Viterbi Algorithm

Following Section 5.3.3.2 of [Müller, FMP, Springer 2015] (http://www.music-processing.de/), we describe in this notebook the Viterbi algorithm, which efficiently solves the uncovering problem of hidden Markov models (HMMs). For a detailed introduction of HMMs, we refer to the famous tutorial paper by Rabiner.

- Lawrence R. Rabiner: **A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition.** Proceedings of the IEEE, 77 (1989), pp. 257–286.
  Bibtex (../data/bibtex/FMP_bibtex_Rabiner89_HMM_IEEE.txt)

Uncovering Problem and Viterbi Algorithm

We now consider the **uncovering problem** (../C5/C5S3_HiddenMarkovModel.html), which is the problem relevant for our chord recognition scenario (../C5/C5S2_ChordRec_Templates.html), in more detail. As described in the FMP notebook on Hidden Markov Models (HMMs) (../C5/C5S3_HiddenMarkovModel.html), the goal is to find the **single** state sequence $S = (s_1, s_2, \ldots, s_N)$ that "best explains" a given observation sequence $O = (o_1, o_2, \ldots, o_N)$. In the chord recognition scenario, the observation sequence is a sequence of chroma vectors (../C3/C3S1_SpecLogFreq-Chromagram.html) extracted from an audio recording. The optimal state sequence is then the sequence of chord labels, which is the result of the chord recognition procedure. As optimization criterion, one possibility is to choose the state sequence S^* that yields the highest probability Prob^* when evaluated against the observation sequence O:

$$\mathrm{Prob}^* = \max_{S=(s_1,s_2,\ldots,s_N)} P[O,S|\Theta], \qquad (1)$$

$$S^* = \operatorname*{argmax}_{S=(s_1,s_2,\ldots,s_N)} P[O,S|\Theta]. \qquad (2)$$

To find the sequence S^* using the naive approach, one would have to compute the probability value $P[O,S|\Theta]$ for each of the I^N possible state sequences of length N and then look for the maximizing argument. Fortunately, there is a technique known as the **Viterbi algorithm** for finding the optimizing state sequence in a much more efficient way. The Viterbi algorithm, which is similar to the DTW algorithm (../C3/C3S2_DTWbasic.html), is based on **dynamic programming**. The idea is to recursively compute an optimal (i.e., probability-maximizing) state sequence from optimal solutions for subproblems, where one considers truncated versions of the observation sequence. Let $O = (o_1, o_2, \ldots o_N)$ be the observation sequence. Then, we define the prefix

$$O(1:n) := (o_1, \ldots, o_n)$$

of length $n \in [1:N]$ and set

$$\mathbf{D}(i,n) := \max_{(s_1,\ldots,s_n)} P[O(1:n),(s_1,\ldots,s_{n-1},s_n=\alpha_i)|\Theta] \qquad (3)$$

for $i \in [1:I]$. In other words, $\mathbf{D}(i,n)$ is the highest probability along a single state sequence $(s_1,\ldots,s_n)$ that accounts for the first n observations and ends in state $s_n = \alpha_i$. The state sequence yielding the maximal value

$$\mathbf{Prob}^* = \max_{i\in[1:I]} \mathbf{D}(i,N) \qquad (4)$$

is the solution to our uncovering problem. The $(I \times N)$ matrix $\mathbf{D}$ can be computed recursively along the column index $n \in [1:N]$. The following table specifies the Viterbi algorithm. For a detailed explanation of the algorithm, we refer to Section 5.3.3.2 of [Müller, FMP, Springer 2015] (http://www.music-processing.de/).

Table 5.2 from [Müller, FMP, Springer 2015]

Algorithm: VITERBI

Input: HMM specified by $\Theta = (\mathcal{A}, A, C, \mathcal{B}, B)$
Observation sequence $O = (o_1 = \beta_{k_1}, o_2 = \beta_{k_2}, \ldots, o_N = \beta_{k_N})$

Output: Optimal state sequence $S^* = (s_1^*, s_2^*, \ldots, s_N^*)$

Procedure: Initialize the $(I \times N)$ matrix $\mathbf{D}$ by $\mathbf{D}(i,1) = c_i b_{ik_1}$ for $i \in [1:I]$. Then compute in a nested loop for $n = 2,\ldots,N$ and $i = 1,\ldots,I$:

$$\begin{aligned}\mathbf{D}(i,n) &= \max_{j\in[1:I]} \left(a_{ji}\cdot\mathbf{D}(j,n-1)\right)\cdot b_{ik_n}\\ \mathbf{E}(i,n-1) &= \operatorname{argmax}_{j\in[1:I]} \left(a_{ji}\cdot\mathbf{D}(j,n-1)\right)\end{aligned}$$

Set $i_N = \operatorname{argmax}_{j\in[1:I]} \mathbf{D}(j,N)$ and compute for decreasing $n = N-1,\ldots,1$ the maximizing indices

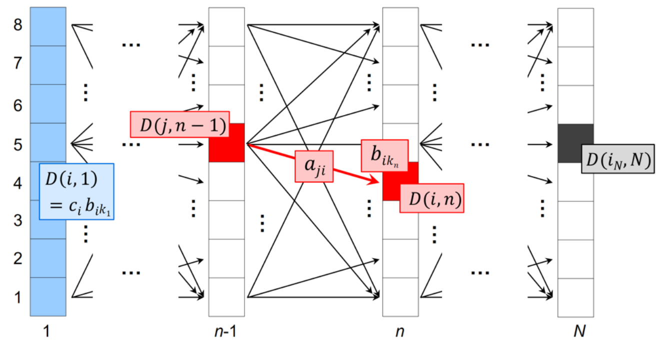
$$i_n = \operatorname{argmax}_{j\in[1:I]} \left(a_{ji_{n+1}}\cdot\mathbf{D}(j,n)\right) = \mathbf{E}(i_{n+1},n).$$

The optimal state sequence $S^* = (s_1^*,\ldots,s_N^*)$ is defined by $s_n^* = \alpha_{i_n}$ for $n \in [1:N]$.

The following figure illustrates the main steps of the Viterbi algorithm.

- The blue cells indicate the entries $\mathbf{D}(i,1)$, which serve as **initialization** of the algorithm.
- The red cells illustrate the computation of the **main iteration**.
- The black cell indicates the **maximizing index** used for the back tracking to obtain the optimal state sequence.
- The matrix E is used to keep track of the maximizing indices in the recursion. This information is needed in the second stage when constructing the optimal state sequence using **backtracking**.

Figure 5.27 from [Müller, FMP, Springer 2015]

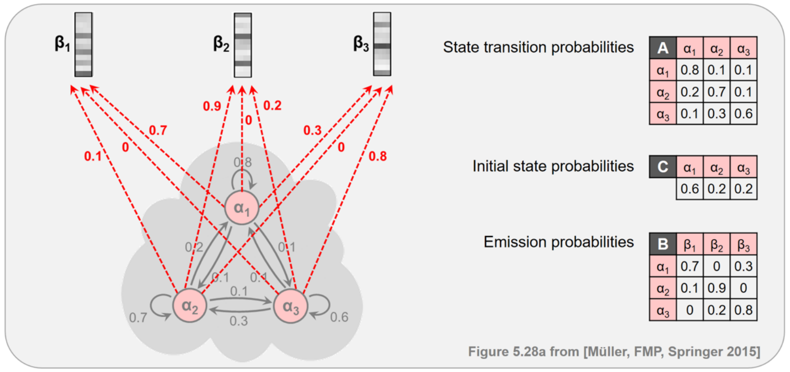


The overall procedure is similar to the DTW algorithm (../C3/C3S2_DTWbasic.html), where one first constructs the accumulated cost matrix and then obtains the optimal warping path using backtracking. As illustrated by the previous figure, the Viterbi algorithm proceeds in an iterative fashion building up a graph-like structure with N layers, each consisting of I nodes (the states). Furthermore, two neighboring layers are connected by I^2 edges, which also determines the order of the number of operations needed to construct a new layer from a given layer. From this follows that the computational complexity of the Viterbi algorithm is $O(N \cdot I^2)$, which is much better

From this follows that the computational complexity of the Viterbi algorithm is $O(N \cdot I^2)$, which is much better than $O(I^N)$ required for the naive approach.

Toy Example

Continuing our toy example from the [FMP notebook on HMMs (../C5/C5S3_HiddenMarkovModel.html)](../C5/C5S3_HiddenMarkovModel.html), the next figure illustrates the principle of the Viterbi algorithm. At the top, one finds a specification of the HMM. At the bottom, the Viterbi algorithm is applied for an input sequence of length $N = 6$.

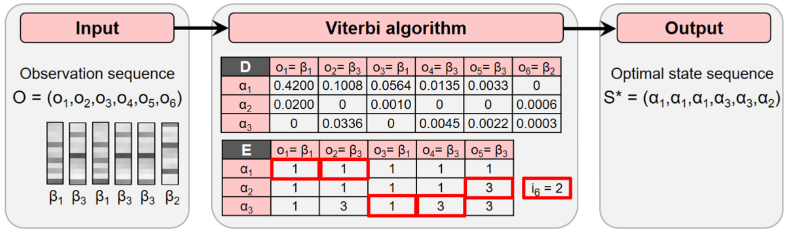


A	α_1	α_2	α_3
α_1	0.8	0.1	0.1
α_2	0.2	0.7	0.1
α_3	0.1	0.3	0.6

C	α_1	α_2	α_3
	0.6	0.2	0.2

B	β_1	β_2	β_3
α_1	0.7	0	0.3
α_2	0.1	0.9	0
α_3	0	0.2	0.8

Figure 5.28a from [Müller, FMP, Springer 2015]



D	$o_1=\beta_1$	$o_2=\beta_3$	$o_3=\beta_1$	$o_4=\beta_3$	$o_5=\beta_3$	$o_6=\beta_2$
α_1	0.4200	0.1008	0.0564	0.0135	0.0033	0
α_2	0.0200	0	0.0010	0	0	0.0006
α_3	0	0.0336	0	0.0045	0.0022	0.0003

E	$o_1=\beta_1$	$o_2=\beta_3$	$o_3=\beta_1$	$o_4=\beta_3$	$o_5=\beta_3$
α_1	1	1	1	1	1
α_2	1	1	1	1	3
α_3	1	3	1	3	3

Figure 5.28b from [Müller, FMP, Springer 2015]

Implementation of Viterbi Algorithm

In the next code cell, we provide an implementation of the Viterbi algorithm.

Note: Due to Python conventions, the indexing in the implementation starts with index `0`. Also, there is an index shift when applying the algorithm to our toy example, which becomes `O = [0, 2, 0, 2, 2, 1]`.

In [1]:
```
import numpy as np
from numba import jit

@jit(nopython=True)
def viterbi(A, C, B, O):
    """Viterbi algorithm for solving the uncovering problem

    Notebook: C5/C5S3_Viterbi.ipynb

    Args:
        A (np.ndarray): State transition probability matrix of dimension
I x I
        C (np.ndarray): Initial state distribution  of dimension I
        B (np.ndarray): Output probability matrix of dimension I x K
        O (np.ndarray): Observation sequence of length N

    Returns:
        S_opt (np.ndarray): Optimal state sequence of length N
        D (np.ndarray): Accumulated probability matrix
        E (np.ndarray): Backtracking matrix
    """
    I = A.shape[0]    # Number of states
    N = len(O)  # Length of observation sequence

    # Initialize D and E matrices
    D = np.zeros((I, N))
    E = np.zeros((I, N-1)).astype(np.int32)
    D[:, 0] = np.multiply(C, B[:, O[0]])

    # Compute D and E in a nested loop
    for n in range(1, N):
        for i in range(I):
            temp_product = np.multiply(A[:, i], D[:, n-1])
            D[i, n] = np.max(temp_product) * B[i, O[n]]
            E[i, n-1] = np.argmax(temp_product)

    # Backtracking
    S_opt = np.zeros(N).astype(np.int32)
    S_opt[-1] = np.argmax(D[:, -1])
    for n in range(N-2, -1, -1):
        S_opt[n] = E[int(S_opt[n+1]), n]

    return S_opt, D, E

# Define model parameters
A = np.array([[0.8, 0.1, 0.1],
              [0.2, 0.7, 0.1],
              [0.1, 0.3, 0.6]])

C = np.array([0.6, 0.2, 0.2])

B = np.array([[0.7, 0.0, 0.3],
              [0.1, 0.9, 0.0],
              [0.0, 0.2, 0.8]])


O = np.array([0, 2, 0, 2, 2, 1]).astype(np.int32)
#O = np.array([1]).astype(np.int32)
#O = np.array([1, 2, 0, 2, 2, 1]).astype(np.int32)

# Apply Viterbi algorithm
S_opt, D, E = viterbi(A, C, B, O)

#
print('Observation sequence:   O = ', O)
print('Optimal state sequence: S = ', S_opt)
np.set_printoptions(formatter={'float': "{: 7.4f}".format})
print('D =', D, sep='\n')
np.set_printoptions(formatter={'float': "{: 7.0f}".format})
print('E =', E, sep='\n')
```

```
Observation sequence:   O =  [0 2 0 2 2 1]
Optimal state sequence: S =  [0 0 0 2 2 1]
D =
[[ 0.4200  0.1008  0.0564  0.0135  0.0033  0.0000]
 [ 0.0200  0.0000  0.0010  0.0000  0.0000  0.0006]
 [ 0.0000  0.0336  0.0000  0.0045  0.0022  0.0003]]
E =
[[0 0 0 0 0]
 [0 0 0 0 2]
 [0 2 0 2 2]]
```

Log-Domain Implementation of Viterbi Algorithm

In each iteration of the Viterbi algorithm, the accumulated probability values of $\mathbf{D}$ at step $n-1$ are multiplied with two probability values from A and B. More precisely, we have

$$\mathbf{D}(i,n) = \max_{j\in[1:I]} \big(a_{ij}\cdot\mathbf{D}(j,n-1)\big)\cdot b_{i_{k_n}}.$$

Since all probability values lie the interval $[0,1]$, the product of such values decreases exponentially with the number n of iterations. As a result, for input sequences $O=(o_1,o_2,\ldots o_N)$ with large N, the values in $\mathbf{D}$ typically become extremely small, which may finally lead to a numerical underflow. A well-known trick when dealing with products of probability values is to work in the **log-domain**. To this end, one applies a logarithm to all probability values and **replaces multiplication by summation**. Since the logarithm is a strictly monotonous function, ordering relations are preserved in the log-domain, which allows for transferring operations such as **maximization** or **minimization**.

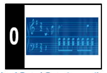
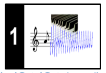
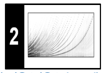
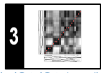
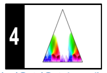
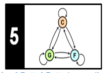
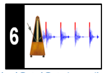
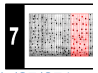
In the following code cell, we provide a **log variant** of the Viterbi algorithm. This variant yields exactly the same optimal state sequence S^* and the same backtracking matrix E as the original algorithm, as well as the accumulated log probability matrix $\log(\mathbf{D})$ (`D_log`) (where the logarithm is applied for each entry). We again test the implementation on our toy example. Furthermore, as a sanity check, we apply the exponential function to the computed log-matrix `D_log` , which should yield the matrix `D` as computed by the original Viterbi algorithm above.

In [2]:
```
@jit(nopython=True)
def viterbi_log(A, C, B, O):
    """Viterbi algorithm (log variant) for solving the uncovering problem

    Notebook: C5/C5S3_Viterbi.ipynb

    Args:
        A (np.ndarray): State transition probability matrix of dimension
I x I
        C (np.ndarray): Initial state distribution  of dimension I
        B (np.ndarray): Output probability matrix of dimension I x K
        O (np.ndarray): Observation sequence of length N

    Returns:
        S_opt (np.ndarray): Optimal state sequence of length N
        D_log (np.ndarray): Accumulated log probability matrix
        E (np.ndarray): Backtracking matrix
    """
    I = A.shape[0]    # Number of states
    N = len(O)  # Length of observation sequence
    tiny = np.finfo(0.).tiny
    A_log = np.log(A + tiny)
    C_log = np.log(C + tiny)
    B_log = np.log(B + tiny)

    # Initialize D and E matrices
    D_log = np.zeros((I, N))
    E = np.zeros((I, N-1)).astype(np.int32)
    D_log[:, 0] = C_log + B_log[:, O[0]]

    # Compute D and E in a nested loop
    for n in range(1, N):
        for i in range(I):
            temp_sum = A_log[:, i] + D_log[:, n-1]
            D_log[i, n] = np.max(temp_sum) + B_log[i, O[n]]
            E[i, n-1] = np.argmax(temp_sum)

    # Backtracking
    S_opt = np.zeros(N).astype(np.int32)
    S_opt[-1] = np.argmax(D_log[:, -1])
    for n in range(N-2, -1, -1):
        S_opt[n] = E[int(S_opt[n+1]), n]

    return S_opt, D_log, E

# Apply Viterbi algorithm (log variant)
S_opt, D_log, E = viterbi_log(A, C, B, O)

print('Observation sequence:   O = ', O)
print('Optimal state sequence: S = ', S_opt)
np.set_printoptions(formatter={'float': "{: 7.2f}".format})
print('D_log =', D_log, sep='\n')
np.set_printoptions(formatter={'float': "{: 7.4f}".format})
print('exp(D_log) =', np.exp(D_log), sep='\n')
np.set_printoptions(formatter={'float': "{: 7.0f}".format})
print('E =', E, sep='\n')
```

```
Observation sequence:   O =  [0 2 0 2 2 1]
Optimal state sequence: S =  [0 0 0 2 2 1]
D_log =
[[  -0.87   -2.29   -2.87   -4.30   -5.73 -714.35]
 [  -3.91 -711.57   -6.90 -713.57 -715.00   -7.44]
 [-710.01   -3.39 -712.30   -5.40   -6.13   -8.25]]
exp(D_log) =
[[ 0.4200  0.1008  0.0564  0.0135  0.0033  0.0000]
 [ 0.0200  0.0000  0.0010  0.0000  0.0000  0.0006]
 [ 0.0000  0.0336  0.0000  0.0045  0.0022  0.0003]]
E =
[[0 0 0 0 0]
 [0 0 0 0 2]
 [0 2 0 2 2]]
```

Acknowledgment: This notebook was created by Meinard Müller (https://www.audiolabs-erlangen.de/fau/professor/mueller) and Christof Weiß (https://www.audiolabs-

erlangen.de/fau/professor/mueller) and Christof Weiß (https://www.audiolabs-erlangen.de/fau/assistant/weiss).


(../C0/C0.html)

(../C1/C1.html)

(../C2/C2.html)

(../C3/C3.html)

(../C4/C4.html)

(../C5/C5.html)

(../C6/C6.html)

(../C7/C7.htm