The Foundation of Probabilistic Sequence Modeling: HMM Viterbi Algorithm

Decoding the Unseen Sequence

AI KUN, CHENGSHENG, DUANXU, ISA WONG, TIANQI

October 14, 2025

Outline

Sequence Data and Latent Variable Models

Takeaway

Core Idea: The data sequence we directly observe is generated by an unobservable, latent (or ”hidden”) stochastic process.

An intuitive analogy: A doctor infers the most likely underlying illness (hidden state) by observing a patient’s symptoms over several days (observable events).

- **Hidden State**: The patient’s true health condition (e.g., ”Healthy” or ”Fever”).
- **Observable Event**: The patient’s reported symptoms (e.g., ”feels normal,” ”chills,” or ”dizzy”).

Formal Definition of a Hidden Markov Model (HMM)

Definition

An HMM can be formally defined by a five-tuple $\lambda = (S, O, \pi, A, B)$:

- S: A finite set of **hidden states** $\{s_1, ..., s_K\}$.
- O: A finite set of **observation symbols** $\{o_1, ..., o_M\}$.
- π: The **initial state probability** distribution, representing the probability of being in each state at $t = 1$.
- A: The **state transition probability** matrix, representing the probability of transitioning from one state to another.
- B: The **observation (emission) probability** matrix, representing the probability of generating a specific observation from a given state.

The Generative Process of an HMM

Takeaway

An HMM is a generative model that provides a clear set of probabilistic rules to generate an observation sequence:

1. Select a starting state according to the initial state distribution π.
2. Generate an observation from that state according to the emission probability B.
3. Transition to a new state according to the transition probability A.
4. Repeat steps 2 and 3 until the full sequence is generated.

This property allows HMMs to be used not only for analysis tasks (like decoding) but also for synthesis tasks (like speech synthesis).

The Two Fundamental Assumptions of HMMs

Definition

The computational tractability of HMMs is based on two key simplifying assumptions:

- **Markov Assumption (First-Order)**:
  The probability of the current state **depends only on the previous state**, and is independent of all earlier states.

$$P(q_t|q_{t-1}, q_{t-2}, ..., q_1) = P(q_t|q_{t-1})$$

- **Output Independence Assumption**:
  The probability of the current observation **depends only on the current hidden state**, and is independent of all other states and observations.

$$P(o_t|q_t, q_{t-1}, ..., o_{t-1}, ...) = P(o_t|q_t)$$

Outline

The Decoding Problem

Within the framework of HMMs, there are three fundamental problems to solve: Evaluation, Learning, and Decoding. The Viterbi algorithm is specifically designed to solve the decoding problem.

Definition

The Decoding Problem
Given a sequence of observations $O = (o_1, o_2, ..., o_T)$ and a known HMM model λ, the goal is to find the single most likely sequence of hidden states $Q^* = (q_1^*, q_2^*, ..., q_T^*)$ that produced this observation sequence.

$$Q^* = \arg\max_Q P(Q|O, \lambda)$$

The Inefficiency of a Brute-Force Search

The most intuitive way to solve the decoding problem is through a brute-force search. This would involve:

- Listing every possible sequence of hidden states.
- Calculating the probability of each sequence generating the given observations.
- Selecting the sequence with the highest probability.

However, this approach is computationally infeasible for any non-trivial problem.

Takeaway

For a model with K states and an observation sequence of length T, there are K^T possible state paths. This number grows exponentially, making a brute-force search impossible in practice. For instance, a simple model with just 2 states and a sequence of 1000 observations would have approximately 10^{301} possible paths.

The Elegance of Dynamic Programming

In 1967, Andrew Viterbi provided an elegant and efficient solution to this problem by applying the principle of **dynamic programming**.

The core idea of dynamic programming is to solve a complex problem by breaking it down into a collection of simpler, overlapping subproblems. The solutions to these subproblems are stored and reused, avoiding redundant calculations.
The Viterbi algorithm builds the final solution from the bottom up, step by step, ensuring that at each step, it only needs to consider the optimal solutions from the previous step.

Visualizing the Problem: The Trellis Diagram

The execution of the Viterbi algorithm can be visualized using a structure known as a **trellis diagram**.

- The horizontal axis represents the time steps, from $t = 1$ to T.
- The vertical axis represents the K possible hidden states.
- Each node (t, j) in the diagram corresponds to being in state s_j at time t.
- Directed edges connect nodes from time $t - 1$ to time t, representing possible state transitions.

In this view, the decoding problem is transformed into finding the single ”best” path through this trellis from the start to the end.

Formalizing the Algorithm: Key Variables

The Viterbi algorithm iteratively populates two matrices to keep track of the optimal path information at each time step. These are typically denoted as:

Definition

- **The Viterbi Probability Matrix (V or δ):**
  $V_t(j)$ stores the probability of the *most likely* state sequence that ends in state s_j at time t, having generated the first t observations.

- **The Backpointer Matrix (ψ or 'ptr'):**
  $\psi_t(j)$ stores the state at time $t-1$ that led to the most likely path ending in state s_j at time t. This allows us to reconstruct the path later.

Step 1: Initialization

The algorithm begins at the first time step ($t = 1$). For each possible hidden state, we calculate the probability of the system starting in that state and emitting the first observation, o_1.

This step establishes the base case for the dynamic programming recursion.

Theorem

For each state $j = 1, \ldots, K$:

- **Probability Calculation:**

$$V_1(j) = \pi_j \cdot B_j(o_1)$$

- **Backpointer Initialization:**

$$\psi_1(j) = 0 \quad \text{(or null)}$$

Step 2: Recursion

The algorithm then proceeds iteratively from time $t = 2$ to T. For each state s_j at each time step t, it calculates the maximum probability of any path that ends at this node.

This is done by considering all possible paths from the previous time step, $t - 1$. For each state s_j, we look at all K states at time $t - 1$ and find the one that provides the most probable path to s_j. This is the core of the algorithm, where the ”Principle of Optimality” is applied: an optimal path must be composed of optimal sub-paths.

Step 2: Recursion Formulas

Theorem

For each time step $t = 2, ..., T$ and for each state $j = 1, ..., K$:

- **Probability Calculation:**

$$V_t(j) = \max_{i=1}^{K}[V_{t-1}(i) \cdot A_{ij}] \cdot B_j(o_t)$$

- **Backpointer Storage:**

$$\psi_t(j) = \arg\max_{i=1}^{K}[V_{t-1}(i) \cdot A_{ij}]$$

The 'argmax' function stores the index of the previous state (i) that maximized the probability, which is our backpointer.

Step 3: Termination

After the recursion step has been completed for all time steps up to T, the forward pass of the algorithm is complete.

The probability of the single most likely path for the entire observation sequence is simply the maximum value in the final column of the Viterbi matrix, V_T. The final state of this optimal path is the state that corresponds to this maximum probability.

Theorem

- **Overall Path Probability:**

$$P^* = \max_{j=1}^{K} V_T(j)$$

- **Final State of Optimal Path:**

$$q_T^* = \arg\max_{j=1}^{K} V_T(j)$$

Step 4: Path Backtracking

While the forward pass gives us the probability of the best path and its final state, it does not tell us the full sequence of states.

To find the complete path, we use the backpointer matrix, ψ. Starting from the final state q_T^*, we trace our steps backward through the trellis, following the pointers stored at each step.

Theorem

For each time step $t = T - 1$ down to 1:

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

This process reconstructs the most likely sequence of hidden states from end to beginning, revealing the complete Viterbi path.

Outline

Example: The Weather and Activities Model

To make the algorithm concrete, we will walk through a classic example that infers the weather (the hidden state) based on a person's activities (the observations).

Definition

Model Components

- **Hidden States (S)**: {Rainy, Sunny}
- **Observations (O)**: {Walk, Shop, Clean}
- **Sequence to Decode**: (Walk, Shop, Clean)

HMM Parameters for the Weather Model

The model is defined by the following probability matrices:
1. Initial Probabilities (π): $P(\text{Rainy}) = 0.6$, $P(\text{Sunny}) = 0.4$
2. Transition Matrix (A) This is a 2x2 matrix representing the probability of transitioning from one weather state to another. Rows represent the previous day's weather (From), and columns represent the current day's weather (To).

$$A = \begin{pmatrix} P(\text{Rainy}|\text{Rainy}) & P(\text{Sunny}|\text{Rainy}) \\ P(\text{Rainy}|\text{Sunny}) & P(\text{Sunny}|\text{Sunny}) \end{pmatrix} = \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix}$$

3. Emission Matrix (B) This is a 2x3 matrix representing the probability of performing a specific activity given a certain weather state. Rows represent the weather state (State), and columns represent the observed activity (Observation).

$$B = \begin{pmatrix} P(\text{Walk}|\text{Rainy}) & P(\text{Shop}|\text{Rainy}) & P(\text{Clean}|\text{Rainy}) \\ P(\text{Walk}|\text{Sunny}) & P(\text{Shop}|\text{Sunny}) & P(\text{Clean}|\text{Sunny}) \end{pmatrix} = \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.6 & 0.3 & 0.1 \end{pmatrix}$$

Calculation: Initialization (t=1)

We begin with the first observation, 'Walk'. We calculate the initial Viterbi probabilities for each state.

For state 'Rainy':

- $V_1(\text{Rainy}) = \pi_{\text{Rainy}} \times B_{\text{Rainy}}(\text{Walk})$
- $V_1(\text{Rainy}) = 0.6 \times 0.1 = 0.06$

For state 'Sunny':

- $V_1(\text{Sunny}) = \pi_{\text{Sunny}} \times B_{\text{Sunny}}(\text{Walk})$
- $V_1(\text{Sunny}) = 0.4 \times 0.6 = 0.24$

Takeaway

At t=1, the most likely starting state is 'Sunny' because its probability (0.24) is higher than 'Rainy' (0.06).

Calculation: Recursion (t=2)

Next, we process the second observation, 'Shop'. For each current state, we find the most probable path leading to it from all possible previous states.

For state 'Rainy' at t=2:

- Path from 'Rainy' (t=1): $V_1(\text{Rainy}) \times A_{\text{Rainy}\to\text{Rainy}} = 0.06 \times 0.7 = 0.042$
- Path from 'Sunny' (t=1): $V_1(\text{Sunny}) \times A_{\text{Sunny}\to\text{Rainy}} = 0.24 \times 0.4 = 0.096$

Theorem

We take the maximum of these path probabilities and multiply by the emission probability:

- $V_2(\text{Rainy}) = \max(0.042, 0.096) \times B_{\text{Rainy}}(\text{Shop})$
- $V_2(\text{Rainy}) = 0.096 \times 0.4 = 0.0384$
- The backpointer $\psi_2(\text{Rainy})$ is set to 'Sunny', as it provided the max probability.

Calculation: Termination and Backtracking

After filling the matrices for all time steps, we find the final state and trace back the path.

Termination (t=3):

- We compare the final probabilities: $V_3(\text{Rainy}) = 0.01344$ and $V_3(\text{Sunny}) = 0.002592$.
- The maximum probability is 0.01344, so the final state of our path is $q_3^* = \text{Rainy}$.

Backtracking:

- At t=3, our state is 'Rainy'. The backpointer is $\psi_3(\text{Rainy}) = \text{Rainy}$. So, $q_2^* = \text{Rainy}$.
- At t=2, our state is 'Rainy'. The backpointer is $\psi_2(\text{Rainy}) = \text{Sunny}$. So, $q_1^* = \text{Sunny}$.

Final Result of the Example

By following the backpointers from the end to the beginning, we reconstruct the most likely sequence of hidden states.

Takeaway

Final Decoding Result
For the observation sequence (Walk, Shop, Clean), the most likely sequence of hidden states is: **(Sunny, Rainy, Rainy)** with a total probability of 0.01344.

Computational Complexity

The efficiency of the Viterbi algorithm is one of its most important features, making it practical for real-world applications.

Definition

Let K be the number of hidden states and T be the length of the observation sequence.

- **Time Complexity:** $O(K^2 T)$
  This arises because for each of the T time steps, and for each of the K states, we must iterate through all K previous states to find the maximum probability.
- **Space Complexity:** $O(KT)$
  This is required to store the Viterbi probability matrix and the backpointer matrix, both of which have dimensions $K \times T$.

Implementation: Numerical Stability

A major practical challenge when implementing the Viterbi algorithm is numerical underflow.

Definition

Problem: Arithmetic Underflow
When multiplying a long sequence of small probabilities (numbers between 0 and 1), the final product can become smaller than the minimum positive number that a computer can represent. This causes the value to be rounded to zero, making the algorithm fail.

Implementation: The Log-Space Solution

The standard solution to underflow is to perform calculations in logarithmic space.

This simple mathematical trick transforms the algorithm's operations, ensuring stability.

- Multiplication becomes addition: $\log(a \cdot b) = \log(a) + \log(b)$
- The 'max' operation remains a 'max' operation on the sum of logs.

Takeaway

The Log-Domain Recursive Formula
The core recursion is transformed into a sum of logarithms, which is numerically stable and is standard practice in all real-world HMM implementations.

$$\log(V_t(j)) = \max_{i=1}^{K}[\log(V_{t-1}(i)) + \log(A_{ij})] + \log(B_j(o_t))$$

Outline

Application: Part-of-Speech (POS) Tagging

Part-of-Speech (POS) tagging is a classic NLP task that serves as a perfect example of the Viterbi algorithm in action.

Definition

HMM for POS Tagging
We frame the problem by defining words as observations and tags as the hidden states we want to uncover.

- **Hidden States**: The set of possible POS tags (e.g., 'NN' for noun, 'VB' for verb).
- **Observations**: The sequence of words in the sentence.

Decoding a Sentence

After learning transition and emission probabilities from a training corpus, the Viterbi algorithm decodes the most likely tag sequence for a new sentence.

Takeaway

Example of Decoding
For the sentence: “Janet will back the bill”
The Viterbi algorithm computes the most probable tag sequence as:
(NNP, MD, VB, DT, NN)
(Proper Noun, Modal, Verb, Determiner, Noun)

Key Limitations of the HMM Tagger

The simplicity of the HMM leads to significant challenges when applied to complex data like natural language.

Takeaway

Core Issues

- **Strict Independence Assumptions**: The model cannot capture long-range dependencies. For example, in grammar, a word's tag can be influenced by words far earlier in the sentence, which the Markov assumption ignores.
- **Data Sparsity (OOV Problem)**: If a word was not in the training data, its emission probability is zero. This causes the Viterbi algorithm to fail, as any path containing the unknown word will have a probability of zero.

Modern Perspectives and Advancements

The limitations of HMMs directly spurred the development of more powerful sequential models.

Models Beyond HMM:

- **Conditional Random Fields (CRFs)**: Relax the strict independence assumptions, allowing for a richer set of features from the entire sentence.
- **Recurrent Neural Networks (RNNs) & Transformers**: Capture much more complex and long-range dependencies, leading to state-of-the-art performance.

The Enduring Idea of Viterbi

While the HMM itself has been largely superseded for tasks like POS tagging, the core algorithm remains highly relevant.

Takeaway

The Viterbi Legacy
The fundamental idea—**using dynamic programming to find an optimal path through a trellis**—is a cornerstone of sequence modeling. It continues to be used as the final decoding step in modern, powerful architectures like the BiLSTM-CRF model.