Market Regime Detection Using Hidden Markov Models

A Comprehensive Tutorial on
Quantitative Trading Strategy Development

Covering: Financial Intuition • Mathematical Foundations • Implementation Details

With LaTeX Mathematical Derivations and Code References

Contents

1 Introduction and Motivation **3**
1.1 The Problem We're Solving . . . 3
1.2 Key Strategy Parameters . . . 3

2 Hidden Markov Models: Mathematical Foundation **4**
2.1 Intuition: What is an HMM? . . . 4
2.2 Formal Definition . . . 4
2.2.1 Component 1: Transition Matrix $\mathbf{A}$ . . . 4
2.2.2 Component 2: Emission Distributions $\mathbf{B}$ . . . 5
2.2.3 Component 3: Initial Distribution $\boldsymbol{\pi}$ . . . 5
2.3 A Small Example . . . 5
2.4 The Viterbi Algorithm . . . 5

3 Training the HMM: The EM Algorithm **7**
3.1 The Maximum Likelihood Problem . . . 7
3.2 Expectation-Maximization (EM) . . . 7
3.2.1 E-Step: Compute Expected Sufficient Statistics . . . 7
3.2.2 M-Step: Update Parameters . . . 7
3.3 Why Multiple Initializations? . . . 8
3.4 Covariance Regularization . . . 8

4 Feature Engineering for Regime Detection **10**
4.1 Feature Categories . . . 10
4.2 Momentum Features . . . 10
4.2.1 Cumulative Returns . . . 10
4.2.2 Exponential Moving Average (EMA) . . . 10
4.3 Volatility Features . . . 11
4.3.1 Rolling Standard Deviation . . . 11
4.3.2 Rolling Range . . . 11
4.4 Cross-Sectional Features (Market Breadth) . . . 11
4.4.1 Cross-Sectional Dispersion . . . 11
4.4.2 Market Breadth . . . 12
4.5 Sector Rotation Features . . . 12
4.6 Standardization . . . 12

5 Automatic Regime Labeling **14**
5.1 The Labeling Algorithm . . . 14
5.2 Expected Duration . . . 14

6 The Trading Strategy **16**
6.1 Strategy Overview . . . 16
6.2 Return-Proportional Weighting . . . 16
6.3 Portfolio Return Calculation . . . 17
6.4 Transaction Costs . . . 17

7 Backtesting Methodology **18**
7.1 Rolling Window Approach . . . 18
7.2 Performance Metrics . . . 18
7.2.1 Sharpe Ratio . . . 18
7.2.2 Maximum Drawdown . . . 18

8 Model Selection: BIC and Cross-Validation **19**
8.1 How Many Regimes? 19
8.2 The BIC Formula 19
8.3 Parameter Count for Gaussian HMM 19

9 Configuration Management **20**
9.1 Configuration Hierarchy 20
9.2 Predefined Configurations 20

10 Putting It All Together **21**
10.1 The Complete Flow 21

11 Summary and Key Takeaways **22**
11.1 What We Covered 22
11.2 Key Mathematical Formulas 22
11.3 Success Criteria 22
11.4 Further Reading 22

1 Introduction and Motivation

Financial markets exhibit distinct behavioral patterns over time. Sometimes prices trend upward persistently (**bull markets**), other times they decline relentlessly (**bear markets**), and often they move sideways with no clear direction. The core insight behind this project is that these *market regimes* are not directly observable but can be inferred from price data using statistical models.

1.1 The Problem We're Solving

Consider a portfolio manager who wants to adjust their allocation based on market conditions. In a bull market, they might want to be fully invested in high-beta sectors. In a bear market, they might prefer defensive sectors or cash. The challenge is: *How do we objectively identify which regime we're in?*

This project implements a systematic approach using Hidden Markov Models (HMMs) to solve this problem. The strategy, as documented in `v3_0_train_and_backtest.py` (lines 1-15), aims to:

- Automatically detect the current market regime from sector return data
- Allocate capital to the top-performing sectors for each regime
- Weight positions proportionally to expected returns
- Control drawdowns by moving to cash during sideways/uncertain periods

1.2 Key Strategy Parameters

The strategy parameters are defined at the module level (`v3_0_train_and_backtest.py`, lines 27-29):

```python
SUCCESS_SHARPE = 0.5        # Minimum acceptable Sharpe ratio
SUCCESS_MAX_DD = 0.10       # Maximum acceptable drawdown (10%)
ANNUALIZATION_FACTOR = 252 # Trading days per year
```

2 Hidden Markov Models: Mathematical Foundation

2.1 Intuition: What is an HMM?

Imagine you're in a room with no windows, trying to guess the weather outside. You can't observe the weather directly, but you notice that your friend's mood changes: when it's sunny, they tend to be happy; when it's rainy, they tend to be sad. By observing their mood over time, you can make inferences about the hidden weather state.

In financial markets, the "weather" is the market regime (Bull, Bear, or Sideways), and the "mood" is the pattern of sector returns we observe. The HMM formalizes this relationship mathematically.

2.2 Formal Definition

Definition 2.1 (Hidden Markov Model)**.** *An HMM is a probabilistic model with:*

1. *A set of K hidden states $\mathcal{S} = \{s_1, s_2, \ldots, s_K\}$*
2. *A sequence of observations $\mathbf{O} = (O_1, O_2, \ldots, O_T)$*
3. *Three sets of parameters $\lambda = (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$*

The model assumes K hidden states. In our implementation (`models.py`, line 47: `n_regimes=3`), these represent Bull, Bear, and Sideways markets.

2.2.1 Component 1: Transition Matrix A

The $K \times K$ transition matrix defines the probability of moving from one regime to another:

Transition Probability

$$A_{ij} = P(q_t = s_j \mid q_{t-1} = s_i) \tag{1}$$

where q_t denotes the hidden state at time t, and $\sum_{j=1}^{K} A_{ij} = 1$ for all i.

The code exposes this via the `transition_matrix` property (`models.py`, lines 321-326):

```python
@property
def transition_matrix(self) -> np.ndarray:
    """Get the transition probability matrix."""
    if not self.is_fitted:
        raise RuntimeError("Model must be fitted first")
    return self.model.transmat_
```

Example: 3-Regime Transition Matrix

A typical transition matrix might look like:

$$\mathbf{A} = \begin{pmatrix} 0.95 & 0.03 & 0.02 \\ 0.05 & 0.90 & 0.05 \\ 0.02 & 0.08 & 0.90 \end{pmatrix}$$

where rows/columns represent Bull, Sideways, Bear. The diagonal elements (0.95, 0.90, 0.90) indicate high **persistence**—regimes tend to persist for many days.

2.2.2 Component 2: Emission Distributions B

Given regime k, the observed returns follow a multivariate Gaussian distribution:

Gaussian Emission

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

where $\mathbf{r}_t \in \mathbb{R}^D$ is the D-dimensional feature vector, $\boldsymbol{\mu}_k$ is the mean, and $\boldsymbol{\Sigma}_k$ is the covariance matrix.

The `covariance_type` parameter in `config.py` (lines 41-42) controls the structure:

```python
# Model structure
n_regimes: int = 3
covariance_type: str = 'full'  # 'full', 'diag', 'spherical', 'tied'
```

Type	Structure	# Parameters per Regime
`full`	Full covariance $\boldsymbol{\Sigma}_k$	$D(D+1)/2$
`diag`	Diagonal: $\text{diag}(\sigma^2_{k,1}, \dots, \sigma^2_{k,D})$	D
`spherical`	Isotropic: $\sigma^2_k\mathbf{I}$	1
`tied`	Shared: $\boldsymbol{\Sigma}_k = \boldsymbol{\Sigma}$ for all k	$D(D+1)/2$ (total)

2.2.3 Component 3: Initial Distribution π

The probability of starting in each state:

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

2.3 A Small Example

Two-Sector

Consider a tiny market with 2 sectors (Tech, Finance) and 2 regimes:

Regime	Mean Returns $\boldsymbol{\mu}$	Characteristics
Bull ($k = 0$)	$(+0.15\%, +0.08\%)^T$	Both positive; Tech leads
Bear ($k = 1$)	$(-0.20\%, -0.05\%)^T$	Both negative; Tech more volatile

When we observe returns similar to the Bull pattern (Tech outperforming), the HMM infers we're likely in a Bull regime. The posterior probability updates with each observation.

2.4 The Viterbi Algorithm

Given a sequence of observed returns, how do we find the most likely sequence of hidden regimes? The **Viterbi algorithm** solves this using dynamic programming.

Viterbi Recursion

Define $\delta_t(j)$ as the probability of the most likely path ending in state j at time t:

$$\delta_t(j) = \max_i \left[\delta_{t-1}(i) \cdot A_{ij}\right] \cdot b_j(O_t) \tag{4}$$

where $b_j(O_t) = P(O_t \mid q_t = j)$ is the emission probability.
The optimal path is recovered by backtracking:

$$\psi_t(j) = \arg\max_i \left[\delta_{t-1}(i) \cdot A_{ij}\right] \tag{5}$$

This is called implicitly in `models.py` (line 224):

```python
def predict(self, returns: pd.DataFrame) -> np.ndarray:
    data = self._preprocess_data(returns)
    X = data.values
    return self.model.predict(X)  # Viterbi decoding
```

3 Training the HMM: The EM Algorithm

3.1 The Maximum Likelihood Problem

Training an HMM means finding parameters $\lambda^* = (\mathbf{A}^*, \boldsymbol{\mu}^*, \boldsymbol{\Sigma}^*)$ that maximize the likelihood of the observed data:

Maximum Likelihood Objective

$$\lambda^* = \arg\max_{\lambda} P(\mathbf{O} \mid \lambda) = \arg\max_{\lambda} \sum_{\mathbf{Q}} P(\mathbf{O}, \mathbf{Q} \mid \lambda) \tag{6}$$

where the sum is over all possible state sequences $\mathbf{Q}$—exponentially many!

Direct optimization is intractable because the regimes are hidden. We can't simply count transitions.

3.2 Expectation-Maximization (EM)

The **Baum-Welch algorithm** is a special case of EM for HMMs. It alternates between:

3.2.1 E-Step: Compute Expected Sufficient Statistics

Define the **forward** and **backward** variables:

$$\alpha_t(i) = P(O_1, \ldots, O_t, q_t = i \mid \lambda) \tag{7}$$
$$\beta_t(i) = P(O_{t+1}, \ldots, O_T \mid q_t = i, \lambda) \tag{8}$$

These allow us to compute:

- $\gamma_t(i) = P(q_t = i \mid \mathbf{O}, \lambda)$ — probability of being in state i at time t
- $\xi_t(i,j) = P(q_t = i, q_{t+1} = j \mid \mathbf{O}, \lambda)$ — probability of transition $i \to j$ at time t

E-Step Formulas

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{K} \alpha_t(j)\beta_t(j)} \tag{9}$$

$$\xi_t(i,j) = \frac{\alpha_t(i)A_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{K}\sum_{j=1}^{K} \alpha_t(i)A_{ij}b_j(O_{t+1})\beta_{t+1}(j)} \tag{10}$$

3.2.2 M-Step: Update Parameters

M-Step Updates

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{11}$$

$$\hat{\boldsymbol{\mu}}_k = \frac{\sum_{t=1}^{T} \gamma_t(k)\mathbf{r}_t}{\sum_{t=1}^{T} \gamma_t(k)} \tag{12}$$

$$\hat{\boldsymbol{\Sigma}}_k = \frac{\sum_{t=1}^{T} \gamma_t(k)(\mathbf{r}_t - \hat{\boldsymbol{\mu}}_k)(\mathbf{r}_t - \hat{\boldsymbol{\mu}}_k)^T}{\sum_{t=1}^{T} \gamma_t(k)} \tag{13}$$

The implementation in `models.py` (lines 122-206) handles this with multiple random initializations:

```python
def fit(self, returns: pd.DataFrame, n_init: int = 10, verbose: bool = True):
    data = self._preprocess_data(returns)
    X = data.values

    best_model = None
    best_score = -np.inf

    for i in range(n_init):
        model = hmm.GaussianHMM(
            n_components=self.n_regimes,
            covariance_type=self.covariance_type,
            n_iter=self.n_iter,
            tol=self.tol,
            random_state=self.random_state + i if self.random_state else None
        )
        model.fit(X)

        # Apply regularization after fitting
        self._regularize_covariance(model)

        score = model.score(X)
        if score > best_score:
            best_score = score
            best_model = model
```

3.3 Why Multiple Initializations?

The EM algorithm only guarantees convergence to a *local* maximum, not the global one. Different starting points can lead to different solutions.

Local vs Global Maxima

Imagine the log-likelihood surface has multiple peaks. Starting from point A might converge to a suboptimal peak, while starting from point B reaches the global maximum. The code tries `n_init` random initializations (default: 10) and keeps the best.

3.4 Covariance Regularization

With many features and limited data, covariance matrices can become singular (non-invertible). The solution is **ridge regularization** (`models.py`, lines 104-120):

Ridge Regularization

$$\hat{\mathbf{\Sigma}}_k \leftarrow \hat{\mathbf{\Sigma}}_k + \lambda \mathbf{I} \tag{14}$$

where $\lambda = 10^{-4}$ (`regularization` parameter). This ensures all eigenvalues are at least $\lambda > 0$.

```python
def _regularize_covariance(self, model: hmm.GaussianHMM) -> None:
    if self.regularization > 0 and self.covariance_type in ["full", "tied"]:
        if self.covariance_type == "full":
            for k in range(self.n_regimes):
                n_features = model.covars_[k].shape[0]
                model.covars_[k] += self.regularization * np.eye(
    n_features)
```

4 Feature Engineering for Regime Detection

Raw daily returns are noisy. The `RegimeFeatureEngine` class (`features.py`) transforms them into more informative signals.

4.1 Feature Categories

The engine computes four categories of features, configured in `config.py` (lines 8-31):

Category	What It Captures	Code Reference
Momentum	Trend direction/strength	`_compute_momentum()`
Volatility	Market turbulence	`_compute_volatility()`
Cross-Sectional	Market breadth	`_compute_cross_sectional()`
Sector Rotation	Risk appetite	`_compute_sector_rotation()`

4.2 Momentum Features

Momentum captures trend direction at multiple time scales. The implementation (`features.py`, lines 171-189) computes:

4.2.1 Cumulative Returns

Cumulative Returns over Window w

$$\text{CumRet}_t(w) = \sum_{i=0}^{w-1} r_{t-i} \tag{15}$$

This is the total return over the past w days. Windows: [5, 20, 60] days.

4.2.2 Exponential Moving Average (EMA)

Exponential Moving Average

$$\text{EMA}_t = \alpha \cdot r_t + (1-\alpha) \cdot \text{EMA}_{t-1}, \quad \alpha = \frac{2}{w+1} \tag{16}$$

The EMA gives more weight to recent observations, making it responsive to regime changes.

```python
def _compute_momentum(self, returns: pd.DataFrame) -> tuple:
    features = []
    names = []

    for window in self.momentum_windows:  # [5, 20, 60]
        # Cumulative returns (sum)
        cum_ret = returns.rolling(window).sum()
        for col in returns.columns:
            features.append(cum_ret[col])
            names.append(f"{col}_mom_cum_{window}d")

        # Exponential moving average
        ema = returns.ewm(span=window).mean()
        for col in returns.columns:
            features.append(ema[col])
            names.append(f"{col}_mom_ema_{window}d")

    return features, names
```

4.3 Volatility Features

Volatility tends to cluster—high volatility days follow high volatility days. Bear markets typically show higher volatility than bull markets.

4.3.1 Rolling Standard Deviation

Rolling Volatility

$$\sigma_t(w) = \sqrt{\frac{1}{w-1}\sum_{i=0}^{w-1}(r_{t-i} - \bar{r})^2} \tag{17}$$

where $\bar{r}$ is the mean return over the window.

4.3.2 Rolling Range

The difference between max and min returns over the window:

$$\text{Range}_t(w) = \max_{i\in[0,w-1]} r_{t-i} - \min_{i\in[0,w-1]} r_{t-i} \tag{18}$$

This captures extreme moves that standard deviation might smooth out.

```python
def _compute_volatility(self, returns: pd.DataFrame) -> tuple:
    for window in self.volatility_windows:
        # Rolling standard deviation
        vol = returns.rolling(window).std()

        # Rolling range (max - min)
        roll_range = (
            returns.rolling(window).max() -
            returns.rolling(window).min()
        )
```

4.4 Cross-Sectional Features (Market Breadth)

These features measure how uniformly the market moves (`features.py`, lines 214-241):

4.4.1 Cross-Sectional Dispersion

Cross-Sectional Dispersion

$$\text{Dispersion}_t = \text{std}_{i\in\text{sectors}}(r_{i,t}) \tag{19}$$

High dispersion means some sectors are winning while others are losing—a possible regime transition signal.

4.4.2 Market Breadth

Market Breadth

$$\text{Breadth}_t(w) = \frac{\#\{i : \sum_{j=0}^{w-1} r_{i,t-j} > 0\}}{N_{\text{sectors}}} \tag{20}$$

The fraction of sectors with positive cumulative returns over window w.

- Breadth $> 80\%$ suggests a broad bull market
- Breadth $< 20\%$ suggests a broad bear market

```python
# Market breadth: % sectors with positive returns over window
for window in [5, 20]:
    breadth = (
        returns.rolling(window).sum() > 0
    ).sum(axis=1) / returns.shape[1]
    features.append(breadth)
    names.append(f"cs_breadth_{window}d")
```

4.5 Sector Rotation Features

The **cyclical vs. defensive** spread is a classic regime indicator (`features.py`, lines 243-285).

Sector Rotation Signal

$$\text{Rotation}_t = \bar{r}_t^{\text{cyclical}} - \bar{r}_t^{\text{defensive}} \tag{21}$$

where $\bar{r}_t^{\text{cyclical}}$ is the average return of cyclical sectors (tech, consumer discretionary) and $\bar{r}_t^{\text{defensive}}$ is the average of defensive sectors (utilities, banks).

- **Bull markets**: Cyclical sectors outperform $\Rightarrow$ Rotation > 0
- **Bear markets**: Defensive sectors hold up better $\Rightarrow$ Rotation < 0

```python
# Cyclical - Defensive spread
cyclical_avg = returns[cyclical_sectors].mean(axis=1)
defensive_avg = returns[defensive_sectors].mean(axis=1)

rotation = cyclical_avg - defensive_avg
features.append(rotation)
names.append("rotation_cyc_def")

# Momentum of rotation (20-day)
rotation_mom = rotation.rolling(20).sum()
features.append(rotation_mom)
names.append("rotation_momentum_20d")
```

4.6 Standardization

All features are standardized to z-scores using `sklearn.preprocessing.StandardScaler`:

Z-Score Standardization

$$z_{i,t} = \frac{x_{i,t} - \hat{\mu}_i}{\hat{\sigma}_i} \tag{22}$$

where $\hat{\mu}_i$ and $\hat{\sigma}_i$ are the sample mean and standard deviation of feature i.

This ensures features with different scales contribute equally to the HMM's Gaussian emissions.

5 Automatic Regime Labeling

The HMM outputs regime indices (0, 1, 2), but we need meaningful labels (Bull, Bear, Sideways). The `label_regimes()` method (`models.py`, lines 259-319) does this automatically.

5.1 The Labeling Algorithm

1. For each regime k, compute the average benchmark return during that regime:

$$\bar{R}_k = \frac{1}{|\{t : q_t = k\}|} \sum_{t:q_t=k} r_t^{\text{benchmark}} \tag{23}$$

2. Sort regimes by $\bar{R}_k$ in descending order

3. Assign labels: highest $\rightarrow$ Bull, middle $\rightarrow$ Sideways, lowest $\rightarrow$ Bear

```python
# Compute statistics per regime
regime_stats = {}
for r in range(self.n_regimes):
    mask = regimes == r
    regime_stats[r] = {
        "mean": ref_returns[mask].mean(),
        "std": ref_returns[mask].std(),
        "count": mask.sum()
    }

# Sort regimes by mean return
sorted_by_return = sorted(regime_stats.keys(),
                          key=lambda x: regime_stats[x]["mean"],
                          reverse=True)

# Assign labels (for 3 regimes)
labels = {
    sorted_by_return[0]: "Bull",
    sorted_by_return[1]: "Sideways",
    sorted_by_return[2]: "Bear"
}
```

5.2 Expected Duration

A useful property is the **expected duration** of each regime, derived from the transition matrix (`models.py`, lines 358-370):

Expected Duration

$$\mathbb{E}[\text{duration of regime } k] = \frac{1}{1 - A_{kk}} \tag{24}$$

This is the expected number of time steps before transitioning out of regime k.

Duration Example

If $A_{\text{Bull,Bull}} = 0.95$, the expected Bull market duration is:

$$\frac{1}{1 - 0.95} = \frac{1}{0.05} = 20 \text{ days}$$

Intuitively: if there's a 95% chance of staying in Bull each day, on average you'll stay for 20 days before transitioning.

```python
def expected_duration(self) -> np.ndarray:
    """Compute expected duration in each regime."""
    # Expected duration = 1 / (1 - p_ii)
    diag = np.diag(self.transition_matrix)
    return 1 / (1 - diag + 1e-10)
```

6 The Trading Strategy

The backtest implementation in `v3_0_train_and_backtest.py` translates regime predictions into portfolio decisions.

6.1 Strategy Overview

The v3.0 strategy (lines 1-15) operates as follows:

Regime	**Allocation**	**Weighting**
Bull	100% in top 5 sectors	Return-proportional
Bear	100% in top 5 (least bad)	Return-proportional
Sideways	0% (Cash)	N/A

6.2 Return-Proportional Weighting

Unlike equal-weighting, this strategy allocates more to sectors with higher expected returns. The implementation (`v3_0_train_and_backtest.py`, lines 65-89):

Return-Proportional Weights

Given top N sectors with average returns $(R_1, R_2, \ldots, R_N)$ during the regime:

$$w_i = \frac{\tilde{R}_i}{\sum_{j=1}^{N} \tilde{R}_j}, \quad \text{where } \tilde{R}_i = R_i - \min_j R_j + \epsilon \tag{25}$$

The shift ensures all weights are positive even when returns are negative (bear market).

```python
def get_top_sectors_with_weights(
    regime_sector_returns: dict,
    regime: str,
    n_sectors: int = 5
) -> dict[str, float]:
    sector_returns = regime_sector_returns[regime]
    sorted_sectors = sorted(sector_returns.items(),
                           key=lambda x: x[1], reverse=True)
    top_sectors = sorted_sectors[:n_sectors]

    # Calculate return-proportional weights
    returns = np.array([ret for _, ret in top_sectors])

    # Shift returns to be positive if any are negative
    if returns.min() < 0:
        returns = returns - returns.min() + 0.01

    # Normalize to sum to 1
    weights = returns / returns.sum()

    return {sector: weight for (sector, _), weight in zip(top_sectors,
    weights)}
```

Weight Calculation Example

Suppose the top 3 sectors in a Bull regime have average daily returns:

Tech: +0.15%, Finance: +0.10%, Energy: +0.05%

Raw returns: $[0.15, 0.10, 0.05]$
All positive, so no shift needed. Normalized weights:

$$w = \left[\frac{0.15}{0.30}, \frac{0.10}{0.30}, \frac{0.05}{0.30}\right] = [50\%, 33\%, 17\%]$$

The best-performing sector (Tech) gets the highest allocation.

6.3 Portfolio Return Calculation

The daily portfolio return is computed as (`v3_0_train_and_backtest.py`, lines 92-103):

Portfolio Return

$$R_t^{\text{portfolio}} = \alpha \cdot \sum_{i=1}^{N} w_i \cdot r_{i,t} \tag{26}$$

where α is the allocation percentage (100% for Bull/Bear, 0% for Sideways).

```python
def calculate_portfolio_return(
    target_portfolio: dict[str, float],
    sector_returns: pd.Series,
    allocation_pct: float
) -> float:
    """Calculate weighted portfolio return for a given day."""
    if not target_portfolio:
        return 0.0

    weighted_return = sum(
        weight * sector_returns[sector]
        for sector, weight in target_portfolio.items()
    )
    return weighted_return * allocation_pct
```

6.4 Transaction Costs

Each rebalance incurs a cost of `rebalance_cost = 0.001` (10 basis points):

```python
# Apply transaction cost if rebalancing
if needs_rebalance:
    daily_return -= rebalance_cost
    current_portfolio = target_portfolio
    days_held = 0
```

7 Backtesting Methodology

7.1 Rolling Window Approach

To avoid look-ahead bias, the backtest uses **rolling out-of-sample windows** (`v3_0_train_and_backtest.py`, lines 32-44).

1. **Train**: Fit HMM on 3 years of historical data
2. **Test**: Apply to the next 6 months (unseen data)
3. **Roll**: Advance by 6 months and repeat

```
TEST_WINDOWS = [
    ('2015-01-01', '2017-12-31', '2018-01-01', '2018-06-30'),
    ('2015-07-01', '2018-06-30', '2018-07-01', '2018-12-31'),
    ('2016-01-01', '2018-12-31', '2019-01-01', '2019-06-30'),
    # ... continues through 2023
]
```

This ensures the model never sees future data during training—critical for realistic performance estimates.

7.2 Performance Metrics

The backtest computes several metrics (`v3_0_train_and_backtest.py`, lines 107-123):

7.2.1 Sharpe Ratio

Annualized Sharpe Ratio

$$\text{Sharpe} = \frac{\bar{r}}{\sigma_r} \times \sqrt{252} \tag{27}$$

where $\bar{r}$ is the mean daily return and σ_r is the daily standard deviation. The $\sqrt{252}$ annualizes from daily to yearly.

```
def calculate_sharpe(returns: np.ndarray) -> float:
    """Calculate annualized Sharpe ratio."""
    if returns.std() == 0:
        return 0.0
    return returns.mean() / returns.std() * np.sqrt(
    ANNUALIZATION_FACTOR)
```

7.2.2 Maximum Drawdown

Maximum Drawdown

$$\text{MaxDD} = \min_t \left[\frac{\text{NAV}_t}{\max_{s \le t} \text{NAV}_s} - 1 \right] \tag{28}$$

The largest peak-to-trough decline in portfolio value.

```
def calculate_max_drawdown(cumulative_returns: pd.Series) -> float:
    """Calculate maximum drawdown from cumulative returns."""
    return (cumulative_returns / cumulative_returns.cummax() - 1).min()
```

8 Model Selection: BIC and Cross-Validation

8.1 How Many Regimes?

Choosing the number of regimes K is critical:

- Too few $\Rightarrow$ miss important market states
- Too many $\Rightarrow$ overfit to noise

The `select_n_regimes()` function (`models.py`, lines 467-517) uses the **Bayesian Information Criterion (BIC)**.

8.2 The BIC Formula

Bayesian Information Criterion

$$\text{BIC} = -2\ln L + k\ln n \tag{29}$$

where L is the likelihood, k is the number of free parameters, and n is the sample size. **Lower BIC is better.**

The first term rewards fit quality (high likelihood). The second term penalizes complexity (many parameters).

8.3 Parameter Count for Gaussian HMM

The total number of parameters depends on model configuration (`models.py`, lines 391-405):

HMM Parameter Count

$$k = \underbrace{K(K-1)}_{\text{transitions}} + \underbrace{(K-1)}_{\text{initial}} + \underbrace{K\cdot D}_{\text{means}} + \underbrace{k_{\text{cov}}}_{\text{covariances}} \tag{30}$$

where:

- Full covariance: $k_{\text{cov}} = K\cdot D(D+1)/2$
- Diagonal covariance: $k_{\text{cov}} = K\cdot D$
- Spherical: $k_{\text{cov}} = K$
- Tied: $k_{\text{cov}} = D(D+1)/2$

```python
# Number of parameters calculation
if self.covariance_type == "diag":
    n_cov_params = k * n_features
elif self.covariance_type == "full":
    n_cov_params = k * n_features * (n_features + 1) / 2
elif self.covariance_type == "spherical":
    n_cov_params = k
else:  # tied
    n_cov_params = n_features * (n_features + 1) / 2

n_params = k * (k - 1) + (k - 1) + k * n_features + n_cov_params
```

9 Configuration Management

The `config.py` module provides a centralized, type-safe way to manage all model parameters.

9.1 Configuration Hierarchy

Config Class	Key Parameters
`FeatureConfig`	`include_momentum`, `momentum_windows`, `standardize`
`HMMConfig`	`n_regimes`, `covariance_type`, `regularization`
`ValidationConfig`	`min_persistence`, `min_separation`
`TradingConfig`	`bull_allocation`, `commission`, `stop_loss`

9.2 Predefined Configurations

Three preset configurations are provided (`config.py`, lines 222-277):

```python
# Baseline: Raw returns only
BASELINE_CONFIG = RegimeDetectionConfig(
    feature_config=FeatureConfig(
        include_momentum=False,
        include_volatility=False,
        include_cross_sectional=False,
        include_sector_rotation=False
    ),
    hmm_config=HMMConfig(
        n_regimes=3,
        covariance_type='diag',
        regularization=0.0
    )
)

# Enhanced: Full feature engineering
ENHANCED_CONFIG = RegimeDetectionConfig(
    feature_config=FeatureConfig(
        include_momentum=True,
        include_volatility=True,
        include_cross_sectional=True,
        include_sector_rotation=True,
        momentum_windows=[5, 20, 60],
        volatility_windows=[5, 20, 60]
    ),
    hmm_config=HMMConfig(
        n_regimes=3,
        covariance_type='full',
        regularization=1e-4,
        smoothing_window=30
    )
)
```

10 Putting It All Together

The `run_v3_backtest()` function (`v3_0_train_and_backtest.py`, lines 134-217) orchestrates the full backtest.

10.1 The Complete Flow

1. **For each rolling window:**
   (a) Extract training data
   (b) Train HMM and label regimes
   (c) Learn which sectors perform best in each regime
   (d) Build optimized portfolios
2. **Testing phase:**
   (a) Predict regimes on unseen test data
   (b) Execute strategy day-by-day
   (c) Apply transaction costs on rebalances
3. **Aggregate results** across all windows

```python
# Training phase
for idx, (train_start, train_end, test_start, test_end) in enumerate(
    TEST_WINDOWS):
     # Extract and train
     train_returns = sector_returns[train_mask]
     model, train_regimes = train_hmm_for_window(train_returns,
    train_benchmark)

     # Learn optimal portfolios
     regime_sector_returns = calculate_regime_sector_returns(
    train_returns, train_regimes)
     bull_portfolio = get_top_sectors_with_weights(regime_sector_returns
    , 'Bull', 5)
     bear_portfolio = get_top_sectors_with_weights(regime_sector_returns
    , 'Bear', 5)

     # Testing phase
     test_regimes = model.get_regime_sequence(test_returns).map(model.
    regime_labels)

     for date, regime in test_regimes.items():
         # Select portfolio based on regime
         if regime == 'Bull':
             target_portfolio = bull_portfolio
         elif regime == 'Bear':
             target_portfolio = bear_portfolio
         else:  # Sideways
             target_portfolio = {}

         # Calculate return and apply costs
         daily_return = calculate_portfolio_return(target_portfolio,
    ...)
         if needs_rebalance:
             daily_return -= rebalance_cost
```

11 Summary and Key Takeaways

11.1 What We Covered

1. **Hidden Markov Models**: A probabilistic framework for inferring unobservable market regimes from return data
2. **Feature Engineering**: Transforming noisy returns into multi-scale momentum, volatility, breadth, and rotation signals
3. **Regime-Based Trading**: Dynamically adjusting sector allocations based on predicted market state
4. **Backtesting**: Rolling out-of-sample windows to simulate realistic trading conditions
5. **Model Selection**: Using BIC to balance fit quality against complexity

11.2 Key Mathematical Formulas

Concept	**Formula**
Transition Probability	$A_{ij} = P(q_t = j \mid q_{t-1} = i)$
Gaussian Emission	$P(\mathbf{r}_t \mid q_t = k) = \mathcal{N}(\mathbf{r}_t; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$
Expected Duration	$\mathbb{E}[\text{duration}] = 1/(1 - A_{kk})$
Sharpe Ratio	$\text{SR} = (\bar{r}/\sigma_r) \times \sqrt{252}$
BIC	$\text{BIC} = -2 \ln L + k \ln n$

11.3 Success Criteria

The strategy is considered successful if (lines 27-28):

- **Sharpe Ratio** > 0.5: Generates reasonable risk-adjusted returns
- **Max Drawdown** $< 10\%$: Controls downside risk through regime detection

The hypothesis (line 14): "Higher allocation + better sector weighting will improve returns while regime detection keeps drawdown controlled."

11.4 Further Reading

- Rabiner, L. R. (1989). "A Tutorial on Hidden Markov Models" — The classic HMM reference
- Hamilton, J. D. (1989). "A New Approach to the Economic Analysis of Nonstationary Time Series" — Regime switching in economics
- Murphy, K. P. (2012). "Machine Learning: A Probabilistic Perspective" — Chapters 17-18 on HMMs

End of Tutorial