_ratios = []
        spreads = []

        for i in range(lookback_window, len(common_dates)):
            # Rolling window data
            window_dates = common_dates[i-lookback_window:i]
            p1_window = price1_aligned.loc[window_dates].values.reshape(-1, 1)
            p2_window = price2_aligned.loc[window_dates].values

            # Regression to find hedge ratio
            reg = LinearRegression()
            reg.fit(p1_window, p2_window)
            hedge_ratio = reg.coef_[0]

            hedge_ratios.append(hedge_ratio)

            # Calculate spread
            current_date = common_dates[i]
            spread = price2_aligned.loc[current_date] - hedge_ratio * price1_aligned.lo
c[current_date]
            spreads.append(spread)

        # Convert to series
        signal_dates = common_dates[lookback_window:]
        hedge_ratio_series = pd.Series(hedge_ratios, index=signal_dates)
        spread_series = pd.Series(spreads, index=signal_dates)

        # Calculate spread statistics
        spread_mean = spread_series.rolling(lookback_window).mean()
        spread_std = spread_series.rolling(lookback_window).std()
        spread_zscore = (spread_series - spread_mean) / spread_std

        # Generate signals
        signals = pd.DataFrame(index=signal_dates)
        signals['spread'] = spread_series
        signals['spread_zscore'] = spread_zscore
```

```
        signals['hedge_ratio'] = hedge_ratio_series

        # Entry signals
        signals['long_entry'] = spread_zscore < -entry_threshold  # Spread too low, ex-
pect reversion
        signals['short_entry'] = spread_zscore > entry_threshold   # Spread too high,
expect reversion

        # Exit signals
        signals['exit'] = abs(spread_zscore) < exit_threshold

        # Position signals
        position = pd.Series(0, index=signal_dates)
        current_position = 0

        for date in signal_dates:
            if signals.loc[date, 'long_entry'] and current_position == 0:
                current_position = 1  # Long spread (long stock2, short stock1)
            elif signals.loc[date, 'short_entry'] and current_position == 0:
                current_position = -1  # Short spread (short stock2, long stock1)
            elif signals.loc[date, 'exit'] and current_position != 0:
                current_position = 0  # Close position

            position.loc[date] = current_position

        signals['position'] = position

        # Calculate position sizes for each stock
        signals['stock1_position'] = -signals['position'] * signals['hedge_ratio']  #
Opposite of spread position
        signals['stock2_position'] = signals['position']

        self.pair_signals[f"{stock1}_{stock2}"] = signals

        return signals

    def backtest_pair_strategy(self, pair_signals, transaction_cost=0.001):
        """
        Backtest pairs trading strategy

        Parameters:
        -----------
        pair_signals : pd.DataFrame
            Signals from generate_pair_signals
        transaction_cost : float
            Transaction cost per trade
        """
        # Extract stock names from pair signals
        pair_name = list(self.pair_signals.keys())[0]  # Assuming single pair for now
        stock1, stock2 = pair_name.split('_')

        # Get returns
        ret1 = self.stock_returns[stock1].loc[pair_signals.index]
        ret2 = self.stock_returns[stock2].loc[pair_signals.index]

        # Calculate strategy returns
        strategy_returns = []
        prev_pos1, prev_pos2 = 0, 0

        for date in pair_signals.index:
            pos1 = pair_signals.loc[date, 'stock1_position']
            pos2 = pair_signals.loc[date, 'stock2_position']
```

```
            # Transaction costs
            tc_cost = 0
            if abs(pos1 - prev_pos1) > 0.01:  # Position change
                tc_cost += transaction_cost * abs(pos1 - prev_pos1)
            if abs(pos2 - prev_pos2) > 0.01:
                tc_cost += transaction_cost * abs(pos2 - prev_pos2)

            # Strategy return
            if date in ret1.index and date in ret2.index:
                daily_return = pos1 * ret1.loc[date] + pos2 * ret2.loc[date] - tc_cost
                strategy_returns.append(daily_return)
            else:
                strategy_returns.append(0)

            prev_pos1, prev_pos2 = pos1, pos2

        strategy_returns = pd.Series(strategy_returns, index=pair_signals.index)

        # Performance metrics
        total_return = (1 + strategy_returns).prod() - 1
        annualized_return = (1 + strategy_returns.mean()) ** 252 - 1
        volatility = strategy_returns.std() * np.sqrt(252)
        sharpe_ratio = annualized_return / volatility if volatility > 0 else 0

        # Drawdown analysis
        cumulative_returns = (1 + strategy_returns).cumprod()
        running_max = cumulative_returns.expanding().max()
        drawdown = (cumulative_returns - running_max) / running_max
        max_drawdown = drawdown.min()

        # Trade analysis
        position_changes = pair_signals['position'].diff().abs() > 0
        num_trades = position_changes.sum()

        return {
            'strategy_returns': strategy_returns,
            'cumulative_returns': cumulative_returns,
            'total_return': total_return,
            'annualized_return': annualized_return,
            'volatility': volatility,
            'sharpe_ratio': sharpe_ratio,
            'max_drawdown': max_drawdown,
            'num_trades': num_trades,
            'pair_name': pair_name
        }
```

Fixed Income and Rates Trading

1. Yield Curve Analysis and Trading

```
import pandas as pd
import numpy as np
from scipy.optimize import minimize
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

class YieldCurveAnalyzer:
    """
    Comprehensive yield curve analysis and trading strategies
    """

    def __init__(self, yield_data):
        """
        Initialize yield curve analyzer

        Parameters:
        -----------
        yield_data : pd.DataFrame
            Yield data with maturities as columns (e.g., '3M', '6M', '1Y', '2Y', etc.)
        """
        self.yield_data = yield_data
        self.maturities = self._parse_maturities(yield_data.columns)
        self.curve_metrics = pd.DataFrame(index=yield_data.index)

    def _parse_maturities(self, maturity_labels):
        """
        Convert maturity labels to years
        """
        maturity_years = []

        for label in maturity_labels:
            if 'M' in label:  # Months
                months = int(label.replace('M', ''))
                maturity_years.append(months / 12)
            elif 'Y' in label:  # Years
                years = int(label.replace('Y', ''))
                maturity_years.append(years)
            else:
                # Try to parse as float
                try:
                    maturity_years.append(float(label))
                except:
                    maturity_years.append(np.nan)

        return np.array(maturity_years)

    def calculate_curve_metrics(self):
        """
        Calculate various yield curve metrics
        """
        # Slope (10Y - 2Y)
        if '10Y' in self.yield_data.columns and '2Y' in self.yield_data.columns:
            self.curve_metrics['slope_10y2y'] = self.yield_data['10Y'] - self.yield_dat
a['2Y']

        # Steepness (30Y - 2Y)
        if '30Y' in self.yield_data.columns and '2Y' in self.yield_data.columns:
            self.curve_metrics['steepness_30y2y'] = self.yield_data['30Y'] - self.yield
_data['2Y']

        # Curvature (2 * 5Y - 2Y - 10Y)
        if all(col in self.yield_data.columns for col in ['2Y', '5Y', '10Y']):
```

```
            self.curve_metrics['curvature'] = (2 * self.yield_data['5Y'] -
                                               self.yield_data['2Y'] - self.yield_data['10Y'])

        # Level (average of key rates)
        key_rates = ['2Y', '5Y', '10Y', '30Y']
        available_rates = [rate for rate in key_rates if rate in self.yield_data.columns]
        if available_rates:
            self.curve_metrics['level'] = self.yield_data[available_rates].mean(axis=1)

        # Volatility of level
        if 'level' in self.curve_metrics.columns:
            self.curve_metrics['level_volatility'] = self.curve_metrics['level'].rolling(21).std()

        return self.curve_metrics

    def nelson_siegel_fit(self, date=None):
        """
        Fit Nelson-Siegel model to yield curve

        Parameters:
        -----------
        date : str or pd.Timestamp, optional
            Specific date to fit. If None, fits to latest date.

        Returns:
        --------
        dict : Nelson-Siegel parameters and fitted curve
        """
        if date is None:
            date = self.yield_data.index[-1]

        yields = self.yield_data.loc[date].dropna()
        maturities = self.maturities[~pd.isna(yields)]
        yields = yields.values

        def nelson_siegel(tau, beta0, beta1, beta2, lambda_param):
            """Nelson-Siegel yield curve model"""
            term1 = beta0
            term2 = beta1 * (1 - np.exp(-lambda_param * tau)) / (lambda_param * tau)
            term3 = beta2 * ((1 - np.exp(-lambda_param * tau)) / (lambda_param * tau) -
                             np.exp(-lambda_param * tau))
            return term1 + term2 + term3

        def objective(params):
            """Objective function for optimization"""
            beta0, beta1, beta2, lambda_param = params
            fitted_yields = nelson_siegel(maturities, beta0, beta1, beta2, lambda_param)
            return np.sum((yields - fitted_yields) ** 2)

        # Initial guess
        initial_guess = [yields.mean(), -1, 0, 1]

        # Bounds
        bounds = [(-5, 15), (-15, 5), (-15, 15), (0.1, 5)]

        # Optimize
        result = minimize(objective, initial_guess, bounds=bounds, method='L-BFGS-B')

        if result.success:
```

```
            beta0, beta1, beta2, lambda_param = result.x

            # Generate fitted curve
            fitted_maturities = np.linspace(0.25, 30, 120)  # 3M to 30Y
            fitted_yields = nelson_siegel(fitted_maturities, beta0, beta1, beta2,
lambda_param)

            return {
                'beta0': beta0,  # Level
                'beta1': beta1,  # Slope
                'beta2': beta2,  # Curvature
                'lambda': lambda_param,  # Decay parameter
                'fitted_maturities': fitted_maturities,
                'fitted_yields': fitted_yields,
                'rmse': np.sqrt(result.fun / len(yields)),
                'date': date
            }
        else:
            return {'error': 'Optimization failed'}

    def principal_component_analysis(self, lookback_period=252):
        """
        Perform PCA on yield curve changes

        Parameters:
        -----------
        lookback_period : int
            Lookback period for PCA analysis

        Returns:
        --------
        dict : PCA results
        """
        from sklearn.decomposition import PCA
        from sklearn.preprocessing import StandardScaler

        # Calculate yield changes
        yield_changes = self.yield_data.diff().dropna()

        if len(yield_changes) < lookback_period:
            lookback_period = len(yield_changes)

        # Use recent data
        recent_changes = yield_changes.tail(lookback_period)

        # Standardize
        scaler = StandardScaler()
        scaled_changes = scaler.fit_transform(recent_changes)

        # PCA
        pca = PCA()
        pca_components = pca.fit_transform(scaled_changes)

        # Create component time series
        component_series = pd.DataFrame(
            pca_components,
            index=recent_changes.index,
            columns=[f'PC{i+1}' for i in range(len(recent_changes.columns))]
        )

        # Loadings
        loadings = pd.DataFrame(
            pca.components_.T,
```

```
            index=recent_changes.columns,
            columns=[f'PC{i+1}' for i in range(len(recent_changes.columns))]
        )

        return {
            'explained_variance_ratio': pca.explained_variance_ratio_,
            'cumulative_variance_ratio': np.cumsum(pca.explained_variance_ratio_),
            'components': component_series,
            'loadings': loadings,
            'scaler': scaler,
            'pca_model': pca
        }

    def butterfly_spread_analysis(self, short_maturity='2Y', middle_maturity='5Y',
long_maturity='10Y'):
        """
        Analyze butterfly spread opportunities

        Parameters:
        -----------
        short_maturity, middle_maturity, long_maturity : str
            Maturity labels for butterfly spread

        Returns:
        --------
        dict : Butterfly spread analysis
        """
        required_maturities = [short_maturity, middle_maturity, long_maturity]

        if not all(mat in self.yield_data.columns for mat in required_maturities):
            return {'error': f'Required maturities not available:
{required_maturities}'}

        # Calculate butterfly spread
        # Long wings (short and long maturity), short body (middle maturity)
        butterfly = (self.yield_data[short_maturity] + self.yield_data[long_maturity] -
                     2 * self.yield_data[middle_maturity])

        # Statistics
        butterfly_stats = {
            'current_level': butterfly.iloc[-1],
            'mean': butterfly.mean(),
            'std': butterfly.std(),
            'percentile_25': butterfly.quantile(0.25),
            'percentile_75': butterfly.quantile(0.75),
            'z_score': (butterfly.iloc[-1] - butterfly.mean()) / butterfly.std()
        }

        # Trading signals
        # Buy butterfly when spread is cheap (below 25th percentile)
        # Sell butterfly when spread is rich (above 75th percentile)
        signals = pd.Series(0, index=butterfly.index)
        signals[butterfly < butterfly.quantile(0.25)] = 1   # Buy butterfly
        signals[butterfly > butterfly.quantile(0.75)] = -1  # Sell butterfly

        return {
            'butterfly_spread': butterfly,
            'statistics': butterfly_stats,
            'signals': signals,
            'maturity_structure': {
                'short': short_maturity,
                'middle': middle_maturity,
                'long': long_maturity
```

```
            }
        }

    def carry_roll_down_analysis(self, holding_period_days=30):
        """
        Analyze carry and roll-down for different maturities

        Parameters:
        -----------
        holding_period_days : int
            Holding period for roll-down calculation

        Returns:
        --------
        dict : Carry and roll-down analysis
        """
        results = {}

        for maturity in self.yield_data.columns:
            if maturity in self.maturities:
                maturity_years = self.maturities[list(self.yield_data.columns).in-
dex(maturity)]

                # Current yield (carry)
                current_yield = self.yield_data[maturity].iloc[-1]

                # Estimate roll-down
                # Find next shorter maturity for interpolation
                shorter_maturities = self.maturities[self.maturities < maturity_years]

                if len(shorter_maturities) > 0:
                    # Time to roll down
                    roll_down_time = holding_period_days / 365.25
                    new_maturity = maturity_years - roll_down_time

                    if new_maturity > 0:
                        # Interpolate yield at new maturity
                        current_curve = self.yield_data.iloc[-1].dropna()
                        current_maturities = self.maturities[~pd.isna(current_curve)]

                        if len(current_maturities) > 1:
                            interp_func = interp1d(current_maturities, current_curve.va
lues,
                                                   kind='linear', fill_value='extrapol-
ate')
                            roll_down_yield = interp_func(new_maturity)

                            # Roll-down return (price appreciation)
                            # Approximate duration
                            duration = new_maturity  # Simplified duration estimate
                            roll_down_return = duration * (current_yield -
roll_down_yield) / 100

                            # Total return (carry + roll-down)
                            carry_return = current_yield * (holding_period_days / 365.2
5) / 100
                            total_return = carry_return + roll_down_return

                            results[maturity] = {
                                'current_yield': current_yield,
                                'carry_return': carry_return * 100,  # Convert to per-
centage
                                'roll_down_return': roll_down_return * 100,
```

```
                                'total_return': total_return * 100,
                                'duration': duration
                            }

        # Rank by total return
        if results:
            total_returns = {mat: data['total_return'] for mat, data in results.items()
}
            ranked_maturities = sorted(total_returns.items(), key=lambda x: x[1], re-
verse=True)

            return {
                'maturity_analysis': results,
                'ranked_by_total_return': ranked_maturities,
                'holding_period_days': holding_period_days
            }
        else:
            return {'error': 'Insufficient data for carry/roll-down analysis'}

class BondTradingStrategy:
    """
    Bond trading strategies based on yield curve analysis
    """

    def __init__(self, yield_data, bond_prices=None):
        """
        Initialize bond trading strategy

        Parameters:
        -----------
        yield_data : pd.DataFrame
            Yield curve data
        bond_prices : pd.DataFrame, optional
            Bond price data
        """
        self.yield_data = yield_data
        self.bond_prices = bond_prices
        self.curve_analyzer = YieldCurveAnalyzer(yield_data)

    def duration_neutral_strategy(self, target_duration=5.0):
        """
        Create duration-neutral trading strategy

        Parameters:
        -----------
        target_duration : float
            Target portfolio duration

        Returns:
        --------
        dict : Duration-neutral strategy
        """
        # Calculate duration for each maturity (simplified)
        durations = {}
        for maturity in self.yield_data.columns:
            if 'Y' in maturity:
                years = float(maturity.replace('Y', ''))
                # Simplified duration calculation (modified duration ≈ maturity for
bonds near par)
                durations[maturity] = years * 0.9  # Approximate modified duration
            elif 'M' in maturity:
                months = float(maturity.replace('M', ''))
                durations[maturity] = (months / 12) * 0.9
```

```
        # Current yield levels
        current_yields = self.yield_data.iloc[-1]

        # Strategy: Long bonds with high yield, short bonds with low yield
        # Subject to duration neutrality

        # Rank bonds by yield
        yield_ranking = current_yields.sort_values(ascending=False)

        # Select long and short candidates
        long_candidates = yield_ranking.head(3).index.tolist()  # Top 3 highest yields
        short_candidates = yield_ranking.tail(3).index.tolist()  # Bottom 3 lowest
yields

        # Optimize weights for duration neutrality
        from scipy.optimize import minimize

        def duration_objective(weights):
            """Minimize deviation from target duration"""
            long_weights = weights[:len(long_candidates)]
            short_weights = weights[len(long_candidates):]

            # Portfolio duration
            portfolio_duration = 0
            for i, bond in enumerate(long_candidates):
                if bond in durations:
                    portfolio_duration += long_weights[i] * durations[bond]

            for i, bond in enumerate(short_candidates):
                if bond in durations:
                    portfolio_duration -= short_weights[i] * durations[bond]  # Short
position

            return (portfolio_duration - target_duration) ** 2

        # Constraints
        def weight_constraint(weights):
            """Long and short weights should sum to 1 each"""
            long_weights = weights[:len(long_candidates)]
            short_weights = weights[len(long_candidates):]
            return abs(long_weights.sum() - 1) + abs(short_weights.sum() - 1)

        # Initial guess
        n_long = len(long_candidates)
        n_short = len(short_candidates)
        initial_weights = np.concatenate([
            np.ones(n_long) / n_long,  # Equal long weights
            np.ones(n_short) / n_short  # Equal short weights
        ])

        # Bounds
        bounds = [(0, 1) for _ in range(n_long + n_short)]

        # Constraints
        constraints = [
            {'type': 'eq', 'fun': lambda w: w[:n_long].sum() - 1},  # Long weights sum
to 1
            {'type': 'eq', 'fun': lambda w: w[n_long:].sum() - 1}
# Short weights sum to 1
        ]

        # Optimize
```

```
        result = minimize(duration_objective, initial_weights,
                          bounds=bounds, constraints=constraints, method='SLSQP')

        if result.success:
            optimal_weights = result.x
            long_weights = optimal_weights[:n_long]
            short_weights = optimal_weights[n_long:]

            # Create position dictionary
            positions = {}
            for i, bond in enumerate(long_candidates):
                positions[bond] = long_weights[i]
            for i, bond in enumerate(short_candidates):
                positions[bond] = -short_weights[i]  # Negative for short

            # Calculate actual portfolio duration
            actual_duration = 0
            for bond, weight in positions.items():
                if bond in durations:
                    actual_duration += weight * durations[bond]

            return {
                'positions': positions,
                'target_duration': target_duration,
                'actual_duration': actual_duration,
                'long_candidates': long_candidates,
                'short_candidates': short_candidates,
                'optimization_success': True
            }
        else:
            # Fallback to equal weights
            positions = {}
            for bond in long_candidates:
                positions[bond] = 1 / len(long_candidates)
            for bond in short_candidates:
                positions[bond] = -1 / len(short_candidates)

            return {
                'positions': positions,
                'target_duration': target_duration,
                'optimization_success': False,
                'message': 'Used equal weights fallback'
            }

    def yield_curve_momentum_strategy(self, lookback_period=21):
        """
        Momentum strategy based on yield curve changes

        Parameters:
        -----------
        lookback_period : int
            Lookback period for momentum calculation

        Returns:
        --------
        dict : Momentum strategy signals
        """
        # Calculate yield changes
        yield_changes = self.yield_data.diff(lookback_period)

        # Current momentum
        current_momentum = yield_changes.iloc[-1]
```

```
        # Strategy: Long bonds with falling yields (positive momentum for bond prices)
        # Short bonds with rising yields (negative momentum for bond prices)

        signals = pd.Series(0, index=self.yield_data.columns)

        # Normalize momentum scores
        momentum_zscore = (current_momentum - yield_changes.mean()) / yield_changes.std
()

        # Generate signals based on z-scores
        signals[momentum_zscore < -1] = 1   # Strong negative yield momentum -> Long
bonds
        signals[momentum_zscore > 1] = -1   # Strong positive yield momentum -> Short
bonds

        # Position sizing based on momentum strength
        position_sizes = abs(momentum_zscore) / 2  # Scale by momentum strength
        position_sizes = np.clip(position_sizes, 0, 1)  # Cap at 100%

        final_positions = signals * position_sizes

        return {
            'signals': signals,
            'momentum_scores': current_momentum,
            'momentum_zscores': momentum_zscore,
            'position_sizes': final_positions,
            'lookback_period': lookback_period
        }

    def mean_reversion_strategy(self, lookback_period=63, entry_threshold=1.5,
exit_threshold=0.5):
        """
        Mean reversion strategy for yield spreads

        Parameters:
        -----------
        lookback_period : int
            Lookback period for mean calculation
        entry_threshold : float
            Z-score threshold for entry
        exit_threshold : float
            Z-score threshold for exit

        Returns:
        --------
        dict : Mean reversion strategy
        """
        # Calculate curve metrics
        curve_metrics = self.curve_analyzer.calculate_curve_metrics()

        strategies = {}

        for metric in curve_metrics.columns:
            metric_series = curve_metrics[metric].dropna()

            if len(metric_series) < lookback_period:
                continue

            # Rolling statistics
            rolling_mean = metric_series.rolling(lookback_period).mean()
            rolling_std = metric_series.rolling(lookback_period).std()
            z_scores = (metric_series - rolling_mean) / rolling_std
```

```
            # Generate signals
            signals = pd.Series(0, index=metric_series.index)

            # Entry signals
            signals[z_scores > entry_threshold] = -1  # Mean revert down
            signals[z_scores < -entry_threshold] = 1  # Mean revert up

            # Exit signals
            exit_condition = abs(z_scores) < exit_threshold

            # Create position series
            positions = pd.Series(0, index=metric_series.index)
            current_position = 0

            for date in metric_series.index:
                if signals.loc[date] != 0 and current_position == 0:
                    current_position = signals.loc[date]
                elif exit_condition.loc[date] and current_position != 0:
                    current_position = 0

                positions.loc[date] = current_position

            strategies[metric] = {
                'z_scores': z_scores,
                'signals': signals,
                'positions': positions,
                'current_zscore': z_scores.iloc[-1],
                'current_position': positions.iloc[-1]
            }

        return strategies
```

Commodities and Futures Trading

1. Commodity-Specific Strategies

```python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

class CommodityTradingStrategy:
"""
Commodity-specific trading strategies
"""
```

```
def __init__(self, commodity_data, commodity_type='energy'):
    """
    Initialize commodity trading strategy

    Parameters:
    -----------
    commodity_data : pd.DataFrame
        Commodity price data
    commodity_type : str
        Type of commodity ('energy', 'metals', 'agriculture')
    """
    self.commodity_data = commodity_data
    self.commodity_type = commodity_type
    self.seasonal_patterns = {}
    self.storage_costs = self._get_storage_costs()

def _get_storage_costs(self):
    """
    Get typical storage costs by commodity type
    """
    storage_costs = {
        'energy': {
            'crude_oil': 0.02,  # 2% per year
            'natural_gas': 0.15,  # 15% per year (high storage cost)
            'gasoline': 0.03
        },
        'metals': {
            'gold': 0.005,  # 0.5% per year
            'silver': 0.01,
            'copper': 0.02,
            'aluminum': 0.015
        },
        'agriculture': {
            'corn': 0.05,  # 5% per year
            'wheat': 0.04,
            'soybeans': 0.06,
            'coffee': 0.08,
            'sugar': 0.07
        }
    }

    return storage_costs.get(self.commodity_type, {})

def seasonal_analysis(self, years_lookback=10):
    """
    Analyze seasonal patterns in commodity prices

    Parameters:
    -----------
    years_lookback : int
        Number of years to analyze for seasonal patterns

    Returns:
    --------
    dict : Seasonal analysis results
    """
    results = {}

    for commodity in self.commodity_data.columns:
        price_series = self.commodity_data[commodity].dropna()

        # Filter to recent years
```

```
        cutoff_date = price_series.index[-1] - pd.DateOffset(years=years_lookback)
        recent_prices = price_series[price_series.index >= cutoff_date]

        if len(recent_prices) < 252:  # Need at least 1 year of data
            continue

        # Calculate monthly returns
        monthly_returns = recent_prices.resample('M').last().pct_change().dropna()

        # Group by month
        monthly_stats = monthly_returns.groupby(monthly_returns.index.month).agg({
            monthly_returns.name: ['mean', 'std', 'count']
        }).round(4)

        monthly_stats.columns = ['mean_return', 'std_return', 'count']
        monthly_stats.index.name = 'month'

        # Calculate seasonal strength (coefficient of variation of monthly means)
        seasonal_strength = monthly_stats['mean_return'].std() / abs(monthly_stats['mea
n_return'].mean())

        # Best and worst months
        best_month = monthly_stats['mean_return'].idxmax()
        worst_month = monthly_stats['mean_return'].idxmin()

        # Statistical significance test
        from scipy.stats import f_oneway
        monthly_groups = [monthly_returns[monthly_returns.index.month == month].values
                          for month in range(1, 13)]
        monthly_groups = [group for group in monthly_groups if len(group) > 0]

        if len(monthly_groups) > 1:
            f_stat, p_value = f_oneway(*monthly_groups)
            is_significant = p_value < 0.05
        else:
            f_stat, p_value, is_significant = np.nan, np.nan, False

        results[commodity] = {
            'monthly_stats': monthly_stats,
            'seasonal_strength': seasonal_strength,
            'best_month': best_month,
            'worst_month': worst_month,
            'best_month_return': monthly_stats.loc[best_month, 'mean_return'],
            'worst_month_return': monthly_stats.loc[worst_month, 'mean_return'],
            'f_statistic': f_stat,
            'p_value': p_value,
            'is_statistically_significant': is_significant
        }

    self.seasonal_patterns = results
    return results

def contango_backwardation_analysis(self, front_month_col, back_month_col):
    """
    Analyze contango/backwardation patterns

    Parameters:
    -----------
    front_month_col : str
        Column name for front month futures
    back_month_col : str
        Column name for back month futures
```

```
    Returns:
    --------
    dict : Contango/backwardation analysis
    """
    if front_month_col not in self.commodity_data.columns or back_month_col not in self
.commodity_data.columns:
        return {'error': 'Required columns not found'}

    front_prices = self.commodity_data[front_month_col].dropna()
    back_prices = self.commodity_data[back_month_col].dropna()

    # Align data
    common_dates = front_prices.index.intersection(back_prices.index)
    front_aligned = front_prices.loc[common_dates]
    back_aligned = back_prices.loc[common_dates]

    # Calculate spread (back - front)
    spread = back_aligned - front_aligned
    spread_pct = (back_aligned - front_aligned) / front_aligned * 100

    # Classify market structure
    market_structure = pd.Series('normal', index=common_dates)
    market_structure[spread > 0] = 'contango'
    market_structure[spread < 0] = 'backwardation'

    # Statistics
    contango_pct = (market_structure == 'contango').mean() * 100
    backwardation_pct = (market_structure == 'backwardation').mean() * 100

    # Average spreads in each regime
    contango_avg_spread = spread[market_structure == 'contango'].mean()
    backwardation_avg_spread = spread[market_structure == 'backwardation'].mean()

    # Persistence analysis
    structure_changes = market_structure != market_structure.shift(1)
    avg_regime_length = len(market_structure) / structure_changes.sum()

    return {
        'spread': spread,
        'spread_percentage': spread_pct,
        'market_structure': market_structure,
        'contango_percentage': contango_pct,
        'backwardation_percentage': backwardation_pct,
        'avg_contango_spread': contango_avg_spread,
        'avg_backwardation_spread': backwardation_avg_spread,
        'avg_regime_length_days': avg_regime_length,
        'current_structure': market_structure.iloc[-1],
        'current_spread': spread.iloc[-1]
    }

def roll_yield_strategy(self, front_month_col, back_month_col, roll_threshold=0.02):
    """
    Roll yield trading strategy

    Parameters:
    -----------
    front_month_col : str
        Front month futures column
    back_month_col : str
        Back month futures column
    roll_threshold : float
        Threshold for roll yield signal
```

```
    Returns:
    --------
    dict : Roll yield strategy
    """
    # Get contango/backwardation analysis
    curve_analysis = self.contango_backwardation_analysis(front_month_col, back_month_c
ol)

    if 'error' in curve_analysis:
        return curve_analysis

    spread_pct = curve_analysis['spread_percentage']

    # Generate signals based on roll yield
    signals = pd.Series(0, index=spread_pct.index)

    # Long when in steep backwardation (negative roll yield for long positions)
    signals[spread_pct < -roll_threshold * 100] = 1

    # Short when in steep contango (positive roll yield for short positions)
    signals[spread_pct > roll_threshold * 100] = -1

    # Calculate strategy returns (simplified)
    front_returns = self.commodity_data[front_month_col].pct_change()
    strategy_returns = signals.shift(1) * front_returns
    strategy_returns = strategy_returns.dropna()

    # Performance metrics
    total_return = (1 + strategy_returns).prod() - 1
    annualized_return = (1 + strategy_returns.mean()) ** 252 - 1
    volatility = strategy_returns.std() * np.sqrt(252)
    sharpe_ratio = annualized_return / volatility if volatility > 0 else 0

    return {
        'signals': signals,
        'strategy_returns': strategy_returns,
        'total_return': total_return,
        'annualized_return': annualized_return,
        'volatility': volatility,
        'sharpe_ratio': sharpe_ratio,
        'roll_threshold': roll_threshold,
        'curve_analysis': curve_analysis
    }

def inventory_momentum_strategy(self, inventory_data, price_col):
    """
    Strategy based on inventory levels and momentum

    Parameters:
    -----------
    inventory_data : pd.Series
        Inventory level data
    price_col : str
        Price column to trade

    Returns:
    --------
    dict : Inventory momentum strategy
    """
    if price_col not in self.commodity_data.columns:
        return {'error': f'Price column {price_col} not found'}

    prices = self.commodity_data[price_col].dropna()
```

```
    # Align inventory and price data
    common_dates = prices.index.intersection(inventory_data.index)
    prices_aligned = prices.loc[common_dates]
    inventory_aligned = inventory_data.loc[common_dates]

    # Calculate inventory metrics
    inventory_change = inventory_aligned.diff()
    inventory_zscore = (inventory_aligned - inventory_aligned.rolling(252).mean()) / in
ventory_aligned.rolling(252).std()

    # Price momentum
    price_momentum = prices_aligned.pct_change(21)  # 1-month momentum

    # Generate signals
    signals = pd.Series(0, index=common_dates)

    # Low inventory + positive momentum = Long
    signals[(inventory_zscore < -1) & (price_momentum > 0.05)] = 1

    # High inventory + negative momentum = Short
    signals[(inventory_zscore > 1) & (price_momentum < -0.05)] = -1

    # Calculate returns
    price_returns = prices_aligned.pct_change()
    strategy_returns = signals.shift(1) * price_returns
    strategy_returns = strategy_returns.dropna()

    # Performance metrics
    if len(strategy_returns) > 0:
        total_return = (1 + strategy_returns).prod() - 1
        annualized_return = (1 + strategy_returns.mean()) ** 252 - 1
        volatility = strategy_returns.std() * np.sqrt(252)
        sharpe_ratio = annualized_return / volatility if volatility > 0 else 0
    else:
        total_return = annualized_return = volatility = sharpe_ratio = 0

    return {
        'signals': signals,
        'inventory_zscore': inventory_zscore,
        'price_momentum': price_momentum,
        'strategy_returns': strategy_returns,
        'total_return': total_return,
        'annualized_return': annualized_return,
        'volatility': volatility,
        'sharpe_ratio': sharpe_ratio
    }

def weather_impact_strategy(self, weather_data, commodity='corn'):
    """
    Weather-based trading strategy for agricultural commodities

    Parameters:
    -----------
    weather_data : pd.DataFrame
        Weather data with temperature, precipitation, etc.
    commodity : str
        Commodity to trade

    Returns:
    --------
    dict : Weather impact strategy
    """
```

```
    if commodity not in self.commodity_data.columns:
        return {'error': f'Commodity {commodity} not found'}

    prices = self.commodity_data[commodity].dropna()

    # Align weather and price data
    common_dates = prices.index.intersection(weather_data.index)
    prices_aligned = prices.loc[common_dates]
    weather_aligned = weather_data.loc[common_dates]

    # Weather stress indicators
    weather_stress = pd.Series(0, index=common_dates)

    if 'temperature' in weather_aligned.columns:
        # Temperature stress (too hot or too cold)
        temp_zscore = (weather_aligned['temperature'] - weather_aligned['temperature'].
rolling(30).mean()) / weather_aligned['temperature'].rolling(30).std()
        weather_stress += abs(temp_zscore) > 2

    if 'precipitation' in weather_aligned.columns:
        # Precipitation stress (too dry or too wet)
        precip_zscore = (weather_aligned['precipitation'] - weather_aligned['precipita-
tion'].rolling(30).mean()) / weather_aligned['precipitation'].rolling(30).std()
        weather_stress += abs(precip_zscore) > 2

    # Generate signals
    signals = pd.Series(0, index=common_dates)

    # Long during weather stress (expect higher prices)
    signals[weather_stress > 1] = 1

    # Consider seasonal timing
    if hasattr(self, 'seasonal_patterns') and commodity in self.seasonal_patterns:
        seasonal_data = self.seasonal_patterns[commodity]

        # Enhance signals during historically strong months
        for date in signals.index:
            month = date.month
            if month in seasonal_data['monthly_stats'].index:
                monthly_return = seasonal_data['monthly_stats'].loc[month,
'mean_return']
                if monthly_return > 0.02:  # Strong seasonal month
                    signals.loc[date] *= 1.5  # Increase signal strength

    # Cap signals at +/-1
    signals = np.clip(signals, -1, 1)

    # Calculate returns
    price_returns = prices_aligned.pct_change()
    strategy_returns = signals.shift(1) * price_returns
    strategy_returns = strategy_returns.dropna()

    # Performance metrics
    if len(strategy_returns) > 0:
        total_return = (1 + strategy_returns).prod() - 1
        annualized_return = (1 + strategy_returns.mean()) ** 252 - 1
        volatility = strategy_returns.std() * np.sqrt(252)
        sharpe_ratio = annualized_return / volatility if volatility > 0 else 0
    else:
        total_return = annualized_return = volatility = sharpe_ratio = 0

    return {
        'signals': signals,
```

```
        'weather_stress': weather_stress,
        'strategy_returns': strategy_returns,
        'total_return': total_return,
        'annualized_return': annualized_return,
        'volatility': volatility,
        'sharpe_ratio': sharpe_ratio
    }
```

class FuturesRollStrategy:
“””
Futures roll and calendar spread strategies
“””

```
def __init__(self, futures_data):
    """
    Initialize futures roll strategy

    Parameters:
    -----------
    futures_data : dict
        Dictionary with contract months as keys, price series as values
    """
    self.futures_data = futures_data
    self.contract_months = list(futures_data.keys())
    self.roll_dates = self._identify_roll_dates()

def _identify_roll_dates(self):
    """
    Identify typical roll dates for futures contracts
    """
    # Simplified: assume monthly rolls on the 15th
    roll_dates = []

    if self.futures_data:
        first_contract = list(self.futures_data.values())[0]
        for date in first_contract.index:
            if date.day == 15:  # Roll on 15th of each month
                roll_dates.append(date)

    return roll_dates

def calendar_spread_strategy(self, front_contract, back_contract,
                             entry_threshold=2.0, exit_threshold=0.5):
    """
    Calendar spread trading strategy

    Parameters:
    -----------
    front_contract : str
        Front month contract
    back_contract : str
        Back month contract
    entry_threshold : float
        Z-score threshold for entry
    exit_threshold : float
        Z-score threshold for exit

    Returns:
    --------
    dict : Calendar spread strategy
    """
    if front_contract not in self.futures_data or back_contract not in self.fu-
tures_data:
        return {'error': 'Required contracts not found'}

    front_prices = self.futures_data[front_contract].dropna()
    back_prices = self.futures_data[back_contract].dropna()

    # Align data
    common_dates = front_prices.index.intersection(back_prices.index)
    front_aligned = front_prices.loc[common_dates]
    back_aligned = back_prices.loc[common_dates]

    # Calculate spread
    spread = back_aligned - front_aligned
```

```
    # Rolling statistics
    spread_mean = spread.rolling(63).mean()
    spread_std = spread.rolling(63).std()
    spread_zscore = (spread - spread_mean) / spread_std

    # Generate signals
    signals = pd.Series(0, index=common_dates)
    positions = pd.Series(0, index=common_dates)

    current_position = 0

    for date in common_dates:
        zscore = spread_zscore.loc[date]

        if pd.notna(zscore):
            # Entry signals
            if zscore > entry_threshold and current_position == 0:
                current_position = -1  # Short spread (short back, long front)
            elif zscore < -entry_threshold and current_position == 0:
                current_position = 1   # Long spread (long back, short front)

            # Exit signals
            elif abs(zscore)
```