COMP0080 - Graphical Models
Assignment 2
DUE: DECEMBER 9, 2019

UCL
Machine Learning MSc
Computational Statistics and Machine Learning MSc

Author: Dorota Jagnesakova
Student ID: 16079098
E-mail: dorota.jagnesakova.19@ucl.ac.uk
Questions: 0,1

Author: Agnieszka Dobrowolska
Student ID: 16034489
E-mail: agnieszka.dobrowolska.16@ucl.ac.uk
Questions: 0,1

Author: Oliver Slumbers
Student ID: 19027699
E-mail: oliver.slumbers.19@ucl.ac.uk
Questions: 2

Author: Tom Grigg
Student ID: 19151291
E-mail: tom.grigg.19@ucl.ac.uk
Questions: 0, 1, 2

1 Weather

0. Fully observed case. Weather station data.

We are given N sequences of data describing weather conditions on consecutive days, $v^n_{1:T} = v^n_1, .., v^n_T, n = 1, .., N$ These are realisations of the random variable $V_{1:T}$ which is a Markov chain with three states, i.e. $V_i \in \{0, 1, 2\}$ (Rainy = 0, Cloudy = 1, Sunny = 2). We have N = 500 (number of sequences) and T = 100 (length of each sequence). We denote the transition matrix as A and the transition probabilities as $A_{ij} = P(V_{t+1} = i \mid V_t = j)$.

The parameters of this model are the elements of the transition matrix i.e. the transition distribution, as well as the initial distribution for v_1. We can enumerate them by considering that the Markov chain $V_{1:T}$ has 3 states, and therefore there are 3^2 possible transitions. However, we know that $\sum_{i=0}^{2} P(V_{t+1} = i \mid V_t = j) = \sum_{i=0}^{2} A_{ij} = 1$, for each state $j \in \{0, 1, 2\}$. This means we only need to know 2 of the transition probabilities for each j, and similarly we just need to know 2 of the values for the initial distribution, and so the number of free parameters in our model is $3 \cdot 2 + 2 = 8$.

Consider the probability of obtaining the samples $v^n_{1:100}, n = 1, .., 500$:

$$P(V^N_{1:100} = \boldsymbol{v_{1:100}}) = \prod_{n=1}^{500} [P(V_1 = v^n_1) \prod_{t=2}^{100} P(V_t = v^n_t \mid V_{1:t-1} = v^n_{1:t-1})] \tag{1}$$

$$= \prod_{n=1}^{500} [P(V_1 = v^n_1) \prod_{t=2}^{100} P(V_t = v^n_t \mid V_{t-1} = v^n_{t-1})] \tag{2}$$

Where we obtain (1) using the definition of conditional probabilities and (2) using the Markov property.

Now we define the likelihood function

$$\mathcal{L}(\theta) = \prod_{n=1}^{500} [P(V_1 = v^n_1) \prod_{t=2}^{100} A_{v^n_t v^n_{t-1}}]$$

$$= \prod_{n=1}^{500} [P(V_1 = v_1) \prod_{i=0}^{2} \prod_{j=0}^{2} A_{ij}^{n_{ij}}], \qquad n_{ij} = \sum_{n=1}^{500} \sum_{t=2}^{100} \mathbb{1}[v^n_t = i, v^n_{t-1} = j]$$

Where n_{ij} denotes the number of times i is followed by j in our sample sequences. To maximise the likelihood, we take the log:

$$\mathcal{L}(\theta) = \sum_n [log P(V_1 = v^n_1) + \sum_{ij} n_{ij} log(A_{ij})]$$

Before taking derivatives with respect to A_{ij} we need to consider the constraint $\sum_{i=0}^{2} A_{ij} = 1$ for every j. We pick one of the transition probabilities to be expressed in terms of the others. This is an arbitrary choice, so let us pick the probability of having a rainy day for each j $A_{0j} = 1 - A_{1j} - A_{2j}$. Now, the derivatives with respect to each A_{ij} apart from A_{0j} are:

$$\frac{\partial \mathcal{L}}{A_{ij}} = \frac{n_{ij}}{A_{ij}} - \frac{n_{0j}}{A_{0j}}$$

$$\Rightarrow \frac{n_{ij}}{\hat{A}_{ij}} = \frac{n_{0j}}{\hat{A}_{0j}}$$

$$\frac{n_{ij}}{n_{0j}} = \frac{\hat{A}_{ij}}{\hat{A}_{0j}}$$

Thus $\hat{A}_{ij} \propto \sum_n n_{ij}$, as the above holds for all $i \neq 0$. In fact,

$$\hat{A}_{ij} = \frac{n_{ij}}{\sum_{i=0}^{2} n_{ij}}$$

Implementing this in our code, we obtained:

$$\hat{A} = \begin{pmatrix} 0.69876424 & 0.10079219 & 0.10071453 \\ 0.30123576 & 0.6995032 & 0.19870704 \\ 0. & 0.19970461 & 0.70057843 \end{pmatrix}$$

Following very similar logic, we obtained:

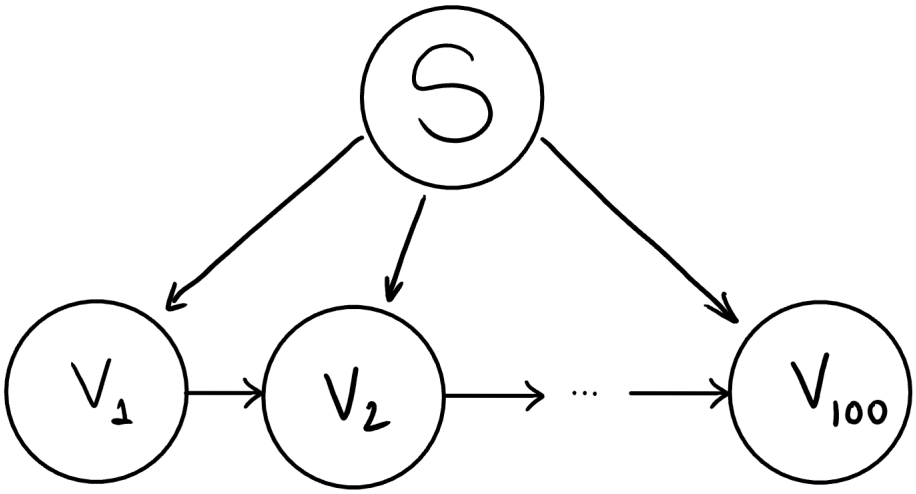
$$\hat{P}(v_1) = \begin{pmatrix} 0.308 \\ 0.512 \\ 0.18 \end{pmatrix}$$

(Please see appendix for implementations)

1. Three weather stations.

(a) Here, we introduce a latent random variable s corresponding to the possible weather stations. Each data sequence $v^n_{1:100}$ we observed came from one of three weather stations, and so we allow s to take on values in $\{s_1, s_2, s_3\}$.

For a given n, the graphical model looks like this:



We now require three matrices in order to specify the distribution, so let us define a set of matrices $M = \{A^{(1)}, A^{(2)}, A^{(3)}\}$, each of dimension 3×3, where each matrix $A^{(k)}$ defines the transition distribution for the corresponding weather station: $A^{(k)}_{ij} = P(v_t = i \mid v_{t-1} = j, s = s_k) \equiv P_{s_k}(v_t = i \mid v_{t-1} = j)$. We define the marginal distribution over s, $\boldsymbol{\pi} = P(s)$, as a vector in $\mathbb{R}^3$. We also require an initial distribution $P_k(v_1)$ for the value v_1 of a sequence given a weather station. Our set of parameters for the EM algorithm is $\theta = \{M, \boldsymbol{\pi}, P_k(v_1)\}$, and the number of free parameters is $3 \times 6 + 2 + 6 = 26$.

In order to calculate the likelihood in our E-step, we calculate the probability of obtaining $v^n_{1:100}$ given our parameters, and use the assumed independence between sequences to formulate a product. The product is over sums of the form:

$$\begin{aligned} P(V_{1:100} = v_{1:100} \mid \theta) &= \sum_{k=1}^{3} [\pi_k P_{s_k}(V_1 = v_1) \prod_{t=2}^{100} P_{s_k}(V_t = v_t \mid V_{t-1} = v_{t-1})] \\ &= \sum_{k=1}^{3} [\pi_k P_{s_k}(V_1 = v_1) \prod_{t=2}^{100} A^{(k)}_{v_t v_{t-1}}] \\ &= \sum_{k=1}^{3} [\pi_k P_{s_k}(V_1 = v_1) \prod_{i=0}^{2} \prod_{j=0}^{2} (A^{(k)}_{ij})^{n_{ij}}], \qquad n_{ij} = \sum_{t=2}^{100} \mathbb{1}[v_t = i, v_{t-1} = j] \end{aligned} \tag{3}$$

Where n_{ij} denotes the number of times i is followed by j in the sequence.

The EM algorithm aims to recover maximum likelihood estimation in the presence of latent variables. The general algorithm iteratively maximizes a lower bound on the log-likelihood of the data.
To specify the steps of the EM algorithm for this problem, we first need to determine a lower bound on the log-likelihood.

$$\begin{aligned} \mathcal{L}(\theta) = log P(v \mid \theta) &= log \sum_k P(v, s_k \mid \theta) \\ &= log \sum_k q(s_k) \frac{P(v, s_k \mid \theta)}{q(s_k)} \geqslant \sum_k q(s_k) log \frac{P(v, s_k \mid \theta)}{q(s_k)} \end{aligned}$$

Where $q(.)$ is any distribution over the hidden variables s_k and the inequality follows from Jensen's inequality (and the fact that the log function is concave on its entire domain). Hence, we define the lower bound to be:

$$\begin{aligned}\sum_k q(s_k) log \frac{P(v, s_k \mid \theta)}{q(s_k)} &= \sum_k q(s_k) log \frac{P(s_k \mid v), \theta) P(\boldsymbol{v} \mid \theta)}{q(s_k)} \\ &= \sum_k q(s_k) log P(v \mid \theta) + \sum_k q(s_k) log \frac{P(s_k \mid v, \theta)}{q(s_k)} \\ &= \mathcal{L}(\theta) - \sum_k q(s_k) log \frac{q(s_k)}{P(s_k \mid v, \theta)}\end{aligned}$$

The term being subtracted in the last line is the Kullback-Leibler divergence.

Now, the E-step maximizes the above w.r.t $q(.)$ and the M-step maximizes it w.r.t. θ. For the E-step, we aim to make the KL term as small as possible - we know from lectures that it will be optimal when we set $q^{(r)}(s_k) = P(s_k \mid v, \theta^{(r-1)})$ at each step r. Using Bayes rule, we can state:

$$\begin{aligned}P(s_k \mid v, \theta) &= \frac{P(s_k, v \mid \theta)}{P(v \mid \theta)} \\ &\propto P(v \mid s_k, \theta) P(s_k \mid \theta) \\ &= \pi_k P_{s_k}(v_1) \prod_{t=2}^{100} P_{s_k}(v_t \mid v_{t-1})\end{aligned}$$

Hence, our E-step corresponds to calculating the above expression. Once this has been done, we then calculate our parameter updates in our M-step as follows:

$$\begin{aligned}\pi^{new} = P^{new}(s = k) &\propto \sum_{n=1}^{N} P^{old}(s = k \mid v_{1:T}^n) \\ A_{new}^{(k)} = P^{new}(v_t = i \mid v_{t-1} = j, s = k) &\propto \sum_{n=1}^{N} \left[P^{old}(s = k \mid v_{1:T}^n) \sum_{t=2}^{T} \mathbb{I}[v_t^n = i] \mathbb{I}[v_{t-1}^n = j] \right] \\ P_k(v_1)^{new} = P^{new}(v_1 = i \mid s = k) &\propto \sum_{n=1}^{N} P^{old}(s = k \mid v_{1:T}^n) \mathbb{I}[v_1^n = i]\end{aligned}$$

Tying all this together, we obtain the **Steps of the Algorithm**:

- First, we **initialise** the parameters θ, by generating a set of three 3×3 transition matrices $A^{(k)}$, and generating a vector $\boldsymbol{\pi}$ of length 3, under the constraint $\sum_{k=1}^{3} \boldsymbol{\pi}_k = 1$, as well as our initial $P_k(v_1)$ distributions for each weather station. In practise, we found that the best way to do this was by randomly assigning each sequence to an initial weather station, and computing the initial parameters from the distribution this induces.
  - **E Step**: Compute $P(s_k \mid v, \theta^{(new)})$ as above, for every sequence and every station. Compute $\log(\mathcal{L}(\theta))$ of the whole data set: $\log(\mathcal{L}(\theta^{(new)})) = \prod_{n=1}^{500} \log(P(v_{1:100}^n) \mid \theta^{(new)})$, i.e. as defined in (3).
  - **M Step**: Compute new parameters as above: new transition matrices $A^{(k)}$ for every station, initial probabilities $P_{s_k}(V_1 = v_1)$ and $\boldsymbol{\pi}$. $A^{(k)}$ are computed using expressions corresponding to the ML estimates (for θ_t derived above as in question 0 (and assignments from E-step). In practise, the probabilities $P_{s_k}(v_1 = i)$, for every i, are computed by counting the number of times $v_1 = i$ over all sequences assigned to s_k and dividing by total number of sequences assigned to s_k. Further, each π_k is computed by dividing the number of sequences assigned to s_k by the number of stations and normalising. These operations are equivalent to the above expressions.
  - **Repeat until convergence** with respect to likelihood.

(b) For implementation, see code in the appendix.
Learned parameters:

$$\boldsymbol{\pi} = \begin{pmatrix} 0.19200000 \\ 0.50830115 \\ 0.29969885 \end{pmatrix} \qquad A^{(1)} = \begin{pmatrix} 0.38869863 & 0.24059140 & 0.54531568 \\ 0.14848337 & 0.51545699 & 0.01756619 \\ 0.46281800 & 0.24395161 & 0.43711813 \end{pmatrix}$$

$$A^{(2)} = \begin{pmatrix} 0.05971004 & 0.13007011 & 0.41759897 \\ 0.13950704 & 0.32378128 & 0.42784294 \\ 0.80078292 & 0.54614861 & 0.15455809 \end{pmatrix} \qquad A^{(3)} = \begin{pmatrix} 0.07354505 & 0.34273044 & 0.30491067 \\ 0.20148914 & 0.22658642 & 0.64191719 \\ 0.72496581 & 0.43068314 & 0.05317214 \end{pmatrix}$$

Log-likelihood after convergence: -45254.51039028404

Posterior probabilities of first 10 sequences, where k^{th} column corresponds k^{th} weather station s_k.

$$\begin{pmatrix} 1. & 0. & 0. \\ 0. & 0.00019072 & 0.99980928 \\ 0. & 0.99999654 & 0.00000346 \\ 0. & 1. & 0. \\ 0. & 0.99999411 & 0.00000589 \\ 0. & 0.00003759 & 0.99996241 \\ 1. & 0. & 0. \\ 0. & 0.99999999 & 0.00000001 \\ 0. & 0.99999684 & 0.00000316 \\ 0. & 0.99999779 & 0.00000221 \end{pmatrix}$$

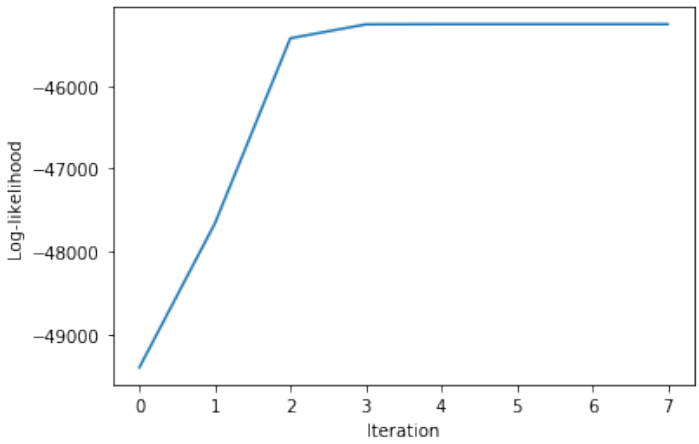


Figure 1: Convergence graph for our EM algorithm in Q1

(c) *Initialisation*: When implementing the EM algorithm, the observed likelihood increases with each iteration, and since the likelihood function defined over the parameter space is not generally convex, we can only converge to a local maximum, i.e. this is not necessarily the global maximum (MLE). Therefore, to find the true maximum one should run the algorithm multiple times with randomly chosen initial guesses, and choose the parameters which give the largest value for the likelihood/log-likelihood. In practise, we found that our algorithm converged to similar parameters despite our fairly stochastic initialisation strategy - though occasionally we would experience a bad initialisation and converge to a different maximum.

We chose to initialise randomly, based on the above. We initially attempted to generate random values for our parameters directly, but this more frequently lead to bad initialisations, typically due to unfortunately small values. Also, this strategy can result in zeros in the transition matrix, which is problematic. Instead, we found that initially assigning each sequence to a weather station, and then calculating the resulting empirical distributions as initial values for our parameters was much more robust and effective at converging. This was done with $\boldsymbol{\pi}$, for example, by

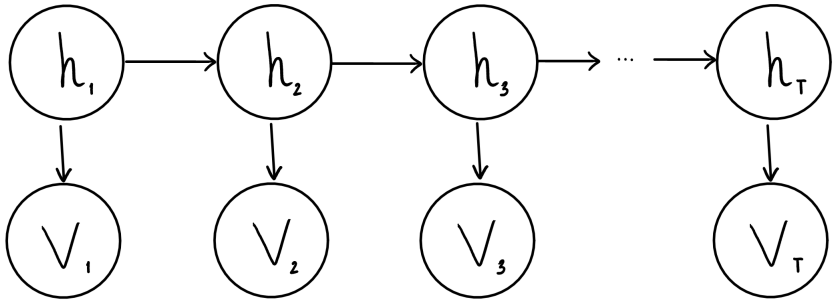
simply counting the number of sequences that were assigned to each weather station, and then normalising. Given this initial random assignment, the other initial distributions were calculated in a similarly natural manner. See our code in the appendix for more details.

Computational issues: The length of the sequences became particularly problematic in the E-step of our implementation of the algorithm. This was because we were multiplying 100 probabilities together, resulting in very small numbers, and frequently causing underflow. We found that this and similar issues elsewhere in our implementation could be avoided by working in the log-domain wherever possible/sensible. We also used some handy `numpy` functions to allow us to normalise while still working in the log-domain!

(d) If we initialise every distribution in θ uniformly, then the EM algorithm would not be able to converge in the manner we would like. What first becomes clear is that the posterior distribution would not be able to update, since all of the terms in the RHS are constant regardless of our choice of sequence and station. If we observe the equations, it becomes clear that no parameter except for the transition matrices would be able to compute a non-zero update. In fact, reviewing the equations we stated above in the context of a uniform distribution, the transition matrices would update exactly once, and this update would correspond to counting the number of empirical transitions across the whole dataset of sequences, and then setting all three matrices equal to this. (The key to seeing that this happens is due to the fact that our posterior distribution will also be forced to be uniform). The algorithm will then have converged. Intuitively, what is happening when we initialise with a uniform distribution is we are forcing the three weather stations to begin in the same place in parameter space, and this means that the EM algorithm is never able to separate them. From the perspective of the algorithm, placing the three weather stations in the same initial spot in parameter space corresponds to having one weather station, which is why the transition matrix is updated with the empirical frequency across all sequences exactly once, and then never again. (So in a sense, the algorithm has converged - but in a degenerate manner). Long story short, we don't trust our friend anymore.

2 Chess

(a) The induced distribution over the hidden variables $P_v(h_{1:T}) = P(h_{1:T}|v_{1:T})$ can be represented graphically as:



In the general case, h_T and h_1 are not independent. This can be seen in the graphical model by observing that h_T and h_1 are d-connected, i.e. there exist distributions satisfying the graph in which h_1 and h_T are dependent.

(b) Given that we observe the sequence $v_{1:10} = [0, 1, 0, 2, 0, 2, 1, 0, 2, 0]$, we can in fact infer that $h_2 = 0$, since $P(v_t = 1|h_t = 1) = 0$, (Anton never draws when he's angry). Hence, h_2 is independent of h_1, and so the distribution induced by the data $v_{1:10}$ would satisfy the graph redrawn with h_1 and h_2 d-separated. Hence, given this data, h_{10} is also d-separated from h_1, and therefore h_{10} is independent of h_1. Explicitly, if we let $\mathcal{S} = [0, 1, 0, 2, 0, 2, 1, 0, 2, 0]$ then:

$$\begin{aligned}
P_{v=\mathcal{S}}(h_{10}|h_1) &= P(h_{10}|h_1, v_{1:10} = \mathcal{S}) \\
&= \sum_{h_2} P(h_{10}|h_2, h_1, v_{1:10} = \mathcal{S}) P(h_2|h_1, v_{1:10} = \mathcal{S}) \\
&= \sum_{h_2} P(h_{10}|h_2, v_{1:10} = \mathcal{S}) P(h_2|v_{1:10} = \mathcal{S}) \\
&= P(h_{10}|h_2 = 0, v_{1:10} = \mathcal{S}) \\
&= P(h_{10}|v_{1:10} = \mathcal{S}) = P_{v=\mathcal{S}}(h_{10})
\end{aligned}$$

(c) To implement the forward-backward algorithm we define a class `FwdBwdSolver`, which can be found in the appendix. The member functions `alpha(t)` and `beta(t)` correspond to the forward and backward pass, respectively. Note that these functions return normalised distributions due to computational issues discussed in part *e*. Of course, this does not affect the final calculation of singleton and pairwise marginals. The following figure displays our results for $P(h_t|v_{1:10})$ for each value of $t \in \{1, \ldots, 10\}$:

```
t=1 => (1.0, 0.0)
t=2 => (1.0, 0.0)
t=3 => (0.5023923444976076, 0.49760765550239244)
t=4 => (0.29824561403508776, 0.7017543859649122)
t=5 => (0.7320574162679425, 0.2679425837320574)
t=6 => (0.17065390749601275, 0.8293460925039873)
t=7 => (1.0, 0.0)
t=8 => (0.4883720930232558, 0.5116279069767443)
t=9 => (0.34108527131782956, 0.6589147286821705)
t=10 => (0.5930232558139534, 0.4069767441860465)
```

Figure 2: $P_v(h_t)$ for $t \in \{1, \ldots, 10\}$, where the distribution is displayed as a row vector $(P_v(h_t = 0), P_v(h_t = 1))$

(d) Using the messages computed in the previous question, we compute the pairwise marginals via the member function `P_ht_ht_plus_1_GIVEN_vs(t)`. The results for $P(h_1, h_2|v_{1:10})$ and $P(h4, h5|v_{1:10})$ are displayed below:

```
P(h_1, h_2 | v_1:T)=
 [[1. 0.]
 [0. 0.]]

P(h_4, h_5 | v_1:T)=
 [[0.15789474 0.57416268]
 [0.14035088 0.12759171]]
```

Figure 3: $P_v(h_1, h_2)$ and $P_v(h_4, h_5)$, where the distribution is displayed as a matrix, with h_{t+1} corresponding to the rows, and h_t corresponding to the columns.

(e) Our implementation of Baum-Welch is included in the appendix. We encountered computational issues in our implementation when using the full `chess.txt` dataset as opposed to the $v_{1:10}$ we had been asked to use in previous questions. These computational issues stemmed from the fact that `alpha(t)` and `beta(t)` calculate $P(h_t, v_{1:t})$ and $P(v_{t+1:T}|h_t)$, respectively. The probability masses at each point in these discrete distributions decreases exponentially on average with the length of the given data sequence, and is hence typically very small for larger sequences of data. This was the heart of the problem, as we ran into significant underflow, making our calculations as we were performing them up until this point infeasible.

To better understand the source of our computational issues, consider `alpha(t)` $= P(h_t, v_{1:t}) = P(h_t, v_1, v_2, v_3, \ldots, v_t)$, this is a distribution over 2 possible values of h_t, and 3 possible values of v_i for each $i \in \{1, \ldots, t\}$. This means that we must assign mass to 2×3^t points. If we assume a uniform distribution, then each point will be assigned a mass of $\frac{1}{2\times 3^t}$. Of course, in the general case the distribution is not uniform, and some mass must be taken from one point and assigned elsewhere, meaning that the smallest value in our distribution is in fact upper-bounded by $\frac{1}{2\times 3^t}$. Hence, for this problem we can see that as t increases, the calculated distribution `alpha(t)` decays pointwise to zero. For $t = 10$, this is not so bad since $\frac{1}{2\times 3^{10}} \approx 8.5 \times 10^{-6}$, which is small, but can certainly be represented by a 64-bit floating point number. This explains why we did not have issues previously. However, for $t = 1000$, the pointwise mass becomes so small that it can no longer be represented by floating point precision, and we reach underflow, which causes undefined behavior/division by zero and a not-a-number result. The issue is almost identical with respect to `beta(t)`.

In order to fix these computational issues, we realised that we were not actually interested in the distributions calculated by our forward and backwards steps, `alpha(t)` and `beta(t)`. Instead, these distributions were used to compute the singleton and pairwise marginals to be used in the Baum-Welch algorithm. Since we were already normalising in order to compute the marginals, we deduced that we were able to normalise step-wise, i.e. at each calculation of `alpha(t)` and `beta(t)`, without affecting the final result. We also found that we could work in the log domain to avoid underflow, but elected to use step-wise normalisation so that we had to make minimal changes to our code.

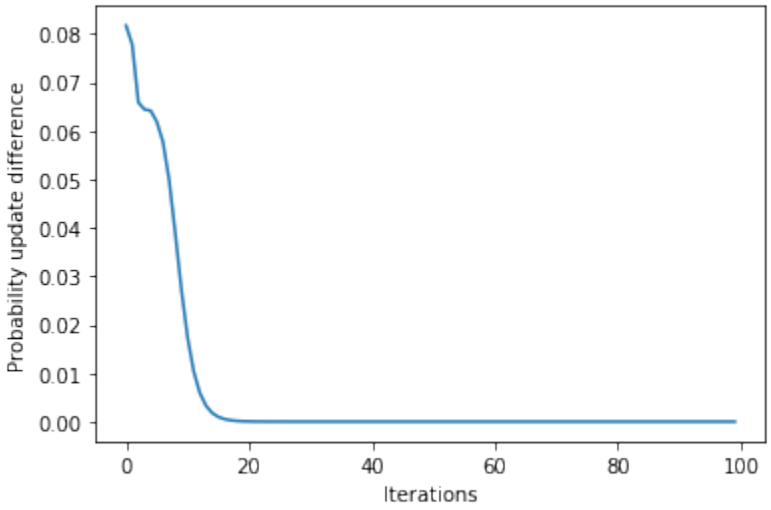
We also found that our code ran very slowly because we were wastefully recomputing `alpha(t)` and `beta(t)` every time our solver required their values during each iteration, (i.e. `alpha(1)` was being computed 1000 times, and `alpha(2)` was being computed 999 times, and so on). To make our code more computationally efficient, we elected to employ memoization, so that at each iteration we only compute `alpha(t)` and `beta(t)` once. This resulted in more than a $100\times$ speed up due to the reduction in computational complexity.

Once we fixed our computational issues, we ran the EM algorithm for the data sequence in `chess.txt`, fixing the emission probability distributions and initial hidden state, and initialising the transition distribution matrix to T_0. With respect to floating point precision, the algorithm converged in 71 iterations (i.e. this is the first point at which the numerical difference in transition matrices could not be represented by the floating point representation we're using). This was sufficiently fast, however, we deemed this notion of convergence to be unnecessarily extreme, since the difference in the resulting transition matrix was very small. So, instead we ran the algorithm to convergence by choosing an $\epsilon = 0.0001$ and waiting until the maximum elementwise difference in the transition matrix was below this threshold. This occurred at iteration 20. The final value of our learned transition matrix was:

```
[[0.83258179 0.0723181 ]
 [0.16741821 0.9276819 ]]
```

Figure 4: Learned transition matrix after our implementation of Baum-Welch reached convergence.

We also plotted a convergence graph showing the mean difference of transition matrices between iterations to demonstrate the appropriateness of our definition of convergence:



(f) In order to predict the outcome of the next game, we formulated the following equation:

$$\begin{aligned} P(v_{t+1}|v_{1:t}) &= \sum_{h_{t+1}} \sum_{h_t} P(v_{t+1}, h_{t+1}, h_t|v_{1:t}) \\ &= \sum_{h_{t+1}} \sum_{h_t} P(v_{t+1}|h_{t+1}, h_t, v_{1:t}) P(h_{t+1}|h_t, v_{1:t}) P(h_t|v_{1:t}) \\ &= \sum_{h_{t+1}} \sum_{h_t} P(v_{t+1}|h_{t+1}) P(h_{t+1}|h_t) P(h_t|v_{1:t}) \end{aligned}$$

Hence, for $P(v_{1001}|v_{1:1000})$, this summation involves only the transition matrix, the emission matrix, and the singleton marginal for the final hidden variable given the entire sequence of data in `chess.txt`. We calculated the latter using our `FwdBwdSolver`, and then computed the formulated expression. Our results were:

```
[0.47736557 0.0679033  0.45473113]
```

Figure 5: The row vector corresponding to the probabilities ($P(v_{1001} = 0|v_{1:1000})$, $P(v_{1001} = 1|v_{1:1000})$, $P(v_{1001} = 2|v_{1:1000})$), i.e. probability of the outcome of the 1001^{th} game.

We observe that in the resulting row vector, the most likely outcome given the data and our learned parameter is: $v_{1001} = 0$, i.e. Anton wins, (angrily)!

In [26]:
```
import pandas as pd
import numpy as np
import random
```

Question 0

In [27]:
```
data = pd.read_csv('meteo0.csv', sep=' ', header=None).values
```

Compute transition matrix

In [28]:
```
N = data.shape[0]
T = data.shape[1]

freq_table = np.zeros((3, 3))

for n in range(N):
    for t in range(T - 1):
        freq_table[data[n, t+1], data[n, t]] += 1

A_ij = freq_table / freq_table.sum(axis=0)
print(A_ij)
```

```
[[0.69876424 0.10079219 0.10071453]
 [0.30123576 0.6995032  0.19870704]
 [0.         0.19970461 0.70057843]]
```

Compute initial probabilities

In [29]:
```
# Obtain probabilities of initial states
initial = np.zeros(shape=(3,1))
for sequence in data:
    for i in range(3):
        if sequence[0]==i:
            initial[i,0] += 1
initial/= initial.sum(axis=0)
print(initial)
```

```
[[0.308]
 [0.512]
 [0.18 ]]
```

Q1

Read Data and Define Constants

In [30]:
```
data = pd.read_csv('meteo1.csv', sep=' ', header=None).values.astype(np.uint8)
```

In [31]:
```
N = data.shape[0]
T = data.shape[1]
K = num_stations = 3
W = num_weather_conds = 3
```

Utility Functions

In [32]:
```
# used to normalize log-distributions
def lognormalize(X):
    A = np.logaddexp.reduce(X, axis=1)
    return np.exp(X - A.reshape(-1, 1))
```

Initialisation

In [33]:
```
# compute the initial probability distribution for P(V1|h)
def compute_V1_vector(sequences):
    initial = np.zeros(shape=(3,1))
    for sequence in sequences:
        for i in range(3):
            if sequence[0]==i:
                initial[i,0] += 1
    return initial/initial.sum(axis=0)
```

In [34]:
```
# compute intial transition matrices A(0|, A(1|, A(2)
def compute_transition_matrix(sequences):
    freq_table = np.zeros((3, 3))
    for n in range(len(sequences)):
        for t in range(T - 1):
            freq_table[sequences[n, t+1], sequences[n, t]] += 1

    return freq_table / freq_table.sum(axis=0)
```

Compute Random Assignment

In [35]:
```
# randomly assign sequences to weather stations in order to generate initial parameters
init_assignments = np.random.choice([0, 1, 2], 500)
```

Compute Initial Parameters

In [36]:
```
# marginal
# p(h) = P(s = k)
P_pi = np.array([sum(init_assignments == 0), sum(init_assignments == 1), sum(init_assignments == 2
)])
P_pi = P_pi / P_pi.sum()

# transition matrix
# A(k)[vs[t], vs[t-1]] = P(v_t | v_t-1, s = k) = Ps(v_t | v_t-1)
A_0 = compute_transition_matrix(data[init_assignments == 0])
A_1 = compute_transition_matrix(data[init_assignments == 1])
A_2 = compute_transition_matrix(data[init_assignments == 2])
P_A = np.array([A_0, A_1, A_2])

# initial
# v1[k, v] = P(v1 = vs[0] | s=k)
v1_0 = compute_V1_vector(data[init_assignments == 0])
v1_1 = compute_V1_vector(data[init_assignments == 1])
v1_2 = compute_V1_vector(data[init_assignments == 2])
P_v1 = np.array([v1_0, v1_1, v1_2]).squeeze().T
```

EM Algorithm Implementation for Markov Mixture Model

Calculate E (E-step)

$E = P^{old}(s = k \mid \mathbf{v}_{1:T}^n) \propto P(s = k)P(\mathbf{v}_{1:T}^n \mid s = k) = P(s = k)[\prod_{t=1}^T P(v_t^n \mid \mathbf{v}_{t-1}^n, s = k) = P(s = k)[\prod_{t=1}^T P(v_t^n \mid \mathbf{v}_{t-1}^n, s = k)$

In [37]:
```
def log_E(n, k, pi, A, P_v1):
    vs = data[n]
    ans = np.log(pi[k])*P_v1[vs[0], k])
    for t in range(1, T):
        ans += np.log(A[k][vs[t], vs[t-1]])
    return ans

# compute E along with log likelihood
def compute_E(pi, A, P_v1):
    E_logged = np.zeros((N, K))
    total_loglike = 0
    for n in range(N):
        for k in range(K):
            E_logged[n, k] = log_E(n, k, pi, A, P_v1)
        n_loglike = np.log(np.sum(np.exp(E_logged[n, :])))
        total_loglike += n_loglike
    E = lognormalize(E_logged)
    print(total_loglike)
    return E, total_loglike
```

Parameter Updates (M-step)

$\pi^{new} = P^{new}(s = k) \propto \sum_{n=1}^N P^{old}(s = k \mid \mathbf{v}_{1:T}^n)$

In [38]:
```
def pi_new(E):
    pi_new = np.zeros(P_pi.shape)
    for k in range(len(P_pi)):
        for n in range(N):
            pi_new[k] += E[n, k]
    return pi_new/pi_new.sum()
```

$A_{new}^{(k)} = P^{new}(v_t = i \mid v_{t-1} = j, s = k) \propto \sum_{n=1}^N \left[P^{old}(s = k \mid \mathbf{v}_{1:T}^n) \sum_{t=1}^T \mathbb{I}[v_t^n = i] \mathbb{I}[v_{t-1}^n = j] \right]$

In [39]:
```
def A_new(E):
    # Too many loops, but it works!
    A_new = np.zeros(P_A.shape)
    for k in range(K):
        for n in range(N):
            vs = data[n]
            for i in range(W):
                for j in range(W):
                    for t in range(1, T):
                        A_new[k][vs[t], vs[t-1]] += (E[n, k]*(vs[t] == i)*(vs[t - 1] == j))
        A_new[k] /= A_new[k].sum(axis=0)
    return A_new
```

$P^{new}(v_t = i \mid s = k) \propto \sum_{n=1}^N P^{old}(s = k \mid \mathbf{v}_{1:T}^n) \mathbb{I}[v_t^n = i]$

In [40]:
```
def v1_new(E):
    v1_new = np.zeros(P_v1.shape)
    for k in range(K):
        for i in range(W):
            for n in range(N):
                vs = data[n]
                v1_new[vs[0], k] += E[n, k]*(vs[0] == i)

    return v1_new/v1_new.sum(axis=0)
```

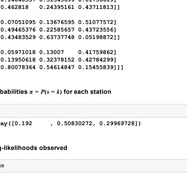
RUN EM Algorithm:

In [16]:
```
from tqdm import tqdm

A = P_A
v1 = P_v1
pi = P_pi

iterations = 15
epsilon = 0.001
logs = []

for iteration in tqdm(list(range(iterations))):
    # E-step
    E, log_likelihood = compute_E(pi, A, v1)

    # M-step
    A = A_new(E)
    v1 = v1_new(E)
    pi = pi_new(E)

    #append log-likelihood results
    logs.append(log_likelihood)

    # stop once sufficiently converged
    if (iteration > 2):
        if (logs[-1] - logs[-2] < epsilon:
            break
```

```
  0%|          | 0/15 [00:00<?, ?it/s]
-49419.03358731352
  7%|▋         | 1/15 [00:30<07:09, 30.71s/it]
-47969.14602251873
 13%|█▎        | 2/15 [00:59<06:33, 30.24s/it]
-45556.87511423374
 20%|██        | 3/15 [01:29<05:59, 29.98s/it]
-45260.41865246454
 27%|██▋       | 4/15 [01:57<05:25, 29.55s/it]
-45254.560557974655
 33%|███▎      | 5/15 [02:26<04:54, 29.45s/it]
-45254.51080764703
 40%|████      | 6/15 [02:57<04:26, 29.62s/it]
-45254.51039384207
```

In [23]:
```
import matplotlib.pyplot as plt
plt.xlabel('Iteration')
plt.ylabel('Log-likelihood')
plt.plot(logs)
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x171914efba8>]
```



In [24]:
```
import numpy as np
np.set_printoptions(precision=8)
np.set_printoptions(suppress=True)
```

Posterior probabilities of first 10 sequences, where the kth column corresponds to the kth weather station

In [25]:
```
print(E[:10])
```

```
[[1.         0.         0.        ]
 [0.         0.00019079 0.99980921]
 [0.         0.99999654 0.00000346]
 [0.         1.         0.        ]
 [0.         0.99999412 0.00000588]
 [0.         0.0000376  0.9999624 ]
 [1.         0.         0.        ]
 [0.         0.99999999 0.00000001]
 [0.         0.99999684 0.00000316]
 [0.         0.99999779 0.00000221]]
```

Posterior Transition Probability Matrices

In [20]:
```
print(A)
```

```
[[[0.38869863 0.2405914  0.54531568]
  [0.14848337 0.51545699 0.01756619]
  [0.462818   0.24395161 0.43711813]]

 [[0.07051095 0.13476595 0.51077572]
  [0.49465376 0.22585657 0.43723556]
  [0.43483529 0.63737748 0.05198872]]

 [[0.05971018 0.13007    0.41759862]
  [0.13950618 0.32378152 0.42784299]
  [0.80078364 0.54614847 0.15455839]]]
```

Probabilities $x = P(s = k)$ for each station

In [21]:
```
pi
```

Out[21]:
```
array([0.192    , 0.50830272, 0.29969728])
```

Log-likelihoods observed

In [22]:
```
logs
```

Out[22]:
```
[-49419.03358731352,
 -47969.14602251873,
 -45556.87511423374,
 -45260.41865246454,
 -45254.560557974655,
 -45254.51080764703,
 -45254.51039384207]
```

Question 2

Setup

In [1]:
```
import numpy as np
```

In [2]:
```
# by default, assume chess.txt is in same directory
data_path = 'chess.txt'
```

HMM Parameters

In [3]:
```
# Emission probability matrix E
E = np.array([[0.3, 0.6, 0.1], [0.5, 0, 0.5]])

# Initial transition probability matrix T
T_0 = np.array([[0.5, 0.8], [0.5, 0.2]])

# Hidden distribution at time t=1
h_1 = np.array([1., 0.])
```

Load Data

In [4]:
```
# v_1:10
vs_10 = [0, 1, 0, 2, 0, 2, 1, 0, 2, 0]

with open(data_path) as f:
    s = f.read()
    vs_1000 = list(map(int, s.strip().split('\n')))
```

ForwardBackwardSolver class

I choose to define a class that computes alpha, beta, and the marginal posterior and pairwise marginal posterior probabilities. Corresponding member functions are named in a self-evident manner, so please refer to those if requiring the computational code relating to this question!

Note: `FwdBwdSolver` returns normalized forward and backward passes (due to computational issues discussed in part e)), and we also use memoization to make our code more computationally efficient (~10s/it for non-memoized versus ~10it/s for memoized)).

In [5]:
```
class FwdBwdSolver():


    def __init__(self, E, vs, T, h_1):
        self.E = E
        self.vs = vs
        self.T = T
        self.h_1 = h_1

        # In the true spirit of dynamic programming, we'll
        # use memoization to speed things up when we can.
        self.alpha_memory = [np.nan]*len(vs)
        self.beta_memory = [np.nan]*len(vs)


    # forward pass
    # calculates (NORMALISED) p(h_t, v_1:t)
    def alpha(self, t):
        # zero index time internally
        t = t - 1
        if np.isnan(self.alpha_memory[t]).all():
            # alias for readability
            E = self.E
            vs = self.vs
            T = self.T
            h_1 = self.h_1

            if t == 0:
                p_star = E[:, vs[0]]*h_1
                self.alpha_memory[t] = p_star/p_star.sum()
            else:
                alpha_t_minus_1 = self.alpha(t) # we already subtracted 1
                p_star = E[:, vs[t]] * (T.dot(alpha_t_minus_1))

                self.alpha_memory[t] = p_star/p_star.sum()

        return self.alpha_memory[t]


    # backward pass
    # calculates (NORMALISED) p(v_t+1:T | h_t)
    def beta(self, t):
        if t == len(self.vs):
            return np.ones(2)
        if np.isnan(self.beta_memory[t]).all():
            # alias for readability
            E = self.E
            T = self.T
            vs = self.vs

            beta_t_plus_1 = self.beta(t + 1)
            p_star = beta_t_plus_1.dot(E[:, vs[t]].reshape(-1, 1) * T)
            self.beta_memory[t] = p_star/p_star.sum()

        return self.beta_memory[t]

    # calculate marginals P(h_t | v_1:T)
    def P_ht_GIVEN_vs(self, t):
        p_star = self.alpha(t)*self.beta(t)
        return p_star/p_star.sum()


    # calculate pairwise marginals P_(h_t, h_t+1 | v_1:T)
    # NOTE: rows correspond to h_t+1, cols to h_t (same as T matrix)
    def P_ht_ht_plus_1_GIVEN_vs(self, t):
        # alias for readability
        E = self.E
        vs = self.vs
        T = self.T

        a = self.alpha(t)
        b = self.beta(t+1)
        emiss = E[:, vs[t]]
        ans = np.zeros((2, 2))
        for i in [0, 1]:
            for j in [0, 1]:
                ans[i, j] = T[i, j]*a[j]*b[i]*emiss[i]

        return ans/ans.sum()


    # we update T and clear cached/memoized alpha and beta values
    def update_T(self, T_update):
        self.T = T_update
        self.alpha_memory = [np.nan]*len(self.vs)
        self.beta_memory = [np.nan]*len(self.vs)
```

Code Questions

c) Implement the forward-backward algorithm to compute the singleton marginals $P(h_t|\mathbf{v}_{1:T})$. Compute and report them for the [given sequence]

In [6]:
```
solver = FwdBwdSolver(E, vs_10, T_0, h_1)
for t in range(1, 10 + 1):
    dist = solver.P_ht_GIVEN_vs(t)
    print('t={} => ({}, {})'.format(t, dist[0], dist[1]))
```

```
t=1 => (1.0, 0.0)
t=2 => (1.0, 0.0)
t=3 => (0.5023923444976076, 0.4976076555023924)
t=4 => (0.29824561403508776, 0.7017543859649122)
t=5 => (0.7320574162679425, 0.2679425837320574)
t=6 => (0.17065390749601275, 0.8293460925039873)
t=7 => (1.0, 0.0)
t=8 => (0.4883720930232558, 0.5116279069767443)
t=9 => (0.34108527131782956, 0.6589147286821705)
t=10 => (0.5930232558139534, 0.4069767441860465)
```

d) Use the messages computed in the previous part to compute the pairwise marginals $P(h_2, h_{1:2}|\mathbf{v}_{1:T})$. Report $P(h_{1:2}, h_2|\mathbf{v}_{1:10})$ and $P(h_4, h_5|\mathbf{v}_{1:10})$ for the same observed sequence and transition matrix as above.

In [7]:
```
print('P(h_1, h_2 | v_1:T)=\n', solver.P_ht_ht_plus_1_GIVEN_vs(1))
print('\nP(h_4, h_5 | v_1:T)=\n', solver.P_ht_ht_plus_1_GIVEN_vs(4))
```

```
P(h_1, h_2 | v_1:T)=
 [[1. 0.]
 [0. 0.]]

P(h_4, h_5 | v_1:T)=
 [[0.15789474 0.57416268]
 [0.14035088 0.12759171]]
```

e) Implement the EM (Baum-Welch) algorithm. Describe any computational issues you encountered when implementing EM and how you solved them. Run the EM alorithm for the observed sequence $\mathbf{v}_{1:1000}$ in the file chess.csv starting from the [initial guess]. Report the number of iterations that it took your implementation to converge and the value of the learned matrix T.

Only code and results are reported here. For discussion on computational issues and computational efficiency, please refer to LaTex.

In [8]:
```
import time
from IPython.display import clear_output

print_time_delay = 1

# BAUM-WELCH IMPLEMENTATION (for updating T)

solver = FwdBwdSolver(E, vs_1000, T_0, h_1)
iterations = 100

# define initial difference matrix (just to keep track of convergence)
diff = np.zeros(T_0.shape)
# track aggregated difference in values for convergence graph
diffs = []

for iteration in range(iterations):
    # print so we can watch it converge :)
    print('iteration=', iteration)
    print('T_{}='.format(iteration))
    print(solver.T)
    print('Δ_{}='.format(iteration))
    print(diff)

    # update
    numerator = np.zeros(T_0.shape)
    denominator = np.zeros(h_1.shape)
    for t in range(1, len(solver.vs)):
        p_ht_ht_plus_1_given_vs = solver.P_ht_ht_plus_1_GIVEN_vs(t)
        p_ht_given_vs = solver.P_ht_GIVEN_vs(t)
        numerator += p_ht_ht_plus_1_given_vs
        denominator += p_ht_given_vs

    T_update = numerator/denominator
    diff = solver.T - T_update

    # take the difference for convergence as mean value of pairwise probability update
    diffs.append(np.abs(diff).sum()/4)
    solver.update_T(T_update)

    # for printing purposes
    time.sleep(print_time_delay)
    clear_output()
```

Value of learned matrix T:

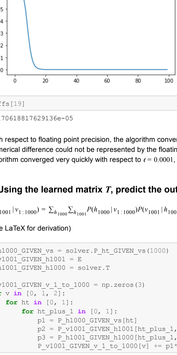
In [9]:
```
solver.T
```

Out[9]:
```
array([[0.83258179, 0.0723181 ],
       [0.16741821, 0.9276819 ]])
```

Convergence graph:

In [15]:
```
import matplotlib.pyplot as plt
```

In [16]:
```
plt.plot(diffs);
```



In [11]:
```
diffs[19]
```

Out[11]:
```
8.170618817629136e-05
```

With respect to floating point precision, the algorithm converged in ~71 iterations (i.e. this is the first point at which the numerical difference could not be represented by the floating point representation we're using). However, as can be seen, the algorithm converged very quickly with respect to $\epsilon = 0.0001$, at iteration ~19.

f) Using the learned matrix T, predict the outcome of the next game, $P(\mathbf{v}_{1001}|\mathbf{v}_{1:1000})$.

$P(\mathbf{v}_{1001}|\mathbf{v}_{1:1000}) = \sum_{\mathbf{h}_{1000}}\sum_{\mathbf{h}_{1001}} P(\mathbf{h}_{1000}|\mathbf{v}_{1:1000})P(\mathbf{v}_{1001}|\mathbf{h}_{1001})P(\mathbf{h}_{1001}|\mathbf{h}_{1000})$

(see LaTeX for derivation)

In [12]:
```
P_hl000_GIVEN_vs = solver.P_ht_GIVEN_vs(1000)
P_vl001_GIVEN_hl001 = E
P_hl001_GIVEN_hl000 = solver.T

P_vl001_GIVEN_v_1_to_1000 = np.zeros(3)
for v in [0, 1, 2]:
    for ht in [0, 1]:
        for ht_plus_1 in [0, 1]:
            p1 = P_hl000_GIVEN_vs[ht]
            p2 = P_vl001_GIVEN_hl001[ht_plus_1, v]
            p3 = P_hl001_GIVEN_hl000[ht_plus_1, ht]
            P_vl001_GIVEN_v_1_to_1000[v] += p1*p2*p3
```

In [13]:
```
P_vl001_GIVEN_v_1_to_1000/P_vl001_GIVEN_v_1_to_1000.sum()
```