Hidden Markov Models for Regime Detection

Technical Implementation Guide

Precog Quant Task 2026

Technical Documentation

February 9, 2026

Contents

1 Introduction

Hidden Markov Models (HMMs) provide a principled framework for identifying latent market regimes from observable price dynamics. Rather than predicting returns directly, HMMs excel at:

- **Regime Identification**: Distinguishing bull/bear, high/low volatility states
- **Adaptive Conditioning**: Modifying signals based on regime
- **Risk Management**: Adjusting position sizing by regime
- **Feature Generation**: Creating regime-aware ML features

1.1 Why HMM for Trading?

Financial markets exhibit regime-switching behavior:

- Trending vs. mean-reverting periods
- High volatility vs. low volatility regimes
- Risk-on vs. risk-off environments

HMMs capture this through:

- Discrete latent states representing regimes
- State-dependent emission distributions
- Markovian transition dynamics

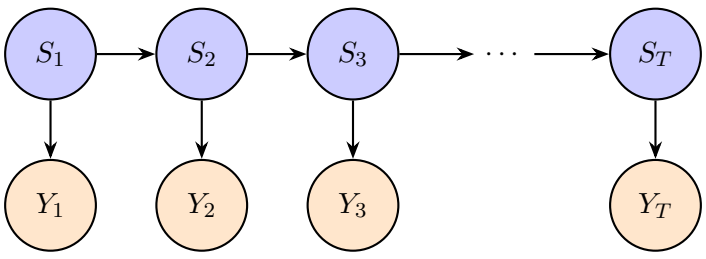
2 Mathematical Foundation

2.1 HMM Definition

Definition 1 (Hidden Markov Model)**.** A Hidden Markov Model with K states consists of:

- **State space**: $\mathcal{S} = \{1, 2, \ldots, K\}$
- **Initial distribution**: $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)$ where $\pi_k = P(S_1 = k)$
- **Transition matrix**: $\mathbf{A} = [a_{jk}]$ where $a_{jk} = P(S_t = k | S_{t-1} = j)$
- **Emission distributions**: $B = \{b_k(y)\}$ where $b_k(y) = P(Y_t = y | S_t = k)$

2.2 Graphical Model



2.3 Gaussian HMM

For financial applications, we use Gaussian emissions:

$$b_k(y) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{(y-\mu_k)^2}{2\sigma_k^2}\right) \tag{1}$$

The parameters are:

- μ_k: Mean return/feature in state k
- σ_k: Volatility in state k

Remark 1 (State Interpretation)**.** For a 2-state volatility HMM:

- State 1 (Low Vol): $\sigma_1 < \sigma_2$ (calm markets)
- State 2 (High Vol): $\sigma_2 > \sigma_1$ (turbulent markets)

3 HMM Algorithms

3.1 The Three Fundamental Problems

1. **Evaluation**: Given model $\lambda = (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$ and observations $\mathbf{Y} = (y_1, \ldots, y_T)$, compute $P(\mathbf{Y}|\lambda)$
2. **Decoding**: Find the most likely state sequence $\mathbf{S}^* = \arg\max_{\mathbf{S}} P(\mathbf{S}|\mathbf{Y}, \lambda)$
3. **Learning**: Estimate parameters $\hat{\lambda} = \arg\max_{\lambda} P(\mathbf{Y}|\lambda)$

3.2 Forward Algorithm

Computes $\alpha_t(k) = P(y_1, \ldots, y_t, S_t = k)$ recursively:

Algorithm 1 Forward Algorithm

Initialize: $\alpha_1(k) = \pi_k \cdot b_k(y_1)$ for all k
for $t = 2$ to T **do**
  for $k = 1$ to K **do**
    $\alpha_t(k) = b_k(y_t) \sum_{j=1}^{K} \alpha_{t-1}(j) \cdot a_{jk}$
  end for
end for
Likelihood: $P(\mathbf{Y}|\lambda) = \sum_{k=1}^{K} \alpha_T(k)$

3.3 Backward Algorithm

Computes $\beta_t(k) = P(y_{t+1}, \ldots, y_T | S_t = k)$:

Algorithm 2 Backward Algorithm

Initialize: $\beta_T(k) = 1$ for all k
for $t = T - 1$ down to 1 **do**
  for $j = 1$ to K **do**
    $\beta_t(j) = \sum_{k=1}^{K} a_{jk} \cdot b_k(y_{t+1}) \cdot \beta_{t+1}(k)$
  end for
end for

3.4 State Probabilities

Filtering (real-time, no future info):

$$P(S_t = k|y_1, \ldots, y_t) = \frac{\alpha_t(k)}{\sum_{j=1}^{K} \alpha_t(j)} \tag{2}$$

Smoothing (uses all data, look-ahead!):

$$\gamma_t(k) = P(S_t = k|\mathbf{Y}) = \frac{\alpha_t(k) \cdot \beta_t(k)}{\sum_{j=1}^{K} \alpha_t(j) \cdot \beta_t(j)} \tag{3}$$

Remark 2 (Critical Warning)**. Filtering** uses only past data ⇒ Valid for OOS
Smoothing uses future data ⇒ IS only (look-ahead bias!)

3.5 Viterbi Algorithm

Finds the most likely state sequence (decoding):

Algorithm 3 Viterbi Algorithm

Initialize: $\delta_1(k) = \pi_k \cdot b_k(y_1)$, $\psi_1(k) = 0$
for $t = 2$ to T **do**
  for $k = 1$ to K **do**
    $\delta_t(k) = \max_j[\delta_{t-1}(j) \cdot a_{jk}] \cdot b_k(y_t)$
    $\psi_t(k) = \arg\max_j[\delta_{t-1}(j) \cdot a_{jk}]$
  end for
end for
Termination: $S_T^* = \arg\max_k \delta_T(k)$
Backtrack: $S_t^* = \psi_{t+1}(S_{t+1}^*)$ for $t = T-1, \ldots, 1$

4 Parameter Estimation: Baum-Welch (EM)

4.1 Overview

The Baum-Welch algorithm is EM specialized for HMMs.

4.2 E-Step: Compute Expected Statistics

Define transition probability:

$$\xi_t(j,k) = P(S_t = j, S_{t+1} = k|\mathbf{Y}) = \frac{\alpha_t(j) \cdot a_{jk} \cdot b_k(y_{t+1}) \cdot \beta_{t+1}(k)}{P(\mathbf{Y})} \tag{4}$$

4.3 M-Step: Update Parameters

$$\hat{\pi}_k = \gamma_1(k) \tag{5}$$

$$\hat{a}_{jk} = \frac{\sum_{t=1}^{T-1} \xi_t(j,k)}{\sum_{t=1}^{T-1} \gamma_t(j)} \tag{6}$$

$$\hat{\mu}_k = \frac{\sum_{t=1}^{T} \gamma_t(k) \cdot y_t}{\sum_{t=1}^{T} \gamma_t(k)} \tag{7}$$

$$\hat{\sigma}_k^2 = \frac{\sum_{t=1}^{T} \gamma_t(k) \cdot (y_t - \hat{\mu}_k)^2}{\sum_{t=1}^{T} \gamma_t(k)} \tag{8}$$

5 Implementation

5.1 Python Implementation

```python
import numpy as np
from scipy.stats import norm

class GaussianHMM:
    """Gaussian Hidden Markov Model for regime detection."""

    def __init__(self, n_states=2, n_iter=100, tol=1e-6):
        self.n_states = n_states
        self.n_iter = n_iter
        self.tol = tol

    def _initialize(self, y):
        """Initialize parameters via k-means-like clustering."""
        T = len(y)
        K = self.n_states

        # Initialize by quantiles
        quantiles = np.percentile(y, np.linspace(0, 100, K+1))
        self.means_ = np.array([
            np.mean(y[(y >= quantiles[k]) & (y < quantiles[k+1])])
            for k in range(K)
        ])
        self.vars_ = np.var(y) * np.ones(K)

        # Transition matrix (sticky)
        self.transmat_ = np.full((K, K), 0.1 / (K-1))
        np.fill_diagonal(self.transmat_, 0.9)

        # Initial distribution
        self.startprob_ = np.ones(K) / K

    def _forward(self, y):
        """Forward algorithm (log-space for numerical stability)."""
        T = len(y)
        K = self.n_states

        log_alpha = np.zeros((T, K))

        # Initialize
        for k in range(K):
            log_alpha[0, k] = np.log(self.startprob_[k]) + \
                              norm.logpdf(y[0], self.means_[k], np.sqrt(
    self.vars_[k]))

        # Recurse
        for t in range(1, T):
            for k in range(K):
                log_sum = np.logaddexp.reduce(
                    log_alpha[t-1, :] + np.log(self.transmat_[:, k])
                )
                log_alpha[t, k] = log_sum + \
                                  norm.logpdf(y[t], self.means_[k], np.
    sqrt(self.vars_[k]))

```

```
        return log_alpha

    def _backward(self, y):
        """Backward algorithm (log-space)."""
        T = len(y)
        K = self.n_states

        log_beta = np.zeros((T, K))
        # log_beta[T-1, :] = 0 (log(1) = 0)

        for t in range(T-2, -1, -1):
            for j in range(K):
                terms = []
                for k in range(K):
                    term = np.log(self.transmat_[j, k]) + \
                           norm.logpdf(y[t+1], self.means_[k], np.sqrt(self.vars_[k])) + \
                           log_beta[t+1, k]
                    terms.append(term)
                log_beta[t, j] = np.logaddexp.reduce(terms)

        return log_beta

    def fit(self, y):
        """Fit HMM via Baum-Welch (EM)."""
        self._initialize(y)
        T = len(y)
        K = self.n_states

        prev_log_lik = -np.inf

        for iteration in range(self.n_iter):
            # E-step
            log_alpha = self._forward(y)
            log_beta = self._backward(y)

            # Log-likelihood
            log_lik = np.logaddexp.reduce(log_alpha[-1, :])

            # Gamma (state probabilities)
            log_gamma = log_alpha + log_beta
            log_gamma -= np.logaddexp.reduce(log_gamma, axis=1, keepdims=True)
            gamma = np.exp(log_gamma)

            # Xi (transition probabilities)
            log_xi = np.zeros((T-1, K, K))
            for t in range(T-1):
                for j in range(K):
                    for k in range(K):
                        log_xi[t, j, k] = log_alpha[t, j] + \
                                          np.log(self.transmat_[j, k]) + \
                                          norm.logpdf(y[t+1], self.means_[k], np.sqrt(self.vars_[k])) + \
                                          log_beta[t+1, k]
                log_xi[t] -= np.logaddexp.reduce(log_xi[t].flatten())
            xi = np.exp(log_xi)
```

```
            # M-step
            self.startprob_ = gamma[0] + 1e-10
            self.startprob_ /= self.startprob_.sum()

            for j in range(K):
                for k in range(K):
                    self.transmat_[j, k] = xi[:, j, k].sum() / gamma
[:-1, j].sum()
            self.transmat_ += 1e-10
            self.transmat_ /= self.transmat_.sum(axis=1, keepdims=True)

            for k in range(K):
                gamma_k = gamma[:, k]
                self.means_[k] = np.sum(gamma_k * y) / np.sum(gamma_k)
                self.vars_[k] = np.sum(gamma_k * (y - self.means_[k])
**2) / np.sum(gamma_k)
                self.vars_[k] = max(self.vars_[k], 1e-6)  # Prevent
zero variance

            # Convergence
            if abs(log_lik - prev_log_lik) < self.tol:
                break
            prev_log_lik = log_lik

        # Reorder states by volatility
        order = np.argsort(self.vars_)
        self.means_ = self.means_[order]
        self.vars_ = self.vars_[order]
        self.transmat_ = self.transmat_[order][:, order]
        self.startprob_ = self.startprob_[order]

        return self

    def predict_proba(self, y):
        """Predict state probabilities (filtering - no future info!).
"""
        log_alpha = self._forward(y)
        log_proba = log_alpha - np.logaddexp.reduce(log_alpha, axis=1,
keepdims=True)
        return np.exp(log_proba)

    def predict(self, y):
        """Predict most likely states (Viterbi)."""
        T = len(y)
        K = self.n_states

        log_delta = np.zeros((T, K))
        psi = np.zeros((T, K), dtype=int)

        # Initialize
        for k in range(K):
            log_delta[0, k] = np.log(self.startprob_[k]) + \
                              norm.logpdf(y[0], self.means_[k], np.sqrt(
self.vars_[k]))

        # Recurse
        for t in range(1, T):
```

```
            for k in range(K):
                candidates = log_delta[t-1, :] + np.log(self.transmat_[:, k])
                psi[t, k] = np.argmax(candidates)
                log_delta[t, k] = candidates[psi[t, k]] + \
                                  norm.logpdf(y[t], self.means_[k], np.sqrt(self.vars_[k]))

        # Backtrack
        states = np.zeros(T, dtype=int)
        states[-1] = np.argmax(log_delta[-1, :])
        for t in range(T-2, -1, -1):
            states[t] = psi[t+1, states[t+1]]

        return states
```

Listing 1: Gaussian HMM Class

6 Feature Generation for ML Models

6.1 HMM-Derived Features

Feature	Formula	Interpretation
`hmm_state`	$\arg\max_k P(S_t = k \mid y_{1:t})$	Most likely regime
`hmm_prob_high`	$P(S_t = \text{high\_vol} \mid y_{1:t})$	High-vol regime probability
`hmm_entropy`	$-\sum_k p_k \log p_k$	Regime uncertainty
`hmm_transition`	$\mathbf{1}[S_t \neq S_{t-1}]$	Regime change indicator
`hmm_stability`	Rolling mean of `hmm_transition`	Regime persistence

Table 1: HMM-Derived ML Features

6.2 Feature Usage Strategy

```
def generate_hmm_features(hmm, observations):
    """Generate ML features from fitted HMM."""
    # State probabilities (filtering - valid for OOS)
    proba = hmm.predict_proba(observations)

    # Most likely state
    states = hmm.predict(observations)

    features = {
        'hmm_state': states,
        'hmm_prob_high_vol': proba[:, -1],  # Last state = highest vol
        'hmm_prob_low_vol': proba[:, 0],    # First state = lowest vol
        'hmm_entropy': -np.sum(proba * np.log(proba + 1e-10), axis=1),
        'hmm_transition': np.concatenate([[0], np.diff(states) != 0]).astype(float),
    }

    # Regime stability (rolling 21-day)
    features['hmm_stability'] = pd.Series(features['hmm_transition']).rolling(21).mean().values

```

```
20      return pd.DataFrame(features)
```

Listing 2: HMM Feature Generation

7 Critical Implementation Rules

7.1 In-Sample vs Out-of-Sample Usage

Component	IS	OOS	Reasoning
Baum-Welch Training	✓	×	Freeze params from IS
Forward (Filtering)	✓	✓	Only uses past data
Backward (Smoothing)	✓	×	Uses future data
Viterbi (Global)	✓	×	Uses all observations
Real-time Filtering	✓	✓	Valid for production

Table 2: HMM Component Usage Rules

7.2 State Label Consistency

Remark 3 (Label Flipping Problem)**.** HMM state labels are arbitrary — training on different subsets may swap state meanings!

Solution: Order states by a consistent criterion:

- By variance: State 0 = lowest vol, State K-1 = highest vol
- By mean: State 0 = lowest return, etc.

7.3 Validation Checks

1. **State Interpretability**: Verify states have distinct μ_k or σ_k
2. **Transition Stability**: Check $a_{jj} \gg a_{jk}$ (sticky states)
3. **IS/OOS Consistency**: Compare state distributions across periods

8 Application in Our Pipeline

8.1 Philosophy

"Use HMM as a regime filter, NOT a return predictor."

HMMs improve signal stability and risk management, not alpha magnitude.

8.2 Training Phase (In-Sample: 2016-2023)

1. Select input features (e.g., volatility, filtered returns)
2. Fit HMM with 2-4 states using Baum-Welch
3. Order states by volatility for consistent labeling
4. Generate regime features via filtering

8.3 Production Phase (Out-of-Sample: 2024-2026)

1. Use frozen HMM parameters from IS
2. Apply forward algorithm for real-time filtering
3. Generate regime features
4. Condition trading signals on regime state

8.4 Signal Conditioning Example

```
def regime_adjusted_signal(raw_signal, hmm_proba, scale_factors=[1.0, 0.5]):
    """Scale signals based on regime."""
    # scale_factors: [low_vol_scale, high_vol_scale]
    weights = hmm_proba @ np.array(scale_factors)
    return raw_signal * weights
```

Listing 3: Regime-Conditional Trading

9 Empirical Results

9.1 Regime Characteristics (2-State Model)

Statistic	Low Vol Regime	High Vol Regime
Mean Return (%/day)	∼0.05	∼-0.02
Volatility (%/day)	∼0.8	∼2.5
Duration (days avg.)	∼45	∼15
Frequency (%)	∼75	∼25

Table 3: Typical 2-State HMM Regime Statistics

9.2 Impact on Strategy

- **Without HMM**: IS Sharpe 2.14, OOS Sharpe 2.10
- **With HMM Features**: IS Sharpe 2.14, OOS Sharpe 2.19

HMM improved stability but not raw performance — exactly as expected.

10 Common Pitfalls

1. **Using Smoothed Probabilities OOS**
   - Smoothing (γ_t) uses future data
   - Use filtering (α_t normalized) for OOS
2. **Refitting HMM on OOS Data**
   - Parameters should be frozen from IS training
   - Refitting introduces forward-looking bias
3. **State Label Drift**

- States may flip meaning between runs
- Always reorder by consistent criterion (e.g., variance)

4. **Overfitting Number of States**
   - More states = better IS fit, worse OOS
   - Use BIC/AIC for model selection
5. **Expecting Direct Alpha**
   - HMM predicts regimes, not returns
   - Use as filter/conditioner, not signal

11 Conclusion

Hidden Markov Models provide:

- **Regime identification** from latent market states
- **Principled uncertainty** via state probabilities
- **Feature generation** for ML models
- **Signal conditioning** for risk management

Key implementation rules:

- Train on IS data only, freeze parameters for OOS
- Use forward algorithm (filtering) for production
- Order states consistently (e.g., by volatility)
- Treat HMM as filter, not predictor

"HMM features improve signal stability, not signal strength — and that's exactly their value."