

Introduction to Hidden Markov Models

Enno Hermann

Contents

1	Preamble	2
1.1	Formulas and definitions	2
1.1.1	Parameters of a HMM	2
1.1.2	Non-emitting initial and final states	2
1.1.3	Ergodic versus left-right HMMs	2
1.2	Values used throughout the experiments	3
2	Generating samples from HMMs	6
2.1	Experiment	6
2.2	Example	6
2.3	Questions	7
2.4	Answers	8
3	Pattern recognition with HMMs	9
3.1	Likelihood of an observation sequence given a HMM	9
3.1.1	Probability of a state sequence Q	9
3.1.2	Likelihood of an observation sequence X given a path Q	10
3.1.3	Joint likelihood of an observation sequence X and a path Q	10
3.1.4	Likelihood of observations with respect to a HMM	10
3.1.5	The Forward Algorithm	11
3.1.6	Questions	12
3.1.7	Answers	13
3.2	Bayesian classification	13
3.2.1	Question	13
3.2.2	Answer	14
3.3	Maximum Likelihood classification	14
3.3.1	Experiment	14
4	Optimal state sequence	15
4.1	Viterbi Algorithm	16
4.1.1	Summary	17
4.1.2	Questions	18
4.1.3	Answers	18
4.2	Experiments	18
4.2.1	Question	22
4.2.2	Answer	22
5	Training of HMMs	22
5.1	Questions	22
5.2	Answers	23

1 Preamble

1.1 Formulas and definitions

- A **Markov chain** or **process** is a sequence of events, usually called **states**, the probability of each of which is dependent only on the event immediately preceding it.
- A **hidden Markov model** (HMM) represents stochastic sequences as Markov chains where the states are not directly observed, but are associated with a probability density function (pdf). The generation of a random sequence is then the result of a random walk in the chain (i.e. the browsing of a random sequence of states $Q = \{q_1, \dots, q_T\}$) and of a draw (called an *emission*) at each visit of a state.

The sequence of states, which is the quantity of interest in speech recognition and in most of the other pattern recognition problems, can be observed only *through* the stochastic processes defined into each state (i.e. you must know the parameters of the pdfs of each state before being able to associate a sequence of states $Q = \{q_1, \dots, q_T\}$ to a sequence of observations $X = \{x_1, \dots, x_T\}$). The true sequence of states is therefore *hidden* by a first layer of stochastic processes. HMMs are *dynamic models*, in the sense that they are specifically designed to account for some macroscopic structure of the random sequences. In the previous lab, concerned with *Gaussian Statistics and Statistical Pattern Recognition*, random sequences of observations were considered as the result of a series of *independent* draws in one or several Gaussian densities. To this simple statistical modeling scheme, HMMs add the specification of some *statistical dependence* between the (Gaussian) densities from which the observations are drawn.

1.1.1 Parameters of a HMM

- The **emission probabilities** are the pdfs that characterize each state q_i , i.e. $p(x|q_i)$. To simplify the notations, they will be denoted $b_i(x)$. For practical reasons, they are usually Gaussian or mixtures of Gaussians, but the states could be parameterized in terms of any other kind of pdf (including discrete probabilities and artificial neural networks).
- The **transition probabilities** are the probability to go from a state i to a state j , i.e. $P(q_j|q_i)$. They are stored in matrices where each term $a_{i,j}$ denotes a probability $P(q_j|q_i)$.

1.1.2 Non-emitting initial and final states

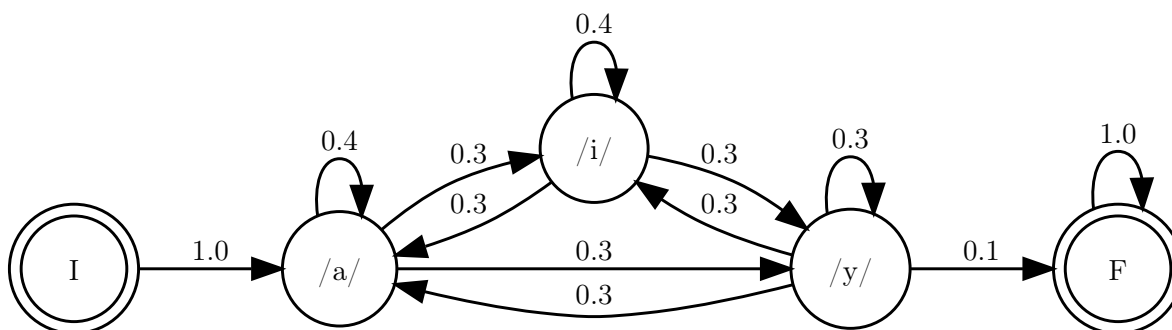
If a random sequence $X = \{x_1, \dots, x_T\}$ has a finite length T , the fact that the sequence begins or ends has to be modeled as two additional discrete events. In HMMs, this corresponds to the addition of two *non-emitting states*, the initial state and the final state. Since their role is just to model the *start* or *end* events, they are not associated with any emission probabilities.

The transitions starting from the initial state correspond to the modeling of an *initial state distribution* $P(I|q_j)$, which indicates the probability to start the state sequence with the emitting state q_j .

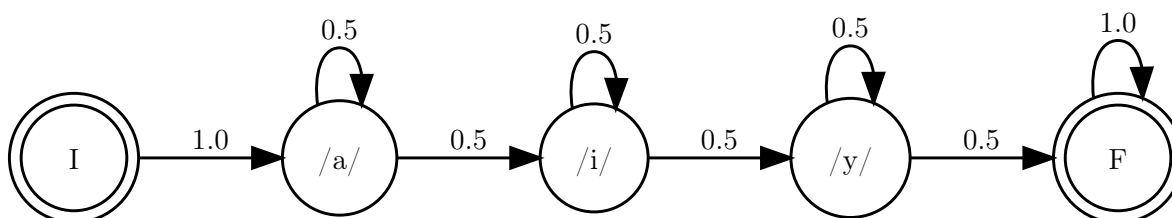
The final state usually has only one non-null transition that loops onto itself with a probability of 1 (it is an *absorbent state*), so that the state sequence gets "trapped" into it when it is reached.

1.1.3 Ergodic versus left-right HMMs

A HMM allowing for transitions from any emitting state to any other emitting state is called an **ergodic HMM**.



Alternately, an HMM where the transitions only go from one state to itself or to a unique follower is called a **left-right HMM**.



1.2 Values used throughout the experiments

The following 2-dimensional Gaussian densities will be used to model simulated vowel observations, where the considered features are the two first formants. They will be combined into Markov Models that will be used to model some observation sequences.

```
import numpy as np
from data import GAUSSIANS
for vowel, gaussian in GAUSSIANS.items():
    print(f"Gaussian('{vowel}'): mean={gaussian.mean.tolist()}, "
          f"cov={gaussian.cov.tolist()}")
```

```
Gaussian('0'): mean=[0.0, 0.0], cov=[[0.0, 0.0], [0.0, 0.0]]
Gaussian('a'): mean=[730.0, 1090.0], cov=[[1625.0, 5300.0], [5300.0, 53300.0]]
Gaussian('e'): mean=[530.0, 1840.0], cov=[[15025.0, 7750.0], [7750.0, 36725.0]]
Gaussian('i'): mean=[270.0, 2290.0], cov=[[2525.0, 1200.0], [1200.0, 36125.0]]
Gaussian('o'): mean=[570.0, 840.0], cov=[[2000.0, 3600.0], [3600.0, 20000.0]]
Gaussian('y'): mean=[440.0, 1020.0], cov=[[8000.0, 8400.0], [8400.0, 18500.0]]
```

Let's look at the resulting HMMs. We represent the initial state I and final state F with mean and variance of zero, but they don't actually emit any observations.

```
print(hmm1.labels)
print(hmm1.transitions)
for gaussian in hmm1.gaussians:
    print(f"mean={gaussian.mean.tolist()}, cov={gaussian.cov.tolist()}")
```

```
['I', '/a/', '/i/', '/y/', 'F']
[[0.  1.  0.  0.  0. ]
```

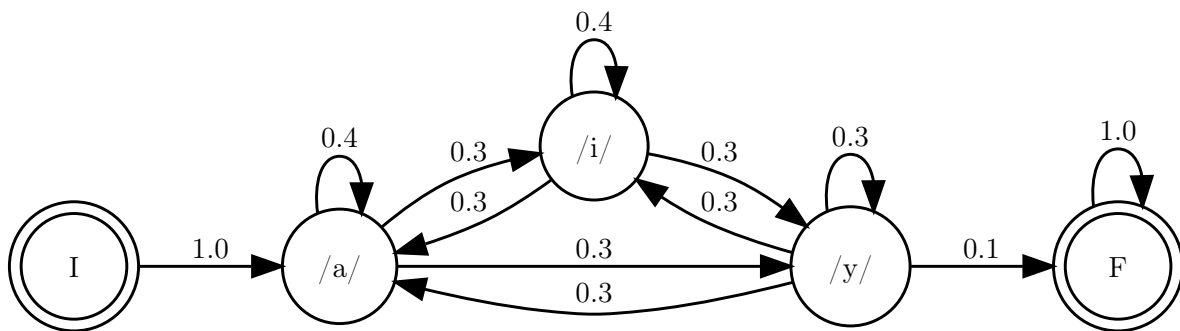
```

[0.  0.4 0.3 0.3 0. ]
[0.  0.3 0.4 0.3 0. ]
[0.  0.3 0.3 0.3 0.1]
[0.  0.  0.  0.  1. ]]
mean=[0.0, 0.0], cov=[[0.0, 0.0], [0.0, 0.0]]
mean=[730.0, 1090.0], cov=[[1625.0, 5300.0], [5300.0, 53300.0]]
mean=[270.0, 2290.0], cov=[[2525.0, 1200.0], [1200.0, 36125.0]]
mean=[440.0, 1020.0], cov=[[8000.0, 8400.0], [8400.0, 18500.0]]
mean=[0.0, 0.0], cov=[[0.0, 0.0], [0.0, 0.0]]

```

We can also visualise the transition matrix as a graph.

```
hmm1.plot()
```



Here are the remaining HMMs.

```
hmm2.pprint()
```

States: ['I', '/a/', '/i/', '/y/', 'F']

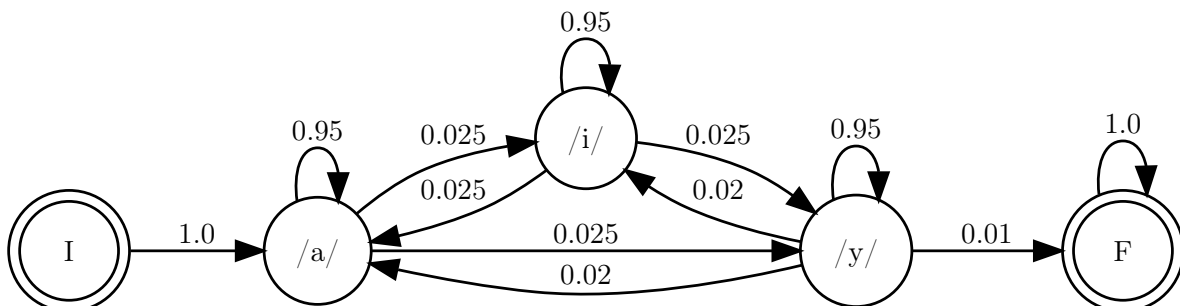
Transition matrix:

```

[[0.  1.  0.  0.  0. ]
 [0.  0.95 0.025 0.025 0. ]
 [0.  0.025 0.95 0.025 0. ]
 [0.  0.02 0.02 0.95 0.01 ]
 [0.  0.  0.  0.  1.  ]]

```

Graph:



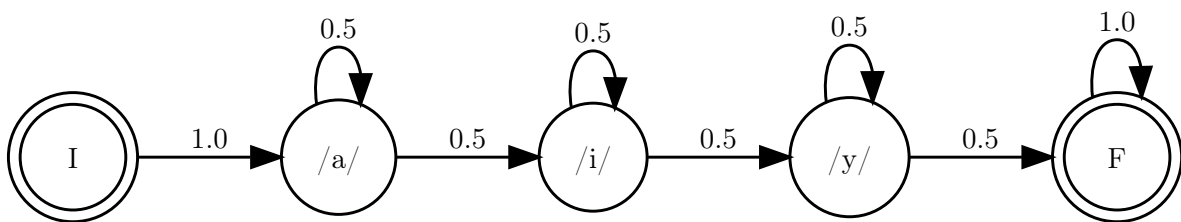
```
hmm3.pprint()
```

States: ['I', '/a/', '/i/', '/y/', 'F']

Transition matrix:

```
[[0.  1.  0.  0.  0. ]
 [0.  0.5 0.5 0.  0. ]
 [0.  0.  0.5 0.5 0. ]
 [0.  0.  0.  0.5 0.5]
 [0.  0.  0.  0.  1. ]]
```

Graph:



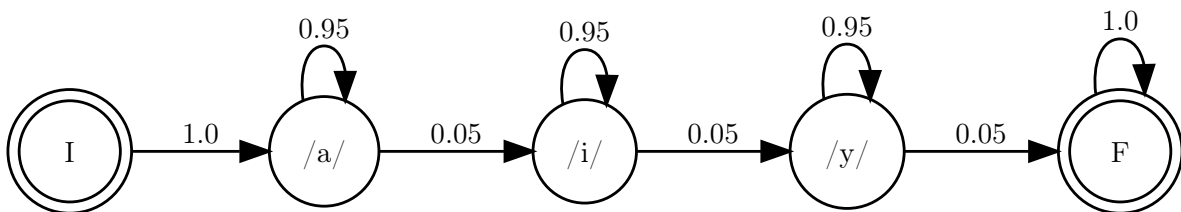
```
hmm4.pprint()
```

States: ['I', '/a/', '/i/', '/y/', 'F']

Transition matrix:

```
[[0.  1.  0.  0.  0. ]
 [0.  0.95 0.05 0.  0. ]
 [0.  0.  0.95 0.05 0. ]
 [0.  0.  0.  0.95 0.05]
 [0.  0.  0.  0.  1. ]]
```

Graph:



```
hmm5.pprint()
```

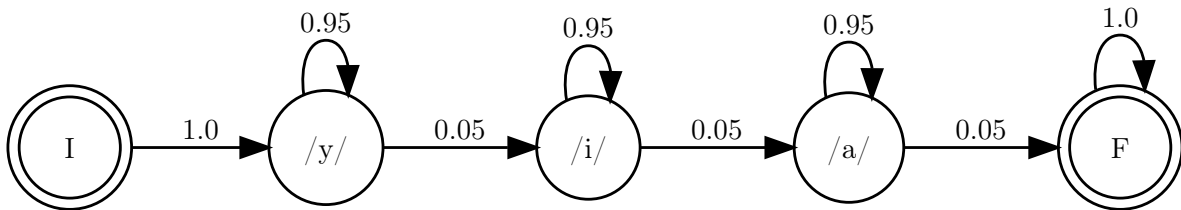
States: ['I', '/y/', '/i/', '/a/', 'F']

Transition matrix:

```
[[0.  1.  0.  0.  0. ]
 [0.  0.95 0.05 0.  0. ]]
```

```
[0.  0.  0.95 0.05 0.  ]
[0.  0.  0.  0.95 0.05]
[0.  0.  0.  0.  1.  ]]
```

Graph:



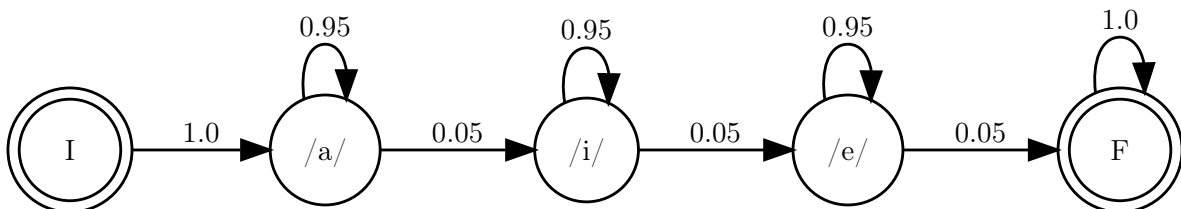
```
hmm6.pprint()
```

States: ['I', '/a/', '/i/', '/e/', 'F']

Transition matrix:

```
[[0.  1.  0.  0.  0.  ]
 [0.  0.95 0.05 0.  0.  ]
 [0.  0.  0.95 0.05 0.  ]
 [0.  0.  0.  0.95 0.05]
 [0.  0.  0.  0.  1.  ]]
```

Graph:



2 Generating samples from HMMs

2.1 Experiment

Generate a sample X coming from the Hidden Markov Models `hmm1`, `hmm2`, `hmm3` and `hmm4`. Use the `HMM.sample()` method to do several draws with each of these models and plot them.

2.2 Example

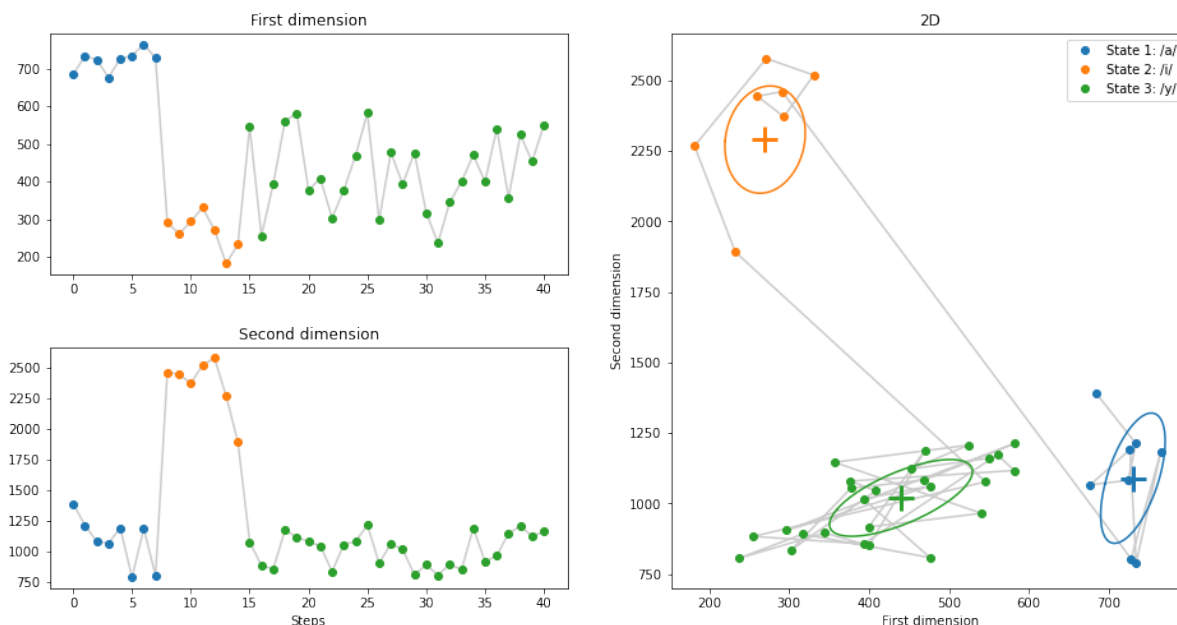
Draw a sample and plot the resulting sequence. The sequence is represented by a gray line where each point is overlaid with a colored dot. The different colors indicate the state from which any particular dot has been drawn.

The lefthand plots highlight the notion of a sequence of states associated with a sequence of observations. The 2-dimensional righthand plot highlights the spatial distribution of the observations and also shows the Gaussian distributions from which the samples for each state were drawn.

```
X, states, labels = hmm1.sample(plot=True)
print(X)
print(states)
print(labels)
```

Repeat this several times and also draw samples from the other models.

```
X, states, labels = hmm4.sample(plot=True)
```



2.3 Questions

1. How can you verify that a transition matrix is valid?
2. What is the effect of the different transition matrices on the sequences obtained during the current experiment? Hence, what is the role of the transition probabilities in the Markovian modeling framework?
3. What would happen if we didn't have a final state ?
4. In the case of HMMs with plain Gaussian emission probabilities, what quantities should be present in the complete parameter set Θ that specifies a particular model?
If the model is ergodic with N states (including the initial and final), and represents data of dimension D , what is the total number of parameters in Θ ?
5. Which type of HMM (ergodic or left-right) would you use to model words?

2.4 Answers

1. In a transition matrix A , the element A_{ij} specifies the probability to go from state i to state j . Hence, the values on row i specify the probability of all the possible transitions that start from state i . This set of transitions must be a complete set of discrete events. Hence, the terms of the i^{th} row of the matrix must sum up to 1. Similarly, the sum of all the elements of the matrix is equal to the number of states in the HMM.
2. The transition matrix of `hmm1` indicates that the probability of staying in a particular state is close to the probability of transitioning to another state. Hence, it allows for frequent jumps from one state to any other state. The observation variable therefore frequently jumps from one phoneme to any other, forming sharply changing sequences like `/a,i,a,y,y,i,a,y,y,.../`.
 Alternatively, the transition matrix of `hmm2` specifies high probabilities of staying in a particular state. Hence, it allows for more "stable" sequences, like `/a,a,a,y,y,i,i,i,i,y,y,.../`.
 Finally, the transition matrix of `hmm4` also fixes the order in which the states are browsed: the given probabilities force the observation variable to go through `/a/`, then to go through `/i/`, and finally to stay in `/y/`, e.g. `/a,a,a,i,i,i,y,y,y,y,.../`.
 Hence, the role of the transition probabilities is to introduce a temporal (or spatial) structure in the modeling of random sequences.
 Furthermore, the obtained sequences have variable lengths: the transition probabilities implicitly model a variability in the duration of the sequences. As a matter of fact, different speakers or different speaking conditions introduce a variability in the phoneme or word durations. In this respect, HMMs are particularly well adapted to speech modeling.
3. If we didn't have a final state, the model would wander from state to state indefinitely, and necessarily correspond to sequences of infinite length.

```
def validate_transition_matrix(hmm: HMM) -> bool:
    """Ensure that each row in the transition matrix sums to one."""
    row_sums = np.sum(hmm.transitions, axis=1)
    return np.allclose(row_sums, np.ones(hmm.n_states))

for hmm in [hmm1, hmm2, hmm3, hmm4, hmm5, hmm6]:
    assert validate_transition_matrix(hmm)
```


5. Words are made of ordered sequences of phonemes: /h/ is followed by /e/ and then by /l/ in the word "hello". Each phoneme can in turn be considered as a particular random process (possibly Gaussian). This structure can be adequately modeled by a left-right HMM. In "real world" speech recognition, the phonemes themselves are often modeled as left-right HMMs rather than plain Gaussian densities (e.g. to model separately the beginning, then the stable middle part of the phoneme and finally the end of it). Words are then represented by large HMMs made of concatenations of smaller phonetic HMMs.
- Hence, in this case, the total number of parameters is $(N - 2) \times (N + D \times (D + 1))$. Note that this number grows exponentially with the number of states and the dimension of the data.
- $(N - 2)$ emitting states where each pdf is characterized by a D dimensional mean and a $D \times D$ covariance matrix.
 - $(N - 2) \times (N - 2)$ transitions, plus $(N - 2)$ initial state probabilities and $(N - 2)$ probabilities to go to the final state;
- In the case of an ergodic HMM with N emitting states and Gaussian emission probabilities, we have:
- The initial state distribution is sometimes modeled as an additional parameter instead of being represented in the transition matrix.
- the parameters of the Gaussian densities characterizing each state, i.e. the means μ_i and the variances Σ_i ;
 - the transition probabilities A ;
4. In the case of HMMs with Gaussian emission probabilities, the parameter set Θ comprises:

3 Pattern recognition with HMMs

3.1 Likelihood of an observation sequence given a HMM

In the previous section, we have generated some stochastic observation sequences from various HMMs. Now, it is useful to study the reverse problem, namely: given a new observation sequence and a set of models, which model explains best the sequence, or in other terms which model gives the highest likelihood to the data?

To solve this problem, it is necessary to compute $p(X|\Theta)$, i.e. the likelihood of an observation sequence given a model.

3.1.1 Probability of a state sequence Q

The probability of a state sequence $Q = \{q_1, \dots, q_T\}$ coming from a HMM with parameters Θ corresponds to the product of the transition probabilities from one state to the following:

$$P(Q|\Theta) = \prod_{t=1}^{T-1} a_{t,t+1} = a_{1,2} \cdot a_{2,3} \cdots a_{T-1,T}$$

In practice we will do the computations in log space to avoid numerical underflow:

$$\log P(Q|\Theta) = \sum_{t=1}^{T-1} \log a_{t,t+1} = \log a_{1,2} + \log a_{2,3} \cdots \log a_{T-1,T}$$

```

hmm = hmm3
X, states, labels = hmm.sample()
print("States:", states)

from_states = states[:-1] # Row indices into hmm3.transitions
to_states = states[1:]     # Column indices
log_a = hmm.log_transitions[from_states, to_states]
print("Transition log probs:", log_a)

log_P_Q = sum(log_a)
print("log P(Q):", log_P_Q)

```

3.1.2 Likelihood of an observation sequence X given a path Q

Given an observation sequence $X = \{x_1, x_2, \dots, x_T\}$ and a state sequence $Q = \{q_1, \dots, q_T\}$ (of the same length) determined from a HMM with parameters Θ , the likelihood of X along the path Q is equal to:

$$p(X|Q, \Theta) = \prod_{i=1}^T p(x_i|q_i, \Theta) = b_1(x_1) \cdot b_2(x_2) \cdots b_T(x_T)$$

i.e. it is the product of the emission probabilities computed along the considered path.

In the previous lab, we had learned how to compute the likelihood of a single observation with respect to a Gaussian model. This method can be applied here, for each term x_i , if the states contain Gaussian pdfs.

```

log_p_X_given_Q = sum(np.log(hmm.gaussians[state].pdf(x))
                      for x, state in zip(X, states[1:-1]))

print("log p(X|Q):", log_p_X_given_Q)

```

3.1.3 Joint likelihood of an observation sequence X and a path Q

The probability that X and Q occur simultaneously, $p(X, Q|\Theta)$, decomposes into a product of the two quantities defined previously:

$$p(X, Q|\Theta) = p(X|Q, \Theta)P(Q|\Theta)$$

```

print("log p(X,Q):", log_p_X_given_Q + log_P_Q)

```

3.1.4 Likelihood of observations with respect to a HMM

The likelihood of an observation sequence $X = \{x_1, x_2, \dots, x_T\}$ with respect to a Hidden Markov Model with parameters Θ expands as follows:

$$p(X|\Theta) = \sum_{\text{every possible } Q} p(X, Q|\Theta)$$

i.e. it is the sum of the joint likelihoods of the sequence over all possible state sequence allowed by the model.

3.1.5 The Forward Algorithm

In practice, the enumeration of every possible state sequence is infeasible even for small values of N and T . Nevertheless, $p(X|\Theta)$ can be computed in a recursive way (dynamic programming) by the **forward algorithm**. This algorithm defines a forward variable $\alpha_t(i)$ corresponding to:

$$\alpha_t(i) = p(x_1, x_2, \dots, x_t, q^t = q_i | \Theta)$$

i.e. $\alpha_t(i)$ is the probability of having observed the partial sequence $\{x_1, x_2, \dots, x_t\}$ and being in the state i at time t (event denoted q_i^t in the course), given the parameters Θ . For a HMM with N states (where states 1 and N are the non-emitting initial and final states, and states $2 \dots N-1$ are emitting), $\alpha_t(i)$ can be computed recursively as follows:

1. Initialization

$$\alpha_1(i) = a_{1,i} \cdot b_i(x_1), \quad 2 \leq i \leq N-1$$

where $a_{1,i}$ are the transitions from the initial non-emitting state to the emitting states with pdfs $b_{i,i=2 \dots N-1}(x)$. Note that $b_1(x)$ and $b_N(x)$ do not exist since they correspond to the non-emitting initial and final states.

```
from data import X1, X2, X3, X4, X5, X6

# Let's pick a fixed sequence defined in `data.py` to make
# the results reproducible
hmm = hmm3
X = X2
print(X)
```

```
# First precompute the b(x), i.e. pdfs, for all observations
# and emitting states
log_bs = np.zeros((len(X), hmm.n_states))
for state in range(1, hmm.n_states - 1):
    log_bs[:,state] = np.log(hmm.gaussians[state].pdf(X))
print(log_bs)
```

```
# Then compute the initial alphas
alphas = np.ones((len(X), hmm.n_states)) * -np.inf
alphas[0] = hmm.log_transitions[0] + log_bs[0]
print(alphas)
```

2. Recursion

$$\alpha_{t+1}(j) = \left[\sum_{i=2}^{N-1} \alpha_t(i) \cdot a_{i,j} \right] b_j(x_{t+1}), \quad \begin{matrix} 1 \leq t \leq T \\ 2 \leq j \leq N-1 \end{matrix}$$

```
# We will show how to compute the alphas in 2 different ways
alphas_2 = alphas.copy()

# Basic Python way with 3 for-loops
for t in range(1, len(X)):
    for j in range(1, hmm.n_states - 1):
```

```

log_as = -np.inf
for i in range(1, hmm.n_states - 1):
    log_as = np.logaddexp(
        log_as, alphas[t-1, i] + hmm.log_transitions[i, j])
alphas[t, j] = log_as + log_bs[t, j]

# Remove the innermost loop thanks to Numpy
for t in range(1, len(X)):
    for j in range(1, hmm.n_states - 1):
        alphas_2[t, j] = np.logaddexp.reduce(
            alphas_2[t-1] + hmm.log_transitions[:, j]) + log_bs[t, j]

# Check that they are indeed the same
print(alphas)
assert np.allclose(alphas, alphas_2)

```

3. Termination

$$p(X|\Theta) = \left[\sum_{i=2}^{N-1} \alpha_T(i) \cdot a_{i,N} \right]$$

i.e. at the end of the observation sequence, sum the probabilities of the paths converging to the final state N . (For more detail about the forward procedure, refer to Lawrence Rabiner's Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition).

```

log_p_X = np.logaddexp.reduce(
    alphas[-1] + hmm.log_transitions[:, hmm.n_states - 1])
print("p(X|hmm):", log_p_X)

```

This procedure raises a very important implementation issue. As a matter of fact, the computation of the α_t vector consists in products of a large number of values that are less than 1 (in general, *significantly* less than 1). Hence, after a few observations ($t \approx 10$), the values of α_t head exponentially to 0, and the floating point arithmetic precision is exceeded (even in the case of double precision arithmetics). Two solutions exist for that problem. One consists in scaling the values and undo the scaling at the end of the procedure: see Rabiner's tutorial for more explanations. The other solution consists in using log-likelihoods and log-probabilities, and to compute $\log p(X|\Theta)$ instead of $p(X|\Theta)$.

3.1.6 Questions

1. The following formula can be used to compute the log of a sum given the logs of the sum's arguments:

$$\log(a + b) = f(\log a, \log b) = \log a + \log \left(1 + e^{(\log b - \log a)} \right)$$

Prove its validity.

Naturally, one has the choice between using $\log(a + b) = \log a + \log \left(1 + e^{(\log b - \log a)} \right)$ or $\log(a + b) = \log b + \log \left(1 + e^{(\log a - \log b)} \right)$, which are equivalent in theory. If $\log a > \log b$, which version leads to the most precise implementation?

2. Express the log version of the forward recursion. (Don't fully develop the log of the sum in the recursion step, just call it "logsum": $\sum_{i=1}^N x_i \xrightarrow{\log} \text{logsum}_{i=1}^N(\log x_i)$.) In addition to the arithmetic precision issues, what are the other computational advantages of the log version?

3.1.7 Answers

These two points show that once the theoretic barrier is crossed in the study of a particular statistical model, the importance of the implementation issues must not be neglected.

In addition to the precision issues, this version transforms the products into sums, which is more computationally efficient. Furthermore, if the emission probabilities are Gaussians, the computation of the log-likelihoods $\log(b_j(x_t))$ eliminates the computation of the Gaussians' exponential (see the previous lab).

- **Termination**

$$\log p(X|\Theta) = \left[\log \sum_{i=1}^N \alpha_{i,j}^{(t)} \right] + \log b_j(x_{t+1}), \quad 1 \leq j \leq T, \quad 1 \leq i \leq N$$
- **Recursion**

$$\alpha_{i,j}^{(t+1)} = \left[\log \sum_{i=1}^N \alpha_{i,j}^{(t)} + \log b_j(x_{t+1}) \right], \quad 1 \leq j \leq T, \quad 1 \leq i \leq N$$
- **Initialization**

$$\alpha_{i,1}^{(1)} = \log a_{i,1} + \log b_1(x_1), \quad 1 \leq i \leq N$$

The computation of the exponential overflows the double precision arithmetics for big values (≈ 700) earlier than for small values. Similarly, the implementations of the exponential operation are generally more precise for small values than for big values (since an error on the input term is exponentially amplified). Hence, if $\log a > \log b$, the first version ($\log(a+b) = \log a + \log(1 + e^{(\log b - \log a)})$) is more precise since in this case $|\log b - \log a|$ is small. If $\log a < \log b$, it is better to swap the terms (i.e. to use the second version). In practice, you would use an existing implementation that handles this automatically, like `np.logaddexp()`.

$$\log(a+b) = \log a + \log\left(1 + e^{(\log b - \log a)}\right) \quad \square$$

$$a + b = e^{\log a} + e^{\log b} = e^{\log a} \left(1 + e^{(\log b - \log a)}\right)$$

$$a = e^{\log a} \quad ; \quad b = e^{\log b}$$

1. Proof:

3.2 Bayesian classification

3.2.1 Question

The forward recursion allows us to compute the likelihood of an observation sequence with respect to a HMM. Hence, given a sequence of features, we are able to find the most likely generative model in a Maximum Likelihood sense. What additional quantities and assumptions do we need to perform a true Bayesian classification rather than a Maximum Likelihood classification of the sequences?

Which additional condition makes the result of Bayesian classification equivalent to the result of ML classification?

3.2.2 Answer

P(Θ_i) can be determined by counting the probability of occurrence of each model (word or phoneme) in a database covering the vocabulary to recognize (see the previous lab). If every model has the same prior probability, then Bayesian classification becomes equivalent to ML classification.

$$P(\Theta_i|X, \Theta) = \frac{P(X|\Theta_i)P(\Theta_i)}{P(X|\Theta)}$$

To perform a Bayesian classification, we need the prior probabilities P(Θ_i|Θ) of each model. In addition, we can assume that all the observation sequences are equi-probable:

3.3 Maximum Likelihood classification

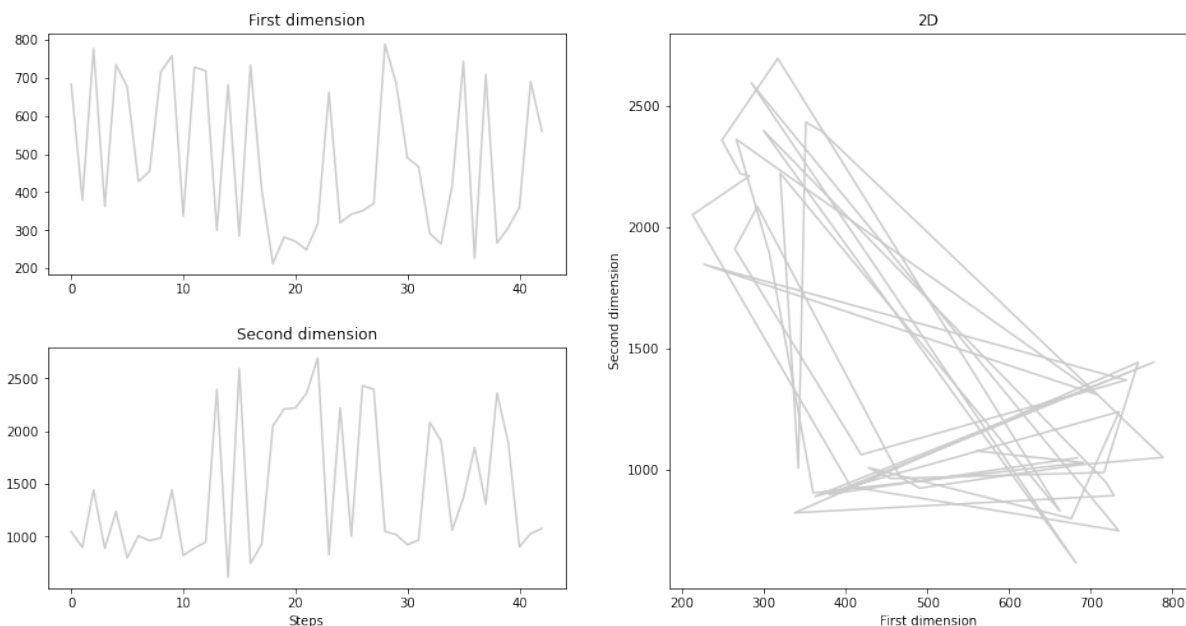
In practice, for speech recognition, it is very often assumed that all the model priors are equal (i.e. that the words or phonemes to recognize have equal probabilities of occurring in the observed speech). Hence, the speech recognition task consists mostly in performing the Maximum Likelihood classification of acoustic feature sequences. For that purpose, we must have of a set of HMMs that model the acoustic sequences corresponding to a set of phonemes or a set of words. These models can be considered as "stochastic templates". Then, we associate a new sequence to the most likely generative model. This part is called the **decoding** of the acoustic feature sequences.

3.3.1 Experiment

Classify the sequences **X1**, **X2**, ..., **X6**, given in the file **data.py**, in a maximum likelihood sense with respect to the six Markov models defined above. Use the method **HMM.forward(X)** to compute the log-forward recursion expressed in the previous section. Store the results in the array **log_prob** (they will be used in the next section) and note them in the table below.

Sequence	$\log p(X \Theta_1)$	$\log p(X \Theta_2)$	$\log p(X \Theta_3)$	$\log p(X \Theta_4)$	$\log p(X \Theta_5)$	$\log p(X \Theta_6)$	Most likely model
X1							
X2							
X3							
X4							
X5							
X6							

```
hmm1.plot_sample(X1)
hmm1.forward(X1)
```



-559.3878877787542

Filling the `log_prob` array can be done automatically with the help of loops:

```
log_prob = np.zeros((6, 6))
for i, X in enumerate([X1, X2, X3, X4, X5, X6]):
    for j, hmm in enumerate([hmm1, hmm2, hmm3, hmm4, hmm5, hmm6]):
        log_prob[i, j] = hmm.forward(X)

print(log_prob.round(2))
```

4 Optimal state sequence

In speech recognition and several other pattern recognition applications, it is useful to associate an "optimal" sequence of states to a sequence of observations, given the parameters of a model. For instance, in the case of speech recognition, knowing which frames of features "belong" to which state allows to locate the word boundaries across time. This is called the *alignment* of acoustic feature sequences.

A "reasonable" optimality criterion consists in choosing the state sequence (or *path*) that has the maximum likelihood with respect to a given model. This sequence can be determined recursively via the **Viterbi algorithm**. This algorithm makes use of two variables:

- The *highest* likelihood $\delta_t(i)$ along a *single* path among all the paths ending in state i at time t :

$$\delta_t(i) = \max_{q_1, q_2, \dots, q_{t-1}} p(q_1, q_2, \dots, q_{t-1}, q^t = q_i, x_1, x_2, \dots, x_t | \Theta)$$

- A variable $\psi_t(i)$ which allows to keep track of the "best path" ending in state i at time t :

$$\psi_t(i) = \arg \max_{q_1, q_2, \dots, q_{t-1}} p(q_1, q_2, \dots, q_{t-1}, q^t = q_i, x_1, x_2, \dots, x_t | \Theta)$$

Note that these variables are vectors of $(N - 2)$ elements, $(N - 2)$ being the number of emitting states. With the help of these variables, the algorithm takes the following steps:

4.1 Viterbi Algorithm

1. Initialization

$$\begin{aligned}\delta_1(i) &= a_{1,i} \cdot b_i(x_1), \quad 2 \leq i \leq N-1 \\ \psi_1(i) &= 0\end{aligned}$$

where, again, $a_{1,i}$ are the transitions from the initial non-emitting state to the emitting states with pdfs $b_{i,i=2\dots N-1}(x)$, and where $b_1(x)$ and $b_N(x)$ do not exist since they correspond to the non-emitting initial and final states.

```
hmm = hmm3
X = X2

# First precompute the b(x), i.e. pdfs, for all observations
# and emitting states
log_bs = np.zeros((len(X), hmm.n_states))
for state in range(1, hmm.n_states - 1):
    log_bs[:, state] = np.log(hmm.gaussians[state].pdf(X))
print(log_bs)
```

```
# Compute the initial deltas
deltas = np.ones((len(X), hmm.n_states)) * -np.inf
deltas[0] = hmm.log_transitions[0] + log_bs[0]
print(deltas)

# Initialize the backpointers
pointers = np.zeros((len(X), hmm.n_states), dtype=int)
print(pointers)
```

2. Recursion

$$\begin{aligned}\delta_{t+1}(j) &= \max_{2 \leq i \leq N-1} [\delta_t(i) \cdot a_{i,j}] \cdot b_j(x_{t+1}), \quad 1 \leq t \leq T-1 \\ &\quad 2 \leq j \leq N-1 \\ \psi_{t+1}(j) &= \arg \max_{2 \leq i \leq N-1} [\delta_t(i) \cdot a_{i,j}], \quad 1 \leq t \leq T-1 \\ &\quad 2 \leq j \leq N-1\end{aligned}$$

Optimal policy is composed of optimal sub-policies: find the path that leads to a maximum likelihood considering the best likelihood at the previous step and the transitions from it; then multiply by the current likelihood given the current state. Hence, the best path is found by induction.

```
for t in range(1, len(X)):
    for j in range(1, hmm.n_states - 1):
        deltas[t, j] = np.max(
            deltas[t-1] + hmm.log_transitions[:, j]) + log_bs[t, j]
        pointers[t, j] = np.argmax(
            deltas[t-1] + hmm.log_transitions[:, j])

print(deltas)
print(pointers)
```


3. Termination

$$p^*(X|\Theta) = \max_{2 \leq i \leq N-1} [\delta_T(i) \cdot a_{i,N}]$$
$$q_T^* = \arg \max_{2 \leq i \leq N-1} [\delta_T(i) \cdot a_{i,N}]$$

Find the best likelihood when the end of the observation sequence is reached, given that the final state is the non-emitting state N .

```
log_p_vit_X = np.max(
    deltas[-1] + hmm.log_transitions[:, hmm.n_states - 1])
print("p*(X|hmm):", log_p_vit_X)
```

```
# Determine from which state the final state was reached
path = np.zeros((len(X)), dtype=int)
path[-1] = np.argmax(
    deltas[-1] + hmm.log_transitions[:, hmm.n_states - 1])
print(path)
```

4. Backtracking

$$Q^* = \{q_1^*, \dots, q_T^*\} \quad \text{so that} \quad q_t^* = \psi_{t+1}(q_{t+1}^*), \quad t = T-1, T-2, \dots, 1$$

Read (decode) the best sequence of states from the ψ_t vectors. Remember that $\psi_t(j)$ stores the state from which we came if the best sequence goes through state j at time t . To get the path, we therefore just need to follow the backpointers in reverse order.

```
for t in range(len(X) - 2, -1, -1):
    path[t] = pointers[t + 1, path[t + 1]]

print(path)
```

Additionally, the state sequence will always include the initial state at the beginning and the final state at the end.

4.1.1 Summary

Hence, the Viterbi algorithm delivers *two* useful results, given an observation sequence $X = \{x_1, \dots, x_T\}$ and a model Θ :

- The selection, among all the possible paths in the considered model, of the *best path* $Q^* = \{q_1^*, \dots, q_T^*\}$, which corresponds to the state sequence giving a maximum of likelihood to the observation sequence X ;
- The *likelihood along the best path*, $p(X, Q^*|\Theta) = p^*(X|\Theta)$. As opposed to the forward procedure, where all the possible paths are considered, the Viterbi computes a likelihood along the best path only.

(For more detail about the Viterbi algorithm, refer to Lawrence Rabiner's Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition).

4.1.2 Questions

1. From an algorithmic point of view, what is the main difference between the computation of the δ variable in the Viterbi algorithm and that of the α variable in the forward procedure?
2. Give the log version of the Viterbi algorithm.

4.1.3 Answers

$$q_1^*, \dots, q_{T-1}^*, q_T^* = \arg \max_{q_1, \dots, q_T} \prod_{t=1}^T p(q_t | q_{1:t-1}^*)$$

• *Backtracking*

$$\begin{aligned} p_{i,j}^* &= \arg \max_{i,j} \left[\delta_{(log)}^T(i,j) + \log p_{i,j} \right] \\ p_{i,j}^* &= \arg \max_{i,j} \left[\delta_{(log)}^T(i,j) + \log p_{i,j} \right] \end{aligned}$$

• *Termination*

$$\begin{aligned} p_{i,j}^* &= \arg \max_{i,j} \left[\delta_{(log)}^T(i,j) + \log p_{i,j} \right] \\ p_{i,j}^* &= \arg \max_{i,j} \left[\delta_{(log)}^T(i,j) + \log p_{i,j} \right] \end{aligned}$$

• *Recursion*

$$\begin{aligned} p_{i,j}^* &= \arg \max_{i,j} \left[\delta_{(log)}^T(i,j) + \log p_{i,j} \right] \\ p_{i,j}^* &= \arg \max_{i,j} \left[\delta_{(log)}^T(i,j) + \log p_{i,j} \right] \end{aligned}$$

• *Initialization*

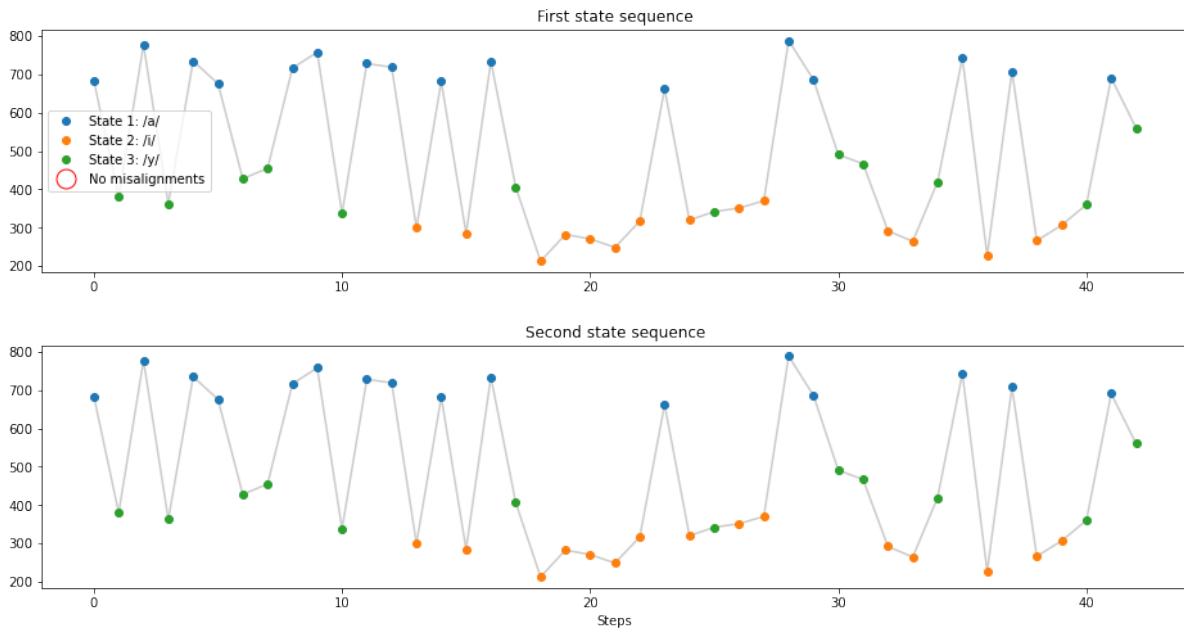
1. The sums that were appearing in the computation of α become max operations in the computation of δ . Hence, the Viterbi algorithm takes less computational power than the forward algorithm.

4.2 Experiments

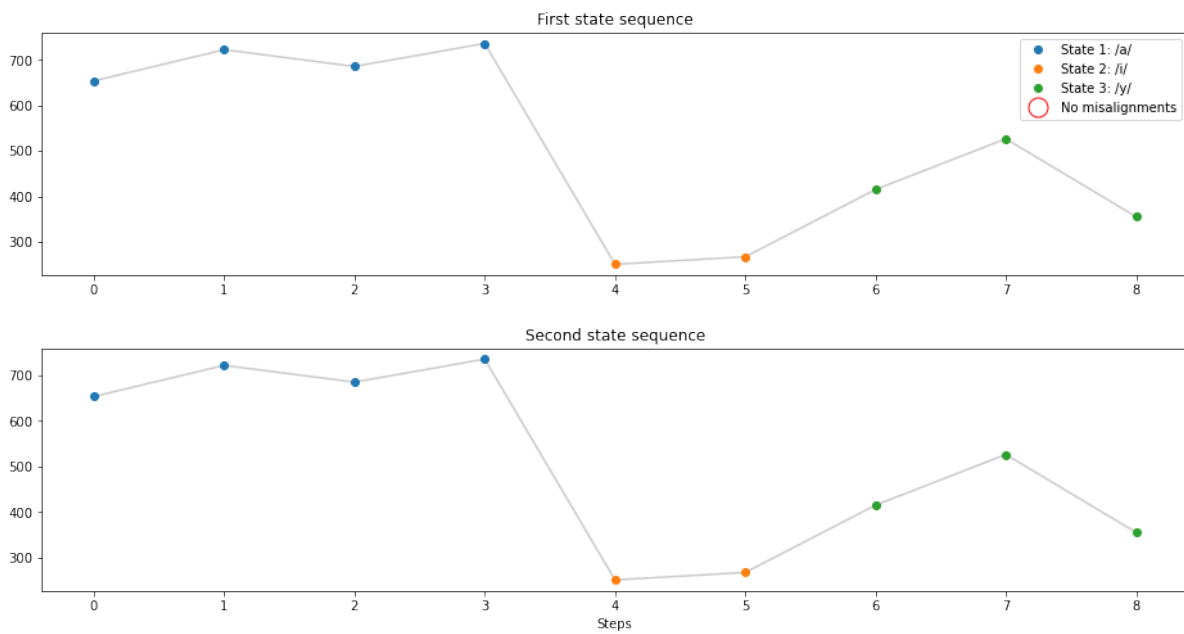
- Use the function `HMM.viterbi(X)` to find the best path of the sequences X_1, \dots, X_6 with respect to the most likely model found with the forward algorithm (i.e. X_1 : `hmm1`, X_2 : `hmm3`, X_3 : `hmm5`, X_4 : `hmm4`, X_5 : `hmm6` and X_6 : `hmm2`). Compare with the state sequences ST_1, \dots, ST_6 originally used to generate X_1, \dots, X_6 (use the function `HMM.compare_sequences(X, S1, S2)`, which provides a view of the first dimension of the observations as a time series, and allows to compare the original alignment to the Viterbi solution).

```
from data import ST1, ST2, ST3, ST4, ST5, ST6

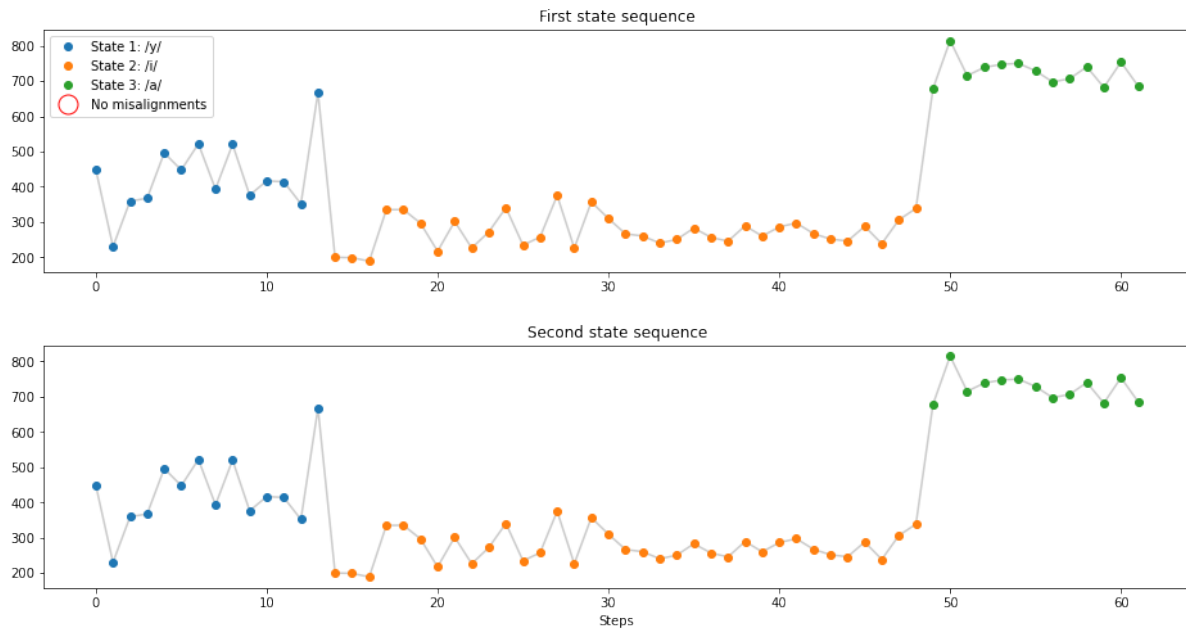
best_states, log_viterbi = hmm1.viterbi(X1)
hmm1.compare_sequences(X1, ST1, best_states)
```



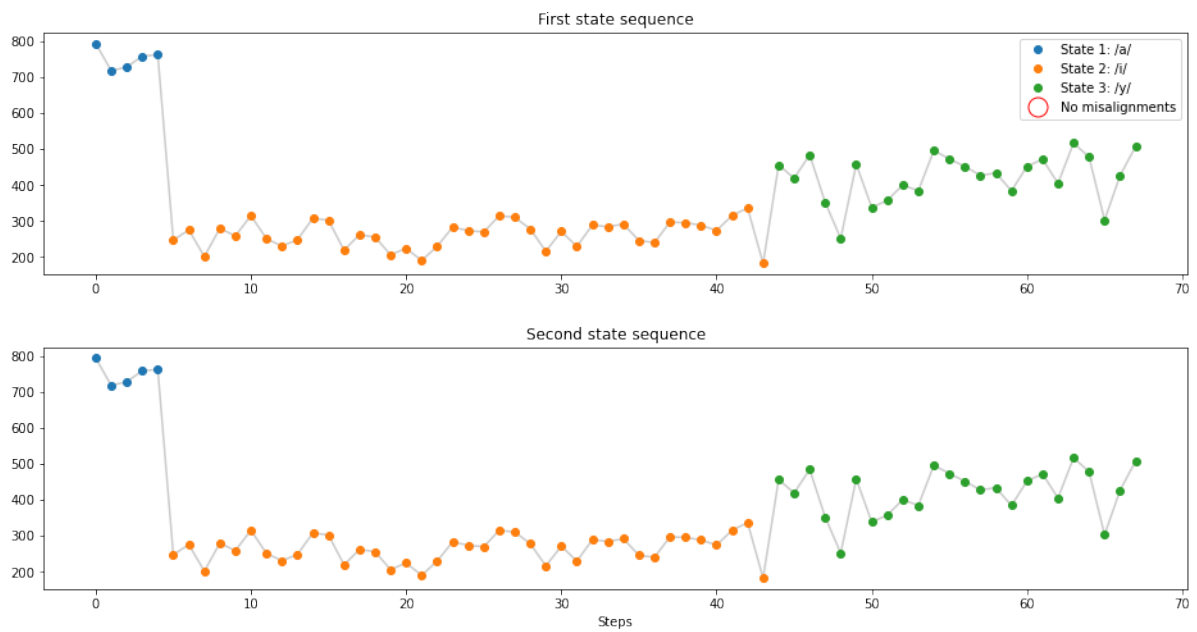
```
best_states, log_viterbi = hmm3.viterbi(X2)
hmm3.compare_sequences(X2, ST2, best_states)
```



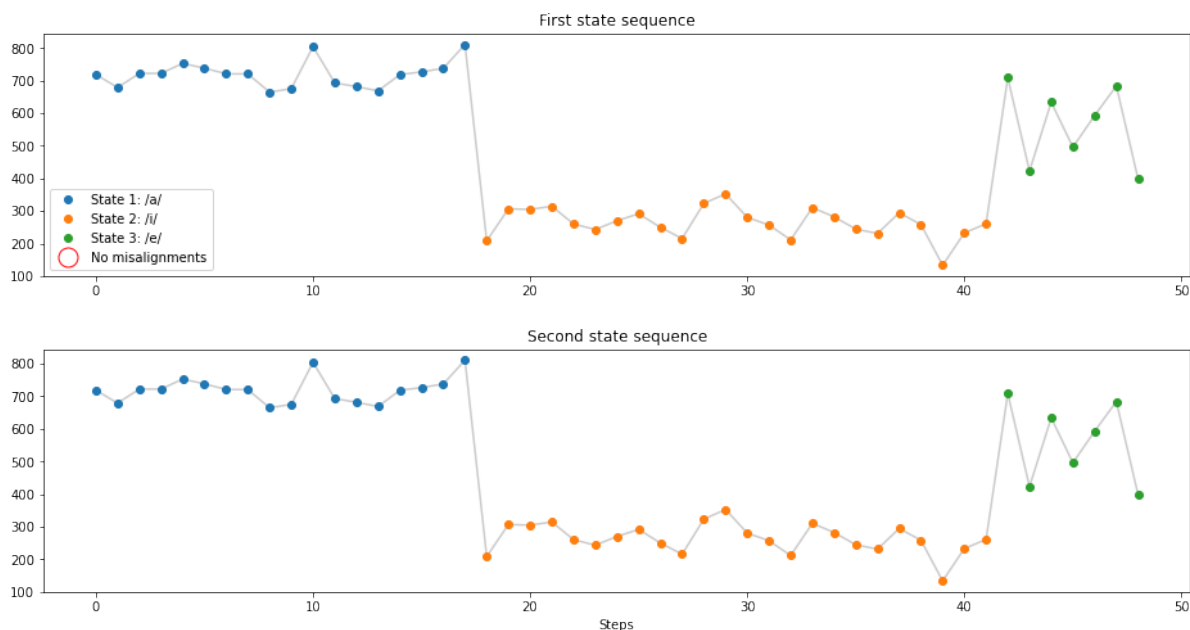
```
best_states, log_viterbi = hmm5.viterbi(X3)
hmm5.compare_sequences(X3, ST3, best_states)
```



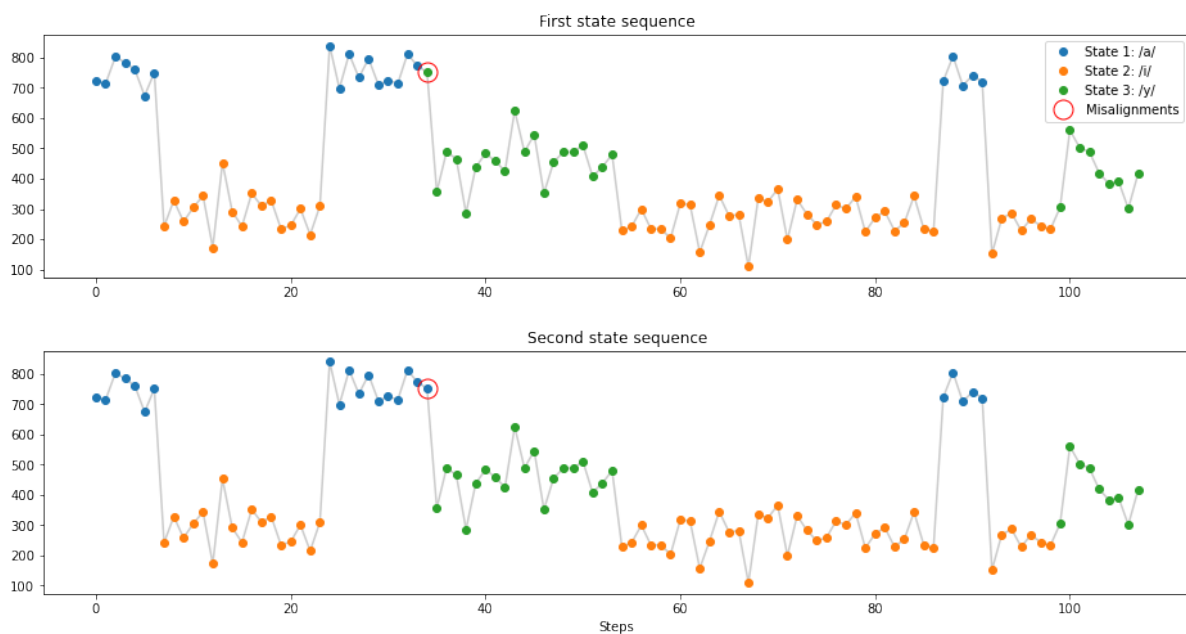
```
best_states, log_viterbi = hmm4.viterbi(X4)
hmm4.compare_sequences(X4, ST4, best_states)
```



```
best_states, log_viterbi = hmm6.viterbi(X5)
hmm6.compare_sequences(X5, ST5, best_states)
```



```
best_states, log_viterbi = hmm2.viterbi(X6)
hmm2.compare_sequences(X6, ST6, best_states)
```



- Use the function `HMM.viterbi(X)` to compute the probabilities of the sequences X_1, \dots, X_6 along the best paths with respect to each model $\Theta_1, \dots, \Theta_6$. Note your results below. Compare with the log-likelihoods obtained previously with the forward algorithm.

Sequence	$\log p^*(X \Theta_1)$	$\log p^*(X \Theta_2)$	$\log p^*(X \Theta_3)$	$\log p^*(X \Theta_4)$	$\log p^*(X \Theta_5)$	$\log p^*(X \Theta_6)$	Most likely model
X1							
X2							
X3							
X4							
X5							
X6							

```

log_viterbi = np.zeros((6, 6))
for i, X in enumerate([X1, X2, X3, X4, X5, X6]):
    for j, hmm in enumerate([hmm1, hmm2, hmm3, hmm4, hmm5, hmm6]):
        log_viterbi[i, j] = hmm.viterbi(X)[1]

print(log_viterbi.round(2))

```

```

# Comparison with the complete log-likelihoods from the forward algorithm
print((log_prob - log_viterbi).round(2))

```

4.2.1 Question

Is the likelihood along the best path a good approximation of the real likelihood of a sequence given a model ?

4.2.2 Answer

The values found for both likelihoods differ within an acceptable error margin. Furthermore, using the best path likelihood does not, in most practical cases, modify the classification results. Finally, it alleviates further the computational load since it replaces the sum or the logsum by a max in the recursive part of the procedure. Hence, the likelihood along the best path can be considered as a good approximation of the true likelihood.

5 Training of HMMs

Decoding or aligning acoustic feature sequences requires the prior specification of the parameters of some HMMs. As explained before, these models have the role of stochastic templates to which we compare the observations. But how to determine templates that represent efficiently the phonemes or the words that we want to model? The solution is to estimate the parameters of the HMMs from a database containing observation sequences, in a supervised or an unsupervised way.

5.1 Questions

In the previous lab session, we have learned how to estimate the parameters of Gaussian pdfs given a set of training data. Suppose that you have a database containing several utterances of the imaginary word / aiy / , and that you want to train a HMM for this word. Suppose also that this database comes with a *labeling* of the data, i.e. some data structures that tell you where are the phoneme boundaries for each instance of the word.

1. Which model architecture (ergodic or left-right) would you choose? With how many states? Justify your choice.
2. How would you compute the parameters of the proposed HMM?
3. Suppose you didn't have the phonetic labeling (i.e. you do *unsupervised training*). Propose a recursive procedure to train the model, making use of one of the algorithms studied during the present session.

5.2 Answers

1. It can be assumed that the observation sequences associated with each distinct phoneme obey specific densities of probability. As in the previous lab, this means that the phonetic classes are assumed to be separable by Gaussian classifiers. Hence, the word / aiy / can be assimilated to the result of drawing samples from the pdf $N_{/a/}$, then transitioning to $N_{/i/}$ and drawing samples again, and finally transitioning to $N_{/y/}$ and drawing samples. It sounds therefore reasonable to model the word / aiy / by a left-right HMM with three emitting states.
 2. If we know the phonetic boundaries for each instance, we know to which state belongs each training observation, and we can give a label (/a/, /i/ or /y/) to each feature vector. Hence, we can use the mean and variance estimators studied in the previous lab to compute the parameters of the Gaussian density associated with each state (or each label).
By knowing the labels, we can also count the transitions from one state to the following (itself or another state). By dividing the transitions that start from a state by the total number of transitions from this state, we can determine the transition matrix.
 3. The Viterbi procedure allows to distribute some labels on a sequence of features. Hence, it is possible to perform unsupervised training in the following way:
 - (a) Start with some arbitrary state sequences, which constitute an initial labeling. (The initial sequences are usually made of even distributions of phonetic labels along the length of each utterance.)
 - (b) Update the model, relying on the current labeling.
 - (c) Use the Viterbi algorithm to re-distribute some labels on the training examples.
 - (d) If the new distribution of labels differs from the previous one, re-iterate (go to (b)). One can also stop when the evolution of the likelihood of the training data becomes asymptotic to a higher bound.
- The principle of this algorithm is similar to the Viterbi-EM, used to train the Gaussians during the previous lab. Similarly, there exists a "soft" version, called the Baum-Welch algorithm, where each state participates to the labeling of the feature frames (this version uses the forward recursion instead of the Viterbi). The Baum-Welch algorithm is an EM algorithm specifically adapted to the training of HMMs (see Lawrence Rabiner's Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition for details), and is one of the most widely used training algorithms in "real world" speech recognition.

Acknowledgements

This lab was originally developed by Sacha Krstulović, Hervé Bourlard, Hemant Misra, and Mathew Magimai-Doss for the *Speech Processing and Speech Recognition* course at École polytechnique fédérale de Lausanne (EPFL). The original Matlab version is available here: <http://publications.idiap.ch/index.php/publications/show/739>