

# Market Regime Detection Using Hidden Markov Models

Expanded Tutorial

*Deep Treatment of HMM Components, Viterbi Decoding,  
and the EM Algorithm — with Worked Numerical Examples*

With References to Implementation Code

## Contents

<b>1</b>	<b>Introduction and the Big Picture</b>	<b>2</b>
<b>2</b>	<b>HMM Settings: Every Component Explained</b>	<b>3</b>
2.1	Hidden States $\mathcal{S}$	3
2.2	Transition Matrix $\mathbf{A}$	3
2.3	Emission Distributions $\mathbf{B}$ (Gaussian)	4
2.4	Initial Distribution $\pi$	5
2.5	The Joint Probability Factorisation	5
<b>3</b>	<b>Why We Need Two Algorithms: EM vs Viterbi</b>	<b>6</b>
3.1	An Analogy: Learning a Language	6
3.2	The Pipeline in Our Code	6
<b>4</b>	<b>The EM Algorithm (Baum-Welch): Full Derivation</b>	<b>8</b>
4.1	The Problem EM Solves	8
4.2	Step 0: The Forward-Backward Algorithm	8
4.2.1	Forward Variable $\alpha_t(i)$	8
4.2.2	Backward Variable $\beta_t(i)$	9
4.3	Step 1 of EM: E-Step (Expectation)	9
4.3.1	State Posterior $\gamma_t(i)$	9
4.3.2	Transition Posterior $\xi_t(i, j)$	10
4.4	Step 2 of EM: M-Step (Maximisation)	10
4.5	Convergence and Multiple Initialisations	10
4.6	Regularisation: Preventing Singular Covariances	11
<b>5</b>	<b>The Viterbi Algorithm: Full Derivation</b>	<b>12</b>
5.1	The Problem Viterbi Solves	12
5.2	Key Idea: Principle of Optimality	12
5.3	The Algorithm	12
5.4	Worked Numerical Example	13
5.5	Viterbi in the Code	14
<b>6</b>	<b>Viterbi vs. Forward-Backward: A Subtle Distinction</b>	<b>15</b>
6.1	Why They Can Disagree	15
6.2	Why the Code Uses Viterbi	15
<b>7</b>	<b>How EM Uses Forward-Backward Internally</b>	<b>16</b>
<b>8</b>	<b>Summary: The Complete Pipeline</b>	<b>17</b>
8.1	Key Formulas at a Glance	17
8.2	Final Intuition	17

## 1 Introduction and the Big Picture

Before diving into any mathematics, let us establish the **big picture** of why this project needs *two* seemingly related algorithms.

### The Two Fundamental Questions of an HMM

**Q1: Learning (EM/Baum-Welch):** Given observed market returns, *what are the model parameters?*

⇒ “What does a Bull market *look like* in terms of returns and volatility?”

**Q2: Decoding (Viterbi):** Given *learned* parameters and today’s returns, *which regime are we in?*

⇒ “Is today a Bull day, a Bear day, or a Sideways day?”

**EM comes first** (training). **Viterbi comes second** (deployment). They solve *completely different problems*.

You can see this two-phase structure directly in the code. In `v3_0_train_and_backtest.py`, the function `train_hmm_for_window()` (lines 167-195) first **trains** and then **decodes**:

```

1 # PHASE 1: EM --- learn the parameters
2 model.fit(sector_returns, n_init=3, verbose=False)           # line 190
3
4 # PHASE 2: Viterbi --- decode the regime sequence
5 regimes = model.get_regime_sequence(sector_returns)          # line 191

```

And during the live backtest (lines 254-255), only Viterbi is used on new test data:

```

1 # Only Viterbi on unseen data (model already trained!)
2 test_regimes_raw = model.get_regime_sequence(test_returns)    # line 254
3 test_regimes = test_regimes_raw.map(model.regime_labels)       # line 255

```

## 2 HMM Settings: Every Component Explained

An HMM is defined by the tuple  $\lambda = (\mathcal{S}, \mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$ . Let us walk through each component, connecting the mathematics to the financial intuition and the code.

### 2.1 Hidden States $\mathcal{S}$

**Definition 2.1** (Hidden States). *The set  $\mathcal{S} = \{s_1, s_2, \dots, s_K\}$  contains  $K$  discrete, unobservable states. At each time step  $t$ , the system is in exactly one state  $q_t \in \mathcal{S}$ .*

In our project,  $K = 3$  and the states represent:

$$\mathcal{S} = \{ \underbrace{s_1}_{\text{Bull}}, \underbrace{s_2}_{\text{Sideways}}, \underbrace{s_3}_{\text{Bear}} \}$$

This is set in `config.py`, line 41:

```
1 n_regimes: int = 3
```

#### Why “Hidden”?

The states are hidden because we never directly observe whether today is a “Bull day” or a “Bear day.” We only observe the *returns* of 28 Chinese market sectors. The regime is a latent variable that we must *infer*.

### 2.2 Transition Matrix $\mathbf{A}$

**Definition 2.2** (Transition Matrix). *The  $K \times K$  matrix  $\mathbf{A}$  where:*

$$A_{ij} = P(q_t = s_j \mid q_{t-1} = s_i), \quad A_{ij} \geq 0, \quad \sum_{j=1}^K A_{ij} = 1 \quad \forall i \quad (1)$$

Each row sums to 1 ( $\mathbf{A}$  is row-stochastic).

This encodes the **First-Order Markov Property** (matching Section 2.1 of your lecture image):

$$P(q_t \mid q_{t-1}, q_{t-2}, \dots, q_1) = P(q_t \mid q_{t-1}) = A_{q_{t-1}, q_t}$$

#### Numerical Example: Reading a Transition Matrix

Suppose after training, we get:

$$\mathbf{A} = \begin{pmatrix} 0.95 & 0.03 & 0.02 \\ 0.10 & 0.85 & 0.05 \\ 0.02 & 0.08 & 0.90 \end{pmatrix} \quad \begin{array}{l} \leftarrow \text{Bull row} \\ \leftarrow \text{Sideways row} \\ \leftarrow \text{Bear row} \end{array}$$

Reading row 1 (Bull): if we’re in a Bull market today, there is a:

- 95% chance of *staying* Bull tomorrow
- 3% chance of transitioning to Sideways
- 2% chance of transitioning to Bear

The **diagonal entries**  $A_{ii}$  measure **persistence**. High values mean regimes are “sticky”—a Bull market tends to persist for many days. The expected duration is:

$$\mathbb{E}[\text{Bull duration}] = \frac{1}{1 - 0.95} = 20 \text{ trading days}$$

This is implemented in `models.py` (lines 358-370):

```

1 def expected_duration(self) -> np.ndarray:
2     # Expected duration = 1 / (1 - p_ii)
3     diag = np.diag(self.transition_matrix)
4     return 1 / (1 - diag + 1e-10)

```

## 2.3 Emission Distributions B (Gaussian)

**Definition 2.3** (Gaussian Emission). *Given that the hidden state at time  $t$  is  $q_t = k$ , the observation  $\mathbf{r}_t \in \mathbb{R}^D$  is drawn from:*

$$P(\mathbf{r}_t | q_t = k) = \mathcal{N}(\mathbf{r}_t; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{r}_t - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1} (\mathbf{r}_t - \boldsymbol{\mu}_k)\right) \quad (2)$$

This encodes the **Output Independence** assumption (Section 2.2 of your lecture image):

$$P(\mathbf{r}_t | q_1, \dots, q_T, \mathbf{r}_1, \dots, \mathbf{r}_T) = P(\mathbf{r}_t | q_t)$$

The observation at time  $t$  depends *only* on the current hidden state—not on any other observations or past states.

### Numerical Example: Emission Parameters

For a simple 2-sector model (Tech, Finance), the learned emissions might be:

**Bull regime ( $k = 0$ ):**

$$\boldsymbol{\mu}_0 = \begin{pmatrix} +0.12\% \\ +0.08\% \end{pmatrix}, \quad \boldsymbol{\Sigma}_0 = \begin{pmatrix} 0.0004 & 0.0002 \\ 0.0002 & 0.0003 \end{pmatrix}$$

**Bear regime ( $k = 2$ ):**

$$\boldsymbol{\mu}_2 = \begin{pmatrix} -0.18\% \\ -0.06\% \end{pmatrix}, \quad \boldsymbol{\Sigma}_2 = \begin{pmatrix} 0.0012 & 0.0006 \\ 0.0006 & 0.0008 \end{pmatrix}$$

Notice two things:

1. The Bear regime has **negative means** and **larger variances** (more volatility)
2. The off-diagonal entries of  $\boldsymbol{\Sigma}$  capture *sector correlations*—sectors are more correlated during crashes

The covariance structure is configured in `config.py` (line 42). The “enhanced” config uses `covariance_type='full'`, meaning each regime has its own unrestricted covariance matrix:

Type	Meaning	# Params per Regime
full	Each regime has its own full $D \times D$ covariance	$D(D + 1)/2$
diag	Only variances, no correlations: $\text{diag}(\sigma_1^2, \dots, \sigma_D^2)$	$D$
spherical	Single variance for all features: $\sigma^2 \mathbf{I}$	1
tied	All regimes share the same $\boldsymbol{\Sigma}$	$D(D + 1)/2$ (shared)

## 2.4 Initial Distribution $\pi$

$$\pi_i = P(q_1 = s_i), \quad \sum_{i=1}^K \pi_i = 1 \quad (3)$$

This specifies the probability of starting in each regime. It is learned as part of EM and stored internally by `hmmlearn`.

## 2.5 The Joint Probability Factorisation

These two independence assumptions let us factorise the full joint probability. This factorisation is what makes HMMs computationally tractable:

### Joint Probability of States and Observations

$$P(\mathbf{Q}, \mathbf{O} | \lambda) = \underbrace{\pi_{q_1}}_{\text{start}} \cdot \underbrace{b_{q_1}(\mathbf{r}_1)}_{\text{emit}} \cdot \prod_{t=2}^T \underbrace{A_{q_{t-1}, q_t}}_{\text{transition}} \cdot \underbrace{b_{q_t}(\mathbf{r}_t)}_{\text{emit}} \quad (4)$$

where  $b_k(\mathbf{r}_t) = \mathcal{N}(\mathbf{r}_t; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ ,  $\mathbf{Q} = (q_1, \dots, q_T)$  is the state sequence, and  $\mathbf{O} = (\mathbf{r}_1, \dots, \mathbf{r}_T)$  is the observation sequence.

This single equation is the foundation of *everything* that follows. Both the EM algorithm and Viterbi algorithm manipulate this expression—but for **different purposes**, as we'll now see.

### 3 Why We Need Two Algorithms: EM vs Viterbi

This is the central conceptual question. Let us be completely explicit.

EM and Viterbi Solve Different Problems		
	EM (Baum-Welch)	Viterbi
<b>Question</b>	What are the model parameters $\lambda = (\mathbf{A}, \boldsymbol{\mu}, \Sigma)$ ?	What is the most likely state sequence $\mathbf{Q}^*$ ?
<b>Input</b>	Observations $\mathbf{O}$ only	Observations $\mathbf{O}$ <i>and</i> parameters $\lambda$
<b>Output</b>	Learned $\hat{\lambda}$	Optimal path $\mathbf{Q}^*$
<b>When used</b>	Training (offline, once per window)	Inference (online, every day)
<b>Analogy</b>	“Studying for the exam”	“Taking the exam”
<b>Code</b>	<code>model.fit(X)</code> (line 174)	<code>model.predict(X)</code> (line 224)

#### 3.1 An Analogy: Learning a Language

Suppose you’re trying to understand Morse code (dots and dashes → letters).

- **EM** is like studying a textbook to learn the encoding rules: which patterns of dots/dashes correspond to which letters, how often certain letter pairs occur, etc. After studying, you “know” the language.
- **Viterbi** is like *using* that knowledge to decode a specific incoming message: given a stream of beeps, determine the most likely sequence of letters.

You can’t skip either one:

- Without EM: Viterbi has no parameters to work with—it wouldn’t know what Bull/Bear “look like”
- Without Viterbi: EM gives you parameters, but doesn’t tell you which regime *today* belongs to

#### 3.2 The Pipeline in Our Code

The function `train_hmm_for_window()` (`v3_0_train_and_backtest.py`, lines 167-195) executes both phases in sequence:

```

1 def train_hmm_for_window(sector_returns, benchmark_returns, config_name
2     = 'enhanced'):
3     # ---- Phase 1: EM (learn parameters) ----
4     model.fit(sector_returns, n_init=3, verbose=False)           # line 190
5     # At this point, model has learned:
6     #   model.model.transmat_ --> A (transition matrix)
7     #   model.model.means_    --> mu (emission means)
8     #   model.model.covars_   --> Sigma (emission covariances)
9     # ---- Phase 2: Viterbi (decode regimes) ----

```

```
10 regimes = model.get_regime_sequence(sector_returns)      # line 191
11 # At this point, regimes = [0, 0, 0, 1, 1, 2, 2, 0, ...]
12
13 # ---- Phase 3: Label (human-readable names) ----
14 model.label_regimes(sector_returns, benchmark_returns)    # line 192
15 regime_labels = regimes.map(model.regime_labels)          # line 193
16 # Now: ["Bull", "Bull", "Bull", "Sideways", "Sideways", "Bear",
...]
```

## 4 The EM Algorithm (Baum-Welch): Full Derivation

### 4.1 The Problem EM Solves

We want to find parameters  $\lambda = (\mathbf{A}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \boldsymbol{\pi})$  that maximize the **marginal likelihood** of the observed data:

#### Maximum Likelihood Objective

$$\hat{\lambda} = \arg \max_{\lambda} P(\mathbf{O} | \lambda) = \arg \max_{\lambda} \sum_{\text{all } \mathbf{Q}} P(\mathbf{O}, \mathbf{Q} | \lambda) \quad (5)$$

The sum is over all  $K^T$  possible state sequences  $\mathbf{Q}$ . For  $K = 3$  regimes and  $T = 750$  days (3 years of training data), this is  $3^{750}$ —astronomically large.

Direct maximisation is intractable because the hidden states couple everything together. EM resolves this with an iterative two-step approach.

### 4.2 Step 0: The Forward-Backward Algorithm

Before EM can run, it needs a subroutine that computes “soft assignments”—the probability that each time step belongs to each regime, given the current parameter guess. This subroutine is the **Forward-Backward algorithm**.

#### 4.2.1 Forward Variable $\alpha_t(i)$

#### Forward Variable

$$\alpha_t(i) = P(O_1, O_2, \dots, O_t, q_t = i | \lambda) \quad (6)$$

The probability of having seen the first  $t$  observations *and* being in state  $i$  at time  $t$ .

#### Recursion:

$$\text{Initialisation: } \alpha_1(i) = \pi_i \cdot b_i(O_1) \quad (7)$$

$$\text{Induction: } \alpha_t(j) = \left[ \sum_{i=1}^K \alpha_{t-1}(i) \cdot A_{ij} \right] \cdot b_j(O_t) \quad (8)$$

#### Worked Example: Forward Pass (T)

Let us trace through a tiny example with 2 states (Bull/Bear), 3 time steps, and scalar observations.

#### Parameters:

$$\boldsymbol{\pi} = (0.6, 0.4), \quad \mathbf{A} = \begin{pmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{pmatrix}$$

Emissions (simplified):  $b_1(O_t)$  = probability of  $O_t$  under Bull,  $b_2(O_t)$  = under Bear.

Suppose  $b_1(O_1) = 0.8$ ,  $b_2(O_1) = 0.3$  (observation looks “Bullish”).

#### Step 1: Initialisation (Eq. 7):

$$\alpha_1(\text{Bull}) = \pi_1 \cdot b_1(O_1) = 0.6 \times 0.8 = 0.48$$

$$\alpha_1(\text{Bear}) = \pi_2 \cdot b_2(O_1) = 0.4 \times 0.3 = 0.12$$

#### Step 2: Induction at $t = 2$ (Eq. 8):

Suppose  $b_1(O_2) = 0.7$ ,  $b_2(O_2) = 0.4$ .

$$\begin{aligned}\alpha_2(\text{Bull}) &= [\alpha_1(\text{Bull}) \cdot A_{11} + \alpha_1(\text{Bear}) \cdot A_{21}] \cdot b_1(O_2) \\ &= [0.48 \times 0.9 + 0.12 \times 0.2] \times 0.7 \\ &= [0.432 + 0.024] \times 0.7 = 0.456 \times 0.7 = \mathbf{0.3192}\end{aligned}$$

$$\begin{aligned}\alpha_2(\text{Bear}) &= [\alpha_1(\text{Bull}) \cdot A_{12} + \alpha_1(\text{Bear}) \cdot A_{22}] \cdot b_2(O_2) \\ &= [0.48 \times 0.1 + 0.12 \times 0.8] \times 0.4 \\ &= [0.048 + 0.096] \times 0.4 = 0.144 \times 0.4 = \mathbf{0.0576}\end{aligned}$$

Notice: after seeing two “Bullish” observations,  $\alpha_2(\text{Bull}) \gg \alpha_2(\text{Bear})$ . The evidence is accumulating that we’re in a Bull state.

#### 4.2.2 Backward Variable $\beta_t(i)$

##### Backward Variable

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, \dots, O_T \mid q_t = i, \lambda) \quad (9)$$

The probability of observing the *remaining* data, given we’re in state  $i$  at time  $t$ .

**Recursion** (runs backwards from  $T$  to 1):

$$\text{Initialisation: } \beta_T(i) = 1 \quad \forall i \quad (10)$$

$$\text{Induction: } \beta_t(i) = \sum_{j=1}^K A_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j) \quad (11)$$

#### 4.3 Step 1 of EM: E-Step (Expectation)

Using  $\alpha$  and  $\beta$ , we compute two quantities:

##### 4.3.1 State Posterior $\gamma_t(i)$

##### State Posterior

$$\gamma_t(i) = P(q_t = i \mid \mathbf{O}, \lambda) = \frac{\alpha_t(i) \beta_t(i)}{\sum_{j=1}^K \alpha_t(j) \beta_t(j)} \quad (12)$$

The probability of being in state  $i$  at time  $t$ , given **all** the observed data.

This is a “soft assignment”—unlike Viterbi which gives a hard 0/1 assignment,  $\gamma$  gives a probability distribution. For example,  $\gamma_t = (0.85, 0.10, 0.05)$  means “85% chance of Bull, 10% Sideways, 5% Bear.”

### 4.3.2 Transition Posterior $\xi_t(i, j)$

#### Transition Posterior

$$\xi_t(i, j) = P(q_t = i, q_{t+1} = j \mid \mathbf{O}, \lambda) = \frac{\alpha_t(i) A_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{i'} \sum_{j'} \alpha_t(i') A_{i'j'} b_{j'}(O_{t+1}) \beta_{t+1}(j')} \quad (13)$$

The probability of being in state  $i$  at time  $t$  and state  $j$  at time  $t + 1$ .

## 4.4 Step 2 of EM: M-Step (Maximisation)

Use the soft assignments to re-estimate parameters:

#### M-Step Update Equations

##### Transition matrix:

$$\hat{A}_{ij} = \frac{\text{expected } \# \text{ of transitions } i \rightarrow j}{\text{expected } \# \text{ of times in state } i} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \quad (14)$$

##### Emission mean:

$$\hat{\mu}_k = \frac{\sum_{t=1}^T \gamma_t(k) \mathbf{r}_t}{\sum_{t=1}^T \gamma_t(k)} \quad (15)$$

##### Emission covariance:

$$\hat{\Sigma}_k = \frac{\sum_{t=1}^T \gamma_t(k) (\mathbf{r}_t - \hat{\mu}_k)(\mathbf{r}_t - \hat{\mu}_k)^T}{\sum_{t=1}^T \gamma_t(k)} \quad (16)$$

##### Initial distribution:

$$\hat{\pi}_i = \gamma_1(i) \quad (17)$$

#### Intuition: Weighted Averages

Notice that  $\hat{\mu}_k$  is a **weighted average** of all observations, where the weight for observation  $\mathbf{r}_t$  is  $\gamma_t(k)$ —the probability that  $\mathbf{r}_t$  was generated by regime  $k$ . Days that are “probably Bull” contribute heavily to the Bull mean; days that are “probably Bear” contribute heavily to the Bear mean.

This is exactly like computing a weighted sample mean in statistics, but the weights come from the E-step.

## 4.5 Convergence and Multiple Initialisations

**Theorem 4.1** (EM Monotonicity). *Each EM iteration is guaranteed to increase (or maintain) the log-likelihood:*

$$\log P(\mathbf{O} \mid \hat{\lambda}^{(n+1)}) \geq \log P(\mathbf{O} \mid \hat{\lambda}^{(n)})$$

However, EM converges to a **local maximum**, not necessarily the global one. The code therefore runs multiple random initialisations (`models.py`, lines 161-183):

```

1 for i in range(n_init):                                # try multiple
2     starts
3     model = hmm.GaussianHMM(
4         n_components=self.n_regimes,
5         covariance_type=self.covariance_type,
6         n_iter=self.n_iter,                                # max 100 EM
7         iterations

```

```

6         tol=self.tol,
7         improvement < 1e-4
8         random_state=self.random_state + i
9         each time
10        )
11        model.fit(X)
12        self._regularize_covariance(model)
13        singularity
14        score = model.score(X)
15        if score > best_score:
16            best_score = score
17            best_model = model
18
19    # stop if
20    # different seed
21    # run EM
22    # prevent
23    # log-likelihood
24    # keep the best

```

# Why Different Initialisations Matter

Imagine the log-likelihood surface has two peaks:

With `n_init=3`, the code would keep Init 2. Without multiple starts, you might be stuck at -5200.

## 4.6 Regularisation: Preventing Singular Covariances

When feature dimensionality  $D$  is large relative to the data in a regime,  $\hat{\Sigma}_k$  can become singular (determinant = 0). This makes the Gaussian density undefined ( $|\Sigma|^{-1/2}$  blows up).

The fix (`models.py`, lines 104-120):

$$\hat{\Sigma}_k \leftarrow \hat{\Sigma}_k + \lambda \mathbf{I} \quad (18)$$

## Regularisation in 2D

Suppose a regime has so few data points that the estimated covariance is:

$$\hat{\Sigma} = \begin{pmatrix} 0.0004 & 0.0004 \\ 0.0004 & 0.0004 \end{pmatrix} \quad \Rightarrow \quad |\hat{\Sigma}| = 0 \text{ (singular!)}$$

After adding  $\lambda = 10^{-4}$ :

$$\hat{\Sigma} + \lambda \mathbf{I} = \begin{pmatrix} 0.0005 & 0.0004 \\ 0.0004 & 0.0005 \end{pmatrix} \Rightarrow |\hat{\Sigma} + \lambda \mathbf{I}| = 0.09 \times 10^{-6} > 0 \checkmark$$

## 5 The Viterbi Algorithm: Full Derivation

### 5.1 The Problem Viterbi Solves

#### Decoding Problem

Given learned parameters  $\hat{\lambda}$  and a sequence of observations  $\mathbf{O} = (O_1, \dots, O_T)$ , find:

$$\mathbf{Q}^* = \arg \max_{\mathbf{Q} \in \mathcal{S}^T} P(\mathbf{Q}, \mathbf{O} | \hat{\lambda}) \quad (19)$$

The most likely sequence of hidden states.

As your lecture image notes, brute-force search over all  $K^T$  sequences is computationally intractable. Viterbi solves it in  $O(K^2 T)$  time using dynamic programming.

### 5.2 Key Idea: Principle of Optimality

The insight is: *the best path to state  $j$  at time  $t$  must pass through the best path to some state  $i$  at time  $t - 1$ .* We never need to track all  $K^T$  paths—just  $K$  paths (one ending in each state).

### 5.3 The Algorithm

Define:

$$\delta_t(j) = \max_{q_1, \dots, q_{t-1}} P(q_1, \dots, q_{t-1}, q_t = j, O_1, \dots, O_t | \lambda) \quad (20)$$

This is the probability of the **single best path** ending in state  $j$  at time  $t$ .

#### Viterbi Recursion

##### Initialisation:

$$\delta_1(j) = \pi_j \cdot b_j(O_1) \quad (21)$$

##### Recursion:

For  $t = 2, \dots, T$ :

$$\delta_t(j) = \left[ \max_{1 \leq i \leq K} \delta_{t-1}(i) \cdot A_{ij} \right] \cdot b_j(O_t) \quad (22)$$

##### Backtracking pointer

(records which state  $i$  achieved the max):

$$\psi_t(j) = \arg \max_{1 \leq i \leq K} \delta_{t-1}(i) \cdot A_{ij} \quad (23)$$

##### Termination:

$$q_T^* = \arg \max_{1 \leq j \leq K} \delta_T(j) \quad (24)$$

##### Backtracking:

For  $t = T - 1, \dots, 1$ :

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \quad (25)$$

## 5.4 Worked Numerical Example

### Viterbi Step-by-Step ( $K = 3$ )

#### Setup:

$$\pi = (0.6, 0.4), \quad \mathbf{A} = \begin{pmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{pmatrix}, \quad \text{States: } s_1 = \text{Bull}, s_2 = \text{Bear}$$

Emission probabilities (already evaluated from Gaussians):

	$b_1(\cdot)$ (Bull)	$b_2(\cdot)$ (Bear)
$O_1$	0.8	0.3
$O_2$	0.7	0.4
$O_3$	0.2	0.9

(Note:  $O_3$  looks “Bearish”—low Bull probability, high Bear probability.)

#### Step 1: Initialisation ( $t = 1$ )

$$\delta_1(\text{Bull}) = \pi_1 \cdot b_1(O_1) = 0.6 \times 0.8 = \mathbf{0.480}$$

$$\delta_1(\text{Bear}) = \pi_2 \cdot b_2(O_1) = 0.4 \times 0.3 = \mathbf{0.120}$$

#### Step 2: Recursion ( $t = 2$ )

$$\begin{aligned} \delta_2(\text{Bull}) &= \max(\delta_1(\text{Bull}) \cdot A_{11}, \delta_1(\text{Bear}) \cdot A_{21}) \cdot b_1(O_2) \\ &= \max(0.480 \times 0.9, 0.120 \times 0.2) \times 0.7 \\ &= \max(\mathbf{0.432}, 0.024) \times 0.7 = 0.432 \times 0.7 = \mathbf{0.3024} \end{aligned}$$

$$\psi_2(\text{Bull}) = \text{Bull} \quad (0.432 \text{ won})$$

$$\begin{aligned} \delta_2(\text{Bear}) &= \max(\delta_1(\text{Bull}) \cdot A_{12}, \delta_1(\text{Bear}) \cdot A_{22}) \cdot b_2(O_2) \\ &= \max(0.480 \times 0.1, 0.120 \times 0.8) \times 0.4 \\ &= \max(0.048, \mathbf{0.096}) \times 0.4 = 0.096 \times 0.4 = \mathbf{0.0384} \end{aligned}$$

$$\psi_2(\text{Bear}) = \text{Bear} \quad (0.096 \text{ won})$$

#### Step 3: Recursion ( $t = 3$ ) — here $O_3$ looks Bearish!

$$\begin{aligned} \delta_3(\text{Bull}) &= \max(0.3024 \times 0.9, 0.0384 \times 0.2) \times 0.2 \\ &= \max(\mathbf{0.2722}, 0.0077) \times 0.2 = \mathbf{0.05443} \end{aligned}$$

$$\psi_3(\text{Bull}) = \text{Bull}$$

$$\begin{aligned} \delta_3(\text{Bear}) &= \max(0.3024 \times 0.1, 0.0384 \times 0.8) \times 0.9 \\ &= \max(\mathbf{0.03024}, 0.03072) \times 0.9 \\ &= 0.03072 \times 0.9 = \mathbf{0.02765} \end{aligned}$$

$$\psi_3(\text{Bear}) = \text{Bear}$$

#### Step 4: Termination

$$q_3^* = \arg \max (\delta_3(\text{Bull}), \delta_3(\text{Bear})) = \arg \max (0.05443, 0.02765) = \mathbf{Bull}$$

#### Step 5: Backtracking

$$q_2^* = \psi_3(\text{Bull}) = \text{Bull}, \quad q_1^* = \psi_2(\text{Bull}) = \text{Bull}$$

$$\text{Result: } \mathbf{Q}^* = (\text{Bull}, \text{Bull}, \text{Bull})$$

Even though  $O_3$  looked Bearish, Viterbi kept Bull because:

1. The strong Bull momentum from  $O_1$  and  $O_2$  accumulated high  $\delta$  values
2. The transition cost of switching ( $A_{12} = 0.1$  is low) penalises regime changes
3. The overall path probability is higher by staying in Bull

This is the power of Viterbi: it considers the **entire sequence**, not just today's observation.

## 5.5 Viterbi in the Code

The call chain in `models.py`:

```

1 def predict(self, returns: pd.DataFrame) -> np.ndarray:           # line
2     208
3     """Predict regime sequence using Viterbi algorithm."""
4     data = self._preprocess_data(returns)
5     X = data.values
6     return self.model.predict(X)      # hmmlearn's Viterbi implementation
7
8 def get_regime_sequence(self, returns: pd.DataFrame) -> pd.Series: # line 244
9     """Get regime sequence as pandas Series with datetime index."""
10    data = self._preprocess_data(returns)
11    regimes = self.predict(returns)
12    return pd.Series(regimes, index=data.index, name="regime")

```

Note also `predict_proba()` (line 226), which returns the *soft* posterior  $\gamma_t(i)$  from the forward-backward algorithm—useful for uncertainty estimation but **not** used in the trading strategy.

## 6 Viterbi vs. Forward-Backward: A Subtle Distinction

There is yet a *third* algorithm involved, and confusing it with Viterbi is a common source of misunderstanding.

Three Algorithms, Three Purposes			
Algorithm	Question	Output	Code
<b>Forward-Backward</b>	What is $P(q_t = k \mid \mathbf{O})$ for each $t$ ?	Marginal posteriors $\gamma_t(k)$	<code>predict_proba()</code>
<b>Viterbi</b>	What is the single best path $\mathbf{Q}^*$ ?	Hard state sequence	<code>predict()</code>
<b>Baum-Welch (EM)</b>	What are the best $\lambda$ ?	Parameters $\hat{\lambda}$	<code>fit()</code>

### 6.1 Why They Can Disagree

#### Forward-Backward vs. Viterbi Disagreement

Consider a 3-state HMM with observations over 5 days.

**Forward-Backward** (marginal posteriors):

$$\gamma = \begin{pmatrix} 0.50 & 0.30 & 0.20 \\ 0.45 & 0.35 & 0.20 \\ 0.10 & 0.80 & 0.10 \\ 0.40 & 0.35 & 0.25 \\ 0.55 & 0.30 & 0.15 \end{pmatrix} \quad (\text{rows} = \text{time}, \text{columns} = \text{Bull/Sideways/Bear})$$

Taking the arg max of each row independently: [Bull, Bull, Sideways, Bull, Bull]. Viterbi might return: [Bull, Bull, Bull, Bull, Bull].

**Why the difference?** At  $t = 3$ , the marginal posterior says Sideways is most likely (80%). But Viterbi considers the transition cost: switching Bull→Sideways→Bull requires two unlikely transitions. The globally optimal *path* stays in Bull throughout, even though the locally optimal *state* at  $t = 3$  is Sideways.

For trading, Viterbi is preferred because we want a **coherent sequence** (smooth regime signals → fewer unnecessary rebalances → lower transaction costs).

### 6.2 Why the Code Uses Viterbi

In `v3_0_train_and_backtest.py` (line 254), regime prediction uses `get_regime_sequence()`, which calls `predict()` (Viterbi), not `predict_proba()` (Forward-Backward):

```
1 # Predict regimes for test period
2 test_regimes_raw = model.get_regime_sequence(test_returns)    # Viterbi!
3 test_regimes = test_regimes_raw.map(model.regime_labels)
```

This choice matters for the trading strategy. Fewer spurious regime switches means fewer rebalances, which means lower transaction costs (each switch costs 10 bps as configured on line 203):

```
1 rebalance_cost: float = 0.001,      # 10 basis points per rebalance
```

## 7 How EM Uses Forward-Backward Internally

This is the final piece that ties everything together and explains the relationship.

### The Relationship Between All Three Algorithms

**EM calls Forward-Backward as a subroutine in every E-step.**

`fit() → loop: {Forward-Backward (E-step) → Parameter updates (M-step)}`

After `fit()` finishes, `predict()` runs Viterbi with the learned parameters.

The full picture:

1. `model.fit(X)` (line 174 of `models.py`):
  - (a) Initialise  $\lambda^{(0)}$  randomly
  - (b) *Repeat until convergence:*
    - i. **E-step:** Run Forward-Backward with current  $\lambda^{(n)}$  to get  $\gamma_t(i)$  and  $\xi_t(i, j)$
    - ii. **M-step:** Use  $\gamma$  and  $\xi$  to compute  $\lambda^{(n+1)}$
  - (c) Store the converged  $\hat{\lambda}$  (means, covariances, transition matrix)
2. `model.predict(X)` (line 224 of `models.py`):
  - (a) Run **Viterbi** using the learned  $\hat{\lambda}$  from step 1
  - (b) Return the most likely state sequence  $\mathbf{Q}^*$

So Forward-Backward is used *inside* EM (during training), while Viterbi is used *after* EM (during deployment). They are complementary, not redundant.

## 8 Summary: The Complete Pipeline

Let us trace the full pipeline from raw data to trading decisions, labelling which algorithm is used at each step.

Step	What Happens	Algorithm	Code Reference
1	Raw returns → features	Feature Engineering	<code>features.py</code>
2	Learn $\hat{\lambda} = (\hat{\mathbf{A}}, \hat{\boldsymbol{\mu}}_k, \hat{\boldsymbol{\Sigma}}_k)$	<b>EM (Baum-Welch)</b>	<code>model.fit()</code> , line 190
3	Decode training regimes	<b>Viterbi</b>	line 191
4	Label states (Bull/Bear/Side-ways)	Sorting by mean return	lines 192-193
5	Build portfolios per regime	Weighted sector selection	lines 240-249
6	Decode test regimes	<b>Viterbi</b>	line 254
7	Trade based on regime	Portfolio construction	lines 258-280

### 8.1 Key Formulas at a Glance

Concept	Formula
Joint factorisation	$P(\mathbf{Q}, \mathbf{O}   \lambda) = \pi_{q_1} b_{q_1}(O_1) \prod_{t=2}^T A_{q_{t-1}, q_t} b_{q_t}(O_t)$
Forward variable	$\alpha_t(j) = [\sum_i \alpha_{t-1}(i) A_{ij}] b_j(O_t)$
State posterior (E-step)	$\gamma_t(i) = \alpha_t(i) \beta_t(i) / \sum_j \alpha_t(j) \beta_t(j)$
Mean update (M-step)	$\hat{\boldsymbol{\mu}}_k = \sum_t \gamma_t(k) \mathbf{r}_t / \sum_t \gamma_t(k)$
Viterbi recursion	$\delta_t(j) = [\max_i \delta_{t-1}(i) A_{ij}] b_j(O_t)$
Expected duration	$\mathbb{E}[\text{dur}_k] = 1/(1 - A_{kk})$
Sharpe ratio	$\text{SR} = (\bar{r}/\sigma_r) \times \sqrt{252}$
BIC	$\text{BIC} = -2 \ln L + k \ln n$

### 8.2 Final Intuition

#### Why Both EM and Viterbi are Essential

- **EM** answers: “In this market, what does a Bull regime look like? What does a Bear regime look like? How often do regimes switch?”
- **Viterbi** answers: “Given what we learned about Bull and Bear patterns, and given today’s sector returns, which regime is the market in *right now*?”
- You cannot skip EM, because Viterbi needs parameters. You cannot skip Viterbi, because EM does not tell you today’s regime.