Comprehensive Guide to Cox Probabilistic Trading System

June 29, 2025

Contents

1 Introduction

The Cox Probabilistic Trading System is a Bayesian framework for financial trading that combines:

- Probabilistic market state modeling
- Entropy-based risk management
- Adaptive feature relevance
- Ensemble backtesting

Two implementations are presented:

1. **Continuous Feature Distribution**: Uses normal distributions for feature evidence
2. **Binary Feature System**: Simplified implementation with binary features

2 Mathematical Foundations

2.1 Bayesian Belief Updating

The core of the system uses Bayes' theorem to update market state probabilities:

$$P(S|E) = \frac{P(E|S) \cdot P(S)}{P(E)} \tag{1}$$

Where:

- $P(S|E)$: Posterior probability of state S given evidence E
- $P(E|S)$: Likelihood of evidence E given state S
- $P(S)$: Prior probability of state S
- $P(E)$: Total probability of evidence E

In simplified form for binary evidence:

$$P_{\text{new}} = \frac{P_{\text{prior}} \cdot E}{P_{\text{prior}} \cdot E + (1 - P_{\text{prior}}) \cdot (1 - E)} \tag{2}$$

2.2 Entropy-Based Signals

Shannon entropy determines trading signal confidence:

$$H(p) = -p \log_2(p) - (1 - p) \log_2(1 - p) \tag{3}$$

Trading signals generated when $H(p) < \tau$:

$$\begin{aligned} &\text{BUY} && \text{if} \quad p > \theta_{\text{trade}} \\ &\text{SELL} && \text{if} \quad p < 1 - \theta_{\text{trade}} \end{aligned}$$

2.3 Risk Management

Position sizing based on entropy-price covariance:

$$\text{Risk Factor} = \text{clip}(1 - \tanh(10 \cdot \text{Cov}(H, r), 0.1, 0.9) \tag{4}$$

Where r are portfolio returns.

3 Implementation 1: Continuous Feature Distribution

3.1 Class Architecture

```python
class CoxProbabilisticTrader:
def __init__(self, proposition_system, feature_functions,
lookback_window=30, entropy_threshold=0.2,
prior_belief=0.5, state_k=0.5, trade_threshold=0.7):
# Initialization parameters
self.propositions = proposition_system
```

```python
self.feature_functions = feature_functions
self.lookback = lookback_window
self.entropy_threshold = entropy_threshold
self.prior = prior_belief
self.state_k = state_k
self.trade_threshold = trade_threshold

# State tracking
self.probability_map = defaultdict(lambda: prior_belief)
self.relevance_weights = self._initialize_relevance_weights()
self.entropy_history = []
self.portfolio_log = []
self.market_states = []
```

3.2 Key Methods

3.2.1 Feature Processing

$$\text{Feature}_i = f_i(\text{returns}) \tag{5}$$

```python
def process_market_data(self, prices):
returns = np.diff(prices)/prices[:-1]
features = {}
for feature, func in self.feature_functions.items():
features[feature] = func(returns)
market_state = self._state_from_features(features)
return market_state, features
```

3.2.2 State Discretization

$$\text{State} = \begin{cases} 0 & \text{if } x < \mu - k\sigma \\ 1 & \text{if } \mu - k\sigma \leq x \leq \mu + k\sigma \\ 2 & \text{if } x > \mu + k\sigma \end{cases} \tag{6}$$

```python
def _state_from_features(self, features):
state_vector = []
for feature in sorted(features.keys()):
value = features[feature]
mean, std = self.propositions[feature]
if value < mean - self.state_k * std:
state_val = 0
elif value > mean + self.state_k * std:
state_val = 2
else:
state_val = 1
state_vector.append(str(state_val))
return "_".join(state_vector)
```

3.2.3 Evidence Calculation

$$\text{Evidence} = \sum w_i \cdot \Phi\left(\frac{x_i - \mu_i}{\sigma_i}\right) \tag{7}$$

Where Φ is the CDF of standard normal distribution.

3.3 Example Usage

```python
# Feature calculation functions
def momentum_func(returns):
return np.prod(1 + returns) - 1

# Configuration
proposition_system = {
  'momentum': (0.05, 0.02),
  'volatility': (0.015, 0.005)
}

feature_functions = {
  'momentum': momentum_func,
  'volatility': lambda r: np.std(r)
}

# Initialize trader
trader = CoxProbabilisticTrader(
proposition_system=proposition_system,
feature_functions=feature_functions,
lookback_window=14,
entropy_threshold=0.15,
state_k=0.5
)

# Backtest on S&P 500 data
data = yf.download("SPY", start="2020-01-01", end="2023-01-01")
results = trader.backtest(data['Close'].values, n_ensembles=10)
```

4 Implementation 2: Binary Feature System

4.1 Class Architecture

```python
class CoxProbabilisticTrader:
def __init__(self, lookback_window=30, entropy_threshold=0.2,
prior_belief=0.5, gamma=0.75, momentum_threshold=0.0,
sharpe_threshold=1.0, utility_threshold=0.0,
trade_threshold=0.7):
# Parameters
self.lookback = lookback_window
self.entropy_threshold = entropy_threshold
self.prior = prior_belief
self.gamma = gamma
self.momentum_threshold = momentum_threshold
self.sharpe_threshold = sharpe_threshold
self.utility_threshold = utility_threshold
self.trade_threshold = trade_threshold

# State tracking
self.probability_map = defaultdict(lambda: prior_belief)
self.entropy_history = []
self.portfolio_log = []
self.market_states = []
```

4.2 Feature Engineering

Three binary features computed from returns window r:

4.2.1 Momentum

$$\text{Momentum} = \mathbb{I}\left(\prod_{i=1}^{n}(1 + r_i) - 1 > \theta_m\right) \tag{8}$$

4.2.2 Sharpe Ratio

$$\text{Sharpe} = \mathbb{I}\left(\frac{\bar{r}}{\sigma_r} > \theta_s\right) \tag{9}$$

4.2.3 Utility

$$\text{Utility} = \mathbb{I}\left(\bar{r} - \gamma\sigma_r > \theta_u\right) \tag{10}$$

```python
def process_market_data(self, prices):
returns = np.diff(prices) / prices[:-1]
window = returns[-self.lookback:]

momentum = np.prod(1 + window) - 1
sharpe = np.mean(window) / np.std(window) if np.std(window) > 0 else 0
utility = np.mean(window) - self.gamma * np.std(window)

features = {
  'momentum': 1 if momentum > self.momentum_threshold else 0,
  'sharpe': 1 if sharpe > self.sharpe_threshold else 0,
  'utility': 1 if utility > self.utility_threshold else 0
}

market_state = f"{features['momentum']}{features['sharpe']}{features['utility']}"
return market_state, features

```

4.3 Evidence Calculation

$$\text{Evidence} = \frac{1}{n}\sum_{i=1}^{n}\text{Feature}_i \tag{11}$$

4.4 Example Usage

```python
trader = CoxProbabilisticTrader(
lookback_window=20,
entropy_threshold=0.18,
prior_belief=0.4,
gamma=0.8,
momentum_threshold=0.03,
sharpe_threshold=0.8,
trade_threshold=0.65
)

data = yf.download("AAPL", period="2y")
results = trader.backtest(data['Close'].values, n_ensembles=20)

print(f"Success Probability: {results['success_probability']:.2%}")
print(f"Expected Return: {results['expected_return']:.2%}")

```

5 Backtesting Framework

5.1 Ensemble Backtesting

Monte Carlo simulation with volatility-preserving perturbations:

$$r_{\text{synthetic}} \sim \mathcal{N}(0, \hat{\sigma}) \tag{12}$$

$$P_t = P_0 \prod_{i=1}^{t} (1 + r_i) \tag{13}$$

5.2 Performance Metrics

- Cumulative Return: $R_T = P_T / P_0 - 1$
- Sharpe Ratio: $S = \frac{\mu_r}{\sigma_r}$
- Maximum Drawdown: $\text{MDD} = \max_{t<\tau} \frac{P_\tau - P_t}{P_\tau}$
- Success Probability: $\mathbb{P}(R > 0)$
- Return Entropy: $H_R = -\sum p_i \log_2 p_i$

5.3 Result Interpretation

Metric	**Interpretation**
Success Probability > 0.55	Robust strategy
Mean Sharpe > 0.8	Good risk-adjusted returns
Mean MDD < 0.15	Acceptable risk profile
Return Entropy < 1.5	Consistent performance

Table 1: Performance benchmark guidelines

6 Conclusion

The Cox Probabilistic Trading System provides:

- A Bayesian framework for market state modeling
- Entropy-based position sizing and risk management
- Robust backtesting with Monte Carlo simulations

The two implementations offer complementary approaches:

- **Continuous**: More nuanced feature representation
- **Binary**: Simpler implementation with discrete states

Optimal parameter configuration requires:

- Historical calibration to asset characteristics
- Walk-forward validation
- Sensitivity analysis