Using Hidden Markov Models to Detect Market Regimes

Christopher Lee

March 2025

Contents

1 Introduction

In my last project, I encountered mathematical concepts that were slightly beyond my current knowledge level, particularly due to my limited background in differential equations. As a result, I decided to focus on a more fundamental—yet equally fascinating—concept: Markov Chains.

Markov Chains are mathematical models that describe systems that transition between different states based on probabilistic rules. What makes them particularly interesting is the Markov property, which states that the probability of moving to the next state depends only on the current state, not on the sequence of states that preceded it. This makes Markov Chains especially useful for modeling time series data, stochastic processes, and even financial markets, where past behavior influences but does not deterministically determine future outcomes.

This project explores a more advanced application of Markov Chains: Hidden Markov Models (HMMs). Unlike simple Markov Chains, where states are directly observable, HMMs assume that the true state of the system is hidden, and we can only observe the outcomes it produces. This makes HMMs a powerful tool for tasks such as market regime detection, where we attempt to classify financial market conditions (e.g., bull or bear markets) based on observable price movements.

1.1 Project Overview

The goal of this project is to implement a Hidden Markov Model (HMM) from scratch in Rust to analyze financial market regimes. This involves:

- Defining the model parameters, including transition probabilities, emission probabilities, and initial state distributions.
- Implementing key algorithms such as the Forward-Backward Algorithm (for probability calculations) and the Viterbi Algorithm (for state prediction).
- Testing the model on historical market data to analyze how well it identifies different market conditions.

The rest of the report will be organized in other sections. Section 2 will introduce Markov Chains and Hidden Markov Models. Section 3 will talk about methedology with my implementation. Section 4 will detail results and analysis. Section 5 will be a concluding section reflecting on my progress.

2 Background

I first began my research with the book Markov Chains: From Theory to Implementation and Experimentation by Paul A. Gagniuc. This book provides a foundational understanding of Markov Models, offering an intuitive approach to their implementation. The examples presented in the book helped me grasp the basic principles of stochastic processes and state transitions, which are essential to understanding more complex applications like Hidden Markov Models (HMMs).

2.1 What Are Markov Models?

A Markov Model is a type of stochastic model that represents a system where the future state depends only on the present state, not on the sequence of states that preceded it. This defining characteristic is known as the Markov property. Mathematically, for a discrete-time Markov Chain, this can be expressed as:

$$P(X_{t+1}|X_t, X_{t-1}, ..., X_0) = P(X_{t+1}|X_t)$$

where X_t represents the state of the system at time t.

In the book, Gagniuc uses an analogy of a cup game, where one can calculate transition probabilities of drawing a specific type of ball from a specific type of cup. The game involves drawing a ball from a cup, and drawing the subsequent ball from the correlating cup. This gave me a fundamental understanding of Markov Chains and sparked the initial interest in this project.

Markov Models are widely used in various fields, including finance, genetics, speech recognition, and reinforcement learning. In financial modeling, they can help identify market trends by analyzing sequences of asset price movements.

In the next subsection, we will extend this idea to Hidden Markov Models (HMMs), which introduce an additional layer of complexity by assuming that the underlying states are not directly observable but can only be inferred from observable emissions.

2.2 Hidden Markov Models (HMM)

While basic Markov Models assume that the states of a system are directly observable, Hidden Markov Models (HMMs) introduce an additional layer of complexity: the true states are hidden and can only be inferred through observable outputs. This makes HMMs particularly useful in fields like finance, speech recognition, bioinformatics, and natural language processing (Rabiner, 1989).

An HMM generally consists of the following:

- States (S): A finite set of hidden states that evolve over time.
- Observations (O): The set of observable symbols generated by hidden states.
- Transition probabilities (A): A matrix representing the probability of moving from one state to another.
- Emission Matrix (B): A matrix defining the probability of observing a particular output from a given hidden state.
- Initial State Probabilities (π): : A probability distribution over the initial states at $t = 0$

Mathematically speaking, an HMM characterizes itself as a triplet (π, A, B)

2.3 Applications of HMMs in Financial Markets

In financial modeling, HMMs are used to detect market regimes—periods in which market behavior follows a distinct pattern (Zucchini et al., 2017). For example, an HMM can classify a financial time series into bullish (expansion) and bearish (recession) states based on price movements (Kritzman et al., 2012).

To implement this, an HMM is trained using historical price data and is then used to infer the most likely hidden state at each point in time. This approach can help traders and portfolio managers adjust their strategies based on prevailing market conditions.

In the next section, I will detail the implementation of an HMM-based market regime detection system, outlining the methodology, coding structure, and challenges encountered.

3 Implementation of Hidden Markov Models in Rust

This section details the design, methodology, and implementation of the Hidden Markov Model (HMM) used for market regime detection. The goal is to classify market conditions into distinct states (e.g., bullish or bearish) based on historical price movements.

3.1 Methods and Libraries

I decided once again to program this in rust, which initially seemed like a good idea but in hindsight was the worst decision (maybe I should give python another try.) This project wouldn't be a demonstration of my understanding of the math if I used a pre-existing HMM model, so I decided to learn how to implement it myself. First what I had to do was pull market data for SPY from the internet. I decided to use the yahoo_finance_api crate, as they were the most robust option.

Another (very bad) thing that you'll notice is my use of ndarray, which is a pretty solid crate for creating n-dimensional arrays (better known as vectors.) About halfway through programming I decided that using vectors was probably going to be faster, and so some of the functions have weird parameters. I did not refactor the code to be more consistent because of the time it would take to change everything.

For the HMM model, I decided to implement it using a struct that contained the number of states (which in this case is 3: Bullish, Bearish, and Neutral), number of observations, initial probabilities, and the relevant transition and emission matrices. This model is able to be trained based on observations using a Baum-Welch algorithm, and can be used to predict market regimes.

```rust
pub fn train(&mut self, observations: &[usize], iterations: usize) {
    if observations.is_empty() {
        return;
    }

    for _ in 0..iterations {
        // E-step: Compute scaled forward and backward variables
        let (alpha: ArrayBase<OwnedRepr<f64>, …>, scales: Vec<f64>) = self.scaled_forward(observations);
        let beta: ArrayBase<OwnedRepr<f64>, …> = self.scaled_backward(observations, &scales);

        // Compute posterior probabilities
        let (gamma: ArrayBase<OwnedRepr<f64>, …>, xi: ArrayBase<O…) = self.compute_gamma_xi(&alpha, &beta, observations, &scales);

        // M-step: Update parameters
        self.update_initial_probabilities(&gamma);
        self.update_transition_matrix(&gamma, &xi);
        self.update_emission_matrix(&gamma, observations);

        // Ensure valid probability distributions
        self.normalize_parameters();
    }
}
```

Figure 1: Train Function in HMM

3.2 Implementation

In this subsection, I will include screenshots of code and explain what is happening. (This seems like a better approch than my previous project.) As you can see in figure 1, the train function goes through a certain number of iterations, generating alpha, beta, and gamma/xi values through each iteration and updating itself. But what does any of this mean?

3.2.1 Understanding the Baum-Welch Algorithm

The Baum-Welch algorithm relies on the following inputs:

- Observations $O = (O_1, O_2, ..., O_T)$
- Number of hidden states N
- Initial guesses for:
  - Transition matrix A
  - Emission matrix B
  - Initial state probabilities π

The algorithm returns the following outputs:

- Updated estimates of A, B, and π that maximizes the likelihood of observations

After receiving inputs, the algorithm is as follows:

- Compute forward probabilities
- Compute backward probabilities

- Compute state probabilities
- Update parameters

Now, we must understand what forward/backward/state probabilities are, and how to implement them for our purpose.

3.2.2 Calculating Forward Probabilities

The forward algorithm calculates the proabability of observing the sequence up to time t given state i. It is demonstrated by the following recursive formula:

$$\alpha_t(i) = \sum_{j=1}^{N} a_{t-1}(j) A_{ji} B_{iO_t}$$

Where:

- $\alpha_t(i)$ is the probability of being in state i at time t, given observations up to t
- A_{ji} is the transition probability from state j to state i
- B_{iO_t} is the emission proabability of observing O_t from state i

In our program, this characterizes itself in its implementation in Figure 2.

3.2.3 Calculating Backward Probabilities

The backward algorithm calculates the probability of observing the rest of the sequence given state i at time t. It is demonstrated by the following formula:

$$\beta_t(i) = \sum_{j=1}^{N} A_{ij} B_{jO_{t+1}} \beta_{t+1}(j)$$

Where:

- $\beta_t(i)$ is the probability of observing future observations given that we are in state i at time t

You can see the way I implemented this in the code above. It's a pretty straight forward function, where within a nested for loop we perform the computations and return beta

```rust
// Implementation of scaled forward algorithm with normalization
fn scaled_forward(&self, observations: &[usize]) -> (Array2<f64>, Vec<f64>) {
    let n: usize = observations.len();
    let mut alpha: ArrayBase<OwnedRepr<f64>, …> = Array2::zeros(shape: (n, self.num_states));
    let mut scales: Vec<f64> = vec![0.0; n];

    // Initialize first time step
    for i: usize in 0..self.num_states {
        alpha[[0, i]] = self.initial_probabilities[i] *
                        self.emission_matrix[[i, observations[0]]];
    }
    scales[0] = alpha.row(index: 0).sum();
    if scales[0] > 0.0 {
        alpha.row_mut(index: 0).mapv_inplace(|x: f64| x / scales[0]);
    }

    // Recursion for subsequent steps
    for t: usize in 1..n {
        for j: usize in 0..self.num_states {
            let sum: f64 = (0..self.num_states) Range<usize>
                .map(|i: usize| alpha[[t-1, i]] * self.transition_matrix[[i, j]]) impl Iterat
                .sum();
            alpha[[t, j]] = sum * self.emission_matrix[[j, observations[t]]];
        }
        scales[t] = alpha.row(index: t).sum();
        if scales[t] > 0.0 {
            alpha.row_mut(index: t).mapv_inplace(|x: f64| x / scales[t]);
        }
    }

    (alpha, scales)
} fn scaled_forward
```

Figure 2: Forward Algorithm

```rust
// Implementation of scaled backward algorithm
fn scaled_backward(&self, observations: &[usize], scales: &[f64]) -> Array2<f64> {
    let n: usize = observations.len();
    let mut beta: ArrayBase<OwnedRepr<f64>, …> = Array2::ones(shape: (n, self.num_states));

    // Start from the end of the sequence
    let last_scale: f64 = scales[n-1];
    if last_scale > 0.0 {
        beta.row_mut(index: n-1).mapv_inplace(|x: f64| x / last_scale);
    }

    // Work backwards through the sequence
    for t: usize in (0..n-1).rev() {
        for i: usize in 0..self.num_states {
            let mut sum: f64 = 0.0;
            for j: usize in 0..self.num_states {
                sum += self.transition_matrix[[i, j]] *
                       self.emission_matrix[[j, observations[t+1]]] *
                       beta[[t+1, j]];
            }
            beta[[t, i]] = sum / scales[t];
        }
    }

    beta
} fn scaled_backward
```

Figure 3: Backward Algorithm

3.2.4 Calculating State Probabilities

To estimate how likely it is to be in a certain state at a certain time, we use the following formula:

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

This gives the probability of being in state i at time t.

The probability of transitioning from state i to j at time t is given by:

$$\xi_t(i,j) = \frac{\alpha_t(i)A_{ij}B_{jO_{t+1}}\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_t(i)A_{ij}B_{jO_{t+1}}\beta_{t+1}(j)}$$

This is all implemented in figure 4 using for loops.

After all of this is calculated, we then update the model parameters. With our initial train function, we perform these computations many times (200 iterations was ideal for my data), which will ensure the model is well trained.

3.2.5 Viterbi Algorithm

This function is excessively large, so I will not be including it in this section. You can check it out on my github repo in the hmm.rs predict function.

```rust
// Compute gamma (state probabilities) and xi (transition probabilities)
fn compute_gamma_xi(
    &self,
    alpha: &Array2<f64>,
    beta: &Array2<f64>,
    observations: &[usize],
    scales: &[f64],
) -> (Array2<f64>, Array3<f64>) {
    let n: usize = observations.len();
    let mut gamma: ArrayBase<OwnedRepr<f64>, …> = Array2::zeros(shape: (n, self.num_states));
    let mut xi: ArrayBase<OwnedRepr<f64>, …> = Array3::zeros(shape: (n-1, self.num_states, self.num_states));

    // Compute gamma values
    for t: usize in 0..n {
        for i: usize in 0..self.num_states {
            gamma[[t, i]] = alpha[[t, i]] * beta[[t, i]];
        }
    }

    // Compute xi values
    for t: usize in 0..n-1 {
        let obs: usize = observations[t+1];
        for i: usize in 0..self.num_states {
            for j: usize in 0..self.num_states {
                xi[[t, i, j]] = alpha[[t, i]] *
                                self.transition_matrix[[i, j]] *
                                self.emission_matrix[[j, obs]] *
                                beta[[t+1, j]] /
                                scales[t+1];
            }
        }
    }

    (gamma, xi)
} fn compute_gamma_xi
```

Figure 4: State Probabilities Function

The Viterbi algorithm is a dynamic programming technique used to find the most probable sequence of hidden states in a Hidden Markov Model (HMM), given a sequence of observed events.

It operates in two main phases:

- Forward pass
- Backward tracing

In the forward pass, the algorithm iterates over time steps, computing the maximum probability of reaching each possible hidden state at each step. This is done by considering all possible transitions from previous states and selecting the most probable one. The probabilities are stored in a Viterbi matrix, while a backpointer matrix keeps track of the optimal state transitions.

After processing all observations, the algorithm identifies the most likely final state and then traces backward through the backpointer matrix to reconstruct the most probable sequence of hidden states.

4 Results and Analysis

After implementing the Hidden Markov Model (HMM) and running the Viterbi algorithm on historical market data, I analyzed the predicted market regimes to assess the effectiveness of the model. This section presents the results of the model's predictions, evaluates its accuracy, and discusses its implications.

4.1 Market Regime Predictions

Overall, the HMM model that I implemented had a somewhat successful attempt at predicting market regimes in the market regimes. I had initially set up 3 states, but because of the volume of data it smoothed out into 2 states.

Visually, the model was able to successfully identify key bearish parts of the market history. It was a bit inaccurate as to how long the bearish periods ran, but overall it was accurate.

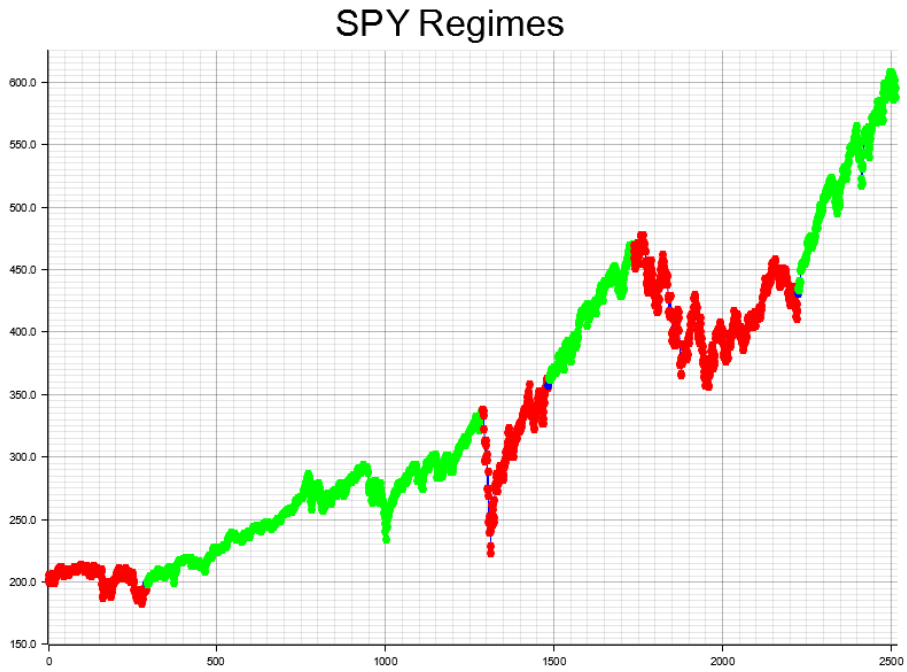

Figure 5: S&P Market Regimes from 2015-2025

4.2 Strengths and Limitations

Strengths:

- The model effectively detected long-term trends and smoothed out short-term noise in the data.
- The Viterbi algorithm efficiently computed the most probable market state sequence.

Limitations:

- Performance depends heavily on initial parameters, which may lead to suboptimal regime classification if not carefully chosen.
- The model does not incorporate external macroeconomic factors, which can influence regime shifts.

4.3 Interpretation of Results

The results suggest that Hidden Markov Models provide a useful framework for detecting financial market regimes. While not perfect, the model's ability to classify trends and reversals makes it a valuable tool for quantitative trading

strategies, portfolio risk management, and macroeconomic forecasting. Further improvements, such as adaptive transition probabilities or incorporating additional market indicators, could enhance accuracy and robustness.

5 Conclusion

In this project, I explored the application of Hidden Markov Models (HMMs) for market regime detection, implementing the Viterbi algorithm to classify financial market conditions into distinct states. My model effectively captured bullish and bearish trends based on historical market data, demonstrating the potential of HMMs as a quantitative tool for financial analysis.

Through this implementation, I found that HMMs can detect underlying market structures while filtering out short-term noise. The model successfully identified trend reversals and long-term patterns, reinforcing the value of probabilistic methods in financial modeling. However, I also encountered certain limitations, such as the assumption of constant transition probabilities and the model's sensitivity to initial parameter selection.

Overall, this project has given me valuable insight into how Markov-based approaches can provide meaningful interpretations of financial markets. It has also laid the foundation for future work in quantitative finance, risk management, and algorithmic trading.

References

[1] Paul A. Gagniuc. *Markov chains: from theory to implementation and experimentation*. eng. Hoboken: John Wiley & sons, 2017. ISBN: 9781119387572 9781119387596.

[2] Jasleen K. Grewal, Martin Krzywinski, and Naomi Altman. "Markov models — training and evaluation of hidden Markov models". en. In: *Nature Methods* 17.2 (Feb. 2020), pp. 121–122. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0702-6. URL: https://www.nature.com/articles/s41592-019-0702-6 (visited on 03/12/2025).

[3] Tech Insights. *Reinforcement learning — part 3*. en. Oct. 2022. URL: https://medium.com/@TechInsight/reinforcement-learning-part-3-d037fa27c5cb (visited on 03/12/2025).

[4] Mark Kritzman, Sébastien Page, and David Turkington. "Regime shifts: implications for dynamic strategies(Corrected)". en. In: *Financial Analysts Journal* 68.3 (May 2012), pp. 22–39. ISSN: 0015-198X, 1938-3312. DOI: 10.2469/faj.v68.n3.3. URL: https://www.tandfonline.com/doi/full/10.2469/faj.v68.n3.3 (visited on 03/12/2025).

[5] Everton Gomede PhD. *Markov chains and its applications*. en. Mar. 2024. URL: https://ai.plainenglish.io/markov-chains-b84ec329a54c (visited on 03/12/2025).

[6] L.R. Rabiner. "A tutorial on hidden Markov models and selected applications in speech recognition". In: *Proceedings of the IEEE* 77.2 (Feb. 1989), pp. 257–286. ISSN: 00189219. DOI: 10.1109/5.18626. URL: http://ieeexplore.ieee.org/document/18626/ (visited on 03/12/2025).

[7] Walter Zucchini, Iain L. MacDonald, and Roland Langrock. *Hidden Markov models for time series: an introduction using R*. eng. Second edition. Monographs on statistics and applied probability 150. Boca Raton London New York: CRC Press, Taylor & Franci Group, 2016. ISBN: 9781482253849 9781322626499.