Applications of the Viterbi Algorithm and Comparisons with Alternative Sequence Inference Methods

Tudor Gulin

Abstract

This project explores the Viterbi algorithm as a general method for decoding the most probable hidden state sequence in Hidden Markov Models (HMMs). A generic implementation of the Viterbi algorithm is presented and applied to three distinct domains: weather prediction, text typo correction, and part-of-speech (POS) tagging. In each case, the hidden states, observations, and probabilistic model parameters are explicitly defined. The Viterbi algorithm is further compared with alternative approaches, including greedy decoding, forward–backward inference, CRF-style max-sum inference, and classical string similarity methods.

1 Introduction

Many real-world problems involve reasoning over sequential data where the underlying system states are not directly observable. Hidden Markov Models (HMMs) provide a principled probabilistic framework for such problems by modeling a sequence of hidden states that emit observable outputs. Given a sequence of observations, one of the central tasks is *decoding*: finding the most likely sequence of hidden states that explains the observations.

The Viterbi algorithm is a dynamic programming method that efficiently solves this decoding problem. In this project, a generic implementation of the Viterbi algorithm is developed and applied to multiple domains in order to demonstrate its generality and limitations.

2 Hidden Markov Models

An HMM is defined by the tuple (E, F, π, P, Q), where:

- E is a finite set of hidden states,
- F is a finite set of observations,
- $\pi(s) = P(x_0 = s)$ is the initial state distribution,
- $P(s', s) = P(x_t = s \mid x_{t-1} = s')$ is the transition probability matrix,
- $Q(s, o) = P(y_t = o \mid x_t = s)$ is the emission probability matrix.

Given an observation sequence $y_{0:T-1}$, the decoding problem consists of finding:

$$x^*_{0:T-1} = \arg\max_{x_{0:T-1}} P(x_{0:T-1} \mid y_{0:T-1}).$$

3 The Viterbi Algorithm

The Viterbi algorithm solves the decoding problem using dynamic programming. It introduces the quantity:

$$\delta_t(s) = \max_{x_{0:t-1}} P(x_{0:t-1}, x_t = s, y_{0:t}),$$

which represents the probability of the most likely partial path ending in state s at time t.

3.1 Initialization

For $t = 0$, the algorithm initializes:

$$\delta_0(s) = \pi(s) \cdot Q(s, y_0).$$

3.2 Recursion

For $t = 1, \ldots, T-1$, the recurrence relation is:

$$\delta_t(s) = \max_{s' \in E} \left[\delta_{t-1}(s') \cdot P(s', s) \cdot Q(s, y_t) \right].$$

At each step, a backpointer is stored to record which previous state maximized the expression.

3.3 Termination and Backtracking

The final state is chosen as:

$$x^*_{T-1} = \arg\max_{s \in E} \delta_{T-1}(s).$$

The optimal path $x^*_{0:T-1}$ is then recovered by following the stored backpointers backward in time.

3.4 Complexity

The time complexity of the Viterbi algorithm is $\mathcal{O}(|E|^2 \cdot T)$, which is significantly more efficient than enumerating all possible state sequences.

4 Generic Implementation

In this project, the Viterbi algorithm is implemented in a generic form that operates solely on:

- a sequence of observations encoded as indices,
- an initial probability vector π,
- a transition matrix P,
- an emission matrix Q.

This design allows the same implementation to be reused across different application domains, with only the model parameters and interpretation of states and observations changing.

5 Application 1: Weather Prediction

5.1 Model Definition

In the weather prediction task, two scenarios are modeled to test algorithm robustness. In the **Standard Scenario**, the hidden states $E = \{\text{Hot, Cold}\}$ represent the latent atmospheric condition. In the **Complex Scenario**, the state space is expanded to $E = \{\text{Very Hot, Hot, Cold, Very Cold}\}$ to test transitions across a gradient of temperatures.

The observations correspond to ice cream consumption levels:

$$F = \{1 \text{ ice cream}, 2 \text{ ice creams}, 3 \text{ ice creams}\}.$$

5.2 Objective

The objective is to infer the weather sequence from observed consumption habits. The transition matrix P is structured to favor state persistence (e.g., a 70% chance to stay Hot), while the emission matrix Q correlates higher consumption with higher temperatures. Since the data is generated synthetically using a random choice based on these probabilities, the Viterbi output is evaluated by its ability to reconstruct the known ground-truth sequence.

6 Application 2: Typo Correction

6.1 Model Definition

This application uses a character-level Hidden Markov Model where the hidden states E are the intended characters and observations F are the characters actually typed.

$$E = F = \{\texttt{a}, \ldots, \texttt{z}, \texttt{space}\}.$$

6.2 Transition and Emission Models

The transition model acts as a bigram language model, capturing the probability of one character following another. This allows the algorithm to prefer character sequences that form valid phonetic or structural patterns.

The emission model is specifically designed around the **QWERTY layout**. It assigns a 70% probability to the correct key, a 20% shared probability among immediate physical neighbors (the "fat-finger" effect), and a 10% background probability for any other key. This ensures that even unlikely typos do not result in a zero-probability path, which would break the Viterbi recursion.

6.3 Interpretation of Viterbi Output

Unlike dictionary-based methods, the Viterbi algorithm uses **global context**. If a user types a character that could be corrected in multiple ways, the algorithm chooses the one that best fits the surrounding characters based on the language model. This makes it a "noisy-channel" approach where words emerge naturally from the sequence of corrected characters.

7 Application 3: Part-of-Speech Tagging

7.1 Model Definition and Training

For POS tagging, the hidden states correspond to the universal grammatical categories, while the observations correspond to the discrete vocabulary found in the corpus.

$$E = \{\text{NOUN}, \text{VERB}, \text{ADJ}, \text{PRON}, \text{DET}, \ldots\}$$

$$F = \{\text{words in the training vocabulary}\}$$

The model is trained on the **Brown Corpus** using a 75/25 train-test split. During training, the transition matrix P is populated by counting tag-bigram frequencies (e.g., how often a NOUN follows a DET), and the emission matrix Q is built from word-tag co-occurrence frequencies.

To handle the **Out-of-Vocabulary (OOV)** problem - where a word appears in the test set but not in the training vocabulary - the implementation introduces an explicit `<UNK>` token in the observation vocabulary. Any unseen word at test time is mapped to `<UNK>`. Emission probabilities are estimated with Laplace smoothing, so `<UNK>` receives a small but nonzero emission probability under each tag. This avoids zero-probability paths and allows decoding to remain well-defined.

7.2 Benchmarking and Evaluation

The HMM-based Viterbi implementation is benchmarked against **spaCy**, a modern industrial-strength NLP library. While spaCy uses deep neural networks (specifically the `en_core_web_sm` model) to predict tags, the HMM relies purely on statistical frequencies. The comparison highlights the difference between global sequence optimization (Viterbi) and the high-dimensional feature extraction used by neural taggers.

8 Comparison with Other Methods

To evaluate the Viterbi algorithm, four alternative inference and similarity methods were implemented:

8.1 Greedy Decoding

A baseline method that ignores the transition matrix P and simply selects the state s that maximizes $Q(s, y_t)$ at each step. This demonstrates the necessity of modeling temporal dependencies, as greedy decoding often fails in ambiguous contexts.

8.2 Forward–Backward (MPM)

The Forward-Backward algorithm calculates the **marginal posterior** probability for each state. The flow of the algorithm is bidirectional:

1. **Forward Pass (α):** Calculates the probability of observing the sequence up to time t and ending in state s.
2. **Backward Pass (β):** Calculates the probability of observing the future sequence from $t+1$ to T given state s at time t.
3. **Marginalization:** The probability of being in state s at time t is computed as $\gamma_t(s) \propto \alpha_t(s) \cdot \beta_t(s)$.

The final path is constructed by independently selecting $\arg\max \gamma_t(s)$ at each step. While this maximizes the expected number of correct individual states, it does not guarantee that the resulting sequence of transitions is valid.

8.3 CRF-Style Inference (Max-Sum)

This method utilizes a **log-space** implementation of the Viterbi recursion. Instead of multiplying probabilities (which risks numerical underflow in long sequences), this algorithm sums log-probabilities:

$$\text{Score}_t(s) = \max_{s'} \left[\text{Score}_{t-1}(s') + \log P(s \mid s') + \log Q(s, y_t) \right].$$

By treating the probabilities as additive weights, the algorithm finds the global maximum path similarly to inference in Conditional Random Fields (CRFs), ensuring stability for long text sequences.

8.4 String Similarity: Levenshtein and Jaro–Winkler

For the typo correction task, Viterbi is compared against classical string metrics which do not use context:

- **Weighted Levenshtein Distance:** This algorithm builds a dynamic programming matrix of size $(|y| + 1) \times (|w| + 1)$ to find the cheapest edit path. The flow involves iteratively filling cells by choosing the minimum cost among Insertion, Deletion, and Substitution. In this implementation, the **Substitution cost is weighted**: swapping physically adjacent keys on the keyboard costs 0.5, while other swaps cost 1.0.
- **Jaro–Winkler Similarity:** This algorithm calculates a similarity score between 0 and 1 using a multi-step flow:
    1. **Matching Window:** It counts matching characters that are within a specific distance (sliding window) of each other.
    2. **Transpositions:** It counts how many matching characters are in the wrong order.
    3. **Prefix Boost:** Finally, it applies a "Winkler" scaling factor that increases the score if the two strings share a common prefix (up to 4 characters), making it highly effective for catching typos at the end of words.

9 Conclusion

This project demonstrates that the Viterbi algorithm is a versatile and reusable method for sequence decoding across diverse domains. While modern neural models often outperform HMM-based approaches, Viterbi remains a valuable baseline due to its interpretability, determinism, and clear probabilistic foundations.