CS726: Programming Assignment 1

August 22, 2025

General Instructions

1. Code plagiarism will be strictly penalized including but not limited to reporting to DADAC and zero in assignments.
2. If you use tools such as ChatGPT, Copilot, you must explicitly acknowledge their usage in your report. Code borrowed from other sources must be cited in the comments.
3. Submit a report explaining your approach, implementation details, and results. Clearly mention the contributions of each team member in the report.
4. Edit the `template.py`, `loopy.py`, `turbocodes.py` files given and use Python's default libraries such as `json`, `math`, `itertools`, `collections`, `functools`, `random`, `heapq`.
5. If you use external sources (e.g., tutorials, papers, or open-source code), you must cite them properly in your report and comments in the code. Submit your code and report as a compressed `<TeamName>_<student1rollno>_<student2rollno>_<student3rollno>.zip` file. Fill a student as `NOPE` if less than 3 members.
6. Start well ahead of the deadline. Submissions up to one day late will be capped at 80% of the total marks, and no marks will be awarded beyond that.

1 Problem Statement

1.1 Inference on Factorial Hidden Markov Models (FHMMs)

A **Factorial Hidden Markov Model (FHMM)** is a probabilistic model used to describe systems composed of multiple independent processes evolving over time, where the combined state of these processes jointly influences a single sequence of observations.

A standard Hidden Markov Model (HMM) consists of a single chain of hidden states influencing observations. An FHMM extends this by introducing multiple, parallel hidden chains:

- **Parallel State Chains (Factors):** An FHMM has F independent hidden state sequences. Each chain i has its own state S_t^i at time t and transition probabilities $P(S_t^i \mid S_{t-1}^i)$.
- **Single Observation Sequence:** At each time step t, there is one observation O_t for the entire system.
- **Joint Influence on Observations:** The observation O_t depends on the *joint state* of all hidden chains at time t. This is represented by a joint emission potential:

$$\phi(O_t, S_t^0, S_t^1, \ldots, S_t^{F-1}).$$

This structure allows FHMMs to model complex systems where multiple hidden causes contribute to a single observable effect. The provided data specifies such a model. For each Conditional Probability Table, the table for a factor is of size K^2, where K is the number of states.

Each set of K consecutive entries is to be normalized (this is done in the skeleton code already) and to be interpreted as the conditional probabilities of values $1 \cdots K$ of the next state given the previous state. For instance, the first entry (after normalizing the row) is the probability of the next state being 0 given a 0, and the second entry is the probability of the next state being 1 given a 0, and so on.
For the state-factor clique, a similar normalization is carried out by the template. If the index is represented as $a_1 a_2 .. a_M a_{obs}$ in base K, then the value of the observed variable is a_{obs} and the value of factor 0 is a_1, and so on.

Task Description

You are required to complete the `Inference` class in the provided `template.py` file. Implement exact inference for FHMMs by filling in the following methods:

1. `get_z_value()` – Compute the normalization constant Z, defined as the sum over all possible assignments of hidden states of the product of potential functions. Note that, under an assumption that all states of all factors are equiprobable in the first epoch, this will simply be K^M times the probability of seeing that particular observation sequence, where K is the number of states per factor and M is the number of factors. The latter observation provides methods to compute the required value leveraging only the properties of FHMMs and without having to build a clique tree. **Return:** A single floating-point number representing Z.

2. `compute_marginals()` – Marginal Probabilities For each hidden variable S_t^i, compute its probability distribution $P(S_t^i)$. **Return:** A list of size $N \times S$, where N = Factors_Count × Number of Observations and S = State_Count. Entry `output[i][j]` gives the probability that variable i is in state j.

3. `compute_top_k()` – Top-K Assignments Find the k most probable complete assignments of all hidden variables. **Return:** A list of k dictionaries, each containing:

   - `"assignment"`: a list of integers representing the state of every variable.
   - `"probability"`: the joint probability of that assignment.

   The list must be sorted in descending order of probability. We guarantee there will be no ties between the k$^{\text{th}}$ and k+1$^{\text{th}}$ most probable assignments.

To solve these queries, one can leverage the special structure of FHMMs. This is the idea used in Ghahramani and Jordan 1997, which gives explicit algorithms for exact inference in FHMMs with a much better time complexity compared to $O(TK^{2M})$ naive inference.
Alternatively, one can try to abstract the FHMM as a standard HMM with TM states, and try to use optimize standard HMM algorithms on the same. Massachusetts Institute of Technology 2014 gives some nice ideas about how to implement forward and backward passes for the sum-product algorithm efficiently. For the other queries, do refer to Koller and Friedman 2009.
The expected time complexity is $O(TMK^{M+1})$, where M is the number of factors, K is the number of states each factor can take, T is the number of observations.

1.2 Loopy Belief Propagation

As explained in Das, Liu, and Gupta 2014, Loopy Belief Propagation allows one to pass messages on a loopy graph rather than having to convert it to a tree, which might blow up the size of the largest CPD. Theoretically, it is most easy to implement the same for composition functions which are invertible (such as the product function for positive distributions). The paper Murphy, Weiss, and Jordan 2013 gives quite a few useful suggestions for implementing LBP.
For this assignment, you will implement Loopy Belief Propagation for the factorial HMMs introduced in the previous subproblem. It suffices to compute the marginal probabilities of each variable in the factorial HMM, given the observed sequence. Note that, due to numerical instability, post-normalizing, the probabilities might differ from the true probabilities slightly. Errors in the fourth decimal place are completely acceptable, and the third decimal place too might have rare errors. LBP is, after all, an approximate inference algorithm.
In the report, argue about the time complexity of the algorithm, assuming $O(1)$ iterations for convergence.

1.3 Turbo Codes

1.3.1 Background

An early application of Loopy Belief Propagation was in Turbo Codes. In the current assignment, we simulate a simplified discrete version of noise, with the probability of observing level i given that the bit is actually 0 being given by the i^{th} entry of the first row of the given probability matrix. This also equals the probability of being in the i^{th} highest level, given that the actual bit is a 1.
Turbo Codes aim to spread out errors as much as possible by interleaving the sent code and its RSC parity bits with the RSC parity bits of a random permutation of the data. This permutation is known to the decoder. Thus, even if a burst of errors occur (by random chance), the permutation will ensure that the errors are spread out in the permuted data, and thus can be corrected.
For this assignment, the following decoding algorithms are implemented:

1. **Viterbi Decoding of a convolutional code**: A convolutional code can be visualized as creating a trellis (see MIT OpenCourseWare 2012 for an example using simple convolutional codes. The algorithm is much the same for RSC). The Viterbi algorithm returns the most probable "path" or string of bits to generate a particular noisy output (with the output being the interleaved data + RSC code). **It suffices to use the interleaved true data and one of the two parity codes for this subpart**. Using these two sequences, create a trellis as explained in the MIT OCW slides, and return the most probable path using the Viterbi algorithm. On a string of size 1000, the bit error rate can get quite high, reaching around 10%, as the most probable sequence overall might not be locally optimal everywhere, and since only one parity code is used, the algorithm has limited information to work with.

2. **BCJL Decoding** BCJL Decoding, explained in Bahl et al. 1974, finds for each bit in the original data the probability that it is a 0 or a 1, given the noisy data and the noisy parity bits from one encoder. In practice, later using these *a posteriori probabilities* to find whether 0 or 1 is more likely for each bit, is just as good as Viterbi, and often better. In the context of UGMs, it can be imagined as analogous to the Forward-Backward Algorithm. The main advantage of BCJL is that, unlike Viterbi, it generates a guess of probabilities for each bit, rather than just a decision regarding the most probable bit. These guesses can be fine-tuned

iteratively to significantly improve performance. This too is restrained to using only one parity code. The Bit Error Rate is roughly the same as Viterbi's.

3. **Turbo Codes** Turbo Codes use two RSC encoders and two decoders, essentially implementing Loopy Belief Propagation. Each decoder uses the noisy data, one of the two parity codes and the previous iteration's guesses of the other decoder to generate new guesses by running the BCLJ algorithm. As with LBP, the initial guesses can be assumed to be random. On bitstrings of length 1000, the algorithm almost always never makes a mistake.

1.3.2 Details for Report

The final report must include detailed steps, explanations, and structured pseudocode, formatted in LaTeX for clarity and precision.

1. Generating the trellis: Explain how the trellis is generated for later usage by the Viterbi and BCJL algorithms

2. Viterbi Algorithm: Explain how your code implements an algorithm to find the most probable path in the trellis. Also, explain the time complexity and any optimizations you made.

3. Turbo Codes/BCJL Algorithm: Explain how *a posteriori probabilities* are calculated for each bit, and how the algorithm iteratively refines these probabilities. Discuss the convergence criteria and any stopping conditions you implemented. Talk about any time or space complexity optimizations you made.

4. Compare the results. By creating larger testcases or more testcases, try to estimate the Bit Error Rate (BER) of Turbo Codes as well. Also argue about which of Viterbi or BCJL (one iteration) might be better in practice. What if the *a posteriori probabilities* after the final iteration are fed to the Viterbi algorithm instead of simply taking the most probable value of each bit greedily? Does such a combination of algorithms perform better in practice, or roughly the same (You might want to use smaller testcases to check this)?

1.4 Comments on Testcases

For Factorial HMMs, the number of states one factor can take will never exceed 10, so as to allow for a state to be represented easily as a string with the number of states as the base. Testcases can be designed to test if the implementation is robust to a few 0s in the potential tables.
For Loopy Belief Propagation, it is guaranteed that all distributions will be positive.

References

Bahl, L. et al. (1974). "Optimal decoding of linear codes for minimizing symbol error rate". In: *IEEE Transactions on Information Theory* 20.2. IEEE article (arnumber 1055186), pp. 284–287. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1055186&tag=1 (visited on 08/21/2025).

Das, Rajarshi, Zhengzhong Liu, and Dishan Gupta (2014). *Lecture 13: Variational Inference – Loopy Belief Propagation*. Scribe notes for 10-708: Probabilistic Graphical Models, Spring 2014, Carnegie Mellon University. URL: https://www.cs.cmu.edu/~epxing/Class/10708-14/scribe_notes/scribe_note_lecture13.pdf.

Ghahramani, Zoubin and Michael I. Jordan (1997). *Factorial Hidden Markov Models*. Tech. rep. AIM-1561. MIT Artificial Intelligence Laboratory. URL: https://mlg.eng.cam.ac.uk/zoubin/papers/fhmmML.pdf.

Koller, Daphne and Nir Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press. URL: https://books.google.co.in/books?id=7dzpHCHzNQ4C.

Massachusetts Institute of Technology (2014). *Lecture 8: Inferences on trees: Sum-product algorithm*. 6.438 Algorithms for Inference, Fall 2014. URL: https://ocw.mit.edu/courses/6-438-algorithms-for-inference-fall-2014/1faaccb44f78c4f4e99c6814842082a0_MIT6_438F14_Lec8.pdf.

MIT OpenCourseWare (Fall 2012). *6.02 Lecture 7: Viterbi Decoding*. Lecture slides — Viterbi decoding. URL: https://ocw.mit.edu/courses/6-02-introduction-to-eecs-ii-digital-communication-systems-fall-2012/f398fa4a366439301b3d17e45e028952_MIT6_02F12_lec07.pdf.

Murphy, Kevin P., Yair Weiss, and Michael I. Jordan (2013). "Loopy belief propagation for approximate inference: An empirical study". In: *arXiv preprint arXiv:1301.6725*. URL: https://arxiv.org/pdf/1301.6725.