CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Machine learning for big sequence data: Wavelet-compressed Hidden Markov Models

Master's thesis in Computer science and engineering

LUCA BELLO

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

MASTER'S THESIS 2020

Machine learning for big sequence data: Wavelet-compressed Hidden Markov Models

LUCA BELLO

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Machine learning for big sequence data: Wavelet-compressed Hidden Markov Models

LUCA BELLO

Supervisor: Alexander Schliep, Department of Computer Science and Engineering
External supervisor: Paolo Garza, Politecnico di Torino
Examiner: Alexander Schliep, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Machine learning for big sequence data: Wavelet-compressed Hidden Markov Models

LUCA BELLO
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Hidden Markov models are among the most important machine learning methods for the statistical analysis of sequential data, but they struggle when applied on big data. Their relative inefficiency has been addressed several times by the use of some compression techniques, either for the computation. This thesis explores the former, with the application of a data compression technique based on wavelets and the subsequent adaptation of the main HMMs algorithms from the literature: the forward, Viterbi and Baum-Welch algorithms used to solve the evaluation, decoding and training problem respectively. The testing phase shows that this new technique generally yields equal or better results, obtaining some extremely high speedups in the training problem, making it even thousands of times faster; this allows to easily train a HMM with big data on a commodity laptop.

Keywords: machine, learning, sequence, wavelet, compression, hidden, markov, models, viterbi, training.

Acknowledgements

Here I am, looking at this page and trying to write something that I can really call my own; something I can unmistakably recognize as marked with my personal style. Anyway, I will limit myself to just sprinkle some little magic here and there, sticking to a traditional acknowledgment format, because there are definitely people I need to thank. Also, I know for sure this will end up being too long, but I'm not the one reading, so the joke's on you.
The crazy series of events, life experiences and people that led me to this moment would be too complex to describe. I started university as a completely different person than I am now, moving out of my small hometown, away from my family, and joining a student dorm in a distant city. I wasn't prepared for what came after: the absurd number of sleepless nights devoted to study; questioning my life choices more times than I can remember; self-reflections that turned my soul inside out multiple times; abandoning everything to start over. And I definitely didn't know that this experience would have given me lifelong friendships with people that I love and respect. I'm incredibly grateful to them, and going through these five years without them would have been impossible. This is the moment I should stop talking about me and my exceptionally moving story and start talking about them. Before starting a foreseeable long list of people, I should thank my family, that has always been incredibly supportive throughout my academic career.
My dorm gave me a second family, and I need to thank every single person I met there since everyone contributed to enriching my personality with a small part of theirs, each in their own way. I want to thank Niccolò, for all the morning coffees, the nightly bitters, the foam swords duels and the flip flops thrown against the wall; Angelo, for his loyalty, the chicken curry, and for shouting my name out loud to all the girls he meets; Fabrizio, for being a constant inspiration and motivation to do better, for all the times he says "sorry", and for his kindness; and I want to thank the whole group "L'Associazione" for all the gaming nights, the shared panic for the exams, the meals we prepared, the laughs, the tears, and all the time we spent and will spend together. I want to thank Luciana for all the tea we drank at night, for the penguin hugs, the talks sitting against the radiator, and for far more things than I could list. I want to thank Martina LL, my "spiritual sister", for being there when it mattered. I want to thank Emma for being each other's rambunctious sidekick during the swedish adventure, and for all the times she tried to drag me out of my room towards social activities.
There are tons of other people who would deserve some big thanks - and a cookie - for being special, but this page won't fit them all. To every single one of you: thanks, from the bottom of my heart.

Luca Bello, Gothenburg, May 2020

Contents

List of Figures **xiii**

1 Introduction **1**
1.1 Motivation and previous research . . . 1
1.2 Hidden Markov Models . . . 2
1.3 Wavelet compression . . . 4

2 Theory **7**
2.1 Evaluation problem . . . 7
2.1.1 Forward algorithm . . . 8
2.1.2 Backward algorithm . . . 9
2.2 Decoding problem . . . 10
2.3 Training problem . . . 11
2.3.1 Starting model . . . 13
2.4 Compressed algorithms . . . 13
2.5 Formal transformations . . . 15
2.5.1 Forward algorithm . . . 16
2.5.2 Backward algorithm . . . 16
2.5.3 Viterbi algorithm . . . 16
2.5.4 Baum-Welch algorithm . . . 17
2.6 Relevant model parameters . . . 18
2.6.1 State separation . . . 18
2.6.2 Self-transition probabilities . . . 20

3 Methods **21**
3.1 Framework description . . . 21
3.2 Implementation details . . . 21
3.2.1 Parser . . . 21
3.2.2 Numerical errors . . . 22
3.2.3 State representation . . . 23
3.2.4 Data generation . . . 23
3.2.5 Wavelet compression . . . 23
3.2.6 Logarithms summation . . . 24
3.2.7 Saving results . . . 24
3.2.8 Test automation . . . 25
3.2.9 Other Python files . . . 25

3.3 Standard algorithms 25
3.3.1 Evaluation problem 26
3.3.2 Decoding problem 26
3.3.3 Training problem 26
3.4 Compressed algorithms 27
3.4.1 Evaluation problem 27
3.4.2 Decoding problem 27
3.4.3 Training problem 28

4 Results **29**
4.1 Testing setup 29
4.1.1 Choosing the parameters 29
4.1.2 Results evaluation 30
4.2 Evaluation problem 31
4.3 Decoding problem 32
4.4 Training problem 33

5 Conclusion **35**
5.1 Main takeaways 35
5.2 Wrapping up 35

Bibliography **37**

A Appendix 1 **I**
A.1 Fully-connected model I
A.2 Circular model II
A.3 Left-to-right model V

B Appendix 2 **IX**
B.1 C++ IX
B.1.1 algorithms_compressed.hpp IX
B.1.2 algorithms.hpp XVIII
B.1.3 commons.hpp XXVI
B.1.4 Compressor.hpp XXVII
B.1.5 Model.hpp XXXIV
B.1.6 parser.hpp XXXVII
B.1.7 State.hpp XXXIX
B.1.8 utilities.hpp XL
B.1.9 WaHMM.hpp XLIII
B.2 Python XLVII
B.2.1 automated_test.py XLVII
B.2.2 create_model_file.py LIII
B.2.3 generate_data.py LIV
B.2.4 generate_states.py LV
B.2.5 plot_data.py LVI
B.2.6 plot_kmeans.py LVII
B.2.7 pomegranate_test.py LVIII

B.2.8 results_aggregation.py LX
B.2.9 utilities_io.py LXVI
B.2.10 utilities_kmeans.py LXVIII
B.2.11 viterbi_comparison.py LXX
B.2.12 WaHMM.py LXXI

List of Figures

1.1 Collocation of this thesis in the context of different HMM types. . . . 2
1.2 A simple example of a Markov model: the state chain through time (top) and the graph representation of the states connection (bottom). 3
1.3 An example of the output of a hidden Markov model through some time steps. The state path Q is not observable, and O are the observations emitted in each state. 4
1.4 A signal decomposed with wavelets in its approximation and different levels of detail. Image taken from [1]. 5
1.5 Typical sequence data undergoing block compression; the vertical red lines show the block borders in the case of an ideal compression. . . . 6
1.6 Example of block generation using a breakpoint array. During the query (thick red), when values are above the threshold (horizontal blue line), a breakpoint is returned (vertical blue line). Figure taken from [2]. 6

2.1 Graphical representation of the lattice structure used for dynamic programming in the standard algorithms. 9
2.2 The structure on which the compressed algorithms operate; it generates from a subset of the links in the lattice structure. 14
2.3 Data generated using a two-state HMM with distributions $N(0, 1^2)$ and $N(10, 1^2)$. 19

3.1 Structure of the implemented code (thin border) and its connection points to the external libraries used (thick border). 22

4.1 Relative difference between the $P(O|\lambda)$ log probabilities of the compressed and standard algorithms. Since the errors are on a big negative log probability, the actual error magnitude is approximately of 10^{-5000}. 31
4.2 Speedup on the evaluation problem using the compressed algorithm, including the input data processing time. 31
4.3 Relative difference between the fractions of errors in the estimated generating path. 32
4.4 Speedup on the decoding problem using the compressed algorithm, including the input data processing time. 32
4.5 Difference between the average KL-divergence for the compressed and standard algorithms. 33

4.6 Speedup on the training problem using the compressed algorithm, including the input data processing time. 34

A.1 Relative difference between the average error on the log transition probabilities of the compressed and standard algorithms. I
A.2 Relative difference between the average error on the log probability of the starting state in the initial distribution for the compressed and standard algorithms. I
A.3 Relative difference between the $P(O|\lambda)$ log probabilities of the compressed and standard algorithms. II
A.4 Relative difference between the fractions of errors in the estimated generating path. II
A.5 Difference between the average KL-divergence for the compressed and standard algorithms. III
A.6 Relative difference between the average error on the log transition probabilities of the compressed and standard algorithms. III
A.7 Relative difference between the average error on the log probability of the starting state in the initial distribution for the compressed and standard algorithms. III
A.8 Speedup on the evaluation problem using the compressed algorithm, including the input data processing time. IV
A.9 Speedup on the decoding problem using the compressed algorithm, including the input data processing time. IV
A.10 Speedup on the training problem using the compressed algorithm, including the input data processing time. IV
A.11 Relative difference between the $P(O|\lambda)$ log probabilities of the compressed and standard algorithms. V
A.12 Relative difference between the fractions of errors in the estimated generating path. V
A.13 Difference between the average KL-divergence for the compressed and standard algorithms. VI
A.14 Relative difference between the average error on the log transition probabilities of the compressed and standard algorithms. VI
A.15 Relative difference between the average error on the log probability of the starting state in the initial distribution for the compressed and standard algorithms. VI
A.16 Speedup on the evaluation problem using the compressed algorithm, including the input data processing time. VII
A.17 Speedup on the decoding problem using the compressed algorithm, including the input data processing time. VII
A.18 Speedup on the training problem using the compressed algorithm, including the input data processing time. VII

1
Introduction

Nowadays, many real-world problems are tackled trying to get insights from data with the application of machine learning techniques; it's easy to see that as the amount of data becomes massive, the algorithms necessarily need to be adapted and optimized towards a lower computational complexity, to retain feasibility in a reasonable execution time. Hidden Markov Models (HMMs) are among the most important machine learning methods for the statistical analysis of sequential data. Their mathematical structure provides a wide range of applications and can serve as a basis for more complex models, making their study extremely meaningful. Some central and well-known algorithms for HMMs tend to struggle when applied on big data; the aim of this thesis is to adapt them so they can be applied on a compressed representation of the data, preserving their usefulness and applicability.

1.1 Motivation and previous research

There are many examples of real world applications showing the importance of hidden Markov models: computational finance [3], speech recognition [4] and more. Unfortunately, limitations have been encountered when working with big data due to the computational complexity of the standard algorithms. This problem has been tackled with many different approaches: one that has often proved its effectiveness is the idea of compression, to significantly reduce either the computations or the scale of the data.
To show some context and establish the relevance of this topic, some efforts and achievements will be described below. For discrete-valued sequences, [5] shows how speed improvements are obtained by text compression techniques, based on identifying repeated substrings in the observed input sequence and obtaining higly parallelizable algorithms. Bayesian computations that were often avoided in practice due to long running times have been accelerated in [6], showing considerable improvements. For continuous observations, Bayesian inference was tackled by [7] and [2] using wavelet compression; as summarized in Figure 1.1, what has been left uncovered are the possible speed improvements obtainable by applying compression techniques to the standard algorithms.
This thesis aims at covering that unexplored case, studying the effects of wavelet compression for regular HMMs and their possible impact on real-world problems. Specifically, I will make use of the concepts in Wiedenhoeft's papers [7] and [2], using part of the HaMMLET tool for wavelet compression; this is described more in detail in Section 1.3.

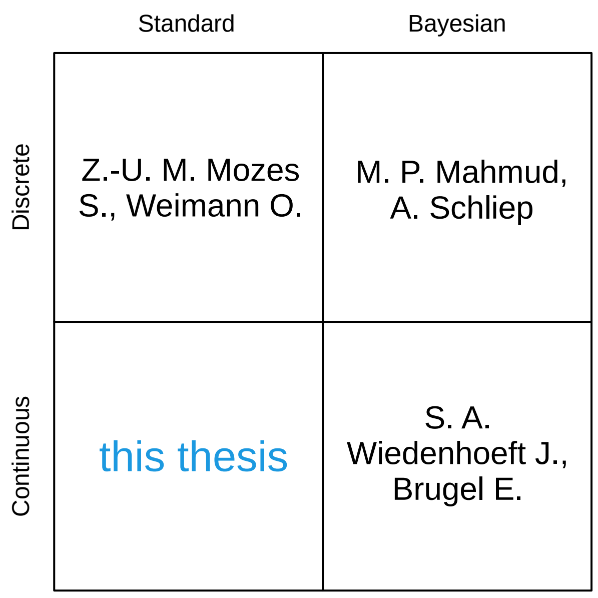


Figure 1.1: Collocation of this thesis in the context of different HMM types.

1.2 Hidden Markov Models

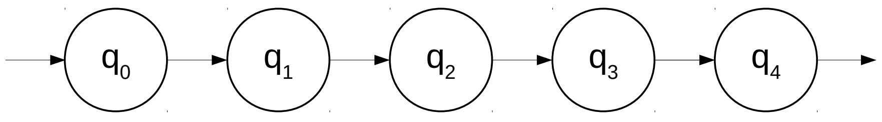
This section provides the theoretical background on Hidden Markov Models and introduces some needed notation, consistently with the one used by Rabiner in [8]. Consider a system which can be described by a set of N distinct states, $S_1, S_2, ..., S_N$, as shown in Figure 1.2. At regular discrete time intervals, the system goes through a state change and moves to another state (or to the same one) according to certain probabilities associated with the state; if the future evolution of the system depends solely on the current state, regardless of the system's history, this is a first-order Markov Process.
Let the time be denoted with $t = 1, 2, ...$, and the state of the system at a certain time with q_t. For a Markov process, it's true that:

$$P(q_t = S_j | q_{t-1} = S_i, q_{t-2} = S_k, ...) = P(q_t = S_j | q_{t-1} = S_i). \tag{1.1}$$

The processes that are more often considered are *time-homogeneous*, meaning that the transition probabilities a_{ij} from a state to another are independent of time; these probabilities can be collected in a matrix A that obeys the standard stochastic constraints:

$$a_{ij} = P(q_t = S_j | q_{t-1} = S_i) \geq 0, \quad \text{and} \tag{1.2a}$$

$$\sum_{j=1}^{N} a_{ij} = 1 \tag{1.2b}$$



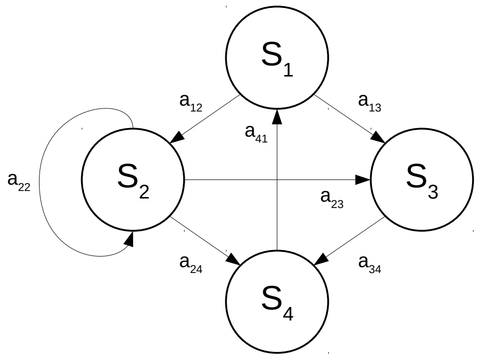


Figure 1.2: A simple example of a Markov model: the state chain through time (top) and the graph representation of the states connection (bottom).

In the Markov model defined above and shown in Figure 1.2, each state corresponds to an observable event. That is not always the case; sometimes the observations are not the states of a Markov model, but they are related to the actual state chain which is *hidden*, or *not observable*. A hidden Markov model (as shown in Figure 1.3) has an underlying Markov model which works as described above, with the observations being a probabilistic function of the current state; this means that for any observation O_t

$$P(O_t|O_{t-1}, O_{t-2}, ..., q_t, q_{t-1}, q_{t-2}, ...) = P(O_t|q_t). \tag{1.3}$$

Based on the type of observations, Markov models can be divided in discrete, where the alphabet of observations is finite, and continuous, where it's not. Looking at the definition of this model, many interesting questions may arise: how likely is that a sequence of observations has been generated by a certain model? What is the most likely state path corresponding to a sequence of observations? What is a good estimation of a model that suits the observations well? How can we apply Bayesian inference to train a model? The first three questions are, as defined by Rabiner in [8], the standard problems of HMMs named respectively evaluation, decoding and training problem. The aim of this thesis is to address these problems in the context of big data with a more efficient approach.

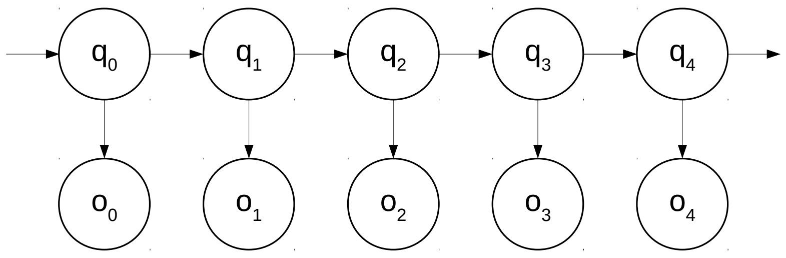


Figure 1.3: An example of the output of a hidden Markov model through some time steps. The state path Q is not observable, and O are the observations emitted in each state.

1.3 Wavelet compression

The compression technique that will be used throughout this thesis is based on wavelets. A wavelet is a function which resembles a small oscillation; the convolution of scaled versions of a wavelet with a signal of interest yields its wavelet transform. This is often used as a mathematical tool used to analyze signals, obtain a different representation of them and extract useful information. It is similar to the Fourier transform, with the main difference that the latter loses all information about the localization of a given frequency component.
A signal is usually decomposed using a certain wavelet and then, after some processing, it is often recomposed using the corresponding reconstruction wavelet. The decomposition yields an *approximation* of the signal and some *detail* coefficients, representing respectively a smoothed version of the signal and the higher frequency information. Many different wavelets can be used in a transform, based on the properties needed for a certain application:

- *size of support*, the interval where the wavelet is non-zero;
- *symmetry*, that influences the quality of localization;
- *number of vanishing moments*, a blindness to polynomials of a certain degree;
- *regularity*, also affecting frequency localization;
- *(bi-)orthogonality*, meaning that the decomposition and reconstruction wavelets form two distinct bases which are mutually orthogonal.

Figure 1.4 shows an example of signal decomposition using wavelets; this representation allows for effective de-noising against additive white Gaussian noise by simply setting to zero some detail coefficients above a certain threshold before reconstructing the signal. An interesting description of a possible use of wavelets is [9].

In the context of this thesis and following Wiedenhoeft's work in [7] and [2], the Haar wavelet is used to detect when a sequence of observations has a significant discontinuity in values, possibly indicating, under certain conditions, a change of state. An example result of the compression process is shown in Figure 1.5. Specifically, two main data structures from the HaMMLET tool will be used: the *breakpoint*

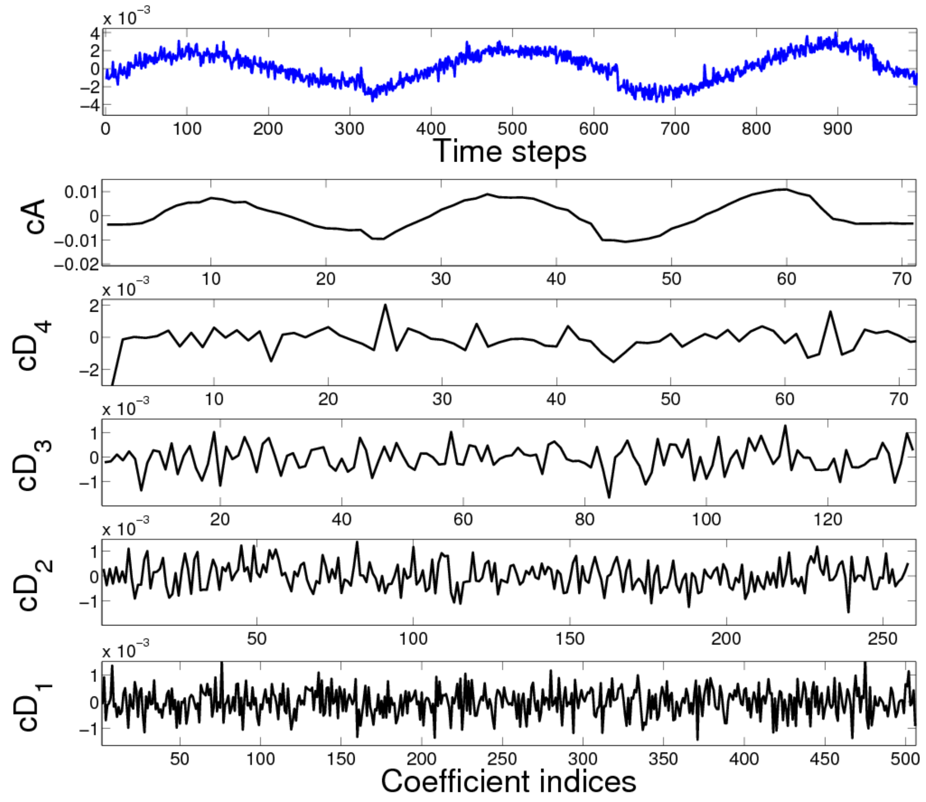


Figure 1.4: A signal decomposed with wavelets in its approximation and different levels of detail. Image taken from [1].

array and the *integral array*. The former uses a certain threshold (discussed later with the implementation) to define a block structure by storing indexes of blocks delimiters, dividing the sequence of observations into blocks where the state can be considered constant; the mechanism is shown in Figure 1.6. The latter contains sufficient statistics for each block, such as the sum of all elements inside it. Using two structures instead of one yields a more efficient implementation, as described in [2].
It's reasonable to expect that this compression method will work well when the states of the HMM are well-separated, so that the block division is accurate, and when the self-loop probability for each state is high enough, so that a single block contains more observations. In principle this approach could also be applied to multivariate data, but that falls out of the scope of this thesis.

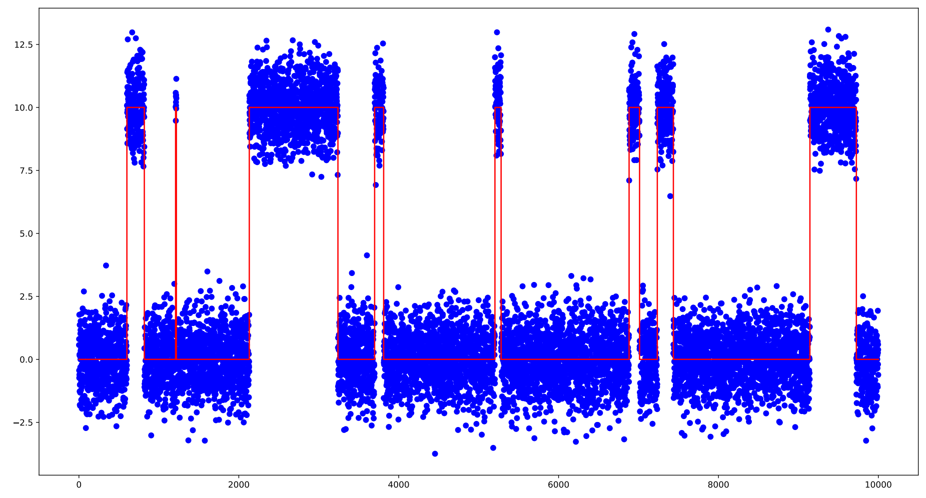


Figure 1.5: Typical sequence data undergoing block compression; the vertical red lines show the block borders in the case of an ideal compression.

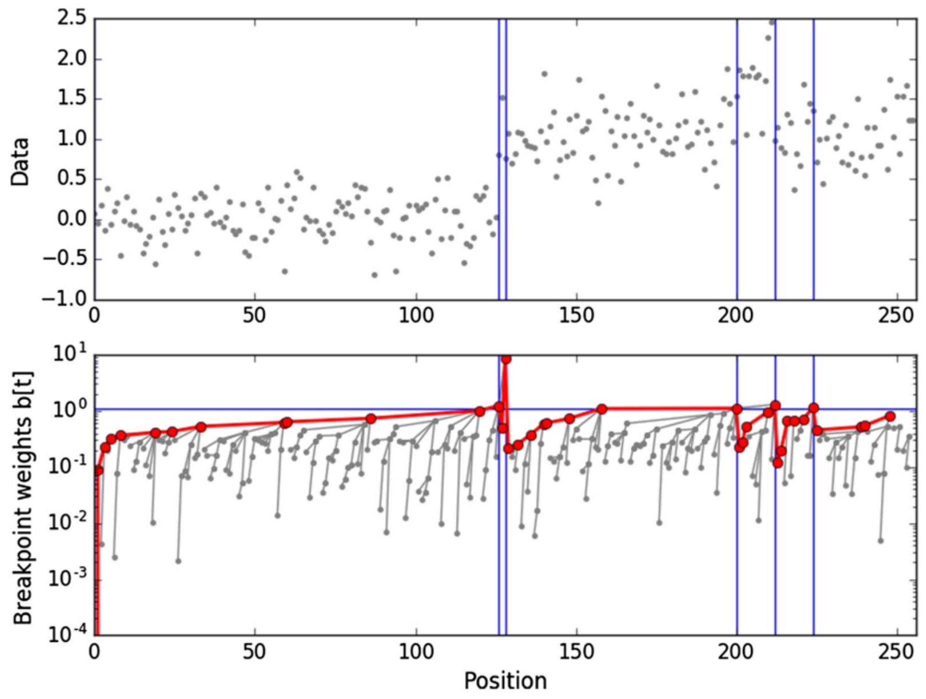


Figure 1.6: Example of block generation using a breakpoint array. During the query (thick red), when values are above the threshold (horizontal blue line), a breakpoint is returned (vertical blue line). Figure taken from [2].

2
Theory

This chapter tackles the theory behind hidden Markov models and their main problems of interest, using that as a stepping stone to analyze the theoretical transformations required by the data compression process. At this point the elements of a HMM should be clearly defined and denoted; again, the notation used is taken from Rabiner [8]:

- N, the number of states in the model; specifically, individual states will be denoted as $S = \{S_1, S_2, ..., S_N\}$ and the state at time t will be denoted as q_t;
- $A = \{a_{ij}\}$, the state transition probability distribution, where a_{ij} indicates the probability of going from state i to state j in one time step;
- $B = \{b_j\}$, the emission probability distribution of observations for each state j;
- $\pi = \{\pi_i\}$, the initial state distribution.

To indicate the complete set of parameters, this compact notation is generally used,

$$\lambda = (N, A, B, \pi). \tag{2.1}$$

To add on this and clarify the notation used: the observation sequence is denoted by O, with the element observed at time t being indicated by O_t; in the same way, the state sequence (also *state path*, *generating path*) is denoted by Q, and the state at time t is indicated by q_t. The HMMs that will be used in this thesis are continuous-valued; each state is associated and characterized by an emission distribution. For many applications, using Gaussian distributions is a good choice (e.g. autoregressive HMMs for speech recognition). As stated in the introduction, there are three main problems that will be tackled; each one will be described and discussed in a separate section below.

2.1 Evaluation problem

The problem is about evaluating how well a specific sequence of observations is represented by a given model, through the computation of the probability that the observed sequence was produced by the model. Solving this problem is important because it allows to compare different models to decide which one better represents a sequence of observations. Formally, given the observation sequence $O = O_1, O_2, ..., O_T$ and a model $\lambda = (A, B, \pi)$, the goal is to efficiently compute its likelihood. A very intuitive but inefficient way of doing it would be applying the law of total probability, enumerating every possible sequence of states and summing the conditional probability of all observations over all those sequences:

$$P(O|\lambda) = \sum_{all\ Q} P(O|Q,\lambda) = \sum_{all\ Q} \pi_{q_1} b_{q_1}(O_1) a_{q_1 q_2} b_{q_2}(O_2) \dots a_{q_{T-1}} b_{q_T}(O_T) \tag{2.2}$$

For a specific state sequence, this equation starts in the state q_1 with probability π_{q_1}, produces the symbol O_1 with probability $b_{q_1}(O_1)$ and moves to the next state q_1 to q_2 with probability $a_{q_1 q_2}$, and then follows the same logic to cover the whole observation sequence. When looking at the computational complexity, the number of state paths obtained by enumeration is given by the dispositions with repetition of the states, that is N^T; for each state, the number of calculations scales linearly with the length of the observation sequence; this yields a total computational complexity that is $\mathcal{O}(T \cdot N^T)$.

2.1.1 Forward algorithm

The standard algorithm use to solve the evaluation problem is the *forward* algorithm, which is much more efficient than the approach considered above. The key element is the forward variable $\alpha_t(i)$, defined as the joint probability of observing the sequence up to time t and being in state S_i at time t

$$\alpha_t(i) = P(O_1 O_2 \dots O_t, q_t = S_i|\lambda). \tag{2.3}$$

Through induction, the following procedure can be defined:

$$\alpha_1(i) = \pi_i b_i(O_1), \qquad 1 \le i \le N \tag{2.4a}$$

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}), \qquad 1 \le t \le T-1, \quad 1 \le j \le N \tag{2.4b}$$

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{2.4c}$$

The first step is the initialization of the forward variable in (2.4a) using the initial probability distribution π. The actual induction happens on the second step (2.4b), where α_{t+1} is calculated using the forward variables at the previous instant α_t; to make that calculation for a certain state S_j, it is necessary to consider the probability of getting there from any state S_i multiplied by the probability of observing the symbol O_{t+1}. This procedure terminates with (2.4c), that simply takes the sum of the forward variables over all the states. The procedure can be visualized well through the Figure 2.1.
Looking at the computational complexity, the number of calculations for each observation is N^2 (N per each state); repeating this for the whole sequence length gives a complexity that is $\mathcal{O}(T \cdot N^2)$, which is evidently faster than the previous approach especially for increasingly long sequences.
Very closely tied to this algorithm is the backward algorithm. Although it's not necessary to solve the evaluation problem, it can be helpful in the solution of both the decoding and the training problems.

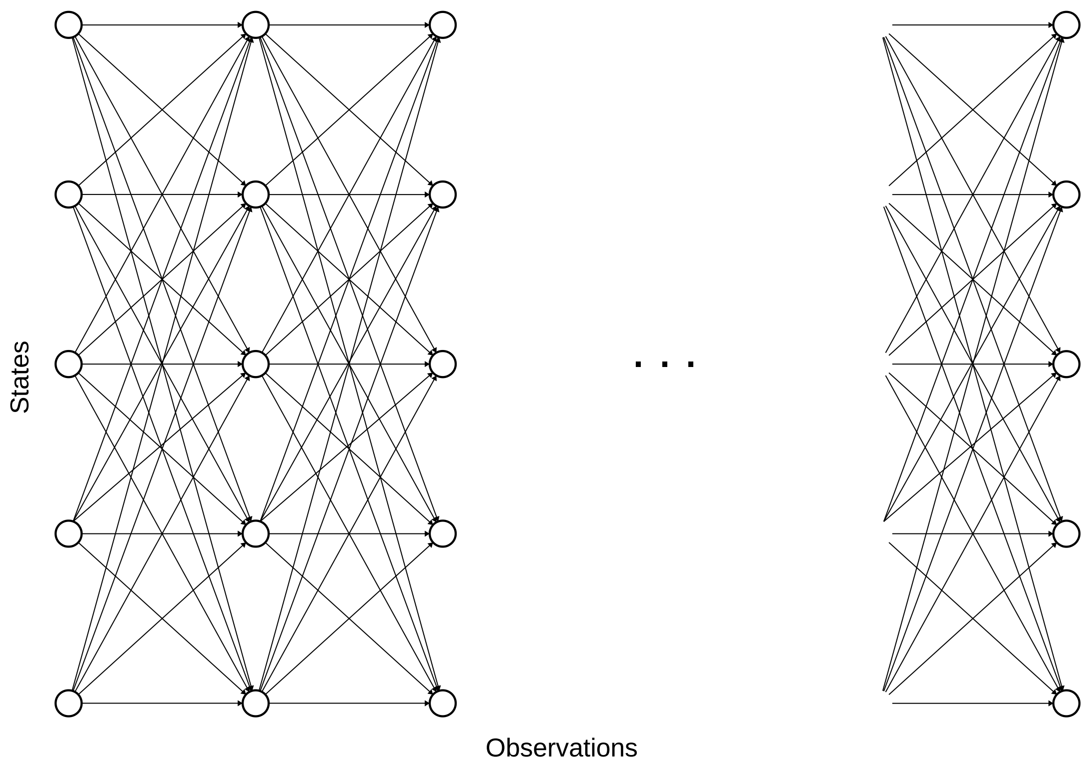


Figure 2.1: Graphical representation of the lattice structure used for dynamic programming in the standard algorithms.

2.1.2 Backward algorithm

The backward variable is defined as the probability of the partial observation sequence from $t+1$ to the end, given a certain state at time t and the model

$$\beta_t(i) = P(O_{t+1}O_{t+2}\dots O_t | q_t = S_i, \lambda). \tag{2.5}$$

This variable can also be calculated using the lattice structure shown in Figure 2.1, in an inductive fashion with the use of dynamic programming:

$$\beta_T(i) = 1, \qquad 1 \le i \le N, \quad \text{and} \tag{2.6a}$$

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1})\ \beta_{t+1}(j), \qquad t = T-1, T-2, \cdots, 1, \quad 1 \le i \le N. \tag{2.6b}$$

The initialization step in (2.6a) chooses an arbitrary starting point; the induction (2.6b) computes the other variables by accounting for all the possible states the system could have been in at the following time step. The calculation of the computational complexity of this algorithm follows the same reasoning of the forward algorithm; this leads to conclude that the backward algorithm also has a complexity that is $\mathcal{O}(T \cdot N^2)$.

2.2 Decoding problem

A very common situation when dealing with hidden Markov models is the need to predict the generating state path of a certain observed sequence. There is no exact answer to this question; the goal is to find the solution that better fits the data. Many optimization criteria can be used based on the definition of *better*. An example is finding the most likely state individually for each observation; using this criterion, the key variable is

$$\gamma_t(i) = P(q_t = S_i | O, \lambda), \tag{2.7}$$

which is the probability of being in state S_i at time t, given both the observation sequence and the model. This variable can be expressed using the forward and backward variables in the following way

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}, \tag{2.8}$$

because the forward variable accounts for the observations before time t, while the backward variable refers to the observations after t. The most likely state at time t is easily obtained through the γ_t variable by looking at which state as the highest associated probability:

$$q_t = \underset{1 \leq i \leq N}{\operatorname{argmax}} \Big[\gamma_t(i) \Big], \qquad 1 \leq t \leq T. \tag{2.9}$$

Although this approach obtains the highest number of expected correct matches between predicted and actual state, it's often discarded since it disregards the transition probabilities of the model. In particular, if some transitions have zero probability, meaning they cannot happen, this criterion would still be able to include them in the result.
For this reason, it's often more interesting to compute the most likely generating path; mathematically, this means finding the maximization of $P(Q|O, \lambda)$. The standard algorithm used to solve this problem is the *Viterbi* algorithm; as the forward algorithm, it uses a dynamic programming approach that allows to reduce computational complexity. The Viterbi algorithm substitutes the summations of the forward algorithm with a maximization; the key variable to calculate is $\delta_t(i)$, defined as the highest probability along a single path, after the first t observations and ending in state i:

$$\delta_t(i) = \max_{q_1, q_2, \ldots, q_{t-1}} P(q_1 q_2 \ldots q_t = i, O_1 O_2 \ldots O_t | \lambda) \tag{2.10}$$

The most likely path will be the argument of this maximization over all the states considering the whole observations sequence; it can be defined in the notation as ψ. Through induction it is possible to write the following equations:

$$\begin{aligned} &\delta_1(i) = \pi_i b_i(O_1), \qquad 1 \leq i \leq N \\ &\psi_1 = 0 \end{aligned} \tag{2.11a}$$

$$\delta_t(j) = \max_{1\le i\le N}\Big[\delta_{t-1}(i)\ a_{ij}\Big] b_j(O_t), \qquad 2 \le t \le T, \quad 1 \le j \le N \tag{2.11b}$$
$$\psi_t(j) = \operatorname*{argmax}_{1\le i\le N}\Big[\delta_{t-1}(i)\ a_{ij}\Big], \qquad 2 \le t \le T, \quad 1 \le j \le N$$

$$P^* = \max_{1\le i\le N}\Big[\delta_T(i)\Big] \tag{2.11c}$$
$$q_T^* = \operatorname*{argmax}_{1\le i\le N}\Big[\delta_T(i)\Big]$$

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \qquad t = T-1, T-2, \dots, 1 \tag{2.11d}$$

The initialization phase in (2.11a) starts with an empty solution for the state ψ_1. The induction in (2.11b) relies on the lattice structured defined previously in Figure 2.1; the contribution of the previous δ_t variables is given through the max operator instead of using summation. During this process, the argmax is saved to later use the variable ψ to recover the most likely path. The termination (2.11c) happens at the end of the observations sequence. After the final step, the most likely state path associated with the sequence is given by backtracking through the ψ variables as shown in (2.11d).
To discuss the computational complexity of this algorithm, the same points of the forward algorithm can be made, leading to affirm that the Viterbi algorithm is also $\mathcal{O}(T \cdot N^2)$.

2.3 Training problem

Real-world applications present many scenarios where the HMMs' parameters are not explicitly known. The relevance of this problem is easily shown by noting that a model gives a lot of insights on the system; moreover, its accuracy was a key assumption in the previous computations. The goal is to find the model λ that maximizes $P(O|\lambda)$, the probability of the observation sequence given the model. Unfortunately, this problem is very complex and there is no known way to analytically solve this maximization problem for any given finite observation sequence.
Despite this, a number of techniques can be used to locally maximize $P(O|\lambda)$; a very popular one is the *Baum-Welch* method, that starts from a guess of the model and iteratively performs reestimations of the parameters to improve it. This algorithm introduces a new key variable: $\xi(i,j)$, the probability of being in state S_i at time t and in state S_j at time $t+1$

$$\xi_t(i,j) = P(q_t = S_i, q_{t+1} = S_j | O, \lambda). \tag{2.12}$$

It can be useful to express this equation using the forward and backward variables. In fact, the forward variable $\alpha_t(i)$ accounts for the observations from the first one up to O_t in state S_i; the backward variable $\beta_{t+1}(j)$ does the complementary job, considering the observation sequence starting in state S_j and from observation O_{t+1} up to the last one. The step between t and $t+1$ has been left out: to tie the two variables, it's necessary to include the probability of transitioning from state S_i to S_j and observing O_{t+1}, which is $a_{ij}b_j(O_{t+1})$. The new formulation of ξ_t can be written as

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{p=1}^{N}\sum_{q=1}^{N}\alpha_t(p)a_{pq}b_q(O_{t+1})\beta_{t+1}(q)}. \tag{2.13}$$

By looking at the definition of the γ_t variable given in (2.7), it can be related with ξ_t:

$$\gamma_t(i) = \sum_{j=1}^{N}\xi_t(i,j) \tag{2.14}$$

Recalling the previous definition of γ_t given at (2.7) is important to notice that by summing $\gamma_t(i)$ over t, the obtained quantity can be interpreted as the expected number of times that the state S_i is visited, or equivalently as the expected number of transitions from state S_i (if we exclude the last observation at time T):

$$\sum_{t=1}^{T-1}\gamma_t(i) = \text{expected number of transitions from state } S_i \tag{2.15}$$

In a similar way, the sum of $\xi_t(i,j)$ over t can be interpreted as the expected number of transitions from S_i to S_j:

$$\sum_{t=1}^{T-1}\xi_t(i,j) = \text{expected number of transitions from state } S_i \text{ to } S_j \tag{2.16}$$

These interpretations lead to the definition of two reestimation formulas for the initial distribution and the transition probabilities:

$$\bar{\pi}_i = \gamma_1(i) \tag{2.17a}$$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)} \tag{2.17b}$$

The HMMs that have been object of study at this point have continuous emission densities, that in the most general case can be written as:

$$b_j(O) = \sum_{m=1}^{M} c_{jm}\mathfrak{R}[O,\mu_{jm},U_{jm}] \tag{2.18}$$

where O is the observations sequence, c_{jm} is the mixture coefficient of the m-th mixture in state S_j and $\mathfrak{R}$ is a log-concave or elliptically symmetric density with mean vector μ_{jm} and covariance matrix U_{jm}, again for the m-th mixture in state S_j. It can be shown ([10–12]) that the reestimation formulas for the coefficients of the mixture density have the following form

$$\bar{c}_{jk} = \frac{\sum_{t=1}^{T}\gamma_t(j,k)}{\sum_{t=1}^{T}\sum_{k=1}^{M}\gamma_t(j,k)}, \tag{2.19a}$$

$$\bar{\mu}_{jk} = \frac{\sum_{t=1}^{T} \gamma_t(j,k) \cdot O_t}{\sum_{t=1}^{T} \gamma_t(j,k)}, \quad \text{and} \tag{2.19b}$$

$$\bar{U}_{jk} = \frac{\sum_{t=1}^{T} \gamma_t(j,k) \cdot (O_t - \mu_{jk})(O_t - \mu_{jk})'}{\sum_{t=1}^{T} \gamma_t(j,k)}. \tag{2.19c}$$

where prime denotes the transposition of the vector and $\gamma_t(j,k)$ is simply $\gamma_t(j)$ relative to the k-th mixture component. According to the context and the scope of the thesis, the mixture model is reduced to a univariate Gaussian distribution; the emission density for a state S_j can be rewritten as

$$b_j(O) = \mathfrak{R}[O, \mu_j, \sigma^2], \tag{2.20}$$

where μ_j is the mean and σ^2 is the variance of the Gaussian distribution associated with the state S_j. Thus, the reestimation formulas can also be simplified (e.g. by getting rid of the mixture weight coefficients) and rewritten:

$$\bar{\mu}_j = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot O_t}{\sum_{t=1}^{T} \gamma_t(j)} \tag{2.21a}$$

$$\bar{\sigma^2}_j = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot (O_t - \mu_j)^2}{\sum_{t=1}^{T} \gamma_t(j)} \tag{2.21b}$$

Applying the reestimation formulas (2.17a), (2.17b), (2.21a) and (2.21b) produces a reestimated model $\bar{\lambda}$; the Baum-Welch algorithm guarantees that either the original model λ is a critical point of the likelihood function (the result would be $\lambda = \bar{\lambda}$) or the model $\bar{\lambda}$ is more likely than the previous one, meaning that $P(O|\bar{\lambda}) > P(O|\lambda)$. The iteration of this procedure converges to a local maximum and produces a maximum likelihood estimate of the model, providing a solution to the training problem.

2.3.1 Starting model

The Baum-Welch procedure requires the definition of a starting model; even though the number of states is generally known (or can be guessed or estimated), to obtain good results a good definition of the starting model $\lambda = (A, B, \pi)$ is necessary. Unfortunately, most of the times little knowledge is possessed about the system; thus, there is no straightforward answer to this problem. As discussed by Rabiner in [8], experience shows that for A and π either random or uniform initial estimates are adequate for useful parameters reestimation. For continuous emission distributions B, the starting parameters are essential. Such parameters can be obtained with several techniques, such as manual segmentation of the observation sequence into states or k-means segmentation with clustering.

2.4 Compressed algorithms

The wavelet compression of the data has a big impact on the mechanisms of the algorithms. In fact, the compression removes the necessity to consider the observations individually, allowing to focus on groups of them called *blocks*. As anticipated

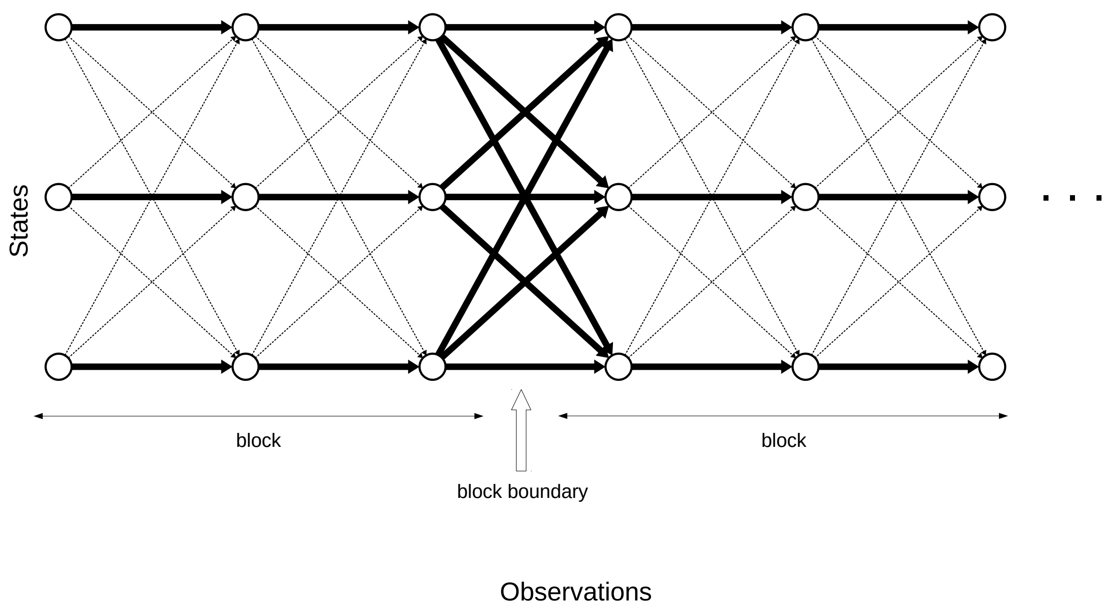


Figure 2.2: The structure on which the compressed algorithms operate; it generates from a subset of the links in the lattice structure.

in Section 1.3, a block is a series of observations where the underlying state can be considered constant and with sufficient statistics to perform the computations required in the algorithms of interest. The structure used by all the algorithms above, as said many times, is the one in Figure 2.1; not knowing anything about the generating path of the observations forces to consider all the possible ones, evaluating at each time step the contribute of every possible state. Compressing the observations opens up new possibilities by modifying that lattice structure into the one represented in Figure 2.2. Since inside a block the state can be assumed to remain the same, the transitions from other states are reputed too unlikely and thus ignored.
To talk formally about compression and its impact on the computations, it's necessary to introduce some notation; this is done taking [13] as a starting point, but applying some changes to avoid conflicts with already defined symbols and to put more emphasis on some concepts.
A partition of the observations in blocks can be denoted as $Y := \{Y_w\}_{w=1}^{W}$, where Y_w is a single block and W is the number of blocks forming the partition. A block contains n_w observations; each one is referred to using the symbol $y_{w,k}$ which is indexed by the block number w and by the position inside the block k. The summary statistics gathered for each block are the following;

$$n_w, \qquad \Sigma_{1,w} := \sum_{k=1}^{n_w} y_{w,k}, \qquad \Sigma_{2,w} := \sum_{k=1}^{n_w} y_{w,k}^2 \tag{2.22}$$

To figure out how the computation varies with the introduction of this block structure, it's useful to thoroughly examine the calculations in Wiedenhoeft's PhD thesis [13] where HaMMLET wass developed. Remembering the definition of the forward variable at (2.3) and its computation formula at (2.4b), being inside a block Y_w only allows self-transitions; this reduces the computation of the next forward variable to

$$\alpha_{t+1}(j) = \alpha_t(j) a_{jj} b_j(O_{t+1}). \tag{2.23}$$

Using induction, the forward variable relative to the whole block has to account for $n_w - 1$ self-transitions, one transition to state S_j and n_w emissions; it can be expressed as

$$\alpha_w(j) = \sum_{i=1}^{N} \Big[\alpha_{w-1}(i) a_{ij} \Big] a_{jj}^{n_w - 1} \prod_{k=1}^{n_w} b_j(y_{w,k}). \tag{2.24}$$

An analogous point can be made on the backward and the Viterbi algorithms; the key part, though, is that this formula still relies on individual observations. To exploit summary blocks statistics, the term accounting emissions and self-transitions within a block can be rewritten by making the Gaussian emissions explicit

$$a_{jj}^{n_w - 1} \prod_{k=1}^{n_w} b_j(y_{w,k}) = \frac{a_{jj}^{n_w - 1}}{\sqrt{2\pi}^{n_w} \sigma_j^{n_w}} \exp\left(- \sum_{k=1}^{n_w - 1} \frac{(y_{w,k} - \mu_j)^2}{2\sigma_j^2} \right). \tag{2.25}$$

The factors outside the exponential can be brought in, also providing implementation advantages discussed later in Section 3.2.2. This yields

$$\exp\left(- \sum_{k=1}^{n_w - 1} \frac{(y_{w,k} - \mu_j)^2}{2\sigma_j^2} + (n_w - 1)\log(a_{jj}) - n_w \log(\sigma_j) - n_w \log(\sqrt{2\pi}) \right). \tag{2.26}$$

The exponent can finally be rewritten using the blocks summary statistics:

$$E_w(j) := \frac{2\mu_j \Sigma_{1,w} - \Sigma_{2,w}}{2\sigma_j^2} + K(n_w, j), \quad \text{and} \tag{2.27a}$$

$$K(n_w, j) := (n_w - 1)\log(a_{jj}) - n_w \Big(\log(\sigma_j) + \frac{\mu_j^2}{2\sigma_j^2} + \frac{1}{2}\log(2\pi) \Big). \tag{2.27b}$$

As pointed out in Wiedenhoeft's PhD thesis [13], an equivalent term can be easily derived also for non-Gaussian emissions that belong to the exponential distribution family.

2.5 Formal transformations

To perform the other calculations, the equations have to be adapted using the reformulation above. The following sections contain the adaptation of the algorithms to the compression scheme.

2.5.1 Forward algorithm

Restructuring the forward algorithm doesn't bring a new meaning to the new variable; for a block, $\alpha_w(i)$ is the approximation of the uncompressed forward variable at the end of the block. The induction phase has already been defined in (2.24); adding the other steps yields:

$$\alpha_1(i) = \pi_i e^{E_1(i)}, \qquad 1 \le i \le N \tag{2.28a}$$

$$\alpha_w(j) = \left[\sum_{i=1}^{N} \alpha_{w-1}(i) a_{ij}\right] e^{E_w(j)}, \qquad 1 \le w \le W, \quad 1 \le j \le N \tag{2.28b}$$

$$P_Y(O|\lambda) = \sum_{i=1}^{N} \alpha_W(i). \tag{2.28c}$$

2.5.2 Backward algorithm

The backward algorithm follows a very similar transformation; $\beta_w(i)$ is defined here as the backward variable at the start of a block:

$$\beta_W(i) = 1, \qquad 1 \le i \le N \tag{2.29a}$$

$$\beta_w(i) = \sum_{j=1}^{N} a_{ij} e^{E_{w+1}(j)} \beta_{w+1}(j), \qquad w = W-1, W-2, \cdots, 1, \quad 1 \le i \le N. \tag{2.29b}$$

2.5.3 Viterbi algorithm

The Viterbi algorithm is based on the forward algorithm with the substitution of the sum over all the states with the max operator; again, all the variables refer to the end of a block:

$$\begin{aligned} \delta_1(i) &= \pi_i e^{E_1(i)}, \qquad 1 \le i \le N \\ \psi_1 &= 0, \end{aligned} \tag{2.30a}$$

$$\begin{aligned} \delta_w(j) &= \max_{1\le i\le N} \Big[\delta_{w-1}(i)\ a_{ij}\Big] e^{E_w(j)}, \qquad 1 \le w \le W, \quad 1 \le j \le N \\ \psi_t(j) &= \operatorname*{argmax}_{1\le i\le N} \Big[\delta_{w-1}(i)\ a_{ij}\Big], \qquad 1 \le w \le W, \quad 1 \le j \le N \end{aligned} \tag{2.30b}$$

$$\begin{aligned} P^* &= \max_{1\le i\le N} \Big[\delta_W(i)\Big], \\ q_W^* &= \operatorname*{argmax}_{1\le i\le N} \Big[\delta_W(i)\Big], \end{aligned} \tag{2.30c}$$

$$q_w^* = \psi_{w+1}(q_{w+1}^*), \qquad w = W-1, W-2, \ldots, 1. \tag{2.30d}$$

2.5.4 Baum-Welch algorithm

The Baum-Welch algorithm is more complex than the others, having more variables to calculate for the parameters reestimation. Following the same order of Section 2.3, the first computation of interest is $\xi_t(i,j)$, as defined in (2.13). To reason about the following computations, it's important to remember that both the compressed forward and backward variables refer to the end of a block. Given this, two different situations happen based on computing ξ inside or outside a block. Inside a block, it's easy to see that $\xi_t(i,j) = 0$ for $i \neq j$; if w indicates the block, when $i = j$ it becomes:

$$\begin{aligned}\xi_t(i,i) &= \frac{\alpha_t(i) a_{ii} b_i(O_{t+1}) \beta_{t+1}(i)}{P(O|\lambda)} = \\ &= \frac{\alpha_w(i)}{a_{ii}^{n_w - t} \prod_{k=t}^{n_w} b_i(O_k)} a_{ii} b_i(O_{t+1}) \beta_w(i) a_{ii}^{n_w-(t+1)} \prod_{k=t+2}^{n_w} b_i(O_k) \frac{1}{P_Y(O|\lambda)} = \\ &= \frac{\alpha_w(i)\beta_w(i)}{P_Y(O|\lambda)}\end{aligned} \tag{2.31}$$

Since the right expression is not dependent on t, inside a block the variable is constant over time. Instead, at the boundary between two blocks:

$$\begin{aligned}\xi_t(i,j) &= \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)} = \\ &= \alpha_w(i) a_{ij} b_j(O_{t+1}) \beta_{w+1}(j) a_{jj}^{n_{w+1}-1} \prod_{k=t+2}^{n_{w+1}} b_j(O_k) \frac{1}{P_Y(O|\lambda)} = \\ &= \frac{\alpha_w(i) a_{ij} e^{E_{w+1}(j)} \beta_{w+1}(j)}{P_Y(O|\lambda)}\end{aligned} \tag{2.32}$$

For the purpose of rewriting reestimation equations, it's useful to define the ξ variable for a block in the following way:

$$\xi_w(i,j) = \sum_{t \in Y_w} \xi_t(i,j) = \tag{2.33}$$

$$= \frac{1}{P(O|\lambda)} \cdot \begin{cases} (n_w - 1)\alpha_w(i)\beta_w(i) + \alpha_w(i) a_{ij} e^{E_{w+1}(j)} \beta_{w+1}(j) \ , & \text{for } i = j \wedge w \neq W \\ (n_W - 1)\alpha_W(i)\beta_W(i) \ , & \text{for } i = j \wedge w = W \\ \alpha_w(i) a_{ij} e^{E_{w+1}(j)} \beta_{w+1}(j) \ , & \text{for } i \neq j \wedge w \neq W \\ 0 \ , & \text{for } i \neq j \wedge w = W \end{cases} \tag{2.34}$$

Moving forward, it's interesting to note that by interpreting $\gamma_t(i)$ as the probability of visiting the state S_i at time t, the variable is also constant over t inside a block (also implied from the result above); this means that for any t inside a block, any $\gamma_t(i)$ can be representative for the whole block; remembering that the forward and backward variables both refer to the end of a block, the equation (2.8) can be rewritten:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} = \frac{\alpha_w(i)\beta_w(i)}{P_Y(O|\lambda)} \tag{2.35}$$

It's worth noting that this reformulation correctly maintains the definition given in (2.14). For convenience it's useful to define $\gamma_w(i)$ as the representative value for a block, which means that for all the t associated with a block, $\gamma_w(i) = \gamma_t(i)$.
The reestimation formulas at (2.17a), (2.17b), (2.21a) and (2.21b) follow the new definitions of $\xi_t(i,j)$ and $\gamma_t(i)$. Using the equations above, they can be rewritten:

$$\bar{\pi}_i = \gamma_1(i) \tag{2.36a}$$

$$\bar{a}_{ij} = \frac{\sum_{w=1}^{W} \xi_w(i,j)}{\sum_{w=1}^{W} \Big[n_w \gamma_w(i)\Big] - \gamma_W(i)} \tag{2.36b}$$

The mean and standard deviation reestimations follow a slightly more complex reformulation, both for the general mixture and the single Gaussian distribution. In particular, (2.21a) multiplies the single observation value by the respective $\gamma_t(i)$. Since $\gamma_t(i)$ is constant inside a block, the equation can be rewritten as:

$$\bar{\mu}_j = \frac{\sum_{t=1}^{T} \gamma_t(j) O_t}{\sum_{t=1}^{T} \gamma_t(j)} = \frac{\sum_{w=1}^{W} \gamma_w(j) \sum_{k=1}^{n_w} y_{w,k}}{\sum_{w=1}^{W} \gamma_w(j) \cdot n_w} = \frac{\sum_{w=1}^{W} \gamma_w(j) \cdot \Sigma_{1,w}}{\sum_{w=1}^{W} \gamma_w(j) \cdot n_w} \tag{2.37}$$

The same reasoning applies to the variance reestimation:

$$\bar{\sigma}_j^2 = \frac{\sum_{t=1}^{T} \gamma_t(j)(O_t - \mu_j)^2}{\sum_{t=1}^{T} \gamma_t(j)} = \frac{\sum_{w=1}^{W} \gamma_w(j) \Big[\Sigma_{2,w} - 2\bar{\mu}_j \Sigma_{1,w} + n_w \bar{\mu}_j^2\Big]}{\sum_{w=1}^{W} \gamma_w(j) \cdot n_w} \tag{2.38}$$

2.6 Relevant model parameters

The computational complexity of the compressed algorithms is lower than the standard versions; since the blocks are used instead of the individual observations, it goes from $\mathcal{O}(T \cdot N^2)$ to $\mathcal{O}(T \cdot W^2)$, where W is the total number of blocks obtained from the compression. The complexity analysis refers to an infinite amount of data; the actual efficiency gain depends on several factors: the separation of the states, the self-transition probabilities and the implementation details (which will be discussed in Chapter 3).
These parameters also impact the results errors due to the approximation introduced by the block compression (as in (2.24)). Trying to find some conditions for which the compressed algorithms always work well (or badly) in terms of speed and accuracy is one of the main goals of this thesis, leading to a sensible choice on which set of algorithms to utilize in different situations.
This section presents the most relevant factors that influence the results quality, discussing their relevance and eventual conditions under which the compressed algorithms should yield a substantial advantage over the standard ones.

2.6.1 State separation

The state separation is a key factor in the compressed algorithms; if the states are well-separated, it's easier to distinguish between two of them. This means that the

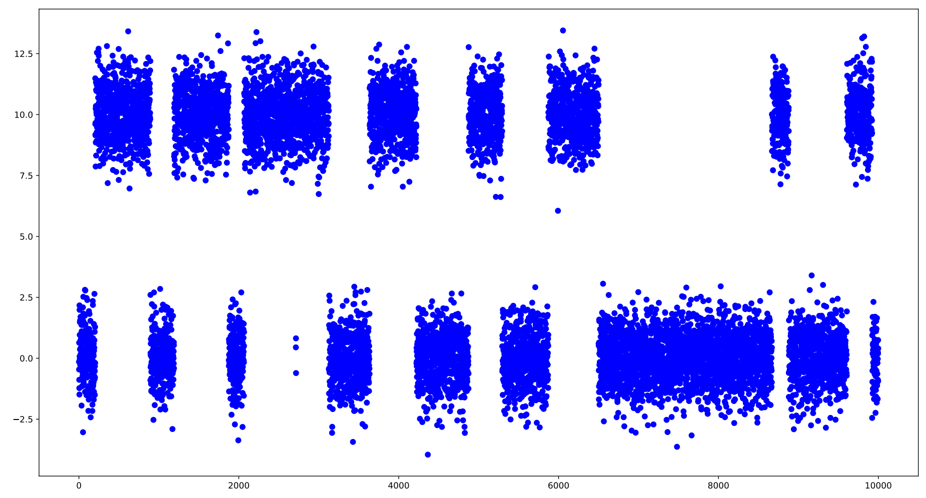


Figure 2.3: Data generated using a two-state HMM with distributions $N(0, 1^2)$ and $N(10, 1^2)$.

wavelet compression will produce blocks with clear boundaries, because some detail coefficients of the Maxlet transform will be high due to a bigger jump between two observations values belonging to different states. Figure 2.3 shows an example of data with clear distinction between the states.

If the states are well-separated, the approximation made by the block compression of neglecting some states contribution becomes more accurate. This is true because the emission probability of an observation for a state that didn't generate it gets very close to zero. In this context, it's useful to formally define a measure η_{S_1,S_2} of how much well-separated two states are. Since for a Gaussian distribution $N(\mu, \sigma^2)$ it's known that 99.97% of the values lie within three standard deviations from the mean, a good measure definition could be:

$$\eta_{S_1,S_2} = \frac{|\mu_{S_1} - \mu_{S_2}|}{3(\sigma_{S_1} + \sigma_{S_2})} \tag{2.39}$$

The power of this definition lies in the fact that it's easy to understand visually: if $\eta_{S_1,S_2} = 1$ it means that the two distributions touch exactly after their respective 3σ; if $\eta_{S_1,S_2} \ll 1$, the distributions overlap for a significant portion; if $\eta_{S_1,S_2} \gg 1$, then the distributions are clearly separated. Also, a measure of $\eta_{S_1,S_2} = 0$ indicates that the two distributions have the same mean. When having more than two states, this separation should apply between every pair of consecutive distributions (ordered by mean). This η will be denoted as *separation coefficient* for easier reference.

It should be clear that a higher state separation should reduce the error between the compressed and the standard algorithms; a more precise analysis will be conducted with the results evaluation in Chapter 4.

2.6.2 Self-transition probabilities

High self-transition probabilities should help the compression process in more than one way: for example, they make the blocks bigger, directly implying the production of fewer blocks for a fixed amount of observations and thus saving precious computation time. The impact on accuracy is hard to evaluate: for a fixed observation sequence, having less blocks could increase the error because more state paths are ignored; on the other hand, the contribution of the other state paths become smaller, thus decreasing the approximation error. To better understand the actual effects of this parameter on it, extensive testing will be conducted in Chapter 4.

3
Methods

This chapter describes the implementation structure, covering in detail the differences between the standard algorithms and their compressed version. Moreover, typical problems (e.g. numerical precision) will be addressed, discussing applied solutions and possible improvements.
The language of choice is C++, after considering others such as Python and R; it sacrifices some simplicity for the sake of efficiency, both in terms of speed and memory management. A great and popular compromise is to expose some Python bindings to an internal C++ structure, providing a simpler interface to the underlying complex implementation.

3.1 Framework description

In the design phase of a new project it's very useful to draw a scheme describing the class structure as the one in Figure 3.1, also listing what external tools will be used and how they will interface to the main components. To have more control over the code and allow for fair speed comparisons, both the HMM representation and the standard algorithms have been implemented from scratch.
The external components used are: CXXopts, to parse input arguments from command line; HaMMLET [13], for wavelet compression and the relative data structures; Pomegranate, a Python framework used for data generation.
The tool has been named WaHMM, after **Wa**velets **H**idden **M**arkov **M**odel and following the style of HaMMLET. The core interfaces with the external components through the `parser` and the `Compressor` elements. The standard algorithms and their compressed version have been separated in different files for easier management.

3.2 Implementation details

This section will describe several implementation choices, the reasoning behind them and their impact on results.

3.2.1 Parser

The parser allows for an easy interaction with the tool through the specifications of various options using the common UNIX style. Although it can be quite verbose, it's a very effective instrument to define input parameters and which algorithms to

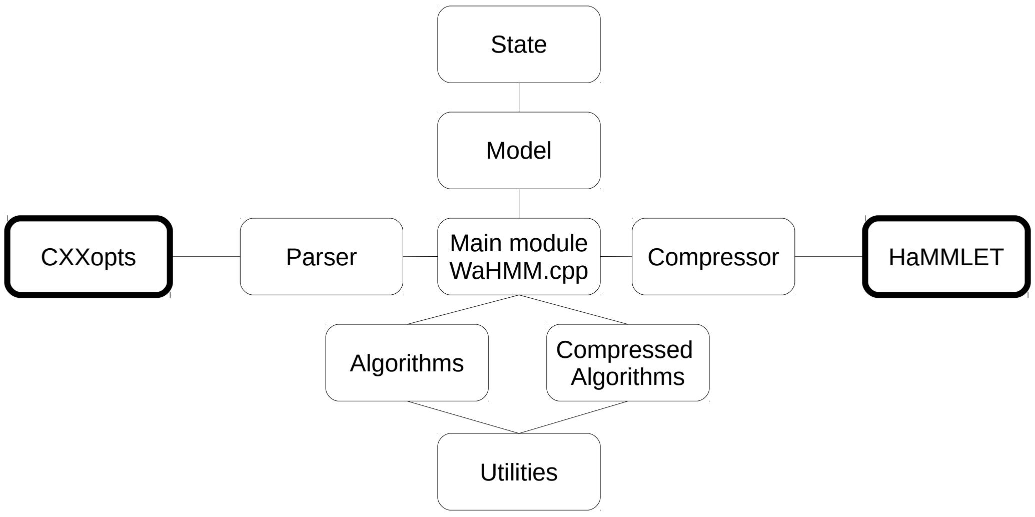


Figure 3.1: Structure of the implemented code (thin border) and its connection points to the external libraries used (thick border).

execute. This interface allows to input a model through command line or file, as well as giving input observations as a space-separated list of floats in a file or with a binary file format. Several options allow to choose which algorithms to execute, together with controls for verbosity and saving the results to files.

3.2.2 Numerical errors

Any kind of numerical method or scientific computation faces the problem of numerical errors. Representing real numbers on a machine is one thing, but observing that representation forces their decimal expansion to be truncated at some point. There is an entire sub-field of programming language theory called "exact real-number computation" devoted to representing real numbers on computers; more information on that can be found on Haskell's wiki page [14]. At the end of the day, the smallest difference between two numbers that a computer can recognize is called *machine epsilon*; if their difference is smaller, it's rounded to zero producing a rounding error. The machine epsilon is platform-dependent, but generally it can be close to 10^{-16}. Since the product of probabilities can get very small fairly quickly, this problem would cause increasingly bigger errors on all the computations (e.g. forward matrix after some time steps).
To address this well-known problem, many approaches are possible: to achieve higher precision, a double data type is used to store results, redefined as `wahmm::real_t`;

the workspace `wahmm` is used to avoid naming conflicts with libraries such as HaMMLET. For numerical problems, the strategy followed in this thesis is to use logarithms of probabilities. There are several advantages with this approach: the logarithm naturally scales the $[0, 1]$ interval to $(-\infty, 0]$ so that when probabilities get smaller, their logarithm becomes more negative; all the products required by the algorithms become summations, which is much easier and faster to perform. Using a logarithmic space also saves computations in the compressed algorithms; in fact, this transformation removes the exponential function in (2.27a), making it a natural choice for the use in these algorithms.

3.2.3 State representation

For the scope of this thesis, a state is associated with a Gaussian distribution that defines its emission probabilities. As such, accepting the trade-off between generalization loss and efficiency gain, the `State` class directly embeds the parameters of the associated Gaussian distribution. This allows for faster retrieval and update of the parameters, and can be easily expanded to other probability distributions (for future work) by turning `State` into an abstract class and deriving distribution-specific State classes from it. It is worth noting that the emission probability is provided directly in logarithmic space, to avoid useless overheads and speed up the computation.

3.2.4 Data generation

To generate data from the model, the framework Pomegranate is being used. It's a general HMM library, but the fact that the implementation language is Python makes any eventual speed comparison unfair, and thus it won't be used to apply standard algorithms to the model. Pomegranate can generate data in a simple and fast way; some Python scripts interface with it by defining a model that is coherent with the one used or estimated in WaHMM.
Specifically, `generate_data.py` reads the model from a file and generates an observation sequence of some length, optionally saving both the sequence and the generating state path to file in both plaintext and binary form; this allows to read the binary file for faster input processing, shrinking the running times considerably. At this time, k-means clustering can be performed on the data after it's generated.

3.2.5 Wavelet compression

The compression of the data happens through the `Compressor` class, that wraps an interface to HaMMLET. The observation sequence undergoes the Maxlet transform and it's encapsulated in a `BreakpointArray`, a data structure that, given a certain threshold, subdivides the sequence in blocks. Each block gathers observations that are sufficiently close and thus are likely to have been generated while the model was in the same hidden state. A summary of each block statistics is stored in a parallel structure called `IntegralArray`. The combination of these two structures allows to define an interface for simple and efficient querying of consecutive blocks

and their statistics summary. More information on how HaMMLET works can be found in [13].
The threshold used to define blocks can surely be object of discussion: a low threshold will yield more blocks than needed, reducing the efficiency of compression; a high threshold will instead generate fewer bigger blocks that may group observations belonging to different states. The choice is taken from HaMMLET: the threshold is obtained by computing and estimate of the noise variance from the finest detail coefficients of the wavelet transform.

3.2.6 Logarithms summation

Some very useful declarations and functions are present in `utilities.hpp` and `commons.hpp`; other than functions to easily print and free matrices, the most important one is `sum_logarithms()`. Since the program operates in the logarithmic space, the probability products are converted into summations. A sum in the original space, though, has no simple logarithmic equivalent; this situation happens often, as in equation (2.4b)).
Using symbols, the function needed to solve this problem is some F so that, given two elements in logarithmic space $\log(x), \log(y)$ it should produce:

$$\log(x+y) = F(\log(x), \log(y)) \tag{3.1}$$

The simplest option would be converting both elements back to the original space through exponentiation, writing:

$$\log(x+y) = \log(e^{\log(x)} + e^{\log(y)}) \tag{3.2}$$

However, this solution can cause underflow when $\log(x)$ or $\log(y)$ are too negative. A simplification of this allows to write:

$$\log(x+y) = \log(x) + log(e^{\log(y)-\log(x)} + 1) \tag{3.3}$$

when $\log(x) > \log(y)$. Some workaround are required if any of the operand is $-\infty$, implying the presence of some `if` clauses before the actual computation. The `sum_logarithms()` function implements this operation using the `std::log1pf()` function from the standard library of C++ for a more efficient computation of the logarithm.

3.2.7 Saving results

The algorithms results can be saved to file by adding the right option when calling the program. Although the results directory may be changed, the default choice is the `/results` folder. After running the algorithms, WaHMM will write the results in properly named files. To produce an example, a set of standard results can also be generated using Pomegranate through the script `pomegranate_test.py`, more for a comparison with a different approach than for speed and performance.

3.2.8 Test automation

The process of testing the algorithms against generated data is of course central to the thesis work; given the huge number of tests to conduct, it makes sense to automate not only the testing process, but also the extraction of meaningful results. The Python script `automated_test.py` takes care of the testing process through a very simple sequence of steps:

- generate the model according to the topology and number of states relative to the current case that is subject to test;
- generate data from the model and save to file both the sequence and the generating path;
- estimate a model for the training problem from the data and save it to file;
- execute the algorithms, timing their execution and computing some performance measures;
- periodically save the test results to file for future analysis.

The Python script `results_aggregation.py` allows to analyze the results in an automated way, by not only computing the differences between the compressed and standard algorithms in the chosen metrics, but also producing their graphical representation in the form of plots, boxplots, and a merge of the two. The last one is the format used for the figures discussed in Chapter 4.

3.2.9 Other Python files

Several Python scripts have been written to ease WaHMM's usage and setup, not only with an interface that may be simpler to use, but that also allows an easier automation of the testing process. The `create_model_file.py` script defines a model and saves it to file, so that it can be used in other scripts and later be imported by WaHMM. `plot_data.py` and `plot_kmeans.py` are utilities to produce some outlook on the generated data and on the estimated model respectively. `viterbi_comparison.py` simply compares Viterbi paths obtained with different algorithms to point out the differences, particulary useful when checking the accuracy of the compressed algorithms against the real generating path. `utilities_io.py` and `utilities_kmeans.py` simply provide helper functions for the other scripts.

3.3 Standard algorithms

This section will discuss the implementation of the standard algorithms; for an easier understanding, it can be useful to reference Figure 2.1 for a structural overview. All the three functions solving the problems accept some boolean flags to influence the type of output: `verbose` prints more information to the standard output; `silence` suppresses all the output messages; `tofile` specifies that the output results should be saved to file.

3.3.1 Evaluation problem

The evaluation problems can be solved by computing the forward matrix. Looking at Figure 2.1, each row is associated to a state; each column represents a time step. After the initialization described in (2.4a), the induction phase is constructed by initializing to $-\infty$ the forward variables for the current time step; then, the `sum_logarithms()` function is applied to accumulate the sum of the products in (2.4b); at last, the emission probability is added and the computation moves to the following time step. A simple sum of the forward variables over all states at the last time step yields the desired $P(O|\lambda)$ probability. Looking at the code, it's easy to confirm the computational complexity of $\mathcal{O}(T \cdot N^2)$ (as it was previously stated in Section 2.1.1).

3.3.2 Decoding problem

From a theoretical perspective, the decoding problem is very similar to the evaluation problem; in fact, the Viterbi algorithm differs from the forward algorithm in applying the *max* operator instead of the sum. The implementation is slightly more complex, requiring an additional matrix (named `statesViterbi`) with the same structure to hold the *argmax* results from (2.11b). After the initialization phase, a loop is used to join the computation with a classic maximum search on-the-fly; this allows to only iterate once over the states. The backtracking described in (2.11d) is then applied by appending each state of the path at the head of a list; in this way, a simple visit of the list yields the Viterbi state path. Again, the computational complexity of $\mathcal{O}(T \cdot N^2)$ that was discussed in Section 2.2 is confirmed by the implementation.

3.3.3 Training problem

The Baum-Welch algorithm is the most complex of the three. It is performed iteratively for a maximum amount of iterations or until the procedure is improving $P(O|\lambda)$ by an amount smaller than a predefined threshold (10^{-9} in the implementation). For this reason, to avoid a big overhead most of the memory allocations happen in `training_problem_wrapper()`, that also handles the iterations of the algorithm. The result returned at each iteration is the evaluation probability $P(O|\lambda)$ relative to the model before the reestimation; this implementation choice avoids one useless computation of the forward matrix at each iteration of the Baum-Welch algorithm at the cost of performing one more iteration than needed after matching the threshold.
The starting model supplied to the procedure can be directly read from a file using the option `estimate`; in the absence of a solid estimation from domain knowledge, it's possible to generate the model estimate through an automatic procedure that, given the number of states, selects some means and variances to be associated with states. The chosen method for this automatic procedure is the application of K-means clustering to the sequence; the proposed Python implementation uses the clusters centroids as estimated means and computes the standard deviation with the canonical squared distance formula. The model is then saved to an output `data/kmeans_model` file, to be imported as a starting model when executing the

training algorithm. This clustering operation can happen either directly when the data is generated, for faster execution, or later on, for externally supplied data.
The theory presented in Section 2.3 advocates for calculating every variable for all the time steps: this implies a huge memory usage to store the forward and backward variables, plus γ_t and ε_t. The procedure doesn't actually need to store all the intermediate results: to save space, the backward variable and γ_t are calculated only for the considered time step; the other variables are simply summed to obtain the final cumulative sum over all time steps. Looking at the reestimation equations, it can be seen that the denominator $P(O|\lambda)$ gets simplified in all of them except for (2.17a); thus, it's more convenient to remove it from the equations and just add it to (2.17a) at the end.
Performing the reestimation of the Gaussian's mean using the formula in (2.21a) produces a problem; in fact, although working in logarithmic space eases the computations for the variable γ_t, it cannot be done when observations are negative; that is, if the reestimated mean is negative, it can't be represented in the logarithmic space. To overcome this problem, the observation sequence must be rescaled by translating it by a value that is strictly greater than the minimum observed value so that only strictly positive observations are present. This rescaling is applied on-the-fly and for the mean reestimation only, to avoid useless computations; the added offset must obviously be removed when moving out of the logarithmic space.

3.4 Compressed algorithms

This section contains a description of how the data compression changes the algorithms in the implementation. To ease the computations, the summary statistics of a block can be retrieved both individually (through `Compressor::blockSize()`, `::blockSum()` and `::blockSumSq()`) and together, through an *ad hoc* data structure called `blockdata` and the related function `Compressor::blockData()`.
A key part of the computation is calculating $e^{E_w(i)}$ for a block. As a first observation, working in logarithmic space allows to avoid the exponentiation and just calculate $E_w(i)$. More interestingly, from (2.27a) it can be seen that $K(n_w, j)$ only depends on the state and the size of the block, and not on the actual observations. To make the algorithms faster, the `Model` class stores an array of hashmaps `Model::mKValues` of the $K(n_w, j)$ values per each state, adding entries as they are computed.

3.4.1 Evaluation problem

The forward algorithm is modified to deal with compressed data. The implementation is very similar to the uncompressed one: the `Compressor` class allows iterating through the blocks to perform the computation of the compressed forward variable following the equations (2.28a), (2.28b) and (2.28c).

3.4.2 Decoding problem

The compressed Viterbi algorithm iterates over the blocks in a similar fashion to the forward algorithm, as it happens for the uncompressed algorithms. To keep track of the actual Viterbi path, though, the block sizes must be recorded somewhere; in this way, during the backtracking phase any block state can be associated to its size, specifying how long does the sequence stay in that state.

3.4.3 Training problem

As shown in previous discussions, the compressed training algorithm is the one that differs the most from the uncompressed version. A first difference that is worth noting is that the backward matrix is computed fully using a compressed backward algorithm; the space overhead is more manageable in this case, since the backward matrix has one column per block and not one per observation. The variables are computed according to the new definitions given in Section 2.5; the mean and the variance computations are done outside of the logarithmic space, to avoid the problems caused by negative values of the sum that would require a translation of all values. An important consideration can be made for (2.27b): $K(n_w, j)$ can be precomputed at each iteration of the Baum-Welch algorithm. To find a balance between avoiding useless computations and performing the same calculation every time, a map is used to keep track of the $K(n_w, j)$ values that have already been encountered; whenever a new one is found, the computation is made for all the states and stored into the map; this allows to speed up the compressed algorithm a little bit more, at the cost of some extra memory.

4

Results

This chapter presents the methodologies that have been chosen to evaluate the developed algorithms, discussing the parameters influence and how the results will be observed; this has the purpose to identify some conditions under which the compressed algorithms are worth using.

4.1 Testing setup

4.1.1 Choosing the parameters

To test the approach developed in this thesis, some HMMs need to be defined. Choo et al. [15] do a very good job defining some of the most frequent and useful model topologies in the field of reference, which is computational biology; nonetheless, they have a much broader scope and are found in many different applications. Three topologies will be analyzed:

- *fully connected model*, with every pair of states being connected and thus with the underlying graph being complete; the fully connected graph also includes the self-loops for each state;
- *circular model*, with an ergodic graph: the states are arranged in a circle, and a transition can only occur towards the same state or to the next one;
- *left-to-right model*, with an acyclic graph with the exception of loops; the states are partially ordered and there are uniquely defined starting and ending states; transitions must be taken to visit the states following that order.

Each topology will be explored using different numbers of states, to see how the performance and accuracy varies. Specifically, models will be defined with 2, 3 and 5 states; this choice should give a perspective on how this parameter influences the results. About the observation sequence, both its length and an expected number of transitions should be discussed. For typical applications, the observations length usually is in the order of 10^5. Experience suggests that an adequate expected number of transitions is 10, which is a realistic magnitude for several kind of sequences of interest. The state separations that will be tested are 10 different values of η as defined in (2.4c), going from 0.1 to 1.0 in increases of 0.1.

Testing algorithms against randomly generated data always presents the risk of introducing a non-deterministic bias into results. To contrast this problem, 100 sequences have been generated for each model: this allows not only to analyze some aggregate values to obtain some summarized information, but also to study the spread of the performance measures and thus the results stability. The chosen aggregation method is the median, since its robustness allows to correctly ignore a solid number of outliers. The number of sequences has been chosen as a compromise between robustness of the statistics and feasibility of the execution times on a laptop.

4.1.2 Results evaluation

Before rushing to the results discussion, it is useful to overview what type of information will be presented to analyze the results and the mathematical tools used to elaborate them. Every measure defined in this section will be plotted against the states separation η.

Defining some measure of relative error is necessary, but it can be tricky, especially when facing zero values; the approach used in this thesis is to use the relative difference, defined as

$$d_r = \frac{|x-y|}{\max(|x|,|y|)}, \tag{4.1}$$

with the caveat of setting the error to zero when both x and y are zero.

For the states estimation, the accuracy is evaluated using another measure borrowed from information theory: the Kullback-Leibler divergence (or KL-divergence), an indicator of how much a probability distribution differs from another one taken as a reference. A deeper explanation of this measure falls out of the scope of this thesis; the formula is presented below:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) log\left(\frac{p(x)}{q(x)}\right) dx \tag{4.2}$$

Now that the mathematical tools have been described, the different performance measures can be properly explained.

The evaluation problem produces the logarithm probability $P(O|\lambda)$ as defined in (2.4c); the performance of the compressed algorithm is measured as the relative difference from the standard result.

For the decoding problem, the indicator has been chosen to be the fraction of errors in the estimated generating path; thus, the performance measure is the relative difference between the compressed and the standard results.

The training problem is much more complex, and summarizing its results to obtain some indicator of how well the algorithm performs is not an easy task. The choice for this thesis is to compare both the compressed and the standard estimates against the real model, along three dimensions: the average accuracy on the states, measured with the KL-divergence; the average error on the transition log probabilities; the average error on the log probability of the initial distribution evaluated at the starting state. These three indicators are computed for both the compressed and the standard algorithms and then compared by simply taking their difference.

Finally, the speedup will be analyzed with a simple ratio of the execution times, separately for the three algorithms; this measure obviously includes the data input processing time, to account for the overhead that the compressed algorithms require to elaborate the data.

To give more insights on the plotted information, each plot has boxplots on the side to describe the distribution of the results. It's easy to see how discussing the results

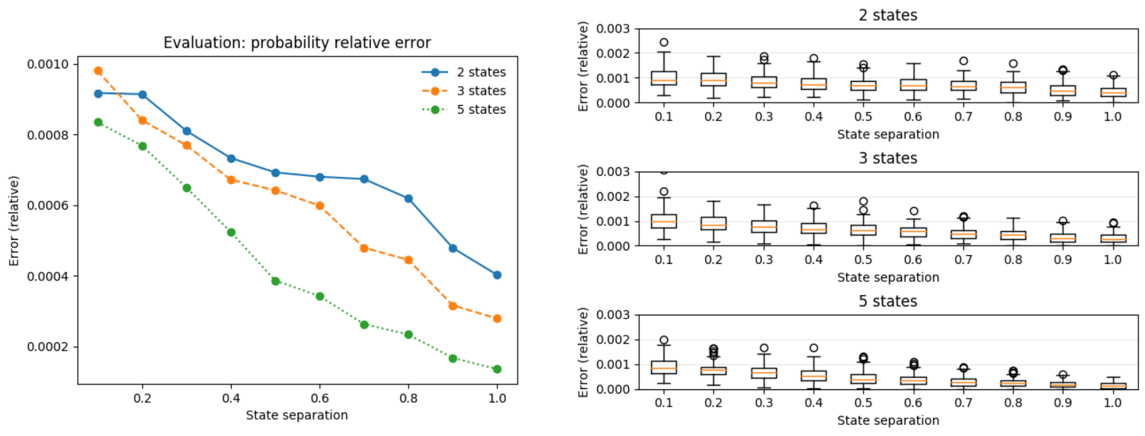


Figure 4.1: Relative difference between the $P(O|\lambda)$ log probabilities of the compressed and standard algorithms. Since the errors are on a big negative log probability, the actual error magnitude is approximately of 10^{-5000}.

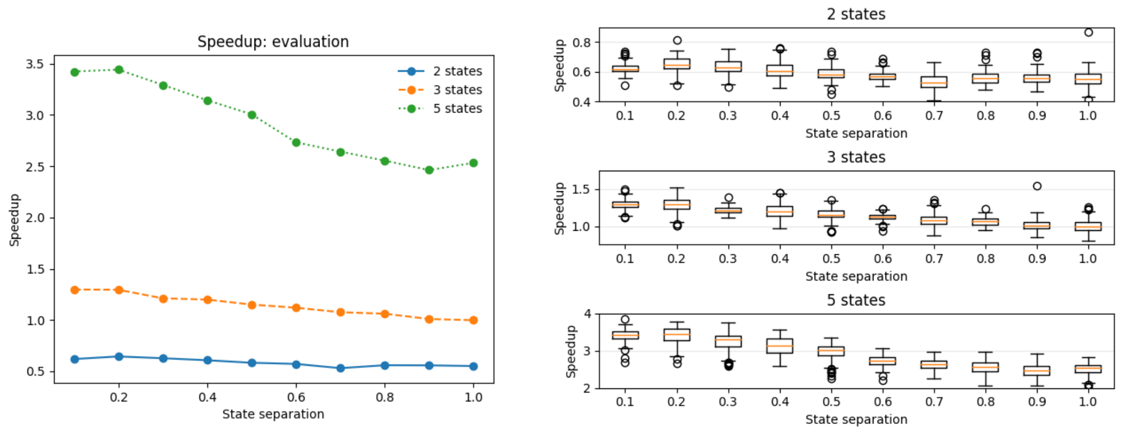


Figure 4.2: Speedup on the evaluation problem using the compressed algorithm, including the input data processing time.

will generate a big amount of figures; for this reason, this chapter will only present the figures for the fully connected model; the other topologies will be discussed in relation with this model and their plots will be grouped in the Appendix A at the end of the thesis.

4.2 Evaluation problem

The information being plotted in Figure 4.1 is the relative difference on the logarithmic probability $P(O|\lambda)$ between the compressed and the standard algorithm. The first thing to notice is that as the state separation increases, the error decreases; this is expected, since having more distinguishable states helps the compression process. In all cases, the error is relatively small and tightly spread, indicating that the compressed algorithm does a good job approximating the standard one on the evaluation problem. Also, the error appears to decrease with a higher number of states in the model. To understand if the use of the compressed algorithm is really worth

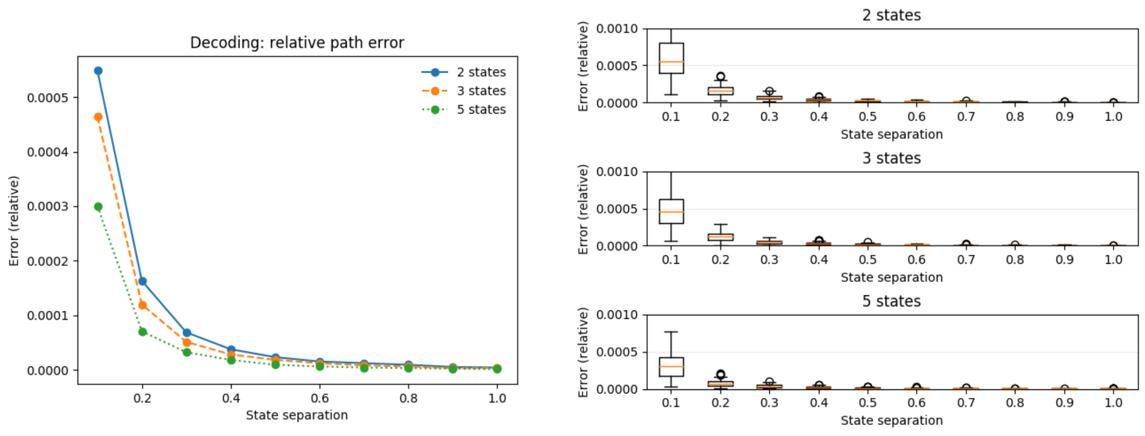


Figure 4.3: Relative difference between the fractions of errors in the estimated generating path.

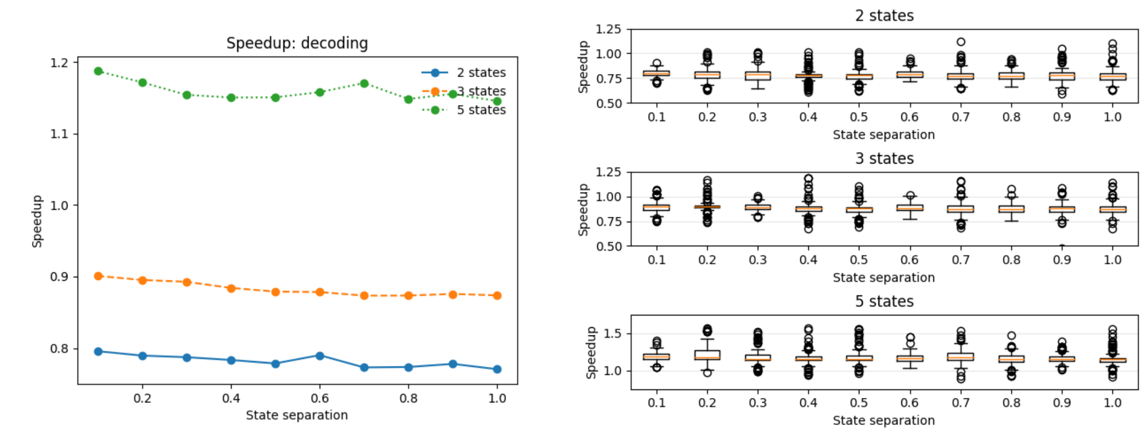


Figure 4.4: Speedup on the decoding problem using the compressed algorithm, including the input data processing time.

it, Figure 4.2 shows the different speedups obtained; although they don't change much with the state separation, the main parameter affecting them is the number of states: the use of the compressed algorithm is thus advised only when the model has a relatively high number of states.
The circular and left-to-right topologies have slightly worse results, but definitely going through the same considerations made for the fully connected model.

4.3 Decoding problem

As properly explained in Section 2.2, the decoding problem is quite similar to the evaluation problem with the forward algorithm being partly modified. Thus, the expected results should more or less align to the performances in the evaluation problem. As Figure 4.3 shows, the general trend is the same. When the state separation is extremely low, the uncertainty on the values is high; thus the actual ordering between the states, although consistent with Figure 4.1, may vary slightly in certain points. Although the performance indicator seems good, Figure 4.4 shows

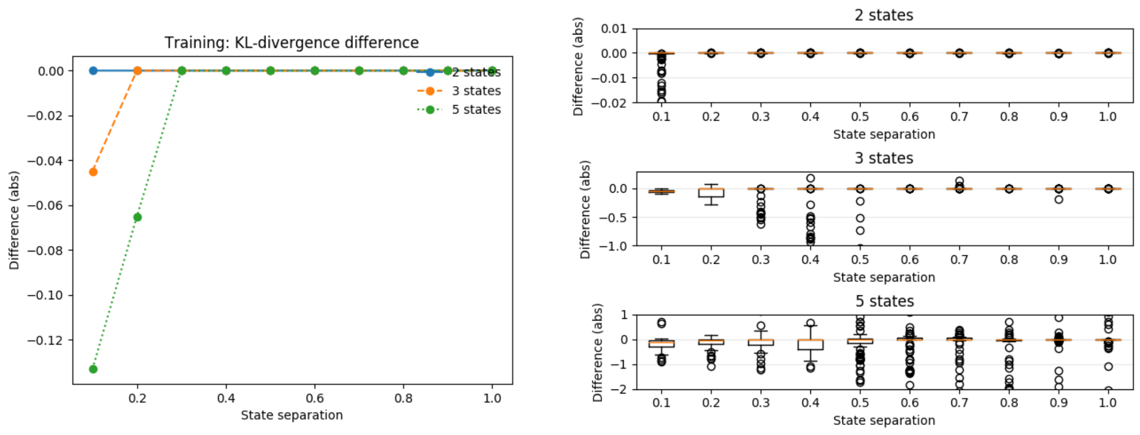


Figure 4.5: Difference between the average KL-divergence for the compressed and standard algorithms.

how the compressed algorithm is generally less efficient than the standard one when the model has a small number of states; moreover, the data has a very high spread, suggesting that the actual speedup depends a lot on the data being generated.
The model that scores better in the decoding problem is the left-to-right model, which intuitively makes sense since each state only has one allowed transition; again, though, the results are pretty similar and thus the compressed algorithm should only be used when the number of states is high enough.

4.4 Training problem

The training problem is the most complex of the three, as it has been said many times at this point. The first plot of interest is in Figure 4.5, showing how well the states are estimated by the compressed algorithm compared to the standard one. While the overall trend is that of a constant small difference for most values of state separation, it's noteworthy that the plotted values are negative when the states are not well-separated. This means that the compressed algorithm is actually more accurate than the standard one, and this accuracy appears to increase with the number of states if the state separation is small enough; the main problem is that, for an increasing number of the states, the spread of the results starts becoming very high. In spite of this, for a 5-states model that is not enough to cause problems or big inaccuracies. Similar considerations can be made for both the transition probabilities and the initial distribution estimations; the compressed training does an overall better job by a small margin, that gets more consistent when the state separation is very low.
The results for other topologies resemble the presented ones quite well. The generally high instability of the results can make the appearance of the plot be less meaningful; but when taking a deeper look at the data, the distribution of the results is mostly skewed towards favoring the compressed algorithm over the standard one. Looking at the speedup in Figure 4.6, there is a very noticeable performance gain that is higher when the state separation is low and the number of states is high. In general,

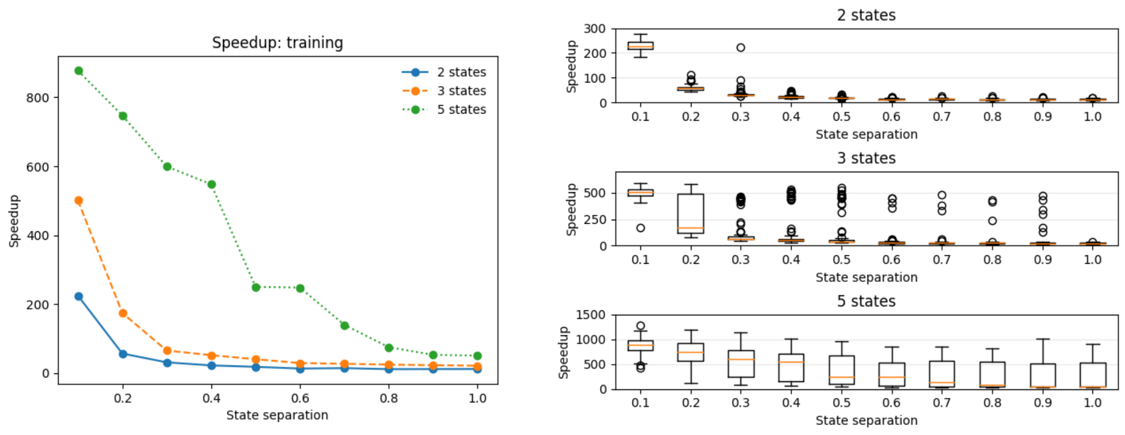


Figure 4.6: Speedup on the training problem using the compressed algorithm, including the input data processing time.

the speedup tends to be extremely high, giving a solid reason to use the compressed training.

5 Conclusion

5.1 Main takeaways

The previous chapter presented a description of the results on a case-by-case basis; after that, it's useful to summarize the insights that have been extracted from the testing process.
A first thing to point out is that the results are pretty consistent over all the tested topologies; this allows to draw some general conclusions that are independent of the topology of the model. Overall, the compressed algorithms perform with an extremely high precision. Specifically, the evaluation and the decoding compressed algorithms perform as well as the standard ones regardless of the states separation. They are slower then their standard counterpart for models with a low number of states, because of the overhead caused by the data compression; fortunately, the speedup gets better for models with a higher number of states, when the running times would start to grow more and more. The training algorithm is the one that definitely performs best: it performs better than the standard algorithm for a low state separation, and the achieved speedup is extremely high.
It's important to remember that there is a number of factors that have been assumed constant throughout the testing process, such as the self-transition probabilities or the sequence length. In the context of big data, for sequences that are much longer than the ones used in testing, the compressed algorithms certainly perform even better than what is showed in Chapter 4.

5.2 Wrapping up

WaHMM gives the opportunity to apply either standard or compressed algorithms to solve the evaluation, decoding, and training problems with an efficient C++ implementation. In the context of scientific research, this thesis will hopefully serve as another confirmation that wavelet compression can work really well to allow hidden Markov models to scale to big sequence data. In particular, it could allow to train a hidden Markov model on a commodity laptop instead of requiring more complex machinery, since the training may be even thousands of times faster.

Bibliography

[1] I. P. Waldmann, "On signals faint and sparse: The ACICA algorithm for blind de-trending of exoplanetary transits with low signal-to-noise," *Astrophys. J.*, vol. 780, p. 23, 2014.

[2] K. R. e. a. Wiedenhoeft J., Cagan A., "Bayesian localization of CNV candidates in WGS data within minutes," *Algorithms Mol Biol*, vol. 14, no. 20, 2019.

[3] I. R. Sipos, A. Ceffer, and J. Levendovszky, "Parallel optimization of sparse portfolios with ar-hmms," *Computational Economics*, vol. 49, pp. 563–578, Apr 2017.

[4] S. Y. M. Gales, "The Application of Hidden Markov Models in Speech Recognition," *Foundations and Trends in Signal Processing*, vol. 1, no. 3, pp. 195–304, 2007.

[5] Z.-U. M. Mozes S., Weimann O., "Speeding Up HMM Decoding and Training by Exploiting Sequence Repetitions," in *Combinatorial Pattern Matching* (B. Ma and K. Zhang, eds.), (Berlin, Heidelberg), Springer Berlin Heidelberg, 2007.

[6] M. P. Mahmud and A. Schliep, "Speeding Up Bayesian HMM by the Four Russians Method," in *Algorithms in Bioinformatics* (T. M. Przytycka and M.-F. Sagot, eds.), (Berlin, Heidelberg), pp. 188–200, Springer Berlin Heidelberg, 2011.

[7] S. A. Wiedenhoeft J., Brugel E., "Fast Bayesian Inference of Copy Number Variants using Hidden Markov Models with Wavelet Compression," *PLoS Comput Biol*, vol. 12, no. 5, 2016.

[8] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. IEEE*, vol. 77, no. 2, pp. 261–266, 1989.

[9] A. B. Romeo, C. Horellou, and J. Bergh, "A wavelet add-on code for new-generation N-body simulations and data de-noising (JOFILUREN)," *Monthly Notices of the Royal Astronomical Society*, vol. 354, pp. 1208–1222, 11 2004.

[10] L. Liporace, "Maximum likelihood estimation for multivariate observations of Markov sources," *IEEE Transactions on Information Theory*, vol. 28, no. 5, pp. 729–734, September 1982.

[11] B. H. Juang, "Maximum-likelihood estimation for mixture multivariate stochastic observations of Markov chains," *AT&T Technical Journal*, vol. 64, no. 6, pp. 1235–1249, July-Aug. 1985.

[12] B. Juang, S. Levinson, and M. Sondhi, "Maximum likelihood estimation for multivariate mixture observations of markov chains," *IEEE Transactions on Information Theory*, vol. 32, pp. 307–309, 3 1986.

[13] J. Wiedenhoeft, *Dynamically compressed Bayesian hidden Markov models using Haar wavelets.* PhD thesis, Rutgers, The State University of New Jersey, 2018.

[14] “Exact real arithmetic - haskellwiki.”

[15] . Z. L. Choo K. H., Tong J. C., “Recent applications of hidden markov models in computational biology,” *Genomics, Proteomics & Bioinformatics*, vol. 2, 5 2004.

A

Appendix 1

A.1 Fully-connected model

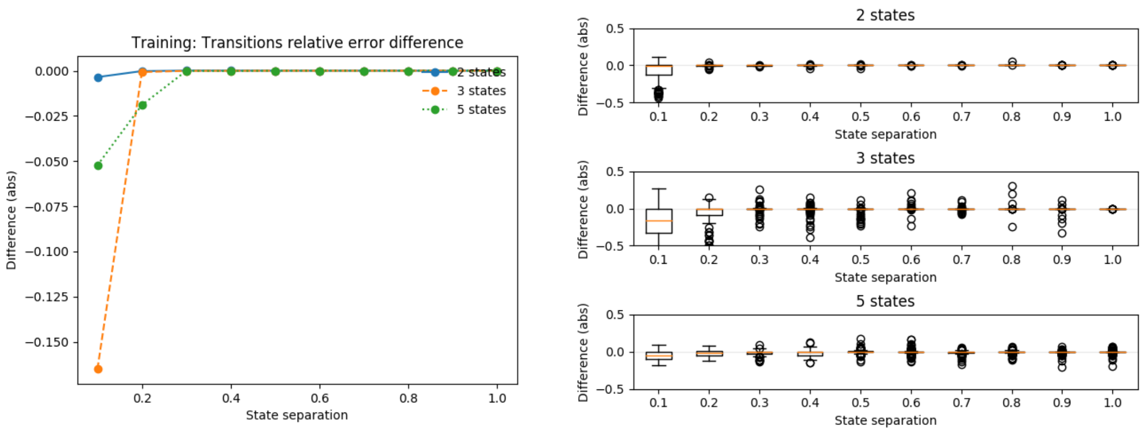


Figure A.1: Relative difference between the average error on the log transition probabilities of the compressed and standard algorithms.

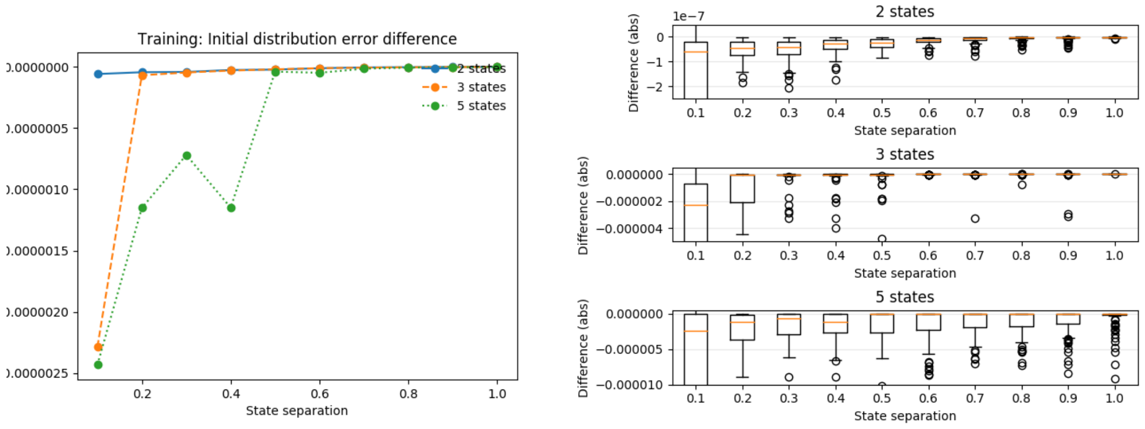


Figure A.2: Relative difference between the average error on the log probability of the starting state in the initial distribution for the compressed and standard algorithms.

A.2 Circular model

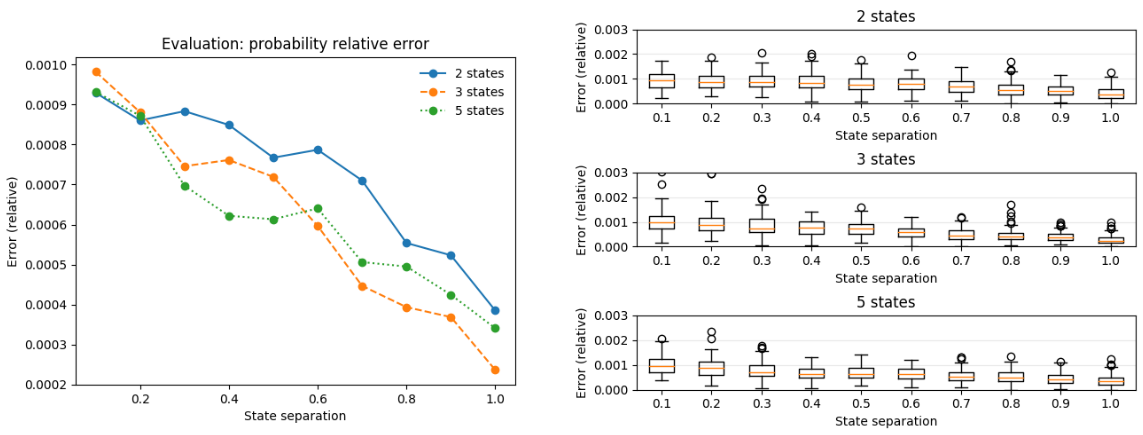


Figure A.3: Relative difference between the $P(O|\lambda)$ log probabilities of the compressed and standard algorithms.

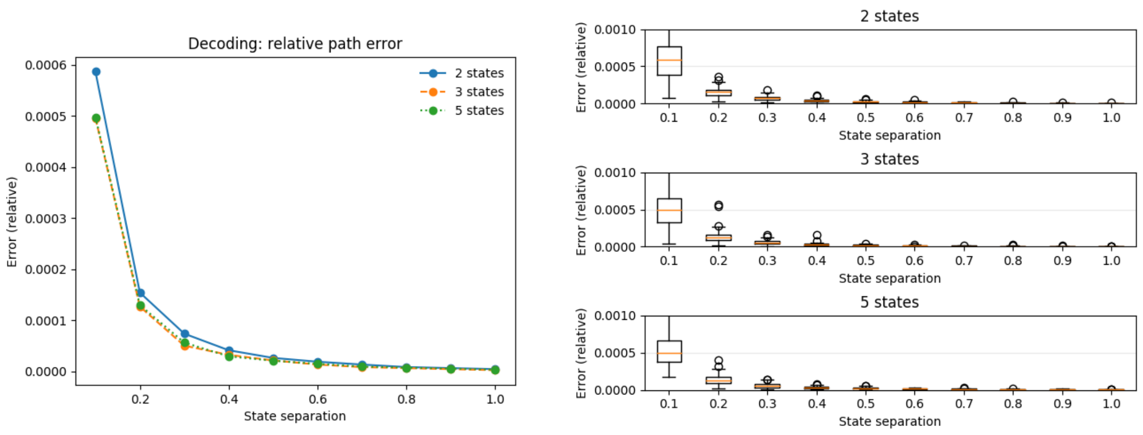


Figure A.4: Relative difference between the fractions of errors in the estimated generating path.

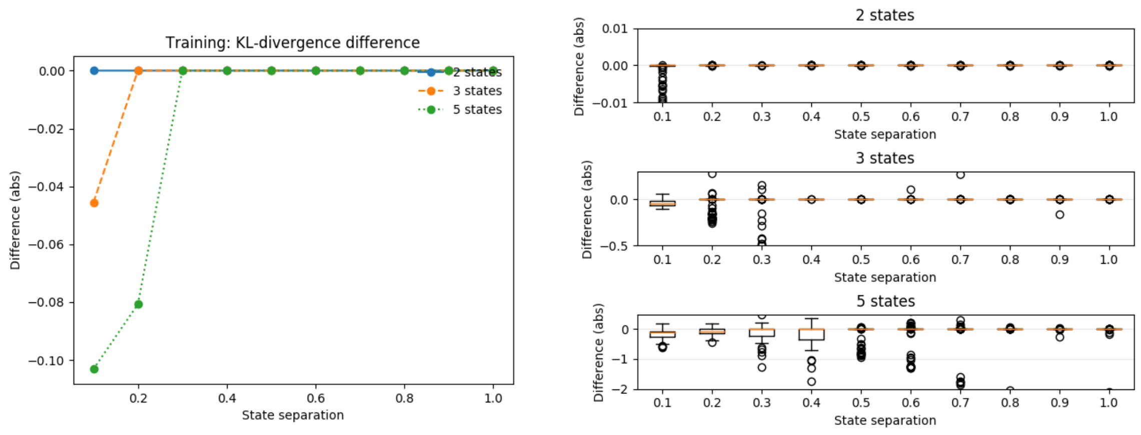


Figure A.5: Difference between the average KL-divergence for the compressed and standard algorithms.

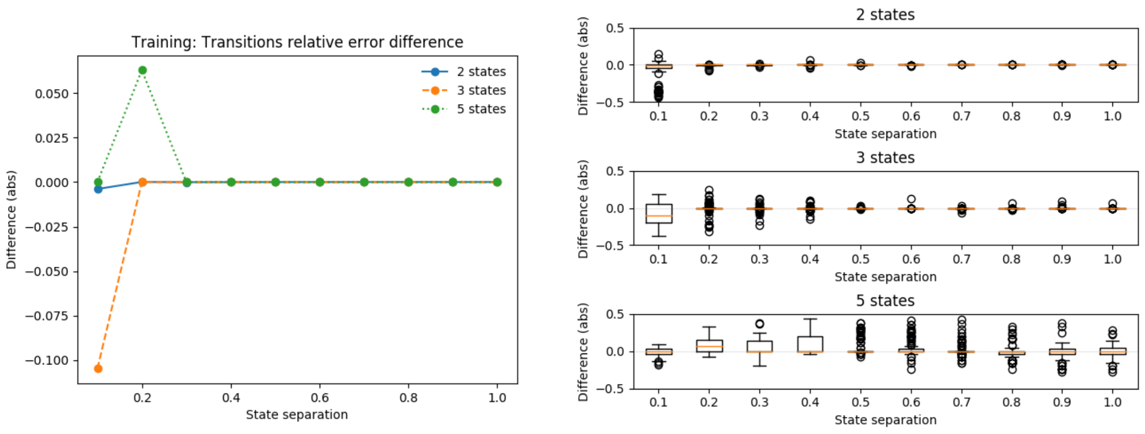


Figure A.6: Relative difference between the average error on the log transition probabilities of the compressed and standard algorithms.

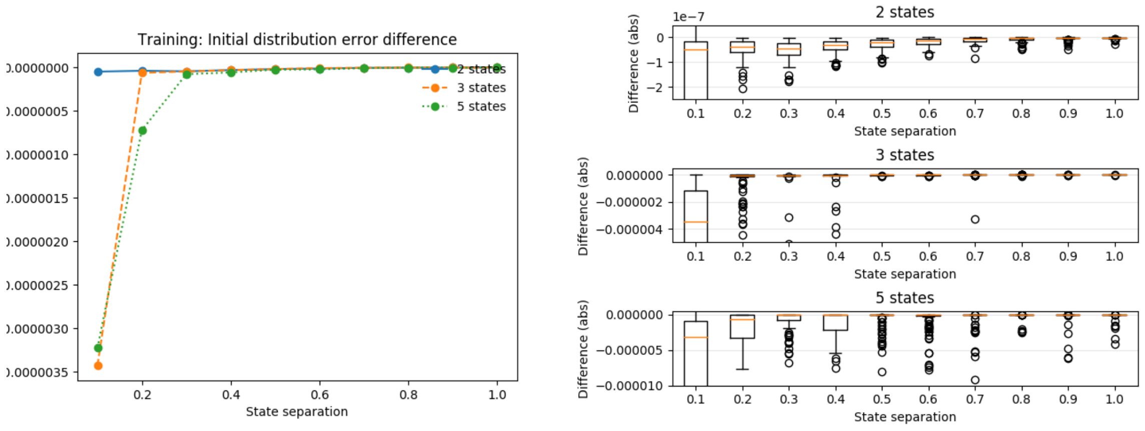


Figure A.7: Relative difference between the average error on the log probability of the starting state in the initial distribution for the compressed and standard algorithms.

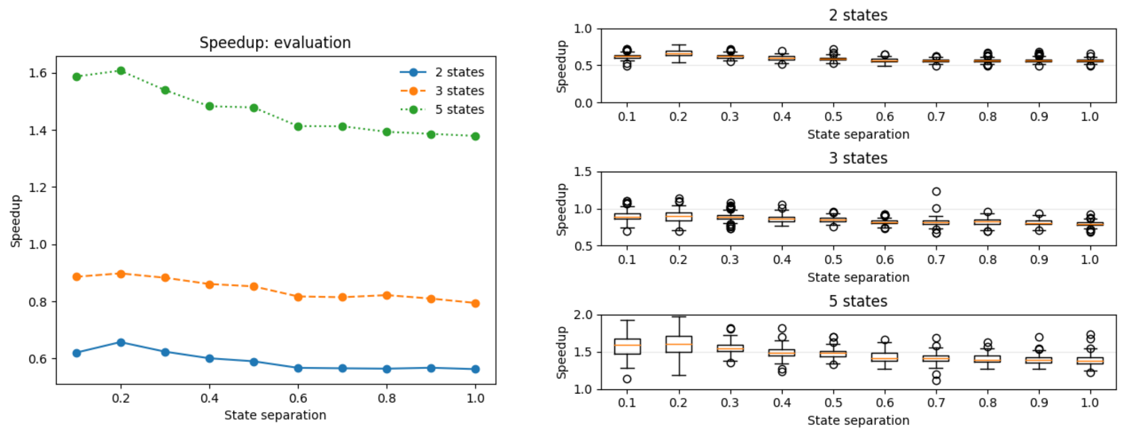


Figure A.8: Speedup on the evaluation problem using the compressed algorithm, including the input data processing time.

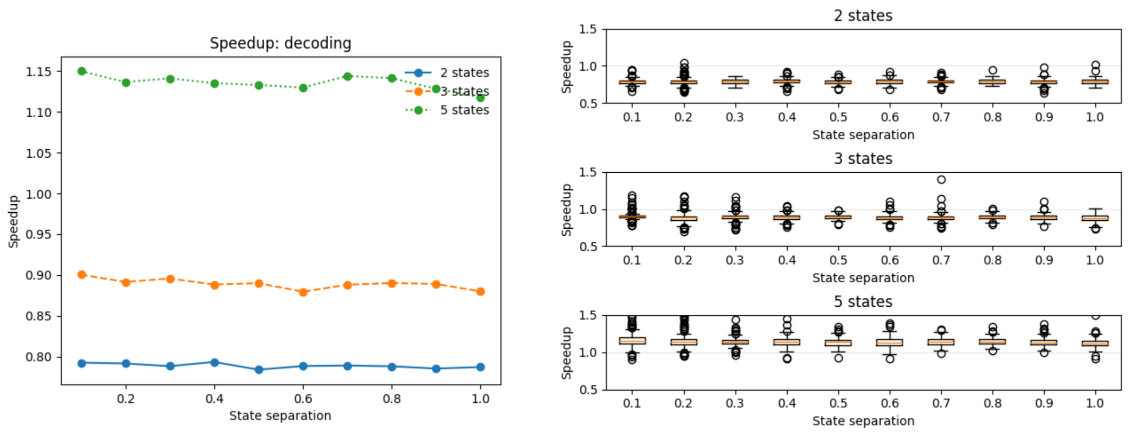


Figure A.9: Speedup on the decoding problem using the compressed algorithm, including the input data processing time.

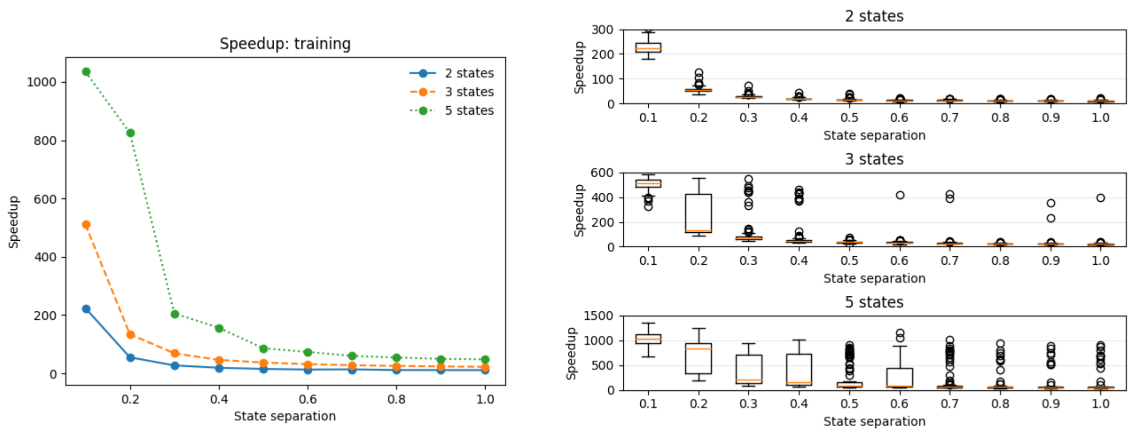


Figure A.10: Speedup on the training problem using the compressed algorithm, including the input data processing time.

A.3 Left-to-right model

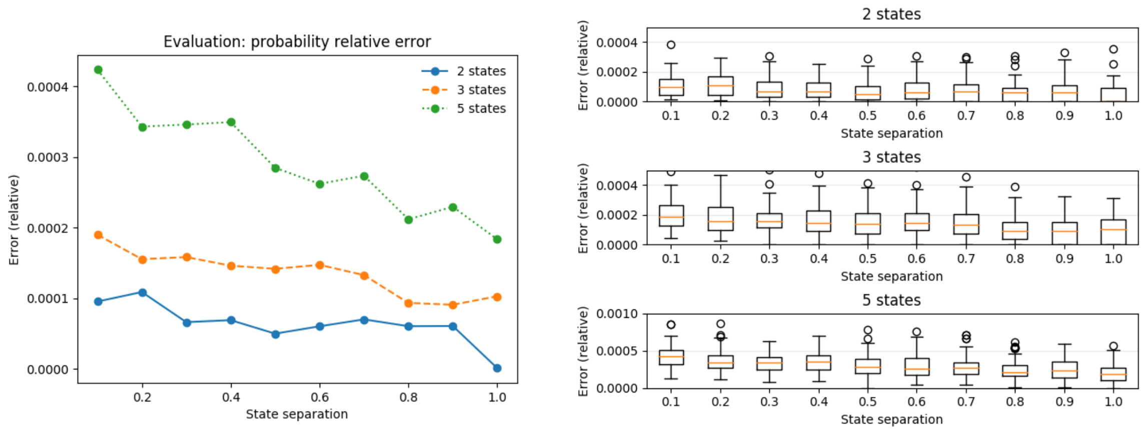


Figure A.11: Relative difference between the $P(O|\lambda)$ log probabilities of the compressed and standard algorithms.

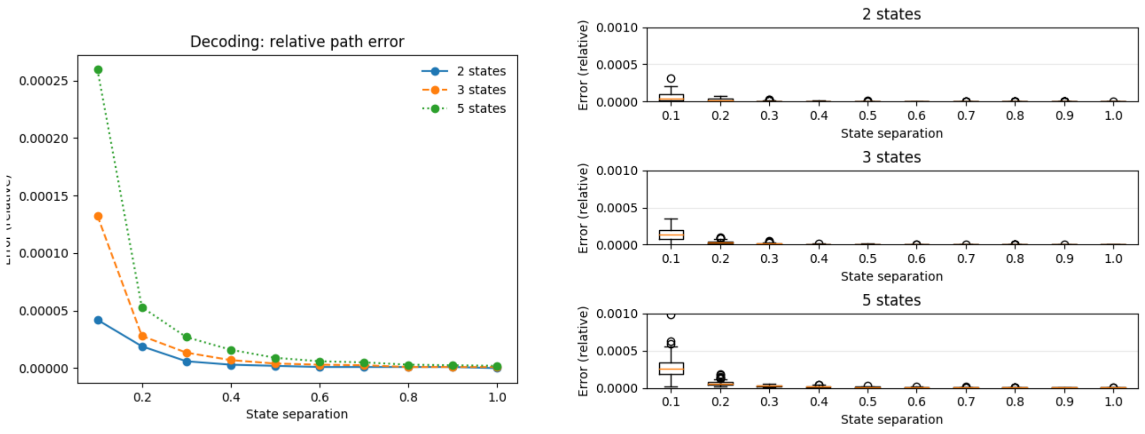


Figure A.12: Relative difference between the fractions of errors in the estimated generating path.

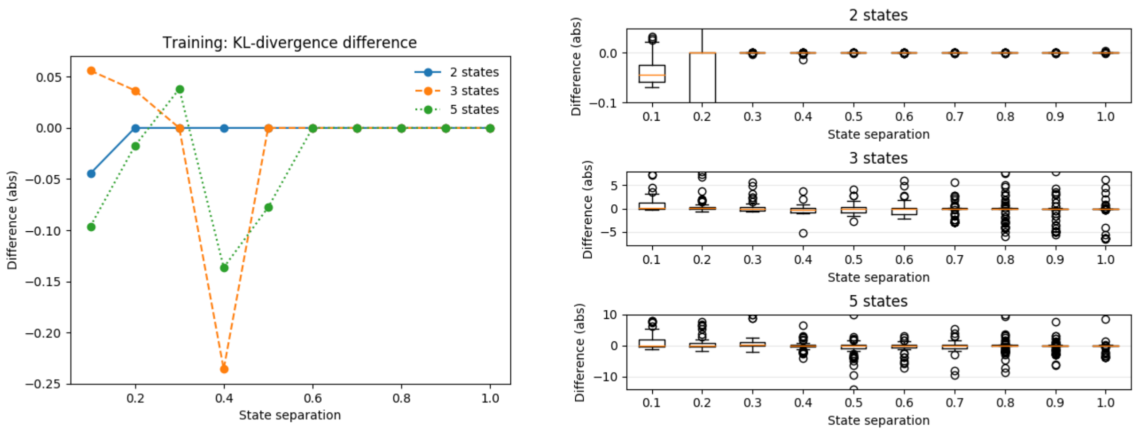


Figure A.13: Difference between the average KL-divergence for the compressed and standard algorithms.

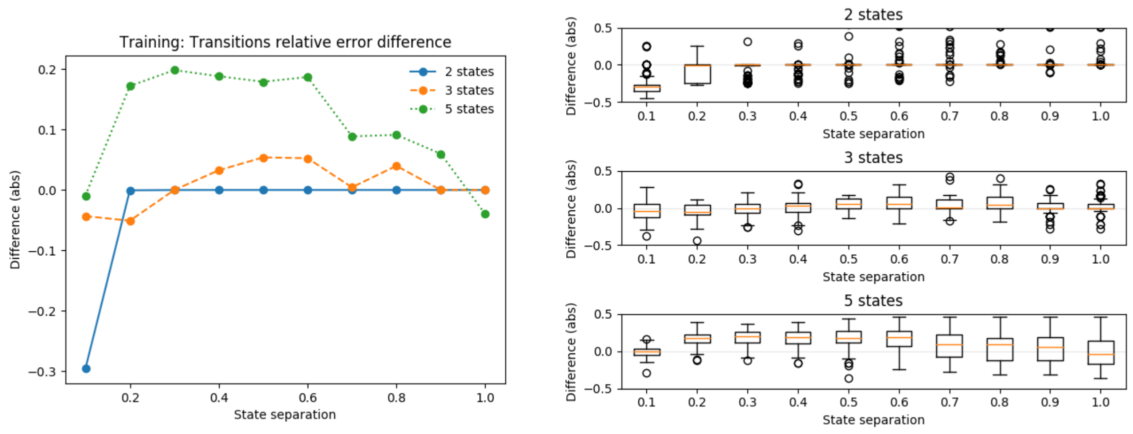


Figure A.14: Relative difference between the average error on the log transition probabilities of the compressed and standard algorithms.

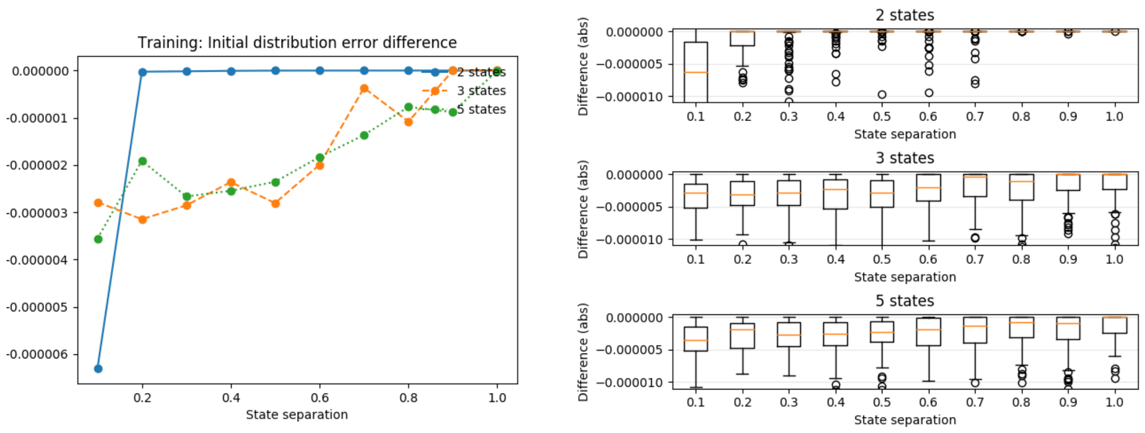


Figure A.15: Relative difference between the average error on the log probability of the starting state in the initial distribution for the compressed and standard algorithms.

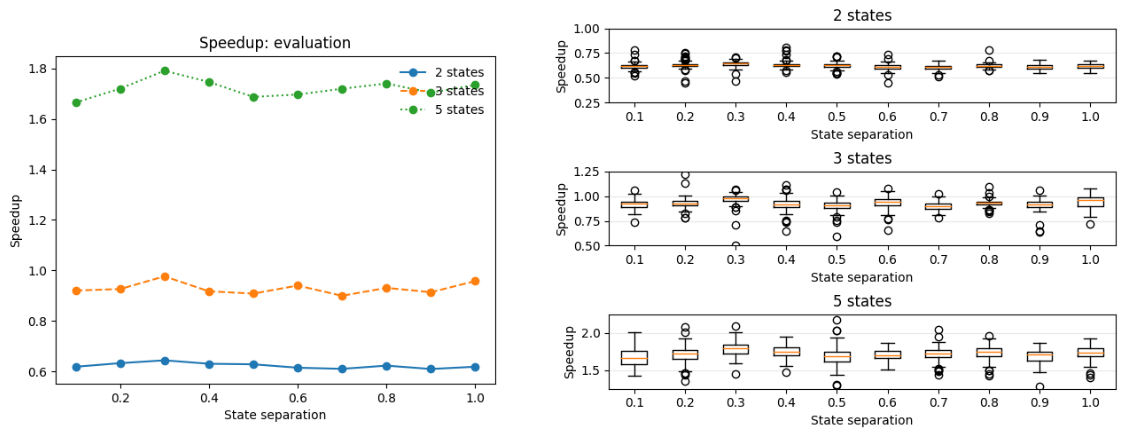


Figure A.16: Speedup on the evaluation problem using the compressed algorithm, including the input data processing time.

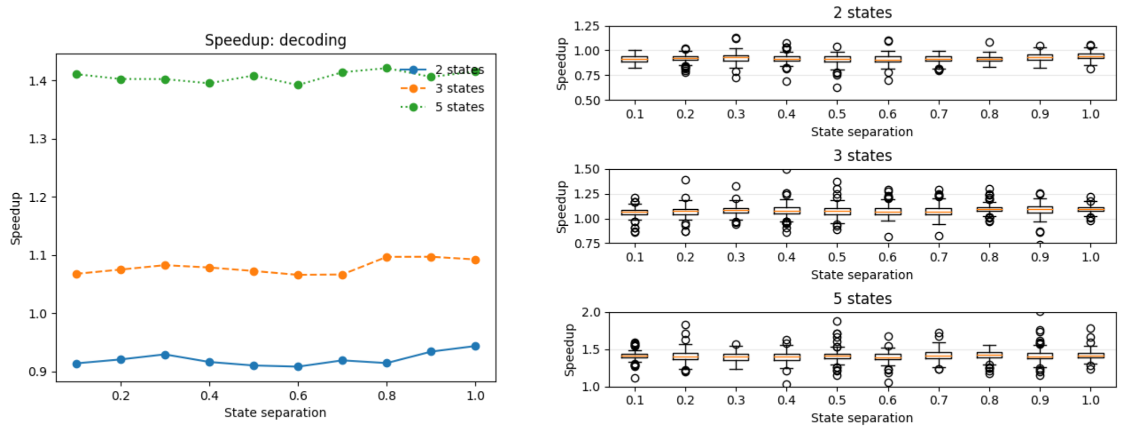


Figure A.17: Speedup on the decoding problem using the compressed algorithm, including the input data processing time.

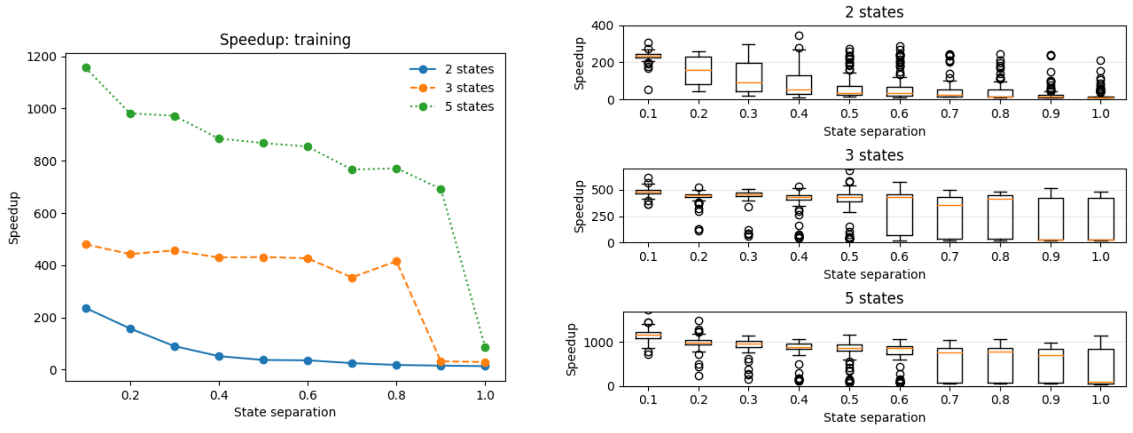


Figure A.18: Speedup on the training problem using the compressed algorithm, including the input data processing time.

B
Appendix 2

B.1 C++

B.1.1 algorithms_compressed.hpp

```
#ifndef WAHMM_ALGORITHMS_COMPRESSED_HPP
#define WAHMM_ALGORITHMS_COMPRESSED_HPP

#include "Compressor.hpp"
#include "commons.hpp"
#include "utilities.hpp"
#include <list>
using std::list;

/**
* Compute the compressed forward matrix given a model and a Compressor holding
* the compressed data. Each element approximates the value at the end of the
* block of the uncompressed matrix.
*
* @param m the model
* @param c the compressor holding the data
*/
wahmm::real_t** forward_matrix_compressed(Model& m, Compressor *c){
    wahmm::real_t **logForward;
    size_t numberOfStates = m.mStates.size();
    logForward = new wahmm::real_t*[numberOfStates]; // forward variables

    c->initForward();
    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        logForward[i] = new wahmm::real_t[c->blocksNumber()];
        // alpha_0(i) = pi_i * E_1(i)
        logForward[i][0] = m.mLogPi[i] + compute_e(m, i, c->blockData());
    }
    // induction
    size_t blockCounter = 1;
    while(c->next()){
        for(size_t j = 0; j < numberOfStates; j++){ // arriving state
            logForward[j][blockCounter] = -infin;
            for(int i = 0; i < numberOfStates; i++){ // starting state
                // alpha_W(j) = sum_{i=0}^N alpha_{w-1}(i)a_{ij} ...
                logForward[j][blockCounter] = sum_logarithms(
                    logForward[j][blockCounter],
                    logForward[i][blockCounter-1] + m.mLogTransitions[i][j]);
            }
            // ... E_w(j)
            logForward[j][blockCounter] += compute_e(m, j, c->blockData());
        }
        blockCounter++;
    }
    c->initForward();

    return logForward;
}


/**
* Compute the compressed backward matrix given a model and a Compressor holding
* the compressed data. Each element approximates the value at the end of the
* block of the uncompressed matrix.
*
* @param m the model
* @param c the compressor holding the data
*/
wahmm::real_t** backward_matrix_compressed(Model& m, Compressor *c){
    wahmm::real_t **logBackward;
    size_t numberOfStates = m.mStates.size();
    logBackward = new wahmm::real_t*[numberOfStates]; // backward variables

    c->initBackward();
    int blockCounter = c->blocksNumber() - 1;
    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        logBackward[i] = new wahmm::real_t[c->blocksNumber()];
        logBackward[i][blockCounter] = 1;
    }
    blockCounter--;
    // induction
    while(blockCounter >= 0){
        for(size_t i = 0; i < numberOfStates; i++){ // arriving state
            logBackward[i][blockCounter] = -infin;
            for(size_t j = 0; j < numberOfStates; j++){ // starting state
                // beta_t(i) = sum_{j=1}^N a_{ij} b_j(O_{t+1}) beta_{t+1}(j)
                logBackward[i][blockCounter] = sum_logarithms(
                    logBackward[i][blockCounter],
                    m.mLogTransitions[i][j] +
                    compute_e(m, j, c->reverseBlockData()) +
                    logBackward[j][blockCounter+1]);
            }
        }
        c->reverseNext();
        blockCounter--;
    }
    c->initBackward();

    return logBackward;
}


/**
```

```
* Solve the evaluation problem through a compressed version of the forward
* algorithm.
*
* @param m the model
* @param c compressor holding the data
* @param verbose if true print result informations
* @param silence suppress all output
* @param tofile save results to file
*/
void evaluation_compressed(Model& m, Compressor *c, bool verbose, bool silence,
    bool tofile){
    wahmm::real_t **logForward;
    wahmm::real_t logEvaluation;
    size_t numberOfStates = m.mStates.size();

    if(!silence)
        std::cout << "[>] +++ Compressed Evaluation Problem +++" << std::endl;

    logForward = forward_matrix_compressed(m, c); //initialization and induction
    // termination
    logEvaluation = -infin;
    for(size_t i = 0; i < numberOfStates; i++){
        logEvaluation = sum_logarithms(logEvaluation,
            logForward[i][c->blocksNumber()-1]);
    }

    // print results
    if(verbose){
        printMatrixSummary(logForward, numberOfStates, c->blocksNumber(),
            "Blocks Forward (log)", false);
    }
    if(!silence)
        std::cout << "[>] log[ P(O|lambda) ]: " << logEvaluation << std::endl;
    if(tofile){
        if(verbose){
            std::cout << "[>] Saving compressed evaluation log probability";
            std::cout << " to file " << PATH_OUT;
            std::cout << "compressed_evaluation_prob ... " << std::flush;
        }
        std::ofstream ofs (PATH_OUT + "compressed_evaluation_prob",
            std::ofstream::out);
        ofs.precision(std::numeric_limits<double>::max_digits10);
        ofs << logEvaluation;
        ofs.close();
        if(verbose)
            std::cout << "done." << std::endl;
    }

    freeMatrix(logForward, numberOfStates);
}


/**
* Solve the decoding problem making use of the compressed version of the
* forward algorithm.
*
```

```
* @param m the model
* @param obs the observation sequence
* @param verbose if true print result informations
* @param silence suppress all output
* @param tofile save results to file
*/
void decoding_compressed(Model &m, Compressor *c, bool verbose, bool silence,
    bool tofile){
    wahmm::real_t **logViterbi, **statesViterbi;
    size_t numberOfStates = m.mStates.size();
    logViterbi = new wahmm::real_t*[numberOfStates];
    statesViterbi = new wahmm::real_t*[numberOfStates];
    std::vector<size_t> blockLengths;

    if(!silence)
        std::cout << "[>] +++ Compressed Decoding Problem +++" << std::endl;

    c->initForward();
    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        logViterbi[i] = new wahmm::real_t[c->blocksNumber()];
        statesViterbi[i] = new wahmm::real_t[c->blocksNumber()];
        // delta_0(i) = pi_i * e^(E_0(i))
        logViterbi[i][0] = m.mLogPi[i] + compute_e(m, i, c->blockData());
        statesViterbi[i][0] = -1; // psi_0(i) = 0
    }
    blockLengths.push_back(c->blockSize());
    // induction
    wahmm::real_t currentMax = -infin;
    wahmm::real_t currentSum = 0;
    int blockCounter = 1;
    int currentState = -1;
    while(c->next()){ // observations
        for(size_t j = 0; j < numberOfStates; j++){ // arriving state
            logViterbi[j][blockCounter] = -infin;
            // delta_t(j) = max_{1<=i<=N} delta_{t-1}(i)a_{ij} ...
            for(size_t i = 0; i < numberOfStates; i++){ // starting state
                currentSum = logViterbi[i][blockCounter-1] +
                    m.mLogTransitions[i][j];
                if(currentSum > currentMax){
                    currentMax = currentSum;
                    currentState = i;
                }
            }
            // ... e^(E_w(j))
            logViterbi[j][blockCounter] = currentMax +
                compute_e(m, j, c->blockData());
            // psi_t(j) = argmax[...]
            statesViterbi[j][blockCounter] = currentState;
            // re-initialize max variables for next loop
            currentMax = -infin;
            currentState = -1;
        }
        blockLengths.push_back(c->blockSize());
        blockCounter++;
    }
```

```
// termination
wahmm::real_t logDecoding;
list<size_t> viterbiPath;
currentMax = -infin; // this will contain the log probability of the path
currentState = -1;
blockCounter--;
for(size_t i = 0; i < numberOfStates; i++){
    if(logViterbi[i][blockCounter] > currentMax){
        currentMax = logViterbi[i][blockCounter];
        currentState = i;
    }
}
viterbiPath.push_front(currentState);
for(; blockCounter > 0; blockCounter--){
    // if currentState == -1, impossible path
    if(currentState >= 0)
        currentState = statesViterbi[currentState][blockCounter];
    else {
        if(!silence)
            std::cerr << "[Warning] Impossible Viterbi path!" << std::endl;
        break;
    }
    viterbiPath.push_front(currentState);
}

// print results
if(verbose){
    printMatrixSummary(logViterbi, numberOfStates, c->blocksNumber(),
        "Block Viterbi (log)", false);
}
if(!silence){
    std::cout << "[>] Most likely path: " << std::endl;
    int i = 0;
    for(auto it = viterbiPath.begin(); it != viterbiPath.end(); it++, i++){
        // only print first 5 and last 5 states
        if(i == 5)
            std::cout << "... ";
        if(i >= 5 && i < viterbiPath.size() - 5)
            continue;
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    std::cout << "[>] log[ P(Q|O,lambda) ]: " << currentMax << std::endl;
}

if(tofile){
    if(verbose){
        std::cout << "[>] Saving compressed Viterbi path to file ";
        std::cout << PATH_OUT << "compressed_decoding_path ...";
        std::cout << std::flush;
    }
    std::ofstream ofsPath (PATH_OUT + "compressed_decoding_path",
        std::ofstream::out);
    size_t lenIndex = 0;
    for(auto it = viterbiPath.begin(); it != viterbiPath.end(); it++){
            for(size_t blen = 0; blen < blockLengths[lenIndex]; blen++){
                ofsPath << *it << " ";
            }
            lenIndex++;
        }
        ofsPath.close();
        if(verbose)
            std::cout << "done." << std::endl;
        if(verbose){
            std::cout << "[>] Saving compressed Viterbi log likelihood";
            std::cout << " to file " << PATH_OUT;
            std::cout << "compressed_decoding_prob ... " << std::flush;
        }
        std::ofstream ofsProb (PATH_OUT + "compressed_decoding_prob",
            std::ofstream::out);
        ofsProb.precision(std::numeric_limits<double>::max_digits10);
        ofsProb << currentMax;
        ofsProb.close();
        if(verbose)
            std::cout << "done." << std::endl;
    }

    c->initForward();

    freeMatrix(logViterbi, numberOfStates);
    freeMatrix(statesViterbi, numberOfStates);
}


/**
* Perform one iteration of a compressed version of the Baum-Welch algorithm.
* Both the forward and backward matrix are computed to achieve this.
* The values for K(n_w, j) could be precomputed; to avoid useless computations,
* whenever a new value of n_w is encountered, K(n_w, j) is computed for every
* state and put into a map for greater efficiency.
*
* Parameters reestimation happens by progressively transforming the numbers
* back into the normal space.
*
* @returns the logEvaluation P(O|lambda) of the previous model
*/
wahmm::real_t compressed_baum_welch_iteration(Model& m, Compressor *c,
    wahmm::real_t minSum, wahmm::real_t **logEpsilon, wahmm::real_t *logPi,
    wahmm::real_t **logGamma, wahmm::real_t *logGammaSum,
    wahmm::real_t *logTrDen, wahmm::real_t *average, wahmm::real_t *variance){

    wahmm::real_t logEvaluation; // P(O|lambda)
    wahmm::real_t **logForward; // forward matrix
    wahmm::real_t **logBackward; // backward matrix
    size_t numberOfStates = m.mStates.size();

    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        for(size_t j = 0; j < numberOfStates; j++)
            logEpsilon[i][j] = -infin;
        logGammaSum[i] = -infin;
```

```
}
logForward = forward_matrix_compressed(m, c);
logEvaluation = -infin;
for(size_t i = 0; i < numberOfStates; i++){
    logEvaluation = sum_logarithms(logEvaluation,
        logForward[i][c->blocksNumber()-1]);
}
logBackward = backward_matrix_compressed(m, c);

c->initForward();
// iterate over blocks
int bc = 0; // block counter
size_t nBlocks = c->blocksNumber();
blockdata currentBd = c->blockData();

wahmm::real_t logEvalGammaSum = -infin;
for(int i = 0; i < numberOfStates; i++){
    logEvalGammaSum = sum_logarithms(logEvalGammaSum,
        logForward[i][0] +
        logBackward[i][0]);
}

while(c->next()){ // "current block" is w+1
    for(int i = 0; i < numberOfStates; i++){
        for(int j = 0; j < numberOfStates; j++){
            logEpsilon[i][j] = sum_logarithms(logEpsilon[i][j],
                logForward[i][bc] +
                m.mLogTransitions[i][j] +
                compute_e(m, j, c->blockData()) +
                logBackward[j][bc+1]);
            if(i == j){
                logEpsilon[i][j] = sum_logarithms(logEpsilon[i][j],
                    log(currentBd.nw - 1) +
                    logForward[i][bc] +
                    logBackward[i][bc]);
            }
        }
        logGamma[i][bc] = logForward[i][bc] + logBackward[i][bc];
        logGammaSum[i] = sum_logarithms(logGammaSum[i],
            log(currentBd.nw) + logGamma[i][bc]);
    }
    currentBd = c->blockData();
    bc++;
}
// last block was not processed
for(int i = 0; i < numberOfStates; i++){
    logEpsilon[i][i] = sum_logarithms(logEpsilon[i][i],
        log(currentBd.nw - 1) +
        logForward[i][bc] +
        logBackward[i][bc]);
    logGamma[i][bc] = logForward[i][bc] + logBackward[i][bc];
    logGammaSum[i] = sum_logarithms(logGammaSum[i],
        log(currentBd.nw) + logGamma[i][bc]);
}

// reestimated parameters
```

```
    for(int i = 0; i < numberOfStates; i++){
        logPi[i] = logGamma[i][0] - logEvalGammaSum;
        for(int j = 0; j < numberOfStates; j++){
            logEpsilon[i][j] -= logGammaSum[i];
        }
        c->initForward();
        average[i] = 0;
        wahmm::real_t v;
        for(int b = 0; b < nBlocks; b++){
            currentBd = c->blockData();
            v = currentBd.s1;
            average[i] += exp(logGamma[i][b]-logGammaSum[i]) * v;
            c->next();
        }
        c->initForward();
        variance[i] = 0;
        for(int b = 0; b < nBlocks; b++){
            currentBd = c->blockData();
            //v can sometimes have some numerical issues caused by HaMMLET code
            v = currentBd.s2 - 2*average[i]*currentBd.s1 +
                currentBd.nw*average[i]*average[i];
            variance[i] += exp(logGamma[i][b]-logGammaSum[i]) * v;
            c->next();
        }
    }

    //update model
    for(size_t i = 0; i < numberOfStates; i++){
        m.mLogPi[i] = logPi[i];
        for(size_t j = 0; j < numberOfStates; j++){
            m.mLogTransitions[i][j] = logEpsilon[i][j];
        }
        m.mStates[i].updateParameters(average[i],
            sqrt(variance[i]));
    }
    //drop KValues matrix
    m.mKValues.clear();

    freeMatrix(logForward, numberOfStates);
    freeMatrix(logBackward, numberOfStates);

    return logEvaluation;
}


/**
* Solve the training problem by performing more iterations of a compressed
* version of the Baum-Welch algorithm.
* The training continues for a certain number of iterations at maximum or until
* the improvement on the evaluation probability falls below a certain threshold.
* Note that one more iteration than necessary is performed.
*/
void training_compressed(Model& m, Compressor *c, wahmm::real_t thresh,
    size_t maxIterations, bool verbose, bool silence, bool tofile){

    wahmm::real_t **logEpsilon; // eps_t(i,j), accumulator over all t
```

```
// logPi is computed at last
wahmm::real_t *logPi = new wahmm::real_t[m.mStates.size()];
wahmm::real_t **logGamma;
wahmm::real_t *logTrDen = new wahmm::real_t[m.mStates.size()];
wahmm::real_t *logGammaSum = new wahmm::real_t[m.mStates.size()];
// both average and variance arrays have one element for each state
wahmm::real_t *average = new wahmm::real_t[m.mStates.size()];
wahmm::real_t *variance = new wahmm::real_t[m.mStates.size()];

logEpsilon = new wahmm::real_t*[m.mStates.size()];
logGamma = new wahmm::real_t*[m.mStates.size()];
for(size_t i = 0; i < m.mStates.size(); i++){
    logEpsilon[i] = new wahmm::real_t[m.mStates.size()];
    logGamma[i] = new wahmm::real_t[c->blocksNumber()];
}

c->buildReverse();
c->initForward();
wahmm::real_t minSum = 0;
do {
    if(c->blockData().s1 < minSum)
        minSum = c->blockData().s1;
} while(c->next());
c->initForward();
minSum -= 1; // to avoid crash when 0 is saves as -0.0000000001
wahmm::real_t evaluation=-infin, newEvaluation=-infin;
wahmm::real_t logImprovement = thresh + 1;
size_t iter;

if(!silence)
std::cout << "[>] +++ Compressed Training Problem +++" << std::endl;

for(iter = 0; iter < maxIterations && logImprovement > thresh; iter++){
    newEvaluation = compressed_baum_welch_iteration(m, c, minSum,
        logEpsilon, logPi, logGamma,
        logGammaSum, logTrDen, average, variance);
    logImprovement = newEvaluation - evaluation;
    evaluation = newEvaluation;
    if(verbose){
        std::cout << "[>] Iteration: " << iter << std::endl;
        std::cout << "[>] Evaluation improvement (log): ";
        std::cout << logImprovement << std::endl;
        std::cout << "[>] New P(O | lambda): ";
        std::cout << newEvaluation << std::endl;
        m.printModel();
    }
}
if(!silence){
    std::cout << "[>] Number of iterations: " << iter << std::endl;
    m.printModel();
}

if(tofile){
    m.sortModel();
    if(verbose){
        std::cout << "[>] Saving trained model to file " << PATH_OUT;
```

```
            std::cout << "training_model ... " << std::flush;
        }
        std::ofstream modelFileOutput(PATH_OUT + "compressed_training_model");
        if(modelFileOutput.is_open()){
            modelFileOutput << m;
        }
        modelFileOutput.close();
        if(verbose)
            std::cout << "done." << std::endl;
    }

    // free variables
    freeMatrix(logEpsilon, m.mStates.size());
    freeMatrix(logGamma, m.mStates.size());
    delete[] logPi;
    delete[] logGammaSum;
    delete[] average;
    delete[] variance;
}


#endif
```

B.1.2 algorithms.hpp

```
#ifndef WAHMM_ALGORITHMS_HPP
#define WAHMM_ALGORITHMS_HPP
#include "commons.hpp"
#include "utilities.hpp"
#include <list>
using std::list;

/**
* Compute the forward matrix given a model and an observations sequence.
*
* @param m the model
* @param obs the observation sequence
*/
wahmm::real_t** forward_matrix(Model& m, std::vector<wahmm::real_t>& obs){
    wahmm::real_t **logForward;
    size_t numberOfStates = m.mStates.size();
    logForward = new wahmm::real_t*[numberOfStates]; // forward variables

    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        logForward[i] = new wahmm::real_t[obs.size()];
        // alpha_0(i) = pi_i * b_i(O_1)
        logForward[i][0] = m.mLogPi[i] +
            m.mStates[i].logPdf(obs[0]);
    }
    // induction
    for(size_t t = 1; t < obs.size(); t++){ // observations
        for(size_t j = 0; j < numberOfStates; j++){ // arriving state
            logForward[j][t] = -infin;
            for(int i = 0; i < numberOfStates; i++){ // starting state
                // alpha_t+1(j) = sum_{i=0}^N alpha_t(i)a_{ij} ...
                logForward[j][t] = sum_logarithms(logForward[j][t],
                    logForward[i][t-1] + m.mLogTransitions[i][j]);
            }
            // ... b_{j}(O_{t+1})
            logForward[j][t] += m.mStates[j].logPdf(obs[t]);
        }
    }

    return logForward;
}

/**
* Compute the backward matrix given a model and an observations sequence.
*
* @param m the model
* @param obs the observation sequence
*/
wahmm::real_t** backward_matrix(Model& m, std::vector<wahmm::real_t>& obs){
    wahmm::real_t **logBackward;
    size_t numberOfStates = m.mStates.size();
    logBackward = new wahmm::real_t*[numberOfStates]; // backward variables

    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        logBackward[i] = new wahmm::real_t[obs.size()];
        // beta_T(i) = 1
        logBackward[i][obs.size()-1] = 0;
    }
    // induction
    for(int t = obs.size()-2; t >= 0; t--){ // observations
        for(size_t i = 0; i < numberOfStates; i++){ // arriving state
            logBackward[i][t] = -infin;
            for(size_t j = 0; j < numberOfStates; j++){ // starting state
                // beta_t(i) = sum_{j=1}^N a_{ij} b_j(O_{t+1}) beta_{t+1}(j)
                logBackward[i][t] = sum_logarithms(logBackward[i][t],
                    m.mLogTransitions[i][j] +
                    m.mStates[j].logPdf(obs[t+1]) +
                    logBackward[j][t+1]);
            }
        }
    }

    return logBackward;
}

/**
* Solve the evaluation problem through the forward algorithm.
*
* @param m the model
* @param obs the observation sequence
* @param verbose if true print result informations
* @param silence suppress all output
* @param tofile save results to file
*/
```

```
void evaluation_problem(Model& m, std::vector<wahmm::real_t>& obs, bool verbose,
    bool silence, bool tofile){
    wahmm::real_t **logForward;
    wahmm::real_t logEvaluation;
    size_t numberOfStates = m.mStates.size();

    if(!silence)
        std::cout << "[>] +++ Evaluation Problem +++" << std::endl;

    logForward = forward_matrix(m, obs); // initialization and induction
    // termination
    logEvaluation = -infin;
    for(size_t i = 0; i < numberOfStates; i++){
        logEvaluation = sum_logarithms(logEvaluation,
            logForward[i][obs.size()-1]);
    }

    // print results
    if(verbose){
        printMatrixSummary(logForward, numberOfStates, obs.size(),
            "Forward (log)", false);
    }
    if(!silence)
        std::cout << "[>] log[ P(O|lambda) ]: " << logEvaluation << std::endl;
    if(tofile){
        if(verbose){
            std::cout << "[>] Saving evaluation log probability to file ";
            std::cout << PATH_OUT << "evaluation_prob ... " << std::flush;
        }
        std::ofstream ofs (PATH_OUT + "evaluation_prob", std::ofstream::out);
        ofs.precision(std::numeric_limits<double>::max_digits10);
        ofs << logEvaluation;
        ofs.close();
        if(verbose)
            std::cout << "done." << std::endl;
    }

    freeMatrix(logForward, numberOfStates);
}


/**
* Solve the decoding problem making use of the forward algorithm.
*
* @param m the model
* @param obs the observation sequence
* @param verbose if true print result informations
* @param silence suppress all output
* @param tofile save results to file
*/
void decoding_problem(Model &m, std::vector<wahmm::real_t>& obs, bool verbose,
    bool silence, bool tofile){
    wahmm::real_t **logViterbi, **statesViterbi;
    size_t numberOfStates = m.mStates.size();
    logViterbi = new wahmm::real_t*[numberOfStates];
    statesViterbi = new wahmm::real_t*[numberOfStates];
```

```
if(!silence)
    std::cout << "[>] +++ Decoding Problem +++" << std::endl;

// initialization
for(size_t i = 0; i < numberOfStates; i++){
    logViterbi[i] = new wahmm::real_t[obs.size()];
    statesViterbi[i] = new wahmm::real_t[obs.size()];
    // delta_0(i) = pi_i * b_i(O_1)
    logViterbi[i][0] = m.mLogPi[i] + m.mStates[i].logPdf(obs[0]);
    statesViterbi[i][0] = -1; // psi_0(i) = 0
}
// induction
wahmm::real_t currentMax = -infin;
wahmm::real_t currentSum = 0;
int currentState = -1;
for(size_t t = 1; t < obs.size(); t++){ // observations
    for(size_t j = 0; j < numberOfStates; j++){ // arriving state
        logViterbi[j][t] = -infin;
        // delta_t(j) = max_{1<=i<=N} delta_{t-1}(i)a_{ij} ...
        for(size_t i = 0; i < numberOfStates; i++){ // starting state
            currentSum = logViterbi[i][t-1] + m.mLogTransitions[i][j];
            if(currentSum > currentMax){
                currentMax = currentSum;
                currentState = i;
            }
        }
        // ... b_{j}(O_{t})
        logViterbi[j][t] = currentMax + m.mStates[j].logPdf(obs[t]);
        // psi_t(j) = argmax[...]
        statesViterbi[j][t] = currentState;
        // re-initialize max variables for next loop
        currentMax = -infin;
        currentState = -1;
    }
}

// termination
wahmm::real_t logDecoding;
list<size_t> viterbiPath;
currentMax = -infin; // this will contain the log probability of the path
currentState = -1;
for(size_t i = 0; i < numberOfStates; i++){
    if(logViterbi[i][obs.size()-1] > currentMax){
        currentMax = logViterbi[i][obs.size()-1];
        currentState = i;
    }
}
viterbiPath.push_front(currentState);
for(int t = obs.size()-2; t >= 0; t--){
    // if currentState == -1, impossible path
    if(currentState >= 0)
        currentState = statesViterbi[currentState][t+1];
    else {
        if(!silence)
            std::cerr << "[Warning] Impossible Viterbi path!" << std::endl;
            break;
        }
        viterbiPath.push_front(currentState);
    }

    // print results
    if(verbose){
        printMatrixSummary(logViterbi, numberOfStates, obs.size(),
            "Viterbi (log)", false);
    }
    if(!silence){
        std::cout << "[>] Most likely path: " << std::endl;
        int i = 0;
        for(auto it = viterbiPath.begin(); it != viterbiPath.end(); it++, i++){
            // only print first 5 and last 5 states
            if(i == 5)
                std::cout << "... ";
            if(i >= 5 && i < viterbiPath.size() - 5)
                continue;
            std::cout << *it << " ";
        }
        std::cout << std::endl;
        std::cout << "[>] log[ P(Q|O,lambda) ]: " << currentMax << std::endl;
    }

    if(tofile){
        if(verbose){
            std::cout << "[>] Saving Viterbi path to file " << PATH_OUT;
            std::cout << "decoding_path ... " << std::flush;
        }
        std::ofstream ofsPath (PATH_OUT + "decoding_path", std::ofstream::out);
        for(auto it = viterbiPath.begin(); it != viterbiPath.end(); it++)
            ofsPath << *it << " ";
        ofsPath.close();
        if(verbose)
            std::cout << "done." << std::endl;
        if(verbose){
            std::cout << "[>] Saving Viterbi log likelihood to file ";
            std::cout << PATH_OUT << "decoding_prob ... " << std::flush;
         }
        std::ofstream ofsProb (PATH_OUT + "decoding_prob", std::ofstream::out);
        ofsProb.precision(std::numeric_limits<double>::max_digits10);
        ofsProb << currentMax;
        ofsProb.close();
        if(verbose)
            std::cout << "done." << std::endl;
    }

    freeMatrix(logViterbi, numberOfStates);
    freeMatrix(statesViterbi, numberOfStates);
}


/**
* Perform one iteration of the Baum-Welch algorithm.
* The forward matrix is calculated entirely; the other things are not to save
* space; the backward variable is computed only for the current time; for the
* reestimation parameters, accumulate sufficient statistics at each time step.
* For example, the current gamma_t is accumulated into logGamma and also used
* to calculate the values for the other accumulators logAverage and logVariance.
* logEpsilon will contain, at the end, the new transition matrix.
* The P(O|lambda) cancels out in everything except for the initial distribution,
* so it's simplified and not included at all.
* For the backward variable, two arrays are needed, for beta_t and beta_{t+1}.
* For this reason, use two arrays and swap them at each time step.
*
* Also, the reestimation of the average requires calculating log(obs[t]);
* to avoid negative observations, translate the sequence (only when performing
* that computation) so the minimum obsevation is zero.
* Actually, translate so that the sequence ranges from [1, inf) because
* zero could be represented as -0.0000000000001 and it would make the log
* function crash.viterbi_likelihood

* @returns the logEvaluation P(O|lambda) of the previous model
*/
wahmm::real_t baum_welch_iteration(Model& m, std::vector<wahmm::real_t>& obs,
    wahmm::real_t minObs, wahmm::real_t **logEpsilon,
    wahmm::real_t *logBackward, wahmm::real_t *prevLogBackward,
    wahmm::real_t *logPi, wahmm::real_t **logGamma, wahmm::real_t *logGammaSum,
    wahmm::real_t *logAverage, wahmm::real_t *logVariance){

    wahmm::real_t logEvaluation; // P(O|lambda)
    wahmm::real_t **logForward; // forward matrix
    wahmm::real_t *tmp; // for swapping of the backward arrays
    size_t numberOfStates = m.mStates.size();

    // initialization
    for(size_t i = 0; i < numberOfStates; i++){
        for(size_t j = 0; j < numberOfStates; j++)
            logEpsilon[i][j] = -infin;
        logBackward[i] = 0;
        prevLogBackward[i] = 0;
        logGammaSum[i] = -infin;
        logAverage[i] = -infin;
        logVariance[i] = -infin;
    }
    logForward = forward_matrix(m, obs);
    logEvaluation = -infin;
    for(size_t i = 0; i < numberOfStates; i++){
        logEvaluation = sum_logarithms(logEvaluation,
            logForward[i][obs.size()-1]);
    }

    // start from T-1
    for(int t = obs.size()-2; t >= 0; t--){
        // calculate backward variable for the next iteration
        for(int i = 0; i < numberOfStates; i++){
            logBackward[i] = -infin;
            for(size_t j = 0; j < numberOfStates; j++){
                logBackward[i] = sum_logarithms(logBackward[i],
                    m.mLogTransitions[i][j] +
                    m.mStates[j].logPdf(obs[t+1]) +
                    prevLogBackward[j]);
            }
        }
        // calculate epsilon and increase estimates for observation t
        for(size_t i = 0; i < numberOfStates; i++){
            for(size_t j = 0; j < numberOfStates; j++){
                logEpsilon[i][j] = sum_logarithms(logEpsilon[i][j],
                    logForward[i][t] +
                    m.mLogTransitions[i][j] +
                    m.mStates[j].logPdf(obs[t+1]) +
                    prevLogBackward[j]); // -logEvaluation (it simplifies)
                    //prevLogBackward is beta_{t+1}
            }
            // calculate gamma_t(i) for the current t
            // logBackward is beta_t
            logGamma[i][t] = logForward[i][t] + logBackward[i];
            logGammaSum[i] = sum_logarithms(logGammaSum[i], logGamma[i][t]);
            // update the accumulators
            logAverage[i] = sum_logarithms(logAverage[i],
                logGamma[i][t] + log(obs[t]-minObs));
        }
        // to avoid copying the array
        tmp = logBackward;
        logBackward = prevLogBackward;
        prevLogBackward = tmp;
    }
    // compute final reestimated parameters
    wahmm::real_t currentNewAverage;
    for(size_t i = 0; i < numberOfStates; i++){
        for(size_t j = 0; j < numberOfStates; j++){
            logEpsilon[i][j] -= logGammaSum[i]; // a_{ij}
        }
        logPi[i] = logGamma[i][0] - logEvaluation; // pi_i = gamma_1(i)
        logAverage[i] -= logGammaSum[i];
        currentNewAverage = exp(logAverage[i])+minObs;
        for(size_t t = 0; t < obs.size()-1; t++){
            logVariance[i] = sum_logarithms(logVariance[i],
                logGamma[i][t] + 2*log(abs(obs[t] - currentNewAverage)));
        }
        logVariance[i] -= logGammaSum[i];
    }

    // update parameters in the model
    for(size_t i = 0; i < numberOfStates; i++){
        m.mLogPi[i] = logPi[i];
        for(size_t j = 0; j < numberOfStates; j++){
            m.mLogTransitions[i][j] = logEpsilon[i][j];
        }
        m.mStates[i].updateParameters(exp(logAverage[i])+minObs,
            sqrt(exp(logVariance[i])));
    }

    freeMatrix(logForward, numberOfStates);

    return logEvaluation;
}
```

```
/**
* Solve the training problem by performing more iterations of the Baum-Welch
* algorithm. The training continues for a certain number of iterations at
* maximum or until the improvement on the evaluation probability falls below
* a certain threshold.
* Note that one more iteration than necessary is performed.
*/
void training_problem(Model& m, std::vector<wahmm::real_t>& obs,
    wahmm::real_t thresh, size_t maxIterations, bool verbose, bool silence,
    bool tofile){

    wahmm::real_t **logEpsilon; // eps_t(i,j), accumulator over all t
    // logBackward only for current t
    wahmm::real_t *logBackward = new wahmm::real_t[m.mStates.size()];
    wahmm::real_t *prevLogBackward = new wahmm::real_t[m.mStates.size()];
    // logPi computed at last
    wahmm::real_t *logPi = new wahmm::real_t[m.mStates.size()];
    wahmm::real_t **logGamma;
    wahmm::real_t *logGammaSum = new wahmm::real_t[m.mStates.size()];
    wahmm::real_t *logAverage = new wahmm::real_t[m.mStates.size()];
    wahmm::real_t *logVariance = new wahmm::real_t[m.mStates.size()];

    logEpsilon = new wahmm::real_t*[m.mStates.size()];
    logGamma = new wahmm::real_t*[m.mStates.size()];
    for(size_t i = 0; i < m.mStates.size(); i++){
        logEpsilon[i] = new wahmm::real_t[m.mStates.size()];
        logGamma[i] = new wahmm::real_t[obs.size()-1];
    }


    wahmm::real_t minObs = 0;
    for(auto it = obs.begin(); it != obs.end(); it++){
        if(*it < minObs)
            minObs = *it;
    }
    minObs -= 1; // to avoid crash when 0 is saves as -0.0000000001
    wahmm::real_t evaluation=-infin, newEvaluation=-infin;
    wahmm::real_t logImprovement = thresh + 1;
    size_t iter;
    if(!silence)
        std::cout << "[>] +++ Training problem +++" << std::endl;
    for(iter = 0; iter < maxIterations && logImprovement > thresh; iter++){
        newEvaluation = baum_welch_iteration(m, obs, minObs,
            logEpsilon, logBackward, prevLogBackward, logPi, logGamma,
            logGammaSum, logAverage, logVariance);
        // newEvaluation = training_problem_scaled(m, obs, minObs,
        //      logEpsilon, logPi, logGamma,
        //      logGammaSum, logAverage, logVariance);
        logImprovement = newEvaluation - evaluation;
        evaluation = newEvaluation;
        if(verbose){
            std::cout << "[>] Iteration: " << iter << std::endl;
            std::cout << "[>] Evaluation improvement (log): ";
            std::cout << logImprovement << std::endl;
            std::cout << "[>] New P(O | lambda): " << newEvaluation;
```

```
                std::cout << std::endl;
                m.printModel();
            }
        }
        if(!silence){
            std::cout << "[>] Number of iterations: " << iter << std::endl;
            m.printModel();
        }

        if(tofile){
            m.sortModel();
            if(verbose)
                std::cout << "[>] Saving trained model to file " << PATH_OUT;
                std::cout << "training_model ... " << std::flush;
            std::ofstream modelFileOutput(PATH_OUT + "training_model");
            if(modelFileOutput.is_open()){
                modelFileOutput << m;
            }
            modelFileOutput.close();
            if(verbose)
                std::cout << "done." << std::endl;
        }

        // free variables
        freeMatrix(logEpsilon, m.mStates.size());
        freeMatrix(logGamma, m.mStates.size());
        delete[] logBackward;
        delete[] prevLogBackward;
        delete[] logPi;
        delete[] logGammaSum;
        delete[] logAverage;
        delete[] logVariance;
}

#endif
```

B.1.3 commons.hpp

```
#ifndef WAHMM_COMMONS_HPP
#define WAHMM_COMMONS_HPP

#define PATH_OUT std::string("results/")

#include <cstdint>

#include <cstddef>
using std::size_t;

#include <vector>
using std::vector;

#include <map>

#include <string>
```

```
using std::string;
using std::to_string;

#include <iostream>
using std::istream;
using std::ostream;
using std::endl;
using std::cin;
using std::cout;

#include <fstream>
using std::ifstream;
using std::ofstream;

#include <cmath>
using std::pow;
using std::exp;         // e^x
using std::log;         // natural log
using std::sqrt;
using std::abs;

#include <math.h>

using std::isfinite;

#include <algorithm>
using std::min;
using std::max;

namespace wahmm {
    typedef double real_t;
}
const wahmm::real_t infin = std::numeric_limits<wahmm::real_t>::infinity();

struct blockdata_t {
    size_t nw;
    wahmm::real_t s1;
    wahmm::real_t s2;
};
typedef blockdata_t blockdata;

#endif
```

B.1.4 Compressor.hpp

```
#ifndef WAHMM_COMPRESSOR_HPP
#define WAHMM_COMPRESSOR_HPP

#include "includes.hpp"
#include "Tags.hpp"
#include "HMM.hpp"
#include "Blocks.hpp"
#include "wavelet.hpp"
#include "Statistics.hpp"
```

```
#include "utils.hpp"
#include "commons.hpp"

template<typename T>
void MaxletTransform(FILE* fin,
    bool binary,
    vector<real_t>& coeffs,
    vector< SufficientStatistics<T> >& suffstats,
    const size_t nrDim = 1,
    const size_t reserveT = 0
);

/**
* Class constituting the interface between WaHMM and HaMMLET. It serves the
* purpose of reading the observations and compressing them into blocks,
* also providing an interace to navigate easily the data structures that
* HaMMLET offers.
*/
class Compressor {
    /** Input observations values */
    vector<real_t> mInputValues;
    /** Sum of the observations in a block */
    vector<SufficientStatistics<Normal>> mStats;
    /** Threshold used to form blocks in the breakpoint array */
    real_t mThreshold;
    /** Integral array, @see HaMMLET documentation */
    Statistics<IntegralArray, Normal> *mIntegralArray;
    /** Breakpoint array, @see HaMMLET documentation */
    Blocks<BreakpointArray> *mWaveletBlocks;
    size_t mBlocksNumber;
    /** Stack to hold the reverse order of the blocks, built on command */
    std::list<blockdata> mReverseList;
    /** Iterator to navigate the reverse list */
    std::list<blockdata>::iterator listIt = mReverseList.begin();
public:
    Compressor(const Compressor& that) = delete;
    /**
    * Construct a Compressor object. Data is read from an input file, creating
    * a breakpoint array and defining a threshold to use; blocks are then
    * defined and the integral array is created. There is initialization to
    * correctly read the first block.
    *
    * @param filename the name of the input file to read the data
    */
    Compressor(std::string& filename, bool binary);
    ~Compressor();
    /** Move the "current block pointer" back to the first block. */
    void initForward();
    /**
    * Move the "current block pointer" one step forward.
    * @return false if the "current block" is the last one
    */
    bool next();
    /** Return the start index of the "current block". */
    size_t start();
    /** Return the end index of the "current block". */
    size_t end();
    /** Return the size of the "current block". */
    size_t blockSize();
    /** Return the sum of the observations in the "current block". */
    wahmm::real_t blockSum();
    /** Return the squared sum of the observations in the "current block". */
    wahmm::real_t blockSumSq();
    /** Return the data associated with the current block. */
    blockdata blockData();
    /** Initialize the iterator for the reverse list. */
    void initBackward();
    /**
    * Advance the reverse list iterator.
    * @return false if the iterator reaches the end of the list
    */
    bool reverseNext();
    /** Get the block size from the reverse list iterator. */
    size_t reverseSize();
    /** Return the sum of the observations using the reverse block iterator. */
    wahmm::real_t reverseSum();
    /** Return the squared sum of observations in the reverse iterator. */
    wahmm::real_t reverseSumSq();
    /** Return the data associated with the block in reverse iterator. */
    blockdata reverseBlockData();
    /** Return the number of blocks. */
    size_t blocksNumber();
    /** Return the number of observations. */
    size_t observationsNumber();
    /** Build a list with the blocks in reverse order */
    void buildReverse();
    /** Print start and end indexes of the "current block" alongside with its
    * size and the sum of the observation values in it; the format used is
    * [start,end) size - "Sum:" sum
    */
    void printBlockInfo();
    /**
    * Print block information for all blocks, preceded by the thresold value
    * used to form them.
    */
    void printAllBlocks();
};

Compressor::Compressor(std::string& f, bool binary){
    try {
        const size_t nrDataDim = 1; // number of dimensions
        FILE* fin;
        if(!binary)
            fin = fopen(f.c_str(), "r");
        else
            fin = fopen(f.c_str(), "rb");
        // Open the file and populate mInputValues
        if(fin != NULL) {
            MaxletTransform(fin, binary, mInputValues, mStats, nrDataDim,
                mInputValues.size() + 2);
        } else {
          throw runtime_error( "Cannot read from input file " + f + "!" );
        }
        fclose(fin);

        // compute an estimate of the noise variance from the finest
        // detail coefficients
                double stdEstimate = 0;
        double estimateAccum = 0;
                for (size_t i = 1; i < mInputValues.size(); i += 2){
                        stdEstimate += mInputValues[i];
            if(stdEstimate > mInputValues.size()){
                estimateAccum += stdEstimate/(mInputValues.size()/2);
                stdEstimate = 0;
            }
                }
        estimateAccum += stdEstimate/(mInputValues.size()/2);
        stdEstimate = estimateAccum;

        HaarBreakpointWeights(mInputValues);
        mIntegralArray = new Statistics<IntegralArray,Normal>(mStats,nrDataDim);
        mWaveletBlocks = new Blocks<BreakpointArray>(mInputValues);

                stdEstimate /= 0.797884560802865355879892119868763736951717262329869315;
        mThreshold = sqrt(2*log((real_t)mWaveletBlocks->size())*stdEstimate);

                mWaveletBlocks->createBlocks(mThreshold);
                mWaveletBlocks->initForward();
                mWaveletBlocks->next();
        mBlocksNumber = 0;
        do {
            mBlocksNumber++;
        } while(mWaveletBlocks->next());
        mWaveletBlocks->initForward();
        mWaveletBlocks->next();
    }
    catch(exception& e) {
        std::cout << std::flush;
                cerr << endl << flush << "[CompressorError] ";
        cerr << e.what()  << endl << flush;
        throw e;
    }
}


Compressor::~Compressor(){
    delete mIntegralArray;
    delete mWaveletBlocks;
}

void Compressor::initForward(){
    mWaveletBlocks->initForward();
    mWaveletBlocks->next();
}

bool Compressor::next(){
    return mWaveletBlocks->next();
}
```

```
size_t Compressor::start(){
    mWaveletBlocks->start();
}

size_t Compressor::end(){
    mWaveletBlocks->end();
}

size_t Compressor::blockSize(){
    mWaveletBlocks->blockSize();
}

wahmm::real_t Compressor::blockSum(){
    mIntegralArray->setStats(*mWaveletBlocks);
    return mIntegralArray->suffStat(0).sum(); // 0 is the dimension index
}

wahmm::real_t Compressor::blockSumSq(){
    mIntegralArray->setStats(*mWaveletBlocks);
    return mIntegralArray->suffStat(0).sumSq(); // 0 is the dimension index
}

blockdata Compressor::blockData(){
    blockdata bd = {blockSize(), blockSum(), blockSumSq()};
    return bd;
}

void Compressor::initBackward(){
    listIt = mReverseList.begin();
}

bool Compressor::reverseNext(){
    if(listIt == mReverseList.end())
        return false;
    listIt++;
    return true;
}

size_t Compressor::reverseSize(){
    return (*listIt).nw;
}

wahmm::real_t Compressor::reverseSum(){
    return (*listIt).s1;
}

wahmm::real_t Compressor::reverseSumSq(){
    return (*listIt).s2;
}

blockdata Compressor::reverseBlockData(){
    blockdata bd = {(*listIt).nw, (*listIt).s1, (*listIt).s2};
    return bd;
}
```

```
size_t Compressor::blocksNumber(){
    return mBlocksNumber;
}

size_t Compressor::observationsNumber(){
    return mInputValues.size();
}

void Compressor::buildReverse(){
    mWaveletBlocks->initForward();
    blockdata bd;
    while(mWaveletBlocks->next()){
        bd.nw = blockSize();
        bd.s1 = blockSum();
        bd.s2 = blockSumSq();
        mReverseList.push_front(bd);
    }
    mWaveletBlocks->initForward();
}

void Compressor::printBlockInfo(){
    mWaveletBlocks->printBlock();
    cout << "- Sum: " << blockSum();
    cout << "- SumSq: " << blockSumSq() << endl;
}

void Compressor::printAllBlocks(){
    do {
        printBlockInfo();
        mWaveletBlocks->next();
    } while (mWaveletBlocks->end() < mWaveletBlocks->size());
    printBlockInfo();
    cout << "Threshold used: " << mThreshold << endl;
    initForward();
}

/*
* Overloads the original function in HaMMLET to use C-style file input
* for efficiency reasons. Some comments have been removed to better adapt the
* formatting to the thesis.
*/
template< typename T>
void MaxletTransform(
    FILE* fin,
    bool binary,
    vector<real_t>& coeffs,
    vector< SufficientStatistics<T> >& suffstats,
    const size_t nrDim = 1,
    const size_t reserveT = 0
) {
        if ( nrDim <= 0 ) {
                throw runtime_error( "Number of dimensions must be positive!" );
        }
        if ( coeffs.size() > 0 ) {
                throw runtime_error( "Coefficient array must be empty!" );
        }
```

```
if ( suffstats.size() > 0 ) {
        throw runtime_error( "Statistics array must be empty!" );
}
if ( fin != NULL ) {
        coeffs.reserve( ( reserveT + nrDim ) / nrDim + nrDim );
        suffstats.reserve( reserveT + nrDim );
vector<real_t> S;
        size_t i = 0;
        real_t v = 0;
        size_t dim = 0;
double inputNum = 0;

bool fileEnd = false;
if(!binary){
    fileEnd = (fscanf(fin, "%lf", &inputNum) == EOF);
}
else {
    fileEnd = (fread(&inputNum,1,sizeof(double),fin) != sizeof(double));
}
        while ( !fileEnd ) {
    v = (real_t)inputNum;
                S.push_back( v );
                suffstats.push_back( SufficientStatistics<T>( v ) );
                dim++;
                if ( dim == nrDim ) {
                        dim = 0;
                        coeffs.push_back( inf );
                        size_t j = i;
                        size_t m = 1;
                        real_t normalizer = sqrt2half;
                        while ( ( j & m ) > 0 ) {
                                real_t maxCoeff = 0;
                                size_t L = S.size() - 2 * nrDim;
                                size_t R = L + nrDim;
                                for ( size_t d = 0; d < nrDim; ++d ) {
                                        maxCoeff = max(maxCoeff,normalizer*abs(S[L]-S[R])
                                        S[L] += S[R];
                                        L++;
                                        R++;
                                }
                                coeffs[j] = maxCoeff;
                                for ( size_t d = 0; d < nrDim; ++d ) {
                                        S.pop_back();
                                }
                                j = j - m;
                                m *= 2;
                                normalizer *= sqrt2half;
                        }
                        i++;
                }
    if(!binary){
        fileEnd = (fscanf(fin, "%lf", &inputNum) == EOF);
    }
    else {
        fileEnd = (fread(&inputNum,1,sizeof(double),fin) !=
            sizeof(double));
            }
        }


        if ( dim != 0 ) {
                throw runtime_error( "Input stream did not contain enough values"
        "to fill all dimensions at last position!" );
        }

        coeffs[0] = inf;

    } else {
        throw runtime_error( "Cannot read input file or stream!" );
    }
}


#endif
```

B.1.5 Model.hpp

```
#ifndef WAHMM_MODEL_HPP
#define WAHMM_MODEL_HPP

#include "State.hpp"
#include "commons.hpp"

/*
This class represents a hidden Markov model for the specific scope
and context of the thesis (Gaussian states, continuous observations, etc).
The member variables are all public for efficiency reasons.
*/
class Model {
public:
    /** Collection of the states of the model */
    std::vector<State> mStates;
    /** Matrix of log probabilities of transitions between states */
    wahmm::real_t **mLogTransitions;
    /** Logarithm of the initial state probability distribution */
    std::vector<wahmm::real_t> mLogPi;
    /** Each entry is < n_w , K(n_w,j) for each state > */
    std::map<size_t, std::vector<wahmm::real_t>> mKValues;
    Model();
    Model(const Model& that);
    Model(std::vector<State>& states, std::vector<wahmm::real_t>&relativeTr,
        std::vector<wahmm::real_t>& relativePi);
    /** Print the model in a readable format, with classic probabilities */
    void printModel();
    /** Sort the model with ascending states order */
    void sortModel();
    // Useful operators for model input/output
    friend ostream& operator<<(ostream& os, const Model& m);
    friend istream& operator>>(istream& is, Model& m);
};
```

```
Model::Model(){}

Model::Model(const Model& that){
    mStates = that.mStates;
    for(int i = 0; i < mStates.size(); i++){
        mLogTransitions[i] = new wahmm::real_t[mStates.size()];
        for(int j = 0; j < mStates.size(); j++){
            mLogTransitions[i][j] = that.mLogTransitions[i][j];
        }
    }
    mLogPi = that.mLogPi;
}

Model::Model(std::vector<State>& states, std::vector<wahmm::real_t>& relativeTr,
    std::vector<wahmm::real_t>& relativePi){
    mStates = std::vector<State>(states);
    mLogTransitions = new wahmm::real_t*[states.size()];
    wahmm::real_t rowSum;
    for(int i = 0; i < states.size(); i++){
        mLogTransitions[i] = new wahmm::real_t[states.size()];
        rowSum = 0;
        for(int j = 0; j < states.size(); j++)
            rowSum += relativeTr[i*states.size() + j];
        for(int j = 0; j < states.size(); j++)
            mLogTransitions[i][j] = log(relativeTr[i*states.size() + j]) -
                log(rowSum);
        rowSum = 0;
        for(int j = 0; j < states.size(); j++)
            rowSum += relativePi[j];
        mLogPi.push_back(log(relativePi[i]) - log(rowSum));
    }
}

void Model::printModel(){
    std::cout << "[>] Model information:" << std::endl;
    for(State s : mStates){
        cout << s.name();
        cout << " | Mean: " << s.mean();
        cout << " | StdDev: " << s.stdDev();
        cout << endl;
    }
    std::cout << "----------" << std::endl;
    std::cout << "Transitions: " << std::endl;
    for(int i = 0; i < mStates.size(); i++){
        for(int j = 0; j < mStates.size(); j++)
            std::cout << exp(mLogTransitions[i][j]) << " ";
        std::cout << std::endl;
    }
    std::cout << "----------" << std::endl;
    std::cout << "Initial Distribution: " << std::endl;
    for(int i = 0; i < mStates.size(); i++)
        std::cout << exp(mLogPi[i]) << " ";
    std::cout << endl;
    std::cout << "----------" << std::endl;
}
```

```
void Model::sortModel(){
    for(size_t i = 0; i < mStates.size(); i++){
        for(size_t j = 0; j < mStates.size()-1; j++){
            if(mStates[j].mean() > mStates[j+1].mean()){
                State tmpState = mStates[j];
                mStates[j] = mStates[j+1];
                mStates[j+1] = tmpState;
                wahmm::real_t *tmpTr = mLogTransitions[j];
                mLogTransitions[j] = mLogTransitions[j+1];
                mLogTransitions[j+1] = tmpTr;
                wahmm::real_t tmpInit = mLogPi[j];
                mLogPi[j] = mLogPi[j+1];
                mLogPi[j+1] = tmpInit;
            }
        }
    }
}

ostream& operator<<(ostream& os, const Model& m){
    os.precision(100);
    os << m.mStates.size() << " ";
    for(State s : m.mStates)
        os << s.mean() << " " << s.stdDev() << " ";
    for(size_t i = 0; i < m.mStates.size(); i++)
        for(size_t j = 0; j < m.mStates.size(); j++)
            os << m.mLogTransitions[i][j] << " ";
    for(size_t i = 0; i < m.mStates.size(); i++)
        os << m.mLogPi[i] << " ";
    return os;
}

istream& operator>>(istream& is, Model& m){
    size_t nStates;
    wahmm::real_t inMean, inStdDev;
    is >> nStates;
    for(size_t n = 0; n < nStates; n++){
        is >> inMean >> inStdDev;
        m.mStates.push_back(State(inMean, inStdDev));
    }
    wahmm::real_t inTr;
    std::string inTrString;
    m.mLogTransitions = new wahmm::real_t*[nStates];
    for(size_t i = 0; i < nStates; i++){
        m.mLogTransitions[i] = new wahmm::real_t[nStates];
        for(size_t j = 0; j < nStates; j++){
            is >> inTrString;
            if(inTrString == "-inf")
                inTr = -infin;
            else
                inTr = std::stod(inTrString);
            m.mLogTransitions[i][j] = inTr;
        }
    }
    wahmm::real_t inPi;
    std::string inPiString;
    for(size_t i = 0; i < nStates; i++){
        is >> inPiString;
        if(inPiString == "-inf")
            inPi = -infin;
        else
            inPi = std::stod(inPiString);
        m.mLogPi.push_back(inPi);
    }
    return is;
}

#endif
```

B.1.6 parser.hpp

```
/*

Copyright (c) 2014 Jarryd Beck

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.

*/
#ifndef WAHMM_PARSER_HPP
#define WAHMM_PARSER_HPP

#include <iostream>

#include "cxxopts.hpp"
#include "commons.hpp"

cxxopts::ParseResult
parse(int argc, char* argv[])
{
  try
  {
    cxxopts::Options options(argv[0], "WaHMM - *Wa*velets on *H*idden *M*arkov"
        " *M*odels");
    bool apple = false;

    options.add_options()
        ("s,state", "Specify a state with a Gaussian distribution",
            cxxopts::value<std::vector<double>>(), "<mean>,<stdDev>")
        ("t,transitions", "Transition probabilities in relative terms. For "
            "example a two-state model could have -t 2.0,4.0,8.0,2.0 that "
            "will translate into [0.33, 0.67; 0.8, 0.2]",
            cxxopts::value<std::vector<double>>(), "<a_11>,...")
        ("i,initial", "Initial probability distribution",
            cxxopts::value<std::vector<double>>(), "<pi_1>,...")
        ("obs", "Input file containing the observations as double separated "
            "by a space",
            cxxopts::value<std::string>(), "<filename>")
        ("binary", "Specify that files for observations and generating path"
            "are in binary format",
            cxxopts::value<bool>())
        ("model", "Input file containing a saved model to use for evaluation "
            "and decoding problems",
            cxxopts::value<std::string>(), "<filename>")
        ("estimate", "Input file containing a saved model to use for the "
            "training problem",
            cxxopts::value<std::string>(), "<filename>")
        ("tofile", "Save results as files in the folder ./results/",
            cxxopts::value<bool>())
        ("evaluation", "Solve the evaluation problem using a standard "
            "implementation of the forward algorithm",
            cxxopts::value<bool>())
        ("decoding", "Solve the decoding problem using the Viterbi algorithm",
            cxxopts::value<bool>())
        ("training", "Solve the training problem with the Baum-Welch "
            "algorithm",
            cxxopts::value<bool>())
        ("compressed", "Use the compressed version of the algorithms",
            cxxopts::value<bool>())
        ("silence", "Suppress every message printed to screen",
            cxxopts::value<bool>())
        ("v,verbose", "Print extensive algorithms information",
            cxxopts::value<bool>())
        ("h,help", "Print this help message",
            cxxopts::value<bool>())
    ;

    auto result = options.parse(argc, argv);

    if (result.count("help")){
      std::cout << options.help({""}) << std::endl;
      exit(0);
    }

    return result;

  } catch (const cxxopts::OptionException& e)
  {
    std::cout << "error parsing options: " << e.what() << std::endl;
    exit(1);
  }
}

#endif
```

B.1.7 State.hpp

```
#ifndef WAHMM_STATE_HPP
#define WAHMM_STATE_HPP

#include "commons.hpp"

/**
* This class represents the State of a hidden Markov model; it is characterized
* by a continuous Gaussian emission probability distribution function.
*/
class State {
    static size_t idCounter;
    /** Numerical ID of the state */
    size_t mId;
    /** Name of the state */
    std::string mName;
    /** Mean of the Gaussian associated to the state */
    wahmm::real_t mMean;
    /** Standard deviation of the Gaussian associated to the state */
    wahmm::real_t mStdDev;
    /** Standard deviation logarithm precomputed for efficiency reasons */
    wahmm::real_t mLogStdDev;
public:
    /** Constructor with default name */
    State(wahmm::real_t mean, wahmm::real_t stdDev);
    /** Constructor with given name */
    State(wahmm::real_t mean, wahmm::real_t stdDev, std::string name);
    size_t id();
    std::string name();
    wahmm::real_t mean();
    wahmm::real_t stdDev();
    wahmm::real_t logStdDev();
    /** Return the log probability of sampling x from the state distribution */
    wahmm::real_t logPdf(wahmm::real_t x);
    /** Update the parameters of the distribution associated to the state */
    void updateParameters(wahmm::real_t mean, wahmm::real_t stdDev);
};

size_t State::idCounter = 0;
const wahmm::real_t log_sqrt2pi = 0.9189385332046727417803297364056176398613974;

State::State(wahmm::real_t mean, wahmm::real_t stdDev) : mMean(mean),
    mStdDev(stdDev),
    mLogStdDev(log(stdDev)),
    mName("State "+std::to_string(idCounter)),
    mId(idCounter){
        idCounter++;
    }
```

```
State::State(wahmm::real_t mean, wahmm::real_t stdDev, std::string name) :
    mMean(mean),
    mStdDev(stdDev),
    mLogStdDev(log(stdDev)),
    mName(name),
    mId(idCounter){
        idCounter++;
    }

size_t State::id(){
    return mId;
}

std::string State::name(){
    return mName;
}

wahmm::real_t State::mean(){
    return mMean;
}

wahmm::real_t State::stdDev(){
    return mStdDev;
}

wahmm::real_t State::logStdDev(){
    return mLogStdDev;
}

wahmm::real_t State::logPdf(wahmm::real_t x){
    return (- log_sqrt2pi - mLogStdDev - 0.5*pow((x - mMean) / mStdDev, 2) );
}

void State::updateParameters(wahmm::real_t mean, wahmm::real_t stdDev){
    mMean = mean;
    mStdDev = stdDev;
    mLogStdDev = log(stdDev);
}

#endif
```

B.1.8 utilities.hpp

```
#ifndef WAHMM_UTILITIES_HPP
#define WAHMM_UTILITIES_HPP
#include <iostream>
#include "commons.hpp"

/** Given (a = log(x) and b = log(y), returns log(x+y)) */
wahmm::real_t sum_logarithms(wahmm::real_t a, wahmm::real_t b){
    /**
        Perform log-sum-exp on a pair of numbers in log space..  This is calculated
        as z = log( e**x + e**y ). However, this causes underflow sometimes
        when x or y are too negative. A simplification of this is thus
        z = x + log( e**(y-x) + 1 ), where x is the greater number. If either of
        the inputs are infinity, return infinity, and if either of the inputs
        are negative infinity, then simply return the other input.
        */
    if(a == infin || b == infin)
        return infin;
    if(a == -infin)
        return b;
    if(b == -infin)
        return a;
    if(a > b)
        return a + log1pf(exp(b-a));
    return b + log1pf(exp(a-b));
}

/** Free a matrix of wahmm::real_t with 'rows' rows. */
void freeMatrix(wahmm::real_t** m, size_t rows){
    for(int i = 0; i < rows; i++)
        delete[] m[i];
    delete[] m;
}

/** Compute K(n_w,j) */
wahmm::real_t compute_k(Model& m, size_t j, size_t nw){
    const wahmm::real_t log_2pi = 1.83787706640934548356065947281123527972279494;
    const wahmm::real_t log_2pi_over2 = log_2pi/2;

    if(m.mKValues.count(nw) != 0){
        return m.mKValues.find(nw)->second.at(j);
    }

    // compute K(n_w, j) for all states and save the value in the map
    m.mKValues[nw] = std::vector<wahmm::real_t>();
    for(size_t i = 0; i < m.mStates.size(); i++){
        wahmm::real_t first = (nw-1)*m.mLogTransitions[i][i];
        wahmm::real_t second = nw*(m.mStates[i].logStdDev() +
            pow(m.mStates[i].mean()/m.mStates[i].stdDev(),2)/2 +
            log_2pi_over2);
        m.mKValues[nw].push_back(first - second);
    }

    return m.mKValues.find(nw)->second.at(j);

}

/** Compute E_w(j) */
wahmm::real_t compute_e(Model& m, size_t j, blockdata bd){
    wahmm::real_t num = 2*m.mStates[j].mean()*bd.s1 - bd.s2;
    wahmm::real_t den = 2*pow(m.mStates[j].stdDev(),2);
    return (num/den) + compute_k(m, j, bd.nw);
}

/**
* Prints a matrix in a compressed form with the following format (example 10x3):
* [ x_00 x_01 x_02 ]
* [ x_10 x_11 x_12 ]
* ...
* [ x_70 x_71 x_72 ]
* [ x_80 x_81 x_82 ]
* [ x_90 x_91 x_92 ]
* If the matrix is smaller than 5 rows, it will be printed fully.
* @param m the matrix to print
* @param rows
* @param cols
* @param matrixName printed before the matrix as a header
* @param byRow if true prints by row, otherwise by columns
*/
void printMatrixSummary(wahmm::real_t **m, size_t rows, size_t cols,
    std::string matrixName, bool byRow){
    cout << "[>] === " << matrixName << " ===" << endl;
    if(byRow){
        if(cols <= 5){
            for(int x = 0; x < rows; x++){
                for(int y = 0; y < rows; y++)
                    cout << m[x][y] << "\t";
                cout << endl;
            }
        }
        else{
            for(int y = 0; y < cols; y++)
                cout << m[0][y] << "\t";
            cout << endl;
            for(int y = 0; y < cols; y++)
                cout << m[1][y] << "\t";
            cout << endl;
            cout << "..." << endl;
            for(int y = 0; y < cols; y++)
                cout << m[rows-3][y] << "\t";
            cout << endl;
            for(int y = 0; y < cols; y++)
                cout << m[rows-2][y] << "\t";
            cout << endl;
            for(int y = 0; y < cols; y++)
                cout << m[rows-1][y-1] << "\t";
            cout << endl;
        }
    } else {
        if(cols <= 5){
            for(int y = 0; y < cols; y++){
                for(int x = 0; x < rows; x++)
                    cout << m[x][y] << "\t";
                cout << endl;
            }
        }
        else{
            for(int x = 0; x < rows; x++)
                cout << m[x][0] << "\t";
            cout << endl;
            for(int x = 0; x < rows; x++)
                cout << m[x][1] << "\t";
            cout << endl;
            cout << "..." << endl;
            for(int x = 0; x < rows; x++)
                cout << m[x][cols-3] << "\t";
            cout << endl;
            for(int x = 0; x < rows; x++)
                cout << m[x][cols-2] << "\t";
            cout << endl;
            for(int x = 0; x < rows; x++)
                cout << m[x][cols-1] << "\t";
            cout << endl;
        }
    }
    cout << "======" << endl;
}

#endif
```

B.1.9 WaHMM.hpp

```
#include "parser.hpp"
#include "commons.hpp"
#include "State.hpp"
#include "Model.hpp"
#include "algorithms.hpp"
#include "Compressor.hpp"
#include "algorithms_compressed.hpp"
#include <stdio.h>

int main(int argc, const char* argv[]){
    // std::string filename("data");
    // Compressor comp(filename);
    // comp.printAllBlocks();
    //std::cout.precision(8);
    //std::cout << std::scientific; // print numbers with scientific notation
    auto result = parse(argc, argv);

    Model model, estimate;
    std::vector<State> states;
    std::vector<wahmm::real_t> relTrans;
    std::vector<wahmm::real_t> relPi;
    std::vector<wahmm::real_t> observations;
    std::string fileObs, fileModelIn, pathOut;
    bool evaluation = false, decoding = false, training = false;
    bool binary = false, tofile = false;
    bool compressed = false;
    bool verbose = false, silence = false;
    Compressor *compressor;
    FILE *finObs, *finPath;

    // parsing arguments from command line
    if(result.count("model")){ // read the model from a file
        fileModelIn = result["model"].as<std::string>();
        // Read the file with input observations
        std::ifstream modelFileInput(fileModelIn);
        if(modelFileInput.is_open()){
            modelFileInput >> model;
        }
        else {
            std::cerr << "Cannot read file " + fileModelIn + " !" << std::endl;
            return -1;
        }
        modelFileInput.close();
    }
    else { // read the model from command line
        if(result.count("state")){
            std::vector<double> stateParams = result["state"]
                .as<std::vector<double>>();
            wahmm::real_t mean, stdDev;
            for(std::size_t i = 0; i < result.count("state"); i++){
                mean = stateParams[i*2];
                stdDev = stateParams[i*2 + 1];
                states.push_back(State(mean, stdDev));
            }
        }
        if(result.count("transitions")){
            std::vector<double> transParams = result["transitions"]
                .as<std::vector<double>>();
            for(double d : transParams)
                relTrans.push_back((wahmm::real_t)d);
        }
        if(result.count("initial")){
            std::vector<double> logParams = result["initial"]
                .as<std::vector<double>>();
            for(double d : logParams)
                relPi.push_back((wahmm::real_t)d);
        }
        model = Model(states, relTrans, relPi);
    }
    if(result.count("estimate")){
        fileModelIn = result["estimate"].as<std::string>();
        // Read the file with input observations
        std::ifstream modelFileInput(fileModelIn);
        if(modelFileInput.is_open()){
            modelFileInput >> estimate;
        }
        else {
            std::cerr << "Cannot read file " + fileModelIn + " !" << std::endl;
            return -1;
        }
        modelFileInput.close();
    }
    if(result.count("obs")){
        fileObs = result["obs"].as<std::string>();
    }
    if(result.count("binary"))
        binary = true;
    if(result.count("tofile"))
        tofile = true;
    if(result.count("evaluation"))
        evaluation = true;
```

```
if(result.count("decoding"))
    decoding = true;
if(result.count("training"))
    training = true;
if(result.count("compressed"))
    compressed = true;
if(result.count("verbose"))
    verbose = true;
if(result.count("silence"))
    silence = true;

// some input checks
if(states.size() != relPi.size()){
    std::cerr << "[Error] Wrong initial distribution size" << std::endl;
    return -1;
}
if(states.size()*states.size() != relTrans.size()){
    std::cerr << "[Error] Wrong number of transition probabilities";
    std::cerr << std::endl;
    return -1;
}
if(fileObs.empty()){
    std::cerr << "[Error] Input file for observations not specified";
    std::cerr << std::endl;
    return -1;
}
if(silence == true && verbose == true){
    std::cerr << "[Error] Silence + Verbose flags cannot be used together";
    std::cerr << std::endl;
    return -1;
}
// read observations
if(!compressed){
    if(!binary){
        if(verbose)
            std::cout << "[>] Reading observations... " << std::endl;
        wahmm::real_t number;
        // efficient file reading in C style
        // Read the file with input observations
        finObs = fopen(fileObs.c_str(), "r");
        if(finObs != NULL){
            while(fscanf(finObs, "%lf", &number) != EOF)
                observations.push_back(number);
        } else {
            std::cerr << "Cannot read file " + fileObs + " !" << std::endl;
            return -1;
        }
        fclose(finObs);
        if(verbose)
            std::cout << "[>] ... done." << std::endl;
    }
    else {
        if(verbose){
            std::cout << "[>] Reading observations from binary file... ";
            std::cout << std::endl;
        }
```

```
            finObs = fopen ( fileObs.c_str() , "rb" );
            if (finObs==NULL){
                std::cerr << "Cannot read file " + fileObs + " !" << std::endl;
                return -1;
            }
            double n;
            // read one number
            while(fread(&n,1,sizeof(double),finObs) == sizeof(double))
                observations.push_back((wahmm::real_t)n);
            // terminate
            fclose (finObs);
            if(verbose)
                std::cout << "[>] ... done." << std::endl;
        }
    }
    else {
        if(verbose)
            std::cout << "[>] Creating Compressor... " << std::endl;
        compressor = new Compressor(fileObs, binary);
        if(verbose)
            std::cout << "[>] ... done." << std::endl;
    }

    // print acquired model
    if(!silence)
        model.printModel();

    // execute the actual algorithms
    if(!compressed){
        if(verbose)
            std::cout << "[>] Starting standard algorithms." << std::endl;
        if(evaluation)
            evaluation_problem(model, observations, verbose, silence, tofile);
        if(decoding)
            decoding_problem(model, observations, verbose, silence, tofile);
        if(training)
            training_problem(estimate, observations, 1e-9, 100, verbose,
                silence, tofile);
    }
    else {
        if(verbose)
            std::cout << "[>] Starting compressed algorithms" << std::endl;
        if(evaluation)
            evaluation_compressed(model, compressor, verbose, silence, tofile);
        if(decoding)
            decoding_compressed(model, compressor, verbose, silence, tofile);
        if(training)
            training_compressed(estimate, compressor, 1e-9, 100, verbose,
                silence, tofile);
    }

    return 0;
}
```

B.2 Python

B.2.1 automated_test.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import subprocess
import sys
import viterbi_comparison
import utilities_io as uio
from math import exp, log, isnan
import numpy as np
import time

# helper functions
# using a relative difference
def compute_error(real, measured):
    if abs(real) > abs(measured):
        maximum = abs(real)
    else:
        maximum = abs(measured)
    if real == 0 and measured == 0:
        return 0
    return abs( (real - measured) / maximum )

# compute KL divergence between two univariate gaussians with means m and
# standard deviation s
# = log(s1/s0) + (s0^2 + (m0-m1)^2)/(2*s1^2) - 1/2
def kl_divergence_gaussians(m0, s0, m1, s1):
    a = log(s1/s0)
    b = (s0**2 + (m0-m1)**2)/(2*(s1**2))
    return a + b - 0.5

def savetofile(suffix, list):
    n = len(list)
    f = prefix + suffix
    out_file = open(f, "w")
    for i in range(0, n):
        out_file.write(str(list[i]) + " ")
    out_file.close()

# OPTIONS
topology = "fully-connected" # not used yet
# states = [2, 3, 5, 7, 11, 13]
states = [2, 3, 5]
# etas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
etas = [1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]

n_tests = 100
sequence_length = 1000000 # used ONLY to calculate relative errors in decoding
topology_prefix = "FC"
verbose = True
f_eval_prob = "results/evaluation_prob"
f_compr_eval_prob = "results/compressed_evaluation_prob"
f_decod_prob = "results/decoding_prob"
```

```python
f_compr_decod_prob = "results/compressed_decoding_prob"
f_train_mod = "results/training_model"
f_compr_train_mod = "results/compressed_training_model"
# output files
f_eval_out = "evaluation"
f_eval_time_std_out = "evaluation_std_time"
f_eval_time_compr_out = "evaluation_compr_time"
f_decod_prob_out = "decoding_prob"
f_decod_path_std_out = "decoding_std_path"
f_decod_path_compr_out = "decoding_compr_path"
f_decod_time_std_out = "decoding_std_time"
f_decod_time_compr_out = "decoding_compr_time"
f_train_std_out = "training_std"
f_train_compr_out = "training_compr"
# f_train_model_std_out = "training_model_std"
# f_train_model_compr_out = "training_model_compr"
f_train_time_std_out = "training_std_time"
f_train_time_compr_out = "training_compr_time"
# SCRIPTS PATHS
f_generate_states = "python/generate_states.py"
f_generate_model = "python/create_model_file.py"
f_generate_data = "python/generate_data.py"
f_wahmm = "bin/WaHMM"
# MAIN PROGRAM ARGUMENTS
# --import data/model --obs data/bin_observations --binary --tofile
# --evaluation --decoding --training
wahmm_args = []
wahmm_args.append("bin/WaHMM")
wahmm_args.append("--model")
wahmm_args.append("data/model")
wahmm_args.append("--estimate")
wahmm_args.append("data/kmeans_model")
wahmm_args.append("--obs")
wahmm_args.append("data/bin_observations")
wahmm_args.append("--binary")
wahmm_args.append("--silence")
wahmm_args.append("--tofile")
eval_std_args = wahmm_args.copy()
eval_std_args.append("--evaluation")
decod_std_args = wahmm_args.copy()
decod_std_args.append("--decoding")
train_std_args = wahmm_args.copy()
train_std_args.append("--training")
eval_compr_args = eval_std_args.copy()
eval_compr_args.append("--compressed")
decod_compr_args = decod_std_args.copy()
decod_compr_args.append("--compressed")
train_compr_args = train_std_args.copy()
train_compr_args.append("--compressed")

if verbose:
    print("=== WaHMM AUTOMATED TESTING ===")
    print("eta:",etas," #states:",states," #tests:",n_tests)

skip_index = 0
test_count = 1
```

```
for eta in etas:
    print("[Test] --- Using Eta:",eta,"---")
    for n_states in states:
        print("[Test] --- Model with",n_states,"states ---")
        if verbose:
            print("[Test] Generating states... ",end="",flush=True)
        arguments = [f_generate_states, str(eta), str(n_states)]
        subprocess.call(arguments)
        if verbose:
            print("done.",flush=True)
        if verbose:
            print("[Test] Generating model... ",end="",flush=True)
        subprocess.call(f_generate_model)
        if verbose:
            print("done.",flush=True)

        # TEST THE MODEL
        evaluation_errors = []
        evaluation_times_std = []
        evaluation_times_compr = []
        decoding_errors = []
        decoding_paths_std_errors = []
        decoding_paths_compr_errors = []
        decoding_times_std = []
        decoding_times_compr = []
        ur_model_diff = []
        cr_model_diff = []
        # u_model = []
        # c_model = []
        training_times_std = []
        training_times_compr = []
        for iteration in range(1, n_tests+1):
            # Step 1: data generation
            if verbose:
                print("[Test",test_count,"] Generating data... ",end="",
                    flush=True)
            subprocess.call(f_generate_data)
            if verbose:
                print("done.",flush=True)

            # Step 2: evaluation problem
            if verbose:
                print("[Test",test_count,"] -- Running WaHMM uncompressed "
                    "evaluation...")
            start = time.perf_counter()
            subprocess.call(eval_std_args)
            end = time.perf_counter()
            evaluation_times_std.append(end - start)
            if verbose:
                print("[Test",test_count,"] WaHMM uncompressed evaluation "
                    "finished.")
            if verbose:
                print("[Test",test_count,"] Running WaHMM compressed "
                    "evaluation...")
            start = time.perf_counter()
            subprocess.call(eval_compr_args)
            end = time.perf_counter()
            evaluation_times_compr.append(end - start)
            if verbose:
                print("[Test",test_count,"] WaHMM compressed evaluation "
                    "finished.")

            in_eval_file = open(f_eval_prob, "r")
            evaluation_prob = float(in_eval_file.read())
            in_eval_file.close()
            in_eval_file = open(f_compr_eval_prob, "r")
            compressed_evaluation_prob = float(in_eval_file.read())
            in_eval_file.close()
            eval_relative_error = compute_error(evaluation_prob,
                compressed_evaluation_prob)
            if verbose:
                print("[Test",test_count,"] Uncompressed evaluation "
                    "probability:", evaluation_prob)
                print("[Test",test_count,"] Compressed evaluation "
                    "probability:", compressed_evaluation_prob)
                print("[Test",test_count,"] Relative Error:",
                    eval_relative_error)
            evaluation_errors.append(eval_relative_error)

            # Step 3: decoding problem
            if verbose:
                print("[Test",test_count,"] -- Running WaHMM uncompressed "
                    "decoding...")
            start = time.perf_counter()
            subprocess.call(decod_std_args)
            end = time.perf_counter()
            decoding_times_std.append(end - start)
            if verbose:
                print("[Test",test_count,"] WaHMM uncompressed decoding "
                    "finished.")
            if verbose:
                print("[Test",test_count,"] Running WaHMM compressed "
                    "decoding...")
            start = time.perf_counter()
            subprocess.call(decod_compr_args)
            end = time.perf_counter()
            decoding_times_compr.append(end - start)
            if verbose:
                print("[Test",test_count,"] WaHMM compressed decoding "
                    "finished.")

            in_decod_file = open(f_decod_prob, "r")
            decoding_prob = float(in_decod_file.read())
            in_decod_file.close()
            in_decod_file = open(f_compr_decod_prob, "r")
            compressed_decoding_prob = float(in_decod_file.read())
            in_decod_file.close()
            decod_relative_error = compute_error(decoding_prob,
                compressed_decoding_prob)
            if verbose:
                print("[Test",test_count,"] Uncompressed decoding probability:",
                    decoding_prob)
```

```
        print("[Test",test_count,"] Compressed decoding probability:",
            compressed_decoding_prob)
        print("[Test",test_count,"] Relative Error:",
            decod_relative_error)
    decoding_errors.append(decod_relative_error)
    path_errors = viterbi_comparison.count_differences_uncompressed() /
        sequence_length
    if verbose:
        print("[Test",test_count,"] Fraction of errors in path for "
            "uncompressed:", path_errors)
    decoding_paths_std_errors.append(path_errors)
    path_errors = viterbi_comparison.count_differences_compressed() /
        sequence_length
    if verbose:
        print("[Test",test_count,"] Fraction of errors in path for "
            "compressed:", path_errors)
    decoding_paths_compr_errors.append(path_errors)

    # Step 4: training problem
    if verbose:
        print("[Test",test_count,"] -- Running WaHMM uncompressed "
            "training...")
    start = time.perf_counter()
    subprocess.call(train_std_args)
    end = time.perf_counter()
    training_times_std.append(end - start)
    if verbose:
        print("[Test",test_count,"] WaHMM uncompressed training "
            "finished.")
    if verbose:
        print("[Test",test_count,"] Running WaHMM compressed "
            "training...")
    start = time.perf_counter()
    subprocess.call(train_compr_args)
    end = time.perf_counter()
    training_times_compr.append(end - start)
    if verbose:
        print("[Test",test_count,"] WaHMM compressed training "
            "finished.")

    # r stands for "real", u for "uncompressed" and c for "compressed"
    r_nstates, r_means, r_stddevs, r_trans, r_init = uio
        .read_model()
    u_nstates, u_means, u_stddevs, u_trans, u_init = uio
        .read_model(f_train_mod)
    c_nstates, c_means, c_stddevs, c_trans, c_init = uio
        .read_model(f_compr_train_mod)

    ur_diff = []
    cr_diff = []

    # Save differences
    # number of states, just to remember that
    ur_diff.append(r_nstates)
    cr_diff.append(r_nstates)
    # states
```

```
            for i in range(0, r_nstates):
                ur_diff.append(kl_divergence_gaussians(r_means[i], r_stddevs[i],
                    u_means[i], u_stddevs[i]))
                cr_diff.append(kl_divergence_gaussians(r_means[i], r_stddevs[i],
                    c_means[i], c_stddevs[i]))
            # transitions
            for i in range(0, r_nstates):
                for j in range(0, r_nstates):
                    ur_diff.append(compute_error(r_trans[i*r_nstates+j],
                        u_trans[i*r_nstates+j]))
                    cr_diff.append(compute_error(r_trans[i*r_nstates+j],
                        c_trans[i*r_nstates+j]))
            # initial distributions
            ur_diff.append(compute_error(r_init[0], u_init[0]))
            cr_diff.append(compute_error(r_init[0], c_init[0]))

            ur_model_diff.append(ur_diff)
            cr_model_diff.append(cr_diff)

            test_count = test_count + 1


        # Save testing results to file
        prefix = "tests/" + topology_prefix + "_" + str(n_states) + "_" +
            str(eta) + "_"
        print("[Test] Saving files with prefix:",prefix)
        # Evaluation
        savetofile(f_eval_out, evaluation_errors)
        savetofile(f_eval_time_std_out, evaluation_times_std)
        savetofile(f_eval_time_compr_out, evaluation_times_compr)
        # Decoding
        savetofile(f_decod_prob_out, decoding_errors)
        savetofile(f_decod_path_std_out, decoding_paths_std_errors)
        savetofile(f_decod_path_compr_out, decoding_paths_compr_errors)
        savetofile(f_decod_time_std_out, decoding_times_std)
        savetofile(f_decod_time_compr_out, decoding_times_compr)
        # Training
        out_file = open(prefix+f_train_std_out, "w")
        for i in range(0, n_tests):
            for x in range(0, len(ur_model_diff[i])):
                out_file.write(str(ur_model_diff[i][x]) + " ")
            out_file.write("\n")
        out_file.close()
        out_file = open(prefix+f_train_compr_out, "w")
        for i in range(0, n_tests):
            for x in range(0, len(cr_model_diff[i])):
                out_file.write(str(cr_model_diff[i][x]) + " ")
            out_file.write("\n")
        out_file.close()
        savetofile(f_train_time_std_out, training_times_std)
        savetofile(f_train_time_compr_out, training_times_compr)


print("\n[Test] -- Testing is finished.")
```

B.2.2 create_model_file.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from math import log
import utilities_io as uio

# Creates a model file; the input params can be inserted by hand

# The file should be written with the following format
# n_states mean std_dev ... log(transitions) ... log(initial_distribution) ...

# OPTIONS
automatic = True
states_file = "data/states"
trans_prob = 10.0/100000.0 # wanted_trans_n / sequence length = 10/10^6
topology = "fully-connected"

out_file = open(uio.model_file, "w")

if automatic:
    in_file = open(states_file, "r")
    line = in_file.read()
    string_list = line.split()
    # number of states
    n_states = len(string_list)
    out_file.write(str(n_states) + " ")
    # states distribution (mean, std_dev)
    for i in range(0, n_states):
        out_file.write(string_list[i] + " 1 ")
    if topology == "fully-connected":
        # transition matrix
        self_trans_prob = 1 - trans_prob
        out_trans_prob = trans_prob / (n_states - 1)
        for i in range(0, n_states):
            for j in range(0, n_states):
                if i == j:
                    out_file.write(str(log(self_trans_prob)) + " ")
                else:
                    out_file.write(str(log(out_trans_prob)) + " ")
        # initial distribution
        out_file.write(str(log(1)) + " ")
        for i in range(1, n_states):
            out_file.write("-inf ")
    if topology == "left-to-right":
        # transition matrix
        self_trans_prob = 1 - trans_prob
        out_trans_prob = trans_prob
        for i in range(0, n_states):
            for j in range(0, n_states):
                # last state is absorbing
                if i == j and i == n_states-1:
                    out_file.write(str(log(1)) + " ")
                # self-transition
                elif i == j:
                    out_file.write(str(log(self_trans_prob)) + " ")
                # transition out
                elif i == j-1:
                    out_file.write(str(log(out_trans_prob)) + " ")
                else:
                    out_file.write("-inf ")
        # initial distribution
        out_file.write(str(log(1)) + " ")
        for i in range(1, n_states):
            out_file.write("-inf ")
    if topology == "circular":
        # transition matrix
        self_trans_prob = 1 - trans_prob
        out_trans_prob = trans_prob
        for i in range(0, n_states):
            for j in range(0, n_states):
                # last state
                if j == 0 and i == n_states-1:
                    out_file.write(str(log(out_trans_prob)) + " ")
                # self-transition
                elif i == j:
                    out_file.write(str(log(self_trans_prob)) + " ")
                # transition out
                elif j == i+1:
                    out_file.write(str(log(out_trans_prob)) + " ")
                else:
                    out_file.write("-inf ")
        # initial distribution
        out_file.write(str(log(1)) + " ")
        for i in range(1, n_states):
            out_file.write("-inf ")
else:
    # number of states
    out_file.write("2 ")
    # states distributions (mean, std_dev)
    out_file.write("0 1 10 1 ")
    # transition matrix
    out_file.write(str(log(0.999)) + " " + str(log(0.001)) + " ")
    out_file.write(str(log(0.001)) + " " + str(log(0.999)) + " ")
    # initial distribution
    out_file.write(str(log(1)) + " " + "-inf ")

out_file.close()
```

B.2.3 generate_data.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import pomegranate as pm
import numpy as np
import utilities_io as uio
import utilities_kmeans as ukm

# OPTIONS
# Lenght of the observations sequence
```

```
sequence_length = 100000
# Choose if kmeans should be performed or not
perform_kmeans = True
# Write the sequence to a binary file
output_binary = True
# Write the sequence1  in a human readable format
output_readable = True
# Also produce a file for the generating path (always human readable)
output_path = True

# read model from input file
n_states, means, std_devs, transitions, initial = uio.read_model()

# create Pomegranate model
dists = []
for i in range(0, n_states):
    dists.append(pm.NormalDistribution(means[i], std_devs[i]))
trans_mat = []
for i in range(0, n_states):
    trans_mat.append([])
    for j in range(0, n_states):
        trans_mat[i].append(transitions[i*n_states + j])
trans_mat = np.array(trans_mat)
starts = np.array(initial)
model = pm.HiddenMarkovModel.from_matrix(trans_mat, dists, starts)
model.bake()

# generate samples from the model
if output_path:
    samples = model.sample(length=sequence_length, path=True, random_state=None)
    observations = samples[0]
    state_path = samples[1]
else:
    samples = model.sample(length=sequence_length, path=False,
        random_state=None)
    observations = samples

if output_readable:
    uio.write_observations(observations)
if output_binary:
    uio.write_observations_binary(observations)
if output_path:
    uio.write_path(state_path)

if perform_kmeans:
    ukm.estimate_model(k=n_states)
```

B.2.4 generate_states.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import sys
# SYNTAX: generate_states.py <eta> <n_states>
# Generate the states means given a certain Eta as defined in the thesis
# (currently at (2.39)); standard deviation is always one

if len(sys.argv) != 3:
    print("Error. Please execute as: generate_states.py <eta> <n_states>")
    exit(1)

# OPTIONS
eta = float(sys.argv[1]) # 0 is total overlap, 1 is non-overlapping within
    # +/- 3 sigmas
n_states = int(sys.argv[2])
filename = "data/states"

means = []
#print("Using eta =",eta,"and n_states =",n_states)
# the first state is always the standard normal Z(0,1)
means.append(0.0)
for i in range(1, n_states):
    means.append(6*eta + means[i-1])
#print("Generated states:",means)

# save to file for test automation
out_file = open(filename, "w")
for i in range(0, n_states):
    out_file.write(str(means[i]))
    if i < n_states-1:
        out_file.write(" ")
out_file.close()
```

B.2.5 plot_data.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import utilities_io as uio

# OPTIONS
use_binary_file = True
input_limit = 1000

if use_binary_file:
    observations = uio.read_observations_binary()
    state_path = uio.read_path()
else:
    observations = uio.read_observations(limit=input_limit)
    state_path = uio.read_path(limit=input_limit)

n_states, means, std_devs, transitions, initial = uio.read_model()
means_path = state_path
for i in range(0, len(state_path)):
    means_path[i] = means[state_path[i]]

if input_limit > 0 and use_binary_file == False:
    x = range(1,input_limit+1)
```

```
else:
    x = range(1,len(observations)+1)

plt.scatter(x, observations, color='blue')
plt.step(x, means_path, color='red')
plt.show()
```

B.2.6 plot_kmeans.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
from math import exp
import utilities_io as uio
import utilities_kmeans as ukm

# read kmeans model from input file
n_states, means, std_devs, transitions, initial = ukm.read_kmeans_model()

# read kmeans labels from file
labels = ukm.read_kmeans_labels()

# read observations from input file
in_file = open(uio.observations_file, "r")
line = in_file.read()
string_list = line.split()
value_list = []
value_x = []
for i in range(0, n_states):
    value_list.append([])
    value_x.append([])
input_limit = 1000000
counter = 0
for s in string_list:
    value_list[labels[counter]].append(float(s))
    value_x[labels[counter]].append(counter)
    counter = counter + 1
    if counter > input_limit:
        break
in_file.close()

for i in range(0, n_states):
    print("State",i,"- Mean:",means[i],"- StdDev:",std_devs[i])

x = range(1, counter+1)
means_plot = []
std_low_plot = []
std_high_plot = []
for i in range(0, n_states):
    means_plot.append([means[i]] * len(x))
    std_low_plot.append([means[i] - 3*std_devs[i]] * len(x))
    std_high_plot.append([means[i] + 3*std_devs[i]] * len(x))

# plt.scatter(x, value_list, color='blue')
```

```
for i in range(0, n_states):
    # -- either differentiate clusters
    plt.scatter(value_x[i], value_list[i])
    # -- or just print observations
    # plt.scatter(value_x[i], value_list[i], color='blue')

    plt.plot(x, means_plot[i], color='black')
    # plt.plot(x, std_low_plot[i], color='red')
    # plt.plot(x, std_high_plot[i], color='red')
plt.show()
```

B.2.7 pomegranate_test.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import pomegranate as pm
import numpy as np
import math
import json
import utilities_io as uio
import utilities_kmeans as ukm

# OPTIONS
do_evaluation = True
do_decoding = True
do_training = True
evaluation_file = "results/pm_evaluation_prob"
viterbipath_file = "results/pm_decoding_path"
viterbilogp_file = "results/pm_decoding_prob"
training_file = "results/pm_training_model"

# read model from input file
n_states, means, std_devs, transitions, initial = uio.read_model()

# create Pomegranate model
dists = []
for i in range(0, n_states):
    dists.append(pm.NormalDistribution(means[i], std_devs[i]))
trans_mat = []
for i in range(0, n_states):
    trans_mat.append([])
    for j in range(0, n_states):
        trans_mat[i].append(transitions[i*n_states + j])
trans_mat = np.array(trans_mat)
starts = np.array(initial)
model = pm.HiddenMarkovModel.from_matrix(trans_mat, dists, starts)
model.bake()

# read observations from input file
observations = uio.read_observations_binary()

# problem 1 : maximum likelihood, forward algorithm
# To compare log probabilities, you can consider the ratio between them:
# specifically, with two states we can see that:
```

```
#     prob_1/prob_2 = exp(log(prob_1) - log(prob_2))
if do_evaluation:
    print("--- PM Evaluation problem ---")
    forward_matrix = model.forward(observations)
    print(forward_matrix)
    # computer evaluation probability
    inf = float("inf")
    neg_inf = float("-inf")
    evaluation = neg_inf
    for alpha in forward_matrix[-1]:
        if evaluation == inf or alpha == inf:
            evaluation = inf
            break # the end result will be +inf no matter what
        elif evaluation == neg_inf:
            evaluation = alpha
        elif alpha == neg_inf:
            evaluation = evaluation
        elif evaluation > alpha:
            evaluation = evaluation + math.log(1 + math.exp(alpha-evaluation))
        else:
            evaluation = alpha + math.log(1 + math.exp(evaluation-alpha))
    print("P( O | lambda ):", evaluation)
    # save evaluation prob to file for future comparisons
    out_file = open(evaluation_file, "w")
    out_file.write(str(evaluation))
    out_file.close()

# problem 2 : Viterbi decoding, Viterbi algorithm
# compute the predicted_path using Viterbi algorithm and count errors
if do_decoding:
    print("--- PM Decoding problem ---")
    path_logp, path_tuples = model.viterbi(observations)
    print("P( O | Q,lambda ):",path_logp)
    state_path = []
    for tuple in path_tuples:
        state_path.append(tuple[0])
    state_path = state_path[1:]
    # save logp to file for future comparisons
    out_file = open(viterbilogp_file, "w")
    out_file.write(str(path_logp))
    out_file.close()
    # save the state path to file for future comparisons
    out_file = open(viterbipath_file, "w")
    for p in state_path:
        out_file.write(str(p) + " ")
    out_file.close()
    # print a shortened version of the state path
    print("Most likely path: ",end='')
    if len(state_path) > 10:
        for i in range(0, 5):
            print(state_path[i], end=' ')
        print("...", end=' ')
        for i in range(-5, 0):
            print(state_path[i], end=' ')
        print("")
    else:
        for i in range(0, len(state_path)):
            print(state_path[i], end=' ')
        print("")


# problem 3 : expectation maximization, baum-welch
# read model estimated with kmeans
if do_training:
    print("--- PM Training problem ---")
    n_states, means, std_devs, transitions, initial = uio.read_model()
    # create Pomegranate model
    dists = []
    for i in range(0, n_states):
        dists.append(pm.NormalDistribution(means[i], std_devs[i]))
    trans_mat = []
    for i in range(0, n_states):
        trans_mat.append([])
        for j in range(0, n_states):
            trans_mat[i].append(transitions[i*n_states + j])
    trans_mat = np.array(trans_mat)
    starts = np.array(initial)
    estimate_model = pm.HiddenMarkovModel.from_matrix(trans_mat, dists, starts)
    estimate_model.bake()

    estimate_model.fit(list([np.array(observations)]))
    # edges in json have the following format:
    # - (start node, end node, probability, pseudocount, label)
    print(estimate_model.to_json())

    out_file = open(training_file, "w")
    out_file.write(estimate_model.to_json())
    out_file.close()
```

B.2.8 results_aggregation.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from __future__ import print_function
import utilities_io as uio
from math import sqrt
import sys
from os import listdir
import subprocess
import matplotlib.pyplot as plt
from statistics import median, pstdev

# OPTIONS
n_tests = 100
save_boxplots = True
topologies = ["FC", "CI", "LR"]
n_states = ["2", "3", "5"]
etas = ["0.1", "0.2", "0.3", "0.4", "0.5", "0.6", "0.7", "0.8", "0.9", "1.0"]
x = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
folder = "plots/"
```

```
base_savename = folder+"PLOT_"

# file names
f_ev = "evaluation"
f_de_pr = "decoding_prob"
f_de_std_pa = "decoding_std_path"
f_de_compr_pa = "decoding_compr_path"
f_tr_std = "training_std"
f_tr_compr = "training_compr"
f_ev_std_t = "evaluation_std_time"
f_ev_compr_t = "evaluation_compr_time"
f_de_std_t = "decoding_std_time"
f_de_compr_t = "decoding_compr_time"
f_tr_std_t = "training_std_time"
f_tr_compr_t = "training_compr_time"


def save_boxplot(data, title, topology, suffix, ybot, ytop, ylabel, xlabel):
    f = base_savename + topology + "_" + suffix
    plt.title(title)
    fig, ax = plt.subplots(len(n_states), constrained_layout=True)
    for i in range(0, len(n_states)):
        ax[i].set(title=n_states[i]+" states")
        ax[i].yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
            alpha=0.5)
        ax[i].set_ylim(ybot[i], ytop[i])
        ax[i].set_xticklabels(etas)
        ax[i].boxplot(data[i])
        # plt.figure(figsize=(2,1))
        # plt.tight_layout()
        ax[i].set_xlabel(xlabel)
        ax[i].set_ylabel(ylabel)
    plt.savefig(f)
    plt.close()
    plt.clf()


def save_plot(x, y_allstates, title, topology, suffix, ylabel, xlabel):
    f = base_savename + topology + "_" + suffix
    line_styles = ['-', '--', ':']
    for i in range(0, len(n_states)):
        plt.title(title)
        plt.plot(x, y_allstates[i], label=n_states[i]+" states", \
            linestyle=line_styles[i], marker="o")
        plt.legend(loc='upper right', frameon=False)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig(f)
    plt.close()
    plt.clf()


def plot_evaluation(topology):
    ev_list = []
    y_all = []
    for n in n_states:
        ev = []
        y = []
        for eta in etas:
            f = "tests/" + topology + "_" + n + "_" + eta + "_"
            list = uio.read_file_to_list(f+f_ev)
            floatlist = [float(e) for e in list]
            y.append(median(floatlist))
            ev.append(floatlist)
        ev_list.append(ev)
        y_all.append(y)
    save_plot(x, y_all, "Evaluation: probability relative error", topology, \
        "evaluation", "Error (relative)", "State separation")
    if topology == "LR":
        tmp_ybot = [0.0,0.0,0.0]
        tmp_ytop = [0.0005, 0.0005, 0.001]
    else:
        tmp_ybot = [0.0,0.0,0.0]
        tmp_ytop = [0.003, 0.003, 0.003]
    save_boxplot(data=ev_list, title="Evaluation: probability error", \
        topology=topology, suffix="evaluation_boxplot", ybot=tmp_ybot, \
        ytop=tmp_ytop, ylabel="Error (relative)", xlabel="State separation")


def plot_decoding(topology):
    de_list = []
    y_all = []
    for n in n_states:
        de = []
        y = []
        for eta in etas:
            f = "tests/" + topology + "_" + n + "_" + eta + "_"
            list = uio.read_file_to_list(f+f_de_std_pa)
            floatlist_std = [float(e) for e in list]
            list = uio.read_file_to_list(f+f_de_compr_pa)
            floatlist_compr = [float(e) for e in list]
            floatlist_diff = []
            for i in range(0, len(floatlist_std)):
                floatlist_diff.append(floatlist_compr[i] - floatlist_std[i])
            de.append(floatlist_diff)
            y.append(median(floatlist_diff))
        de_list.append(de)
        y_all.append(y)
    save_plot(x, y_all, "Decoding: relative path error", topology, \
        "decoding", "Error (relative)", "State separation")
    if topology == "LR":
        tmp_ybot = [0.0,0.0,0.0]
        tmp_ytop = [0.001, 0.001, 0.001]
    else:
        tmp_ybot = [0.0,0.0,0.0]
        tmp_ytop = [0.001, 0.001, 0.001]
    save_boxplot(data=de_list, title="Decoding: path error increment", \
        topology=topology, suffix="decoding_boxplot", ybot=tmp_ybot, \
        ytop=tmp_ytop, ylabel="Error (relative)", xlabel="State separation")


def summarize_training(filename):
    kl = []
    tr = []
    pi = []
    list = uio.read_file_to_list(filename)
    index = 0
    for t in range(0, n_tests):
        statesnum = int(list[index])
        index = index + 1
        # average state kl
        current_kl = 0
        for i in range(0, statesnum):
            current_kl = current_kl + float(list[index])
            index = index + 1
        current_kl = current_kl/statesnum
        kl.append(current_kl)
        # average transition relative error
        current_tr = 0
        for i in range(0, statesnum):
            for j in range(0, statesnum):
                current_tr = current_tr + float(list[index])
                index = index + 1
        current_tr = current_tr / (statesnum**2)
        tr.append(current_tr)
        # average initial distribution relative error
        pi.append(float(list[index]))
        index = index + 1
    return kl, tr, pi


def plot_training(topology):
    tr_kl_list = []
    tr_tr_list = []
    tr_in_list = []
    y_kl_all = []
    y_tr_all = []
    y_in_all = []
    for n in n_states:
        tr_kl = []
        tr_tr = []
        tr_in = []
        y_kl = []
        y_tr = []
        y_in = []
        for eta in etas:
            tr_kl_std = []
            tr_tr_std = []
            tr_in_std = []
            tr_kl_compr = []
            tr_tr_compr = []
            tr_in_compr = []
            tr_kl_diff = []
            tr_tr_diff = []
            tr_in_diff = []
            f = "tests/" + topology + "_" + n + "_" + eta + "_"
            # read standard training results
            tr_kl_std, tr_tr_std, tr_in_std = summarize_training(f +
                f_tr_std)
            # read compressed training results
            tr_kl_compr, tr_tr_compr, tr_in_compr = summarize_training(f +
                f_tr_compr)
            for i in range(0, len(tr_kl_std)):
                tr_kl_diff.append(tr_kl_compr[i] - tr_kl_std[i])
            for i in range(0, len(tr_tr_std)):
                tr_tr_diff.append(tr_tr_compr[i] - tr_tr_std[i])
            for i in range(0, len(tr_in_std)):
                tr_in_diff.append(tr_in_compr[i] - tr_in_std[i])
            tr_kl.append(tr_kl_diff)
            tr_tr.append(tr_tr_diff)
            tr_in.append(tr_in_diff)
            y_kl.append(median(tr_kl_diff))
            y_tr.append(median(tr_tr_diff))
            y_in.append(median(tr_in_diff))
        tr_kl_list.append(tr_kl)
        tr_tr_list.append(tr_tr)
        tr_in_list.append(tr_in)
        y_kl_all.append(y_kl)
        y_tr_all.append(y_tr)
        y_in_all.append(y_in)

    save_plot(x, y_kl_all, "Training: KL-divergence difference", topology,
        "training_kl", "Difference (abs)", "State separation")
    save_plot(x, y_tr_all, "Training: Transitions relative error difference",
        topology, "training_tr", "Difference (abs)", "State separation")
    save_plot(x, y_in_all, "Training: Initial distribution error difference",
        topology, "training_in", "Difference (abs)", "State separation")

    # Training KL
    if topology == "CI":
        tmp_ybot = [-0.01, -0.5, -2]
        tmp_ytop = [0.01, 0.3, 0.5]
    elif topology == "FC":
        tmp_ybot = [-0.02, -1, -2]
        tmp_ytop = [0.01, 0.3, 1]
    elif topology == "LR":
        tmp_ybot = [-0.1, -8, -14]
        tmp_ytop = [0.05, 8, 10]
    save_boxplot(data=tr_kl_list, title="Training: states error", \
        topology=topology, suffix="training_kl_boxplot", ybot=tmp_ybot, \
        ytop=tmp_ytop, ylabel="Difference (abs)", \
        xlabel="State separation")
    # Training TR
    tmp_ybot = [-0.5, -0.5, -0.5]
    tmp_ytop = [0.5, 0.5, 0.5]
    save_boxplot(data=tr_tr_list, title="Training: transitions error", \
        topology=topology, suffix="training_tr_boxplot", ybot=tmp_ybot, \
        ytop=tmp_ytop, ylabel="Difference (abs)", \
        xlabel="State separation")
    # Training IN
    if topology == "LR":
        tmp_ybot = [-0.000011, -0.000011, -0.000011]
        tmp_ytop = [0.0000005, 0.0000005, 0.0000005]
    else:
        tmp_ybot = [-0.00000025, -0.000005, -0.00001]
        tmp_ytop = [0.00000005, 0.0000005, 0.0000005]
    save_boxplot(data=tr_in_list, title="Training: initial distribution error",\
        topology=topology, suffix="training_in_boxplot", ybot=tmp_ybot, \
        ytop=tmp_ytop, ylabel="Difference (abs)", \
        xlabel="State separation")


def plot_speedup(topology):
    titles = ["Speedup: evaluation", "Speedup: decoding", "Speedup: training"]
    problems = ["evaluation", "decoding", "training"]
    files = dict()
    files["evaluation"] = [f_ev_std_t, f_ev_compr_t]
    files["decoding"] =  [f_de_std_t, f_de_compr_t]
    files["training"] = [f_tr_std_t, f_tr_compr_t]
    for p in problems:
        speedup_list = []
        y_all = []
        for n in n_states:
            speedup = []
            y = []
            for eta in etas:
                f = "tests/" + topology + "_" + n + "_" + eta + "_"
                list = uio.read_file_to_list(f+files[p][0])
                floatlist_std = [float(e) for e in list]
                list = uio.read_file_to_list(f+files[p][1])
                floatlist_compr = [float(e) for e in list]
                floatlist_ratio = []
                for i in range(0, len(floatlist_std)):
                    floatlist_ratio.append(floatlist_std[i]/floatlist_compr[i])
                speedup.append(floatlist_ratio)
                y.append(median(floatlist_ratio))
            speedup_list.append(speedup)
            y_all.append(y)
        save_plot(x, y_all, "Speedup: "+p, topology, "speedup_"+p, \
            "Speedup", "State separation")
        if topology == "CI":
            if p == "evaluation":
                tmp_ybot = [0.0, 0.5, 1.0]
                tmp_ytop = [1.0, 1.5, 2.0]
            elif p == "decoding":
                tmp_ybot = [0.5, 0.5, 0.5]
                tmp_ytop = [1.5, 1.5, 1.5]
            elif p =="training":
                tmp_ybot = [0, 0, 0]
                tmp_ytop = [300, 600, 1500]
        elif topology == "FC":
            if p == "evaluation":
                tmp_ybot = [0.4, 0.75, 2]
                tmp_ytop = [0.9, 1.75, 4]
            elif p == "decoding":
                tmp_ybot = [0.5, 0.5, 0.75]
                tmp_ytop = [1.25, 1.25, 1.75]
            elif p == "training":
                tmp_ybot = [0, 0, 0]
                tmp_ytop = [300, 700, 1500]
        elif topology == "LR":
            if p == "evaluation":
                tmp_ybot = [0.25, 0.5, 1.25]
                tmp_ytop = [1, 1.25, 2.25]
            elif p == "decoding":
                tmp_ybot = [0.5, 0.75, 1]
                tmp_ytop = [1.25, 1.5, 2]
            elif p == "training":
                tmp_ybot = [0, 0, 0]
                tmp_ytop = [400, 700, 1700]
        save_boxplot(data=speedup_list, title="Speedup: "+p, \
            topology=topology, suffix="speedup_"+p+"_boxplot", ybot=tmp_ybot, \
            ytop=tmp_ytop, ylabel="Speedup", \
            xlabel="State separation")

if __name__ == "__main__":
    # produce plots
    for topology in topologies:
        f = base_savename + topology + "_"
        plot_evaluation(topology)
        plot_decoding(topology)
        plot_training(topology)
        plot_speedup(topology)
    # join plots with corresponding boxplot
    files = listdir(folder)
    for f in files:
        if "boxplot" not in f and "MERGE" not in f:
            subprocess.call("convert "+folder+f+" "+folder+f[:-4]+ \
                "_boxplot.png +append " + \
                folder+"MERGE_"+f[5:], shell=True)
```

B.2.9 utilities_io.py

```
"""
Functions in this file:
def read_observations(f=observations_file, limit=0):
def read_observations_binary(f=bin_observations_file):
def read_path(f=state_path_file, limit=0):
def read_model(f=model_file):
def read_kmeans_model(f=kmeans_model_file):
"""
import numpy as np
from math import sqrt, log, exp

# common parameters
observations_file = "data/observations"
bin_observations_file = "data/bin_observations"
state_path_file = "data/path"
model_file = "data/model"
tests_folder = "tests/"

# Observations are a series of floats on a single line separated by spaces
def read_observations(f=observations_file, limit=0):
    in_file = open(f, "r")
    line = in_file.read()
    string_list = line.split()
    observations = []
    counter = 1
    for s in string_list:
        observations.append(float(s))
        counter = counter + 1
        if limit != 0 and counter > limit:
            break
    in_file.close()
    return observations

def write_observations(observations, f=observations_file):
    out_file = open(f, "w")
    obs_written = 1
    for o in observations:
        if obs_written == len(observations):
            out_file.write(str(o))
        else:
            out_file.write(str(o) + " ")
        obs_written = obs_written + 1
    out_file.close()

# Observations are the binary representation of a np.float64 array
def read_observations_binary(f=bin_observations_file):
    in_file = open(f, "rb")
    observations = np.fromfile(in_file, dtype=np.float64)
    return observations

def write_observations_binary(observations, f=bin_observations_file):
    out_file = open(f, "wb")
    np.array(observations, dtype=np.float64).tofile(out_file)
    out_file.close()

# Path is a series of ints on a single line separated by spaces
def read_path(f=state_path_file, limit=0):
    path_file = open(f, "r")
    line = path_file.read()
    string_list = line.split()
    path_list = []
    counter = 1
    for s in string_list:
        path_list.append(int(s))
        counter = counter + 1
        if limit != 0 and counter > limit:
            break
    path_file.close()
    return path_list

# state path param is from pomegranate model.sample()
def write_path(state_path, f=state_path_file):
    path_file = open(f, "w");
    for s in state_path:
        if s.name == ("None-start"): # initial silent state
            continue
        # strip the 's' from state name "s0", "s1", ...
        path_file.write(str(s.name)[-1:] + " ")
    path_file.close()

# The model is a series of numbers separated by spaces, following this format:
# n_states mean std_dev ... log(transitions) ... log(initial_distribution) ...
def read_model(f=model_file):
    in_file = open(f, "r")
    line = in_file.read()
    v = line.split()
    n_states = int(v[0])
    means = []
    std_devs = []
    offset = 1
    for i in range(0, n_states):
        means.append(float(v[offset + 2*i]))
        std_devs.append(float(v[offset + 2*i + 1]))
    transitions = []
    offset = 1 + n_states*2
    for i in range(0, n_states):
        for j in range(0, n_states):
            transitions.append(exp(float(v[offset])))
            offset = offset + 1
    initial = []
    for i in range(0, n_states):
        initial.append(exp(float(v[offset])))
        offset = offset + 1
    in_file.close()
    return n_states, means, std_devs, transitions, initial

# Write list to a file
def write_list(f, l):
    out_file = open(f, "w")
    for x in l:
        out_file.write(str(x)+" ")
    out_file.close()

# Read file that is space-separated into a list
def read_file_to_list(f):
    in_file = open(f, "r")
    list = in_file.read().split()
    in_file.close()
    return list
```

B.2.10 utilities_kmeans.py

```
"""
Functions in this file:
def read_observations(f=observations_file, limit=0):
def read_observations_binary(f=bin_observations_file):
def read_path(f=state_path_file, limit=0):
def read_model(f=model_file):
def read_kmeans_model(f=kmeans_model_file):
"""
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from math import sqrt, log, exp
import utilities_io as uio

kmeans_model_file = "data/kmeans_model"
kmeans_labels_file = "data/kmeans_labels"

def read_kmeans_model(f=kmeans_model_file):
    return uio.read_model(f)

def write_kmeans_model(centroids, std_devs, f=kmeans_model_file):
    out_file = open(f, "w")
    k = len(centroids)
    out_file.write(str(k) + " ") # number of states
    for a in range(0, k): # states, from kmeans
        out_file.write(str(centroids[a][0]) + " " + str(std_devs[a]) + " ")
    for a in range(0, k): # transitions, from uniform distribution
        for b in range(0, k):
            out_file.write(str(log(1/k)) + " ");
    for a in range(0, k): # initial distributions, from uniform distribution
        out_file.write(str(log(1/k)) + " ");
    out_file.close()

def read_kmeans_labels(f=kmeans_labels_file):
    in_file = open(f, "r")
    line = in_file.read()
    string_list = line.split()
    labels_list = []
    for s in string_list:
        labels_list.append(int(s))
    return labels_list

def write_kmeans_labels(labels, f=kmeans_labels_file):
    out_file = open(f, "w")
    for x in labels:
        out_file.write(str(x) + " ") # number of states
    out_file.close()

def kmeans(observations, k):
    df = pd.DataFrame(observations)
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(df)
    centroids = kmeans.cluster_centers_ # means
    # calculate the standard deviations
    o_count = 0
    std_devs = [0] * k
    state_counts = [0] * k
    for o in observations:
        i = kmeans.labels_[o_count] # state of the current observation
        std_devs[i] = std_devs[i] + (o - centroids[i])**2
        state_counts[i] = state_counts[i] + 1
        o_count = o_count + 1
    for a in range(0, k):
        std_devs[a] = sqrt(std_devs[a]/state_counts[a])
    labels = kmeans.labels_
    return centroids, std_devs, labels

def estimate_model(k, use_binary_file=True):
    if use_binary_file == True:
        observations = uio.read_observations_binary()
    else:
        observations = uio.read_observations()
    centroids, std_devs, labels = kmeans(observations, k)
    # sort states by increasing mean
    initial_order = []
    new_order = []
    for i in range(0, len(centroids)):
        initial_order.append(i)
        new_order.append(i)
    for i in range(0, len(centroids)):
        for j in range(1, len(centroids)-i):
            if centroids[j-1] > centroids[j]:
                tmp = new_order[j-1]
                new_order[j-1] = new_order[j]
                new_order[j] = tmp
    sorted_centroids = []
    sorted_devs = []
    for i in range(0, len(centroids)):
        sorted_centroids.append(centroids[new_order[i]])
        sorted_devs.append(std_devs[new_order[i]])
    sorted_labels = []
    for i in range(0, len(labels)):
        sorted_labels.append(new_order[labels[i]])

    write_kmeans_model(sorted_centroids, sorted_devs)
    write_kmeans_labels(sorted_labels)
```

B.2.11 viterbi_comparison.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import utilities_io as uio

# OPTIONS
verbose = False

def count_differences(p1, p2):
    difference_count = 0
    for i in range(0, len(p1)):
        if p1[i] != p2[i]:
            difference_count = difference_count + 1
            if verbose:
                print("index:",i,"p1:",p1[i],"p2:",p2[i])
    if verbose:
        print("Number of matches: ", len(p1)-difference_count)
        print("Number of differences: ", difference_count)
    return difference_count

def count_differences_uncompressed():
    state_path = uio.read_path()
    wahmm_path = uio.read_path("results/decoding_path")
    return count_differences(state_path, wahmm_path)

def count_differences_compressed():
    state_path = uio.read_path()
    compressed_path = uio.read_path("results/compressed_decoding_path")
    return count_differences(state_path, compressed_path)

if __name__ == "__main__":
    verbose = True
    # pm_path = uio.read_path("results/pm_decoding_path")
    wahmm_path = uio.read_path("results/decoding_path")
    compressed_path = uio.read_path("results/compressed_decoding_path")

    # print("--- PM decoding vs. WaHMM decoding ---")
    # count_differences(pm_path, wahmm_path)
    #
    # print("--- PM decoding vs. Compressed decoding ---")
    # count_differences(pm_path, compressed_path)

    print("--- WaHMM decoding vs. Compressed decoding ---")
    count_differences(wahmm_path, compressed_path)
```

B.2.12 WaHMM.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import subprocess
import sys

# Wrapper around compiled C++ WaHMM

arguments = sys.argv
arguments[0] = "bin/WaHMM"

#print("###START###")
subprocess.call(arguments)
#print("###END###")
```