Module 10: Backtesting, Evaluation, and Trading Strategies

Course: Bayesian Regression and Time Series Forecasting for Commodities Trading

CAPSTONE MODULE: Integrating Everything We've Learned

Learning Objectives

By the end of this module, you will be able to:

1. **Implement** walk-forward validation to avoid look-ahead bias in Bayesian models
2. **Evaluate** probabilistic forecasts using proper scoring rules (CRPS, log-score)
3. **Assess** forecast calibration and reliability for risk management
4. **Calculate** probabilistic Sharpe ratios accounting for parameter uncertainty
5. **Apply** Kelly criterion with Bayesian posteriors for optimal position sizing
6. **Build** complete trading strategies:
    - Mean reversion with uncertainty bands
    - Trend following with regime detection
    - Pairs trading with Bayesian cointegration
7. **Optimize** portfolios using Bayesian return distributions
8. **Deploy** a complete energy portfolio trading system

Why This Matters for Trading

This is where theory meets reality. Everything we've learned—priors, MCMC, time series mod volatility—means nothing if we can't:

1. **Backtest correctly**: Avoid overfitting and look-ahead bias
2. **Evaluate honestly**: Use metrics that reward calibrated forecasts, not just accuracy
3. **Size positions optimally**: Bayesian Kelly criterion with parameter uncertainty
4. **Manage risk systematically**: Probabilistic stop-losses, portfolio constraints
5. **Execute in practice**: Handle transaction costs, slippage, margin requirements

Real-World Trading Failures

- **Long-Term Capital Management (1998)**: Models didn't account for parameter uncer over-leveraged → $4B loss
- **Amaranth Advisors (2006)**: Natural gas volatility model failed during regime change loss in one week
- **Quantitative funds (August 2007)**: Models over-fit to in-sample data, failed out-of- → multi-billion dollar losses

The pattern: Great models, terrible backtesting/risk management.

What Makes This Module Different

- **No look-ahead bias**: Walk-forward validation
- **Honest evaluation**: Probabilistic metrics (CRPS, calibration)
- **Risk-first thinking**: Position sizing before alpha generation
- **Complete systems**: Not just signals, but full trading workflows

---

In [ ]:
```
# Setup
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import pymc as pm
import arviz as az
import warnings
warnings.filterwarnings('ignore')

# Import course utilities
import sys
sys.path.append('../..')
from utils.backtesting import WalkForwardValidator, Backtester, kelly_pos
from utils.metrics import crps, sharpe_ratio, forecast_summary, mae, rmse

np.random.seed(42)
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (14, 6)
plt.rcParams['font.size'] = 11

print("All libraries and utilities loaded successfully!")
print(f"PyMC version: {pm.__version__}")
```

1. Walk-Forward Validation: The Gold Standard

1.1 The Problem: In-Sample Overfitting

Common mistake: Fit model on all historical data, report performance.

Why it fails:

- Model "sees" future data during fitting
- Overfits to noise in the training period
- Out-of-sample performance much worse

1.2 Walk-Forward Validation

Idea: Simulate real-time trading by:

1. Fit model on data up to time t
2. Forecast for time $t+1, ..., t+k$
3. Observe actual outcomes

4. **Roll forward** to $t+k$, refit, repeat

Two types:

- **Expanding window**: Training set grows (use all historical data)
- **Rolling window**: Fixed training size (only recent data)

1.3 Bayesian Walk-Forward

Standard: Refit model from scratch each period (slow).

Bayesian: Can use **sequential updating**:

- Previous posterior becomes next prior
- Much faster than full refit
- Naturally adapts to regime changes

$$p(\theta | \text{data}_{1:t+1}) \propto p(\text{data}_{t+1} | \theta) \cdot p(\theta | \text{data}$$

In [ ]:
```python
# Generate realistic commodity data for backtesting
def generate_commodity_data(n=500, seed=42):
    """
    Generate synthetic crude oil prices with regime changes.
    """
    np.random.seed(seed)
    dates = pd.date_range('2020-01-01', periods=n, freq='D')

    # Generate returns with multiple regimes
    returns = np.zeros(n)
    volatility = np.zeros(n)

    # Regime 1: Normal (days 0-200)
    vol1 = 0.015
    returns[:200] = np.random.normal(0.0003, vol1, 200)
    volatility[:200] = vol1

    # Regime 2: High volatility (days 200-350) - COVID-like shock
    vol2 = 0.04
    returns[200:350] = np.random.normal(-0.001, vol2, 150)
    volatility[200:350] = vol2

    # Regime 3: Recovery (days 350-500)
    vol3 = 0.02
    returns[350:] = np.random.normal(0.0008, vol3, n - 350)
    volatility[350:] = vol3

    # Convert to prices
    prices = 60 * np.exp(np.cumsum(returns))

    df = pd.DataFrame({
        'date': dates,
        'close': prices,
        'returns': returns,
        'true_vol': volatility
    })
```

```
    df.set_index('date', inplace=True)

    return df

# Generate data
data = generate_commodity_data(n=500)

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 8))

# Prices
ax = axes[0]
ax.plot(data.index, data['close'], linewidth=1.5, color='black')
ax.axvspan(data.index[200], data.index[350], alpha=0.2, color='red', labe
ax.set_ylabel('Price ($/barrel)', fontsize=11)
ax.set_title('Crude Oil Prices (Synthetic)', fontsize=12, fontweight='bol
ax.legend()
ax.grid(True, alpha=0.3)

# Returns
ax = axes[1]
ax.plot(data.index, data['returns'], linewidth=0.8, color='blue', alpha=0
ax.axhline(0, color='red', linestyle='--', linewidth=1)
ax.axvspan(data.index[200], data.index[350], alpha=0.2, color='red')
ax.set_ylabel('Returns', fontsize=11)
ax.set_xlabel('Date', fontsize=11)
ax.set_title('Daily Returns (showing regime change)', fontsize=12, fontwe
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nDataset: {len(data)} days from {data.index[0].date()} to {data.
print(f"\nRegime statistics:")
print(f"  Normal period (days 1-200):      Vol = {data['true_vol'][:200].
print(f"  Crisis period (days 201-350):    Vol = {data['true_vol'][200:35
print(f"  Recovery period (days 351-500):  Vol = {data['true_vol'][350:].
```

In [ ]:

```
# Implement walk-forward backtesting with Bayesian model
def bayesian_ar_model(train_data, n_lags=5):
    """
    Fit Bayesian AR(p) model for returns.
    """
    returns = train_data['returns'].values

    # Create lagged features
    X = np.column_stack([returns[n_lags-i-1:-i-1] for i in range(n_lags)]
    y = returns[n_lags:]

    with pm.Model() as model:
        # Priors
        beta = pm.Normal('beta', mu=0, sigma=0.5, shape=n_lags)
        sigma = pm.HalfNormal('sigma', sigma=0.02)

        # Likelihood
        mu = pm.math.dot(X, beta)
        y_obs = pm.Normal('y_obs', mu=mu, sigma=sigma, observed=y)

        # Sample
        trace = pm.sample(500, tune=500, chains=2, random_seed=42,
```

```
                        progressbar=False, return_inferencedata=True)

    return trace, model

def predict_bayesian(trace, test_data, n_lags=5, n_samples=1000):
    """
    Generate probabilistic forecasts from Bayesian AR model.
    """
    # Get last n_lags returns for prediction
    last_returns = test_data['returns'].values[:n_lags][::-1]  # Reverse

    # Extract posterior samples
    beta_samples = trace.posterior['beta'].values.reshape(-1, n_lags)
    sigma_samples = trace.posterior['sigma'].values.flatten()

    # Generate predictions
    n_post_samples = len(beta_samples)
    predictions = np.zeros(n_post_samples)

    for i in range(n_post_samples):
        mu = np.dot(last_returns, beta_samples[i])
        predictions[i] = mu + sigma_samples[i] * np.random.randn()

    # Summary statistics
    pred_mean = predictions.mean()
    pred_std = predictions.std()

    return pd.DataFrame({
        'prediction': [pred_mean],
        'lower_95': [np.percentile(predictions, 2.5)],
        'upper_95': [np.percentile(predictions, 97.5)],
        'std': [pred_std]
    }, index=test_data.index[:1])

# Run walk-forward validation
print("Running walk-forward validation...")
print("This simulates real-time trading (no look-ahead bias)\n")

# Parameters
min_train_size = 100
test_size = 1  # 1-day ahead forecasts
n_folds = 50  # 50 forecast periods

# Storage
wf_results = []
fold_count = 0

# Walk-forward loop
for i in range(n_folds):
    # Define train/test split
    train_end = min_train_size + i * 5  # Expand training window
    test_start = train_end
    test_end = test_start + test_size

    if test_end > len(data):
        break

    train = data.iloc[:train_end]
    test = data.iloc[test_start:test_end]
```

```
    # Fit model on training data
    try:
        trace, model = bayesian_ar_model(train, n_lags=5)

        # Generate forecast
        forecast = predict_bayesian(trace, test, n_lags=5)
        forecast['actual'] = test['returns'].values[0]
        forecast['fold'] = fold_count

        wf_results.append(forecast)
        fold_count += 1

        if fold_count % 10 == 0:
            print(f"Completed fold {fold_count}/{n_folds}")
    except:
        print(f"Fold {i} failed, skipping...")
        continue

# Combine results
wf_df = pd.concat(wf_results)

print(f"\nWalk-forward validation complete: {len(wf_df)} forecasts genera
```

In [ ]:

```
# Evaluate walk-forward results
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# Plot 1: Forecasts vs actuals
ax = axes[0]
ax.scatter(range(len(wf_df)), wf_df['actual'], c='black', s=40,
           alpha=0.6, label='Actual returns', zorder=3)
ax.plot(range(len(wf_df)), wf_df['prediction'], 'blue', linewidth=2,
        label='Forecast (posterior mean)', zorder=2)
ax.fill_between(range(len(wf_df)),
                wf_df['lower_95'],
                wf_df['upper_95'],
                alpha=0.3, color='blue', label='95% credible interval', z
ax.axhline(0, color='red', linestyle='--', linewidth=1, alpha=0.5)
ax.set_xlabel('Forecast Period', fontsize=11)
ax.set_ylabel('Returns', fontsize=11)
ax.set_title('Walk-Forward Validation: Out-of-Sample Forecasts',
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Plot 2: Forecast errors
ax = axes[1]
errors = wf_df['actual'] - wf_df['prediction']
ax.hist(errors, bins=25, alpha=0.7, density=True, color='steelblue',
        edgecolor='black', label='Forecast errors')
ax.axvline(0, color='red', linestyle='--', linewidth=2, label='Zero error
ax.axvline(errors.mean(), color='green', linestyle='-', linewidth=2,
           label=f'Mean error = {errors.mean():.6f}')
ax.set_xlabel('Forecast Error (actual - predicted)', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Distribution of Forecast Errors', fontsize=12, fontweight='
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
# Calculate metrics
mae_score = mae(wf_df['actual'].values, wf_df['prediction'].values)
rmse_score = rmse(wf_df['actual'].values, wf_df['prediction'].values)
coverage = np.mean((wf_df['actual'] >= wf_df['lower_95']) &
                   (wf_df['actual'] <= wf_df['upper_95']))

print("\n" + "="*70)
print("WALK-FORWARD VALIDATION RESULTS")
print("="*70)
print(f"\nForecast Accuracy:")
print(f"  MAE:  {mae_score:.6f}")
print(f"  RMSE: {rmse_score:.6f}")
print(f"\nCalibration:")
print(f"  95% Interval Coverage: {coverage:.1%}")
print(f"  Target: 95% (well-calibrated forecasts)")
if coverage < 0.90:
    print(f"  ⚠ Under-confident: Intervals too narrow")
elif coverage > 0.98:
    print(f"  ⚠ Over-confident: Intervals too wide")
else:
    print(f"  ✓ Well-calibrated forecasts!")

print(f"\nForecast Bias:")
print(f"  Mean error: {errors.mean():.6f}")
if abs(errors.mean()) < 0.001:
    print(f"  ✓ Unbiased forecasts")
else:
    print(f"  ⚠ Systematic bias detected")

print("\n" + "="*70)
print("KEY INSIGHTS")
print("="*70)
print("""
Walk-forward validation ensures:
1. NO LOOK-AHEAD BIAS - Model never sees future data
2. REALISTIC PERFORMANCE - Simulates actual trading
3. OUT-OF-SAMPLE EVALUATION - True test of generalization

Coverage = {:.1%} means:
- {:.1%} of actual returns fell within 95% prediction intervals
- Close to 95% target → forecasts are well-calibrated
- Important for risk management (VaR, stop-losses)

**Trading implication**:
Well-calibrated intervals → can trust uncertainty estimates for:
- Position sizing (scale by 1/uncertainty)
- Stop-loss placement (outside 95% interval)
- Risk limits (based on CVaR from predictive distribution)
""".format(coverage, coverage))
```

2. Proper Scoring Rules for Probabilistic Forecasts

2.1 Why Standard Metrics Fail

MAE and RMSE:

- Only evaluate point forecasts (mean or median)

- Ignore forecast uncertainty
- Don't penalize poorly calibrated intervals

Example problem:

- Forecast A: "Return = 0.001 ± 0.01" (uncertain)
- Forecast B: "Return = 0.001 ± 0.0001" (very confident)

If actual = 0.01, both have same MAE. But Forecast B is badly calibrated!

2.2 Continuous Ranked Probability Score (CRPS)

The gold standard for probabilistic forecasts.

$$\text{CRPS}(F, y) = \int_{-\infty}^{\infty} [F(x) - \mathbb{1}(x \geq y)]^2 dx$$

Where:

- $F(x)$ is the forecast CDF
- y is the actual outcome
- $\mathbb{1}(x \geq y)$ is the "perfect" forecast (step function at y)

Properties:

1. **Proper scoring rule**: Minimized when forecast = true distribution
2. **Generalizes MAE**: CRPS → MAE when forecast is deterministic
3. **Rewards calibration**: Penalizes both bias and miscalibrated uncertainty

Sample-based approximation: $$\text{CRPS} \approx \mathbb{E}[|X - y|] - \frac{1}{2}\mathb
X'|]$$

where X, X' are independent samples from forecast distribution.

2.3 Log Score (Ignorance Score)

$$\text{Log Score} = -\log p(y | \text{forecast})$$

- Lower is better
- Heavily penalizes forecasts that assign low probability to actual outcome
- Related to information theory ("surprisal")

In [ ]:
```python
# Calculate CRPS for walk-forward forecasts
# We'll need to re-generate posterior predictive samples

# For demonstration, simulate from forecast distributions
def calculate_crps_from_gaussian(actual, mean, std):
    """
    CRPS for Gaussian forecast (closed form).
    """
    z = (actual - mean) / std
    phi = stats.norm.pdf(z)
    Phi = stats.norm.cdf(z)
    crps_val = std * (z * (2 * Phi - 1) + 2 * phi - 1 / np.sqrt(np.pi))
```

```
    return crps_val

# Calculate CRPS for each forecast
crps_scores = []
for idx, row in wf_df.iterrows():
    crps_val = calculate_crps_from_gaussian(row['actual'], row['predictio
    crps_scores.append(crps_val)

wf_df['crps'] = crps_scores
mean_crps = np.mean(crps_scores)

# Also calculate traditional metrics for comparison
mae_score = mae(wf_df['actual'].values, wf_df['prediction'].values)
rmse_score = rmse(wf_df['actual'].values, wf_df['prediction'].values)

# Visualize CRPS over time
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# CRPS time series
ax = axes[0]
ax.plot(range(len(wf_df)), wf_df['crps'], linewidth=2, color='purple', la
ax.axhline(mean_crps, color='red', linestyle='--', linewidth=2,
           label=f'Mean CRPS = {mean_crps:.6f}')
ax.set_xlabel('Forecast Period', fontsize=11)
ax.set_ylabel('CRPS (lower = better)', fontsize=11)
ax.set_title('Continuous Ranked Probability Score Over Time',
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Metric comparison
ax = axes[1]
metrics = ['MAE', 'RMSE', 'CRPS']
values = [mae_score, rmse_score, mean_crps]
colors = ['steelblue', 'orange', 'purple']

bars = ax.bar(metrics, values, color=colors, alpha=0.7, edgecolor='black'
ax.set_ylabel('Score (lower = better)', fontsize=11)
ax.set_title('Forecast Evaluation Metrics Comparison', fontsize=12, fontw
ax.grid(True, alpha=0.3, axis='y')

# Add value labels on bars
for bar, val in zip(bars, values):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2., height,
            f'{val:.6f}',
            ha='center', va='bottom', fontsize=10, fontweight='bold')

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("PROBABILISTIC FORECAST EVALUATION")
print("="*70)
print(f"\nPoint Forecast Metrics (only evaluate mean):")
print(f"  MAE:  {mae_score:.6f}")
print(f"  RMSE: {rmse_score:.6f}")
print(f"\nProbabilistic Metric (evaluates full distribution):")
print(f"  CRPS: {mean_crps:.6f}")
```

```
print("\n" + "="*70)
print("WHY CRPS MATTERS")
print("="*70)
print("""
CRPS vs MAE:
- MAE only cares if mean forecast is close to actual
- CRPS also cares if uncertainty is correctly specified

Example:
  Actual return: 0.01

  Forecast A: Mean = 0.005, Std = 0.02 (wide, honest uncertainty)
  Forecast B: Mean = 0.005, Std = 0.001 (narrow, overconfident)

  MAE: Same for both (|0.01 - 0.005| = 0.005)
  CRPS: Lower for A (correctly captures uncertainty)

**Trading Application**:
Use CRPS to select models for:
1. Position sizing (need correct uncertainty)
2. Option pricing (need correct distribution)
3. Risk management (VaR/CVaR depend on tails)

MAE/RMSE are okay for:
1. Pure alpha generation (only care about directional accuracy)
2. Benchmarking against non-probabilistic models
""")
```

3. Kelly Criterion with Bayesian Uncertainty

3.1 Classical Kelly Criterion

Optimal position size to maximize long-term growth rate:

$$f^* = \frac{\mu - r}{\sigma^2}$$

Where:

- f^* = fraction of capital to invest
- μ = expected return
- r = risk-free rate
- σ^2 = variance of returns

Problem: Assumes we **know** μ and σ with certainty.

3.2 Bayesian Kelly

Reality: We're uncertain about μ and σ.

Solution: $$f^* = \mathbb{E}_{\theta}\left[\frac{\mu(\theta) - r}{\sigma^2(\theta)}\right]$$

where expectation is over posterior $p(\theta | \text{data})$.

Alternative (more conservative): $$f^* = \frac{\mathbb{E}[\mu] - r}{\mathbb{E}[\sigma^2] \text{Var}[\mu]}$$

The extra term $\text{Var}[\mu]$ accounts for epistemic uncertainty about expected return.

3.3 Fractional Kelly

Full Kelly ($f^* $) maximizes growth but has high volatility.

Fractional Kelly: Use fraction of Kelly $$f_{\text{actual}} = \lambda f^*, \quad \lambda \in [0.0, 0.5]$$

Why:

- Reduces volatility dramatically
- Protects against estimation errors
- "Half-Kelly" is common in practice

In [ ]:
```python
# Implement Bayesian Kelly criterion
def bayesian_kelly(trace, risk_free_rate=0.0, fraction=0.5):
    """
    Calculate Bayesian Kelly position sizing.

    Parameters:
    -----------
    trace : InferenceData
        Posterior samples from Bayesian model
    risk_free_rate : float
        Annualized risk-free rate
    fraction : float
        Fraction of Kelly to use (0.5 = half-Kelly)
    """
    # Extract posterior samples for mean and variance
    # Assuming model predicts returns with mean beta * X and variance sig

    # For demonstration, simulate from predictive distribution
    # In practice, use actual posterior predictive samples
    n_samples = 10000

    # Simulate expected returns and volatilities from posterior
    # (In real implementation, this comes from model)
    expected_returns = np.random.normal(0.0005, 0.0002, n_samples)  # Epi
    volatilities = np.random.gamma(4, 0.005, n_samples)  # Uncertainty ab

    # Calculate Kelly fraction for each posterior sample
    kelly_fractions = (expected_returns - risk_free_rate/252) / volatilit

    # Conservative: average Kelly across posterior
    mean_kelly = np.mean(kelly_fractions)

    # Apply fractional Kelly
    actual_kelly = fraction * mean_kelly

    # Clip to reasonable range
    actual_kelly = np.clip(actual_kelly, -1.0, 1.0)

    return {
        'kelly_samples': kelly_fractions,
        'mean_kelly': mean_kelly,
        'fractional_kelly': actual_kelly,
```

```
        'fraction_used': fraction,
        'expected_return': np.mean(expected_returns),
        'expected_vol': np.mean(volatilities)
    }

# Calculate Kelly for different fractions
fractions = [0.25, 0.5, 0.75, 1.0]
kelly_results = {}

for frac in fractions:
    # Use dummy trace (in practice, use actual model trace)
    result = bayesian_kelly(None, risk_free_rate=0.02, fraction=frac)
    kelly_results[frac] = result

# Visualize
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Kelly distribution from posterior
ax = axes[0]
kelly_samples = kelly_results[1.0]['kelly_samples']
ax.hist(kelly_samples, bins=50, alpha=0.7, density=True, color='steelblue
ax.axvline(kelly_results[1.0]['mean_kelly'], color='red', linestyle='--',
           linewidth=2.5, label=f"Mean Kelly = {kelly_results[1.0]['mean_
ax.axvline(0, color='black', linestyle=':', linewidth=1.5)
ax.set_xlabel('Kelly Fraction', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Bayesian Kelly Distribution\n(accounting for parameter unce
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Fractional Kelly comparison
ax = axes[1]
frac_labels = ['Quarter-Kelly', 'Half-Kelly', '3/4-Kelly', 'Full Kelly']
frac_values = [kelly_results[f]['fractional_kelly'] for f in fractions]
colors = ['green', 'blue', 'orange', 'red']

bars = ax.bar(frac_labels, frac_values, color=colors, alpha=0.7, edgecolo
ax.axhline(0, color='black', linestyle='-', linewidth=1)
ax.set_ylabel('Position Size (fraction of capital)', fontsize=11)
ax.set_title('Fractional Kelly Position Sizing', fontsize=12, fontweight=
ax.grid(True, alpha=0.3, axis='y')

# Add value labels
for bar, val in zip(bars, frac_values):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2., height,
            f'{val:.3f}',
            ha='center', va='bottom' if val > 0 else 'top',
            fontsize=10, fontweight='bold')

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("BAYESIAN KELLY CRITERION")
print("="*70)
print(f"\nExpected Return: {kelly_results[1.0]['expected_return']:.4%} (d
print(f"Expected Volatility: {kelly_results[1.0]['expected_vol']:.4%} (da
print(f"\nOptimal Position Sizes:")
```

```
for frac, label in zip(fractions, frac_labels):
    pos = kelly_results[frac]['fractional_kelly']
    print(f"  {label:15s}: {pos:>7.1%} of capital")

print("\n" + "="*70)
print("KELLY GUIDELINES FOR TRADING")
print("="*70)
print("""
1. **Full Kelly is TOO AGGRESSIVE**:
   - Maximizes long-term growth rate
   - But has 50% probability of 50%+ drawdown!
   - Not psychologically sustainable

2. **Half-Kelly is STANDARD**:
   - 75% of growth rate of full Kelly
   - Only 12.5% probability of 50% drawdown
   - Good balance of growth and volatility

3. **Quarter-Kelly is CONSERVATIVE**:
   - 50% of growth rate of full Kelly
   - Very low drawdown risk
   - Good for institutional mandates

4. **Bayesian Kelly accounts for**:
   - Parameter uncertainty (we don't know true μ, σ)
   - Model uncertainty (is AR(5) the right model?)
   - More conservative than classical Kelly

**Practical Implementation**:
- Update Kelly daily/weekly based on new forecasts
- Use fractional Kelly (0.25 - 0.5) for safety
- Set maximum position limits (e.g., no more than 20% in single asset)
- Combine with portfolio-level risk budgeting
""")
```

4. Trading Strategies with Bayesian Forecasts

Now we'll implement three complete trading strategies:

4.1 Mean Reversion with Uncertainty Bands

Idea: Trade when price deviates significantly from forecast.

Signal:

- **Buy**: Price < Lower 95% band (oversold)
- **Sell**: Price > Upper 95% band (overbought)
- **Exit**: Price returns to forecast mean

Position sizing: Scale by forecast uncertainty (wider bands → smaller position)

4.2 Trend Following with Regime Detection

Idea: Follow trends but adjust for volatility regimes.

Signal:

- **Trend**: Sign of forecast (positive → long, negative → short)
- **Strength**: Magnitude of forecast relative to uncertainty
- **Regime**: Scale position by inverse of forecasted volatility

4.3 Pairs Trading with Bayesian Cointegration

Idea: Trade mean-reverting spread between two correlated commodities.

Model: $$\text{Spread}_t = \text{Price}_A - \beta \cdot \text{Price}_B$$

Bayesian model gives distribution of β (hedge ratio).

Signal: Trade when spread exits posterior predictive interval.

In [ ]:
```python
# Strategy 1: Mean Reversion with Uncertainty Bands
class BayesianMeanReversion:
    """
    Mean reversion strategy using Bayesian forecast intervals.
    """
    def __init__(self, entry_threshold=1.96, exit_threshold=0.5,
                 max_position=0.5, volatility_scaling=True):
        self.entry_threshold = entry_threshold  # Number of std devs for
        self.exit_threshold = exit_threshold    # Number of std devs for
        self.max_position = max_position
        self.volatility_scaling = volatility_scaling

    def generate_signals(self, forecasts, prices):
        """
        Generate trading signals from forecasts.

        Parameters:
        -----------
        forecasts : DataFrame
            Must have columns: prediction, std
        prices : Series
            Actual prices
        """
        signals = pd.DataFrame(index=forecasts.index)

        # Calculate z-score (how many std devs is price from forecast)
        price_forecast = forecasts['prediction'].shift(1)  # Previous for
        price_std = forecasts['std'].shift(1)

        # Current price relative to forecast
        aligned_prices = prices.reindex(forecasts.index)
        z_score = (aligned_prices - price_forecast) / price_std

        # Generate signals
        signals['z_score'] = z_score
        signals['signal'] = 0.0

        # Buy when price below lower band (oversold)
        signals.loc[z_score < -self.entry_threshold, 'signal'] = 1.0

        # Sell when price above upper band (overbought)
```

```
        signals.loc[z_score > self.entry_threshold, 'signal'] = -1.0

        # Exit when price returns to mean
        signals.loc[abs(z_score) < self.exit_threshold, 'signal'] = 0.0

        # Position sizing: scale by inverse volatility
        if self.volatility_scaling:
            vol_scale = 1 / (1 + price_std / price_forecast.mean())
            signals['position'] = signals['signal'] * vol_scale * self.ma
        else:
            signals['position'] = signals['signal'] * self.max_position

        return signals

# Strategy 2: Trend Following with Regime Awareness
class BayesianTrendFollowing:
    """
    Trend following with Bayesian volatility adjustment.
    """
    def __init__(self, min_confidence=0.6, max_position=0.5):
        self.min_confidence = min_confidence
        self.max_position = max_position

    def generate_signals(self, forecasts):
        """
        Generate trend-following signals.
        """
        signals = pd.DataFrame(index=forecasts.index)

        # Forecast return and uncertainty
        forecast_return = forecasts['prediction']
        forecast_std = forecasts['std']

        # Confidence = |forecast| / uncertainty
        confidence = abs(forecast_return) / forecast_std
        confidence = np.minimum(confidence, 3.0)  # Cap at 3 (very confid

        # Signal strength based on forecast and confidence
        signals['raw_signal'] = np.sign(forecast_return) * confidence

        # Only trade if confidence exceeds threshold
        signals['signal'] = 0.0
        signals.loc[confidence > self.min_confidence, 'signal'] = \
            signals.loc[confidence > self.min_confidence, 'raw_signal']

        # Normalize to max position
        if signals['signal'].abs().max() > 0:
            signals['position'] = signals['signal'] / signals['signal'].a
        else:
            signals['position'] = 0.0

        return signals

# Test strategies on walk-forward forecasts
print("Testing trading strategies...\n")

# Mean Reversion
mr_strategy = BayesianMeanReversion(entry_threshold=2.0, exit_threshold=0
mr_signals = mr_strategy.generate_signals(wf_df, data['returns'])
```

```
# Trend Following
tf_strategy = BayesianTrendFollowing(min_confidence=0.8)
tf_signals = tf_strategy.generate_signals(wf_df)

print("Strategies generated successfully!")
```

In [ ]:

```
# Backtest strategies
def simple_backtest(signals, returns, initial_capital=100000, transaction
    """
    Simple backtest of strategy signals.
    """
    # Align signals and returns
    common_idx = signals.index.intersection(returns.index)
    signals_aligned = signals.reindex(common_idx)
    returns_aligned = returns.reindex(common_idx)

    # Calculate strategy returns
    positions = signals_aligned['position'].shift(1).fillna(0)  # Use pre
    strategy_returns = positions * returns_aligned

    # Transaction costs
    position_changes = positions.diff().abs()
    costs = position_changes * transaction_cost
    net_returns = strategy_returns - costs

    # Cumulative performance
    cumulative = (1 + net_returns).cumprod() * initial_capital

    return pd.DataFrame({
        'position': positions,
        'returns': net_returns,
        'cumulative': cumulative
    }, index=common_idx)

# Run backtests
mr_backtest = simple_backtest(mr_signals, data['returns'])
tf_backtest = simple_backtest(tf_signals, data['returns'])

# Buy-and-hold benchmark
common_idx = mr_backtest.index
bh_returns = data['returns'].reindex(common_idx)
bh_cumulative = (1 + bh_returns).cumprod() * 100000

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 10))

# Equity curves
ax = axes[0]
ax.plot(mr_backtest.index, mr_backtest['cumulative'], linewidth=2.5,
        color='blue', label='Mean Reversion', alpha=0.8)
ax.plot(tf_backtest.index, tf_backtest['cumulative'], linewidth=2.5,
        color='green', label='Trend Following', alpha=0.8)
ax.plot(common_idx, bh_cumulative, linewidth=2,
        color='red', linestyle='--', label='Buy & Hold', alpha=0.6)
ax.axhline(100000, color='black', linestyle=':', linewidth=1)
ax.set_ylabel('Portfolio Value ($)', fontsize=11)
ax.set_title('Strategy Performance Comparison', fontsize=12, fontweight='
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
```

```python
# Positions over time
ax = axes[1]
ax.plot(mr_backtest.index, mr_backtest['position'], linewidth=1.5,
        color='blue', alpha=0.7, label='Mean Reversion')
ax.plot(tf_backtest.index, tf_backtest['position'], linewidth=1.5,
        color='green', alpha=0.7, label='Trend Following')
ax.axhline(0, color='black', linestyle='-', linewidth=1)
ax.set_xlabel('Date', fontsize=11)
ax.set_ylabel('Position Size', fontsize=11)
ax.set_title('Position Sizes Over Time', fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Calculate performance metrics
def calculate_metrics(returns, name):
    total_return = (1 + returns).prod() - 1
    sharpe = sharpe_ratio(returns.values)
    max_dd = (returns.cumsum().cummax() - returns.cumsum()).max()
    win_rate = (returns > 0).mean()

    return {
        'Strategy': name,
        'Total Return': f"{total_return:.2%}",
        'Sharpe Ratio': f"{sharpe:.2f}",
        'Max Drawdown': f"{max_dd:.2%}",
        'Win Rate': f"{win_rate:.2%}"
    }

metrics_mr = calculate_metrics(mr_backtest['returns'], 'Mean Reversion')
metrics_tf = calculate_metrics(tf_backtest['returns'], 'Trend Following')
metrics_bh = calculate_metrics(bh_returns, 'Buy & Hold')

metrics_df = pd.DataFrame([metrics_mr, metrics_tf, metrics_bh])

print("\n" + "="*70)
print("STRATEGY PERFORMANCE SUMMARY")
print("="*70)
print(metrics_df.to_string(index=False))

print("\n" + "="*70)
print("STRATEGY INSIGHTS")
print("="*70)
print("""
Mean Reversion:
- Trades on deviations from forecast
- Works well in range-bound markets
- Scales position by forecast uncertainty (risk management)

Trend Following:
- Follows forecast direction with confidence weighting
- Only trades when forecast is confident (low uncertainty)
- Adapts to volatility regimes

**Bayesian Advantages**:
1. Uncertainty-aware position sizing
2. Automatic regime detection (via forecast distribution)
3. Natural risk management (wider intervals → smaller positions)
```

```
4. No arbitrary parameters (thresholds derived from posteriors)
""")
```

5. Complete Energy Portfolio Trading System

CAPSTONE PROJECT: Build a complete multi-asset trading system.

5.1 System Components

1. **Universe**: Crude oil, Natural gas, Heating oil (correlated energy commodities)
2. **Models**: Bayesian VAR for multi-asset forecasting
3. **Position Sizing**: Bayesian Kelly with portfolio constraints
4. **Risk Management**:
    - Portfolio VaR limit
    - Individual asset position limits
    - Volatility-based scaling
5. **Execution**: Transaction costs, slippage
6. **Monitoring**: Real-time P&L, drawdown, Sharpe

5.2 Bayesian Vector Autoregression (VAR)

Multivariate time series model:

$$\mathbf{y}_t = \mathbf{A}_ 1 \mathbf{y}_{t- 1 } + ... + \mathbf{A}_p \mathbf{y}_{t-p} + \boldsymbol{\epsilon}_t$$

where $\mathbf{y}_t$ is vector of asset returns.

Bayesian VAR:

- Priors on coefficient matrices $\mathbf{A}_i$
- Shrinkage toward simpler models (regularization)
- Full posterior for coefficients → uncertainty in cross-asset relationships

In [ ]:
```
# Generate multi-asset energy data
def generate_energy_portfolio_data(n=300):
    """
    Simulate crude oil, natural gas, heating oil with correlations.
    """
    np.random.seed(42)
    dates = pd.date_range('2023-01-01', periods=n, freq='D')

    # Correlation structure
    corr_matrix = np.array([
        [1.0, 0.6, 0.8],   # Crude
        [0.6, 1.0, 0.5],   # Natural gas
        [0.8, 0.5, 1.0]    # Heating oil
    ])

    # Cholesky decomposition for correlated normals
    L = np.linalg.cholesky(corr_matrix)

    # Generate correlated returns
```

```
    base_returns = np.random.normal(0, 1, (n, 3))
    correlated_returns = base_returns @ L.T

    # Scale by different volatilities
    crude_returns = 0.0003 + 0.02 * correlated_returns[:, 0]
    ng_returns = 0.0002 + 0.03 * correlated_returns[:, 1]  # More volatil
    ho_returns = 0.0004 + 0.022 * correlated_returns[:, 2]

    # Create DataFrame
    df = pd.DataFrame({
        'crude_ret': crude_returns,
        'ng_ret': ng_returns,
        'ho_ret': ho_returns
    }, index=dates)

    # Add prices
    df['crude_price'] = 70 * np.exp(np.cumsum(crude_returns))
    df['ng_price'] = 3.5 * np.exp(np.cumsum(ng_returns))
    df['ho_price'] = 2.5 * np.exp(np.cumsum(ho_returns))

    return df

# Generate portfolio data
portfolio_data = generate_energy_portfolio_data(n=300)

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# Prices
ax = axes[0]
ax.plot(portfolio_data.index, portfolio_data['crude_price'],
        linewidth=2, label='Crude Oil', alpha=0.8)
ax.plot(portfolio_data.index, portfolio_data['ng_price'] * 15,  # Scale f
        linewidth=2, label='Natural Gas (×15)', alpha=0.8)
ax.plot(portfolio_data.index, portfolio_data['ho_price'] * 20,  # Scale
        linewidth=2, label='Heating Oil (×20)', alpha=0.8)
ax.set_ylabel('Price (normalized)', fontsize=11)
ax.set_title('Energy Portfolio: Price Evolution', fontsize=12, fontweight
ax.legend()
ax.grid(True, alpha=0.3)

# Correlation matrix
ax = axes[1]
returns_corr = portfolio_data[['crude_ret', 'ng_ret', 'ho_ret']].corr()
im = ax.imshow(returns_corr, cmap='RdBu_r', aspect='auto', vmin=-1, vmax=
ax.set_xticks(range(3))
ax.set_yticks(range(3))
ax.set_xticklabels(['Crude', 'Nat Gas', 'Heat Oil'])
ax.set_yticklabels(['Crude', 'Nat Gas', 'Heat Oil'])
ax.set_title('Return Correlations', fontsize=12, fontweight='bold')

# Add correlation values
for i in range(3):
    for j in range(3):
        text = ax.text(j, i, f'{returns_corr.iloc[i, j]:.2f}',
                       ha="center", va="center", color="black", fontsize=1

plt.colorbar(im, ax=ax)
plt.tight_layout()
plt.show()
```

```
print("\nEnergy Portfolio Data Generated")
print(f"Assets: Crude Oil, Natural Gas, Heating Oil")
print(f"Period: {len(portfolio_data)} days")
print(f"\nReturn Correlations:")
print(returns_corr.round(3))
```

6. Summary: Building Production-Ready Bayesian Tra Systems

6.1 Essential Components

Component	Bayesian Implementation
Forecasting	GP, VAR, structural time series with full posteriors
Backtesting	Walk-forward validation (no look-ahead bias)
Evaluation	CRPS, calibration, log-score (not just MAE/RMSE)
Position Sizing	Bayesian Kelly with parameter uncertainty
Risk Management	VaR/CVaR from posterior predictive, volatility scaling
Portfolio	Multi-asset Bayesian models (VAR, copulas)

6.2 What We've Learned

Module 1-3: Bayesian foundations, priors, MCMC **Module** 4-6: Time series models (A space, BSTS) **Module** 7: Hierarchical models for multiple commodities **Module** 8: Gauss Processes for non-linear patterns **Module** 9: Volatility modeling (GARCH, stochastic vol) **Mo** 10: Complete trading systems (this module)

6.3 Bayesian vs Frequentist: Final Comparison

Bayesian Advantages:

1. ✅ Full uncertainty quantification
2. ✅ Parameter uncertainty propagates to decisions
3. ✅ Natural regularization through priors
4. ✅ Sequential updating (efficient online learning)
5. ✅ Honest risk estimates (wider intervals with limited data)

Frequentist Advantages:

1. ✅ Faster computation (no MCMC)
2. ✅ Easier to implement in production
3. ✅ Well-understood asymptotics

Recommendation:

- **Research/Alpha generation**: Bayesian (better forecasts)
- **High-frequency execution**: Frequentist (speed matters)
- **Risk management**: Bayesian (need full distributions)

6.4 Going to Production

Infrastructure needs:

1. **Data pipeline**: Real-time price feeds, cleaning, storage
2. **Model serving**: Daily refits, caching posteriors
3. **Execution**: Order management, broker integration
4. **Monitoring**: P&L tracking, drawdown alerts, model diagnostics
5. **Backtesting**: Automated walk-forward validation

Common pitfalls:

- ❌ Overfitting in-sample
- ❌ Ignoring transaction costs
- ❌ Not accounting for parameter uncertainty
- ❌ Using wrong evaluation metrics (MAE for probabilistic forecasts)
- ❌ Over-leveraging (full Kelly is too aggressive)

6.5 Next Steps

To deepen your knowledge:

1. Read: "Bayesian Methods for Hackers" (Cameron Davidson-Pilon)
2. Read: "Advances in Financial Machine Learning" (Marcos López de Prado)
3. Practice: Kaggle competitions with probabilistic scoring
4. Build: Your own production system with real data
5. Learn: Advanced topics (deep learning + Bayesian, causal inference)

Congratulations! You've completed the Bayesian Forecasting for Commodities Trading course.

Final Knowledge Check Quiz

Q 1: Walk-forward validation prevents:

- A) Model convergence issues
- B) Look-ahead bias in backtesting
- C) Transaction costs
- D) Market regime changes

Q 2: CRPS is better than MAE for Bayesian forecasts because:

- A) It's easier to calculate
- B) It evaluates the full forecast distribution, not just the mean
- C) It always gives higher scores
- D) Regulators require it

Q 3 : Half-Kelly position sizing means:

- A) Use 50% of your capital
- B) Use half the optimal Kelly fraction
- C) Trade half as often
- D) Split position between two assets

Q 4 : Bayesian Kelly criterion accounts for:

- A) Only expected return uncertainty
- B) Only volatility uncertainty
- C) Both parameter and model uncertainty
- D) Neither (uses point estimates)

Q 5 : The main advantage of Bayesian trading systems is:

- A) They always outperform
- B) They're faster to execute
- C) They quantify uncertainty for better risk management
- D) They require less data

In [ ]:
```
# Quiz Answers
print("="*70)
print("FINAL QUIZ ANSWERS")
print("="*70)
print("""
Q1: B) Look-ahead bias in backtesting
    Walk-forward validation simulates real-time trading. Each forecast
    is made using ONLY data available at that time (no future data).
    This prevents overfitting and gives realistic performance estimates.

Q2: B) It evaluates the full forecast distribution, not just the mean
    CRPS is a proper scoring rule that rewards:
    1. Accurate point forecasts (like MAE)
    2. Well-calibrated uncertainty (bonus over MAE)
    Essential for models used in risk management and position sizing.

Q3: B) Use half the optimal Kelly fraction
    Full Kelly maximizes growth but has high volatility.
    Half-Kelly (f* × 0.5) gives:
    - 75% of growth rate
    - Much lower drawdowns
    - More psychologically sustainable

Q4: C) Both parameter and model uncertainty
    Classical Kelly: assumes we KNOW μ and σ
    Bayesian Kelly: integrates over posterior p(μ, σ | data)
    Result: More conservative sizing when data is limited

Q5: C) They quantify uncertainty for better risk management
    Bayesian methods don't always outperform in terms of raw returns.
    But they provide:
    - Full predictive distributions
    - Honest uncertainty estimates
    - Better risk-adjusted performance
```

```
    - Principled position sizing
""")
```

Final Project: Build Your Own Trading System

Project Requirements

Build a complete Bayesian trading system for a commodity of your choice:

1. **Data**: Collect real price data (at least 2 years, daily)
2. **Model**: Choose appropriate Bayesian model:
    - Smooth trends → Gaussian Process
    - Seasonal patterns → Structural time series
    - Regime changes → Switching models
3. **Backtesting**: Implement walk-forward validation
    - Minimum 50 out-of-sample forecasts
    - Calculate CRPS and calibration
4. **Strategy**: Design trading strategy
    - Mean reversion, trend following, or pairs trading
    - Bayesian Kelly position sizing
    - Transaction costs and slippage
5. **Risk Management**:
    - Daily VaR monitoring
    - Maximum drawdown limits
    - Volatility-based position scaling
6. **Evaluation**:
    - Sharpe ratio > 1.0 (after costs)
    - Max drawdown < 20%
    - Win rate > 50%
7. **Documentation**:
    - Model choice justification
    - Prior selection rationale
    - Backtesting methodology
    - Performance analysis

Deliverables

1. Jupyter notebook with complete implementation
2. Performance report (PDF)
3. Production-ready code (modular, documented)

Evaluation Criteria

- **Technical (40%)**: Correct Bayesian implementation, no look-ahead bias
- **Performance (30%)**: Risk-adjusted returns, calibration
- **Rigor (20%)**: Proper evaluation metrics, honest reporting
- **Presentation (10%)**: Code quality, documentation

Course Complete!

You've learned:

- Bayesian inference fundamentals
- Time series modeling with uncertainty
- Volatility and risk management
- Complete trading system development

You can now:

- Build production Bayesian forecasting systems
- Properly backtest without overfitting
- Size positions optimally with Kelly criterion
- Manage risk with probabilistic forecasts

Go forth and trade wisely! 📈

End of Course