ASEN 6519 - Assignment 1

Carl Mueller

February 26, 2019

Problem 1

State Trace of Short HMM Dataset

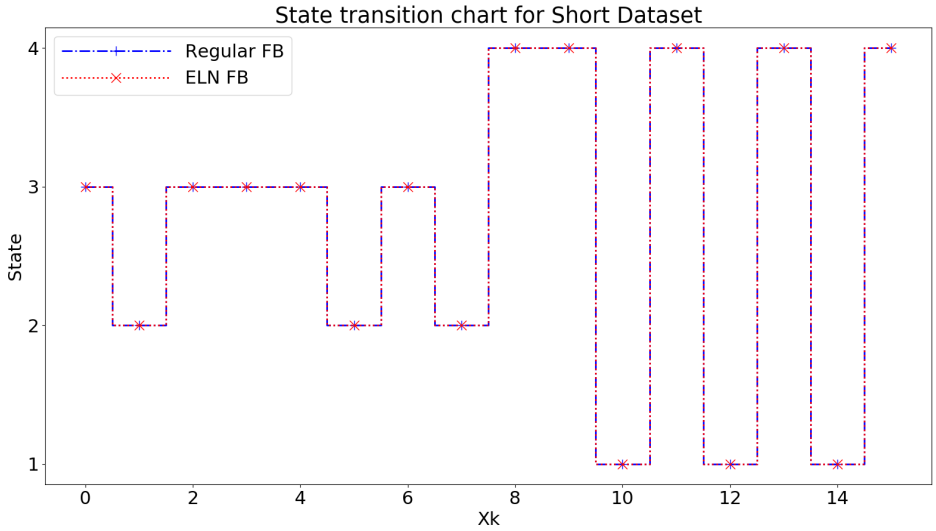

Figure 1: For the short dataset, the state trace is in lockstep for both the normal forward backwards and the logarithmic-based forward backward algorithm.

Posterior Probabilities

States	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X11	X11	X12	X13	X14	X15
1	0	0	0	0	0	0	0	0	0.183297	0.0100234	0.976834	0.00115028	0.930981	0.00233115	0.950946	0.0548677
2	0	0.951268	0.0122975	0.153837	0.105652	0.89824	0.00223589	0.941818	0	0	0	0	0	0	0	0
3	1	0.0487323	0.987703	0.846163	0.894348	0.10176	0.997764	0.0581818	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0.816703	0.989977	0.0231662	0.99885	0.0690191	0.997669	0.049054	0.945132

Log Likelihood of Short Data

-28.13852617314715

Problem 2

Sampling was performed by creating a chain of sampled state values for each of the random variables x_k, which in the case of a HMM, are already conveniently in topological ordering. Weights were determined for each sample by the following update function:

$$w_k = w_{k-1} P(y = ev | x_k = s) \tag{1}$$

The current weight at timestep k is the prior weight w_{k-1} times the probability that the sampled state for x_k emits the observed evidence $y = ev$. This iterative update is done until the end of the chain to give the weight for the entire sample.

Posterior distribution estimates for each state of X_k were generated by using the following formula:

$$P(x_k = s|Y_{1:T}) = \frac{\sum_{i=1}^{N} w_i \delta(x_i = s)}{\sum_{i=1}^{N} w_i} \tag{2}$$

Where N is number of samples, δ is to Kronecker Delta function / indicator that sifts out the desired states across all samples for a given state random variable x_k. This is a weighted frequency count of the sampled states for each state random variable x_k.

Exact Inference vs. Approximate Inference (LW) with 100 Samples

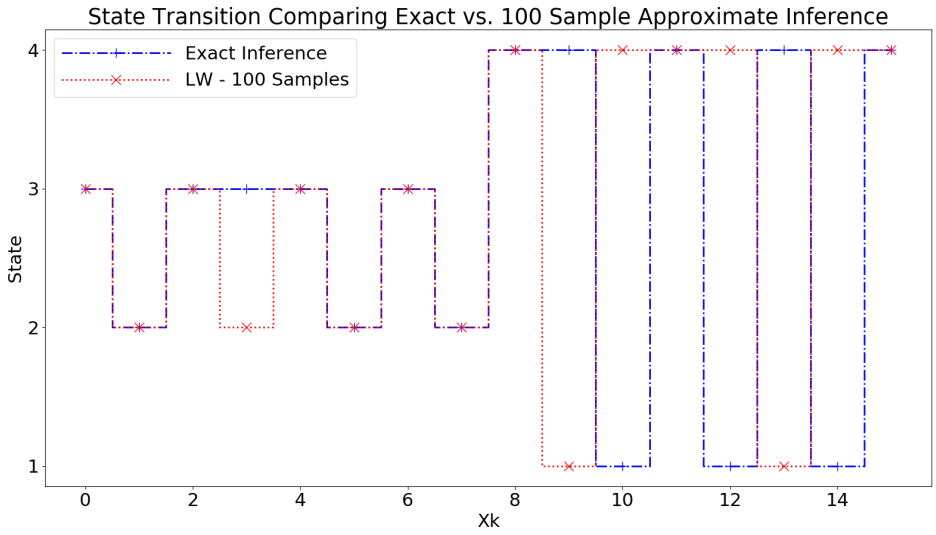

Figure 2: For 100 samples, oftentimes there are mismatched states in a max-likelihood state trace.

Exact Inference vs. Approximate Inference (LW) with 1000 Samples

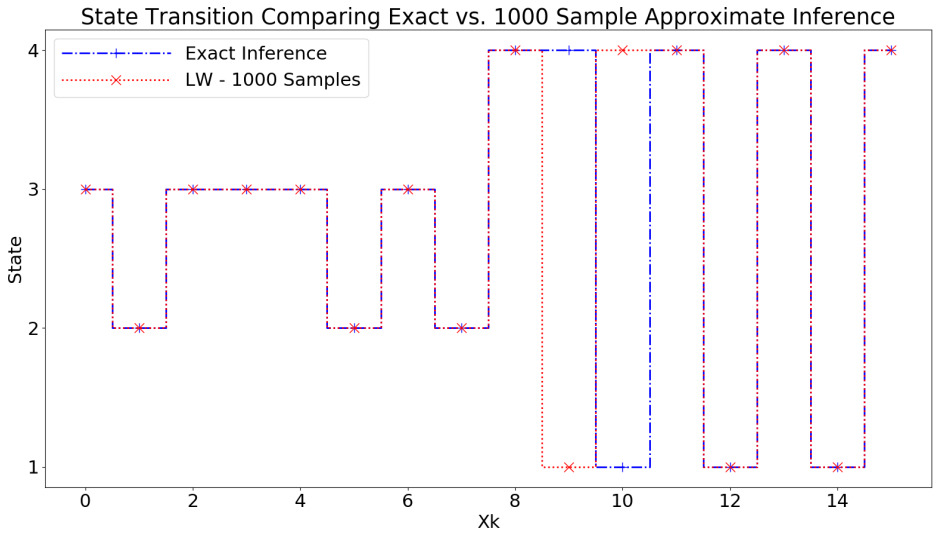

Figure 3: For 1000 samples, oftentimes there are mismatched states in a max-likelihood state trace.

Exact Inference vs. Approximate Inference (LW) with 10000 Samples

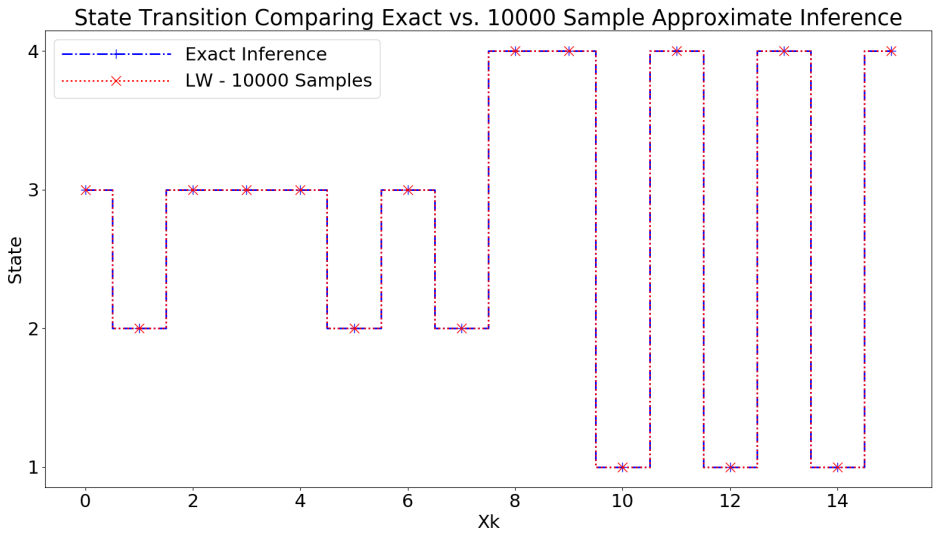

Figure 4: For 10000 samples, most runs result in perfect state trace alignment.

As seen in the above state traces, the likelihood weighting algorithm does provide fairly accurate state distribution inference. As the number of samples increases, the accuracy generally increases. However, if the long HMM dataset is substituted in place of the short dataset, the likelihood weighting (at least with the chosen q function) makes no inference. This is likely because emission probabilities have a zero probability for at least one of the states which makes it very, very unlikely to sample a full state sequence that does not 'zero out' the final weight.

Problem 3

For the long dataset, the state trace is in lockstep early in the sequence, but starts to diverge later in the sequence. This is likely the the standard forward-backward algorithm (without Mann's logarithmic approach) is suffering from floating point underflow which creates incorrect values during the calculation of alpha and beta.

State Trace for Long HMM Dataset

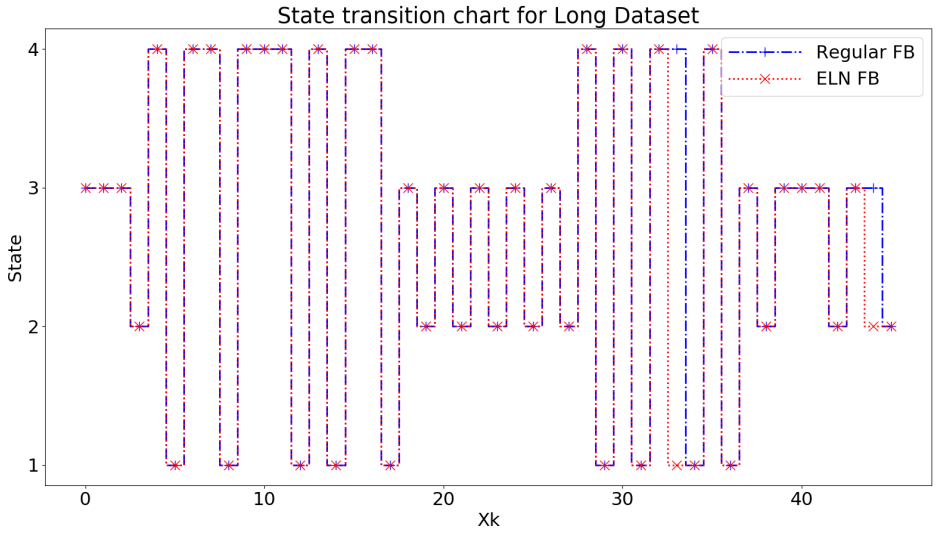

Figure 5: For the long dataset, the state trace is in lockstep for both the normal forward backwards and the logarithmic-based forward backward algorithm earlier in the sequence, but diverges later in the sequence.

Posterior Probabilities for Long Dataset

State	X1	X2	X3	X4	X5	X42	X43	X44	X45	X46
1	0	0	0	0	0.00618432	0	0	0	0	0
2	0	0.172346	0.00578533	0.995108	0	0.0132082	0.943541	0.0133714	0.157476	0.831321
3	1	0.827654	0.994215	0.00489207	0	0.986792	0.0564594	0.986629	0.842524	0.168679
4	0	0	0	0	0.993816	0	0	0	0	0

Log Likelihood of Long Data

-84.74339048594653

Extra Credit: Viterbi

State Trace for Short HMM Dataset

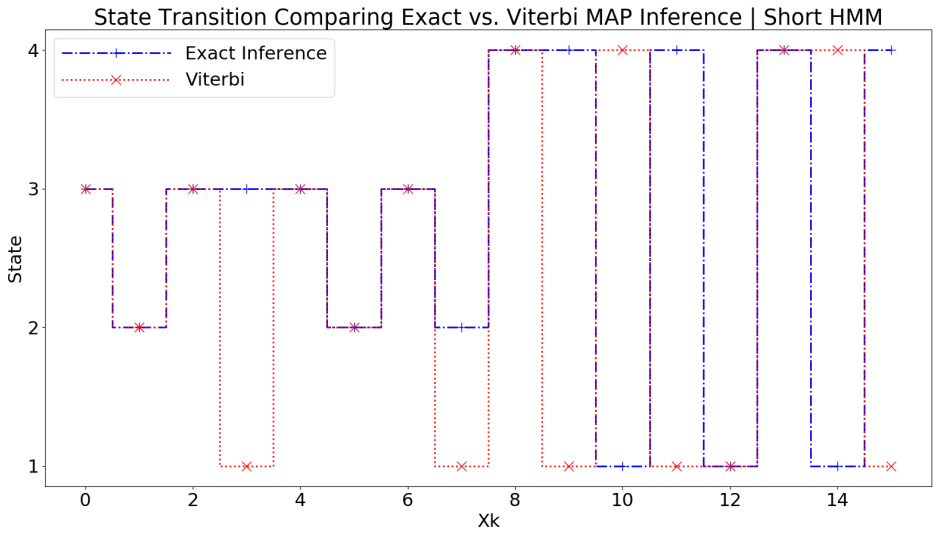

Figure 6: For the short dataset, the Viterbi algorithm is not as accurate as exact inference.

State Trace for Long HMM Dataset

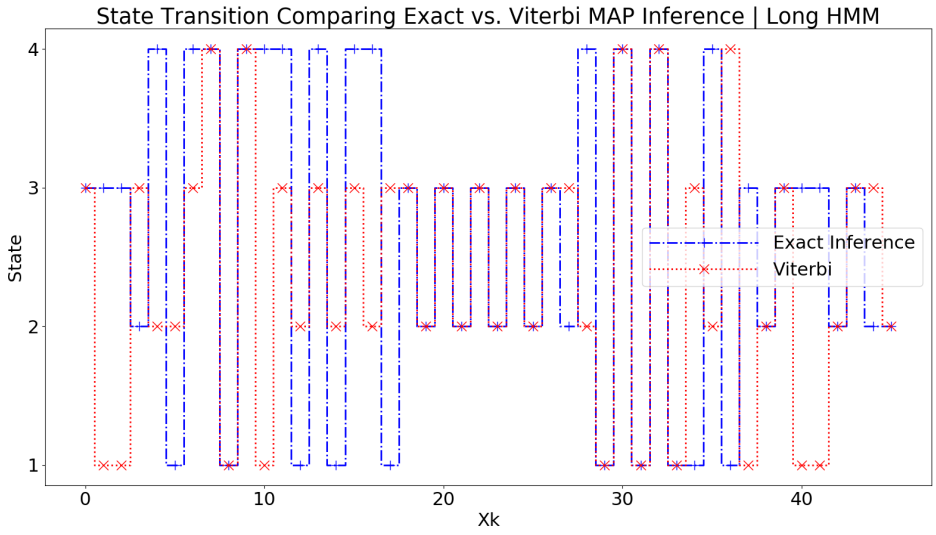

Figure 7: For the long dataset, the Viterbi algorithm is not as accurate as exact inference.

Code:

```
# forward_backward.py
import numpy as np
from exponential_scaling import eexp, eln, elnsum, elnproduct
import colored_traceback
colored_traceback.add_hook()


class ForwardBackwardHMM():
```

```
    def __init__(self, transition_probs, evidence_probs, initial_probs, obs):
        self.n_states = len(initial_probs)
        self.init = initial_probs
        self.emis = evidence_probs
        self.trans = transition_probs
        self.n_obs = len(obs)
        self.obs = obs

    def max_likelihood_state_estimate(self, probs):
        probs = probs.transpose()
        states = [np.argmax(prob_vec) + 1 for prob_vec in probs]
        return states

    def forward_backward(self):
        """
        Implements the forward backward algorithm for a simple HMM example. This
            function calls
        private methods _foward and _backwards to obtain the state probabilities for the
             forward
        and backward passes. They are then multiplied and normalized to get the full
            probabilities
        for each state at each time step.
        """
        alphas = self._forward_iter()
        betas = self._backward_iter()

        # recast as np.arrays to perform element-wise multiplication
        probs = np.array(alphas)[:, 0:] * np.array(betas)[:, :self.n_obs + 1]
        probs = probs / np.sum(probs, 0)

        # for i in range(self.n_obs):
        #     print(np.sum(alphas[:, i] * betas[:, i]))

        return probs, alphas, betas

    def forward_backward_eln(self):
        logalphas = self._forward_iter_eln()
        logbetas = self._backward_iter_eln()
        probs = np.zeros((self.n_states, self.n_obs + 1))
        for k in range(0, self.n_obs + 1):
            norm = -np.inf
            for i in range(self.n_states):
                probs[i, k] = elnproduct(logalphas[i, k], logbetas[i, k])
                norm = elnsum(norm, probs[i, k])
            for i in range(self.n_states):
                probs[i, k] = eexp(elnproduct(probs[i, k], -norm))
        # for i in range(self.n_obs):
        #     print(sum([p for p in list(probs[:, i]) if p != -np.inf]))
        return probs, logalphas, logbetas

    def _forward(self):
        """
        The forward (filtering) pass starting from the initial time step to the ending
            time step.
        alphas stores the calculated alpha value at each iteration normalized across
            states to avoid
        vanishing probabilities. The initial time step t_0 is initialized with the
            initial state
        probabilities.
        """
        alphas = np.zeros((self.n_states, self.n_obs + 1))
        alphas[:, 0] = self.init
        for k in range(0, self.n_obs):
            alphas[:, k + 1] = alphas[:, k].dot(self.trans.transpose()) * self.emis[
                self.obs[k], :]
        return alphas

    def _backward(self):
        """
        The backward (smoothing) pass starting from the an arbitrary future time step to
        the beginning time step. The matrix 'betas' stores the calculated beta value at
            each
```

```
        iteration normalized across states to avoid vanishing probabilities.
        """
        betas = np.zeros((self.n_states, self.n_obs))
        betas[:, -1] = 1
        for k in range(self.n_obs, 0, -1):
            beta_vec = np.matrix(betas[:, k]).transpose()
            betas[:, k - 1] = (np.matrix(self.trans.transpose()) * np.matrix(np.diag(
                self.emis.transpose()[:, self.obs[k - 1]])) * beta_vec).transpose()
        return betas

    def _forward_iter(self):
        alphas = np.zeros((self.n_states, self.n_obs + 1))

        # base case
        alphas[:, 0] = self.init
        # recursive case
        for k in range(0, self.n_obs):
            for i in range(self.n_states):
                for j in range(self.n_states):
                    alphas[i, k + 1] += alphas[j, k] * self.trans[i,
                                                                  j] * self.emis[self.
                                                                      obs[k], i]
        return alphas

    def _backward_iter(self):
        betas = np.zeros((self.n_states, self.n_obs + 1))
        # base case
        betas[:, -1] = 1
        # recursive case
        for k in range(self.n_obs, -1, -1):
            for i in range(self.n_states):
                for j in range(self.n_states):
                    betas[i, k - 1] += self.trans[j, i] * \
                        self.emis[self.obs[
                            k - 1], j] * betas[j, k]
        return betas

    def _forward_iter_eln(self):
        logalphas = np.zeros((self.n_states, self.n_obs + 1))

        # base case
        logalphas[:, 0] = [eln(x) for x in self.init]
        # recursive case
        for k in range(1, self.n_obs + 1):
            for j in range(self.n_states):
                logalpha = -np.inf
                for i in range(self.n_states):
                    logalpha = elnsum(logalpha, elnproduct(
                        logalphas[i, k - 1], eln(self.trans.transpose()[i, j])))
                logalphas[j, k] = elnproduct(logalpha, eln(
                    self.emis[self.obs[k - 1], j]))
        return logalphas

    def _backward_iter_eln(self):
        logbetas = np.zeros((self.n_states, self.n_obs + 1))
        # base case
        logbetas[:, -1] = 0
        # recursive case
        for k in range(self.n_obs, -1, -1):
            for i in range(self.n_states):
                logbeta = -np.inf
                for j in range(self.n_states):
                    logbeta = elnsum(logbeta, elnproduct(eln(self.trans.transpose()[i, j
                        ]), elnproduct(
                        self.emis[self.obs[k - 1], j], logbetas[j, k])))
                logbetas[i, k - 1] = logbeta
        return logbetas

# likelihood_sampling_infererence.py

import numpy as np
from numpy.random import choice
from observations import y_obs_long, y_obs_short
```

```
from parameters import pxk_xkm1, pyk_xk, px0
from exponential_scaling import eexp, eln, elnsum, elnproduct
import colored_traceback
colored_traceback.add_hook()


class LikelihoodSamplingInference():

    def __init__(self, transition_probs, evidence_probs, initial_probs, observations):
        self.n_states = len(initial_probs)
        self.init_probs = initial_probs
        self.ev_probs = evidence_probs
        self.trans_probs = transition_probs
        self.n_observations = len(observations)
        self.observations = observations

    def max_likelihood_state_estimate(self, probs):
        probs = probs
        states = [np.argmax(np.array(prob_vec)) + 1 for prob_vec in probs]
        return states

    def run_inference(self, n_samples=100):
        samples, weights = self.run_sampling(n_samples)
        probs = []
        for k in range(self.n_observations + 1):
            prob = []
            xk_stack = list(list(zip(*samples))[k])
            for s in range(self.n_states):
                idxs = [idx for idx, value in enumerate(xk_stack) if value == s]
                prob.append(sum([w for idx, w in enumerate(weights) if idx in idxs]) /
                    sum(weights))
            probs.append(prob)
        return probs

    def run_sampling(self, n_samples):
        samples = []
        weights = []
        for n in range(n_samples):
            sample, weight = self.liklihood_sampling()
            samples.append(sample)
            weights.append(weight)
        return samples, weights

    def liklihood_sampling(self):
        w = 1
        X = []
        xkm1 = self.sample_state(self.init_probs)
        X.append(xkm1)
        for k in range(0, self.n_observations):
            xk = self.sample_state(list(self.trans_probs[:, xkm1]))
            ob_idx = self.observations[k]
            py_xk = self.ev_probs[ob_idx, xk]
            w = w * py_xk
            X.append(xk)
            xkm1 = xk
        return(X, w)

    def sample_state(self, probs):
        return choice(4, 1, p=probs)[0]

# exponential_scaling.py

import numpy as np


def eexp(x):
    if np.isinf(x):
        return 0
    else:
        return np.exp(x)


def eln(x):
```

```
    if x == 0:
        return -np.inf
    if x > 0:
        return np.log(x)
    else:
        raise ValueError("x must be non-negative for eln")


def elnsum(elnx, elny):
    if np.isinf(elnx) or (np.isinf(elny)):
        if (np.isinf(elnx)):
            return elny
        else:
            return elnx
    else:
        if (elnx > elny):
            return elnx + eln(1 + np.exp(elny - elnx))
        else:
            return elny + eln(1 + np.exp(elnx - elny))


def elnproduct(elnx, elny):
    if np.isinf(elnx) or np.isinf(elny):
        return -np.inf
    else:
        return elnx + elny

# viterbi.py

import numpy as np
from observations import y_obs_long, y_obs_short
from parameters import pxk_xkm1, pyk_xk, px0
from exponential_scaling import eexp, eln, elnsum, elnproduct
import colored_traceback
colored_traceback.add_hook()


class ViterbiHMM():

    def __init__(self, transition_probs, evidence_probs, initial_probs, observations):
        self.n_states = len(initial_probs)
        self.init = initial_probs
        self.emis = evidence_probs
        self.trans = transition_probs
        self.n_obs = len(observations)
        self.obs = observations

    def viterbi_path(self):

        init = np.array([eln(val) for val in np.nditer(self.init)])
        trans = np.array([[eln(v) for v in np.nditer(axis)] for axis in np.nditer(self.
            trans)]).reshape(self.n_states, self.n_states)
        print(np.array([[eln(v) for v in np.nditer(axis)] for axis in np.nditer(self.
            emis)]))
        emis = np.array([[eln(v) for v in np.nditer(axis)] for axis in np.nditer(self.
            emis)]).reshape(self.emis.shape[1], self.emis.shape[0])

        best_states = []

        logprob = init
        for k in range(0, self.n_obs):
            trans_p = np.zeros([self.n_states, self.n_states])
            for i in range(self.n_states):
                for j in range(self.n_states):
                    trans_p[i, j] = elnsum(logprob[i], elnproduct(trans[i, j], emis[i,
                        self.obs[k-1]]))
            # Get the indices of the max probs that give the best prior states.
            best_states.append(np.argmax(trans_p, axis=0))
            logprob = np.max(trans_p, axis=0)
        print(len(best_states))
        # Most likely final state.
        final_state = np.argmax(logprob)
        print(final_state)
```

```
    # Reconstruct path by backtracking through likeliest states.
    prior_state = final_state
    best_path = [prior_state + 1]
    for best in reversed(best_states):
        prior_state = best[prior_state]
        best_path.append(prior_state + 1)
    return list(reversed(best_path)), logprob[final_state]
```