Project in Bioinformatics

Bioinformatics Research Centre (BiRC)
AARHUS UNIVERSITY

Algorithmic Engineering of Hidden Markov Models

Carl M. Kobel

Student ID: 201404379

Thor B. Jakobsen

Student ID: 201404985

January 4, 2020

Summary

1 Introduction

Hidden Markov Models[1] are a group of powerful probabilistic models for modeling sequential data with a hidden underlying structure. They are widely used in the field of bioinformatics for gene finding and gene prediction. The reason for their popularity lies in their accuracy, robustness and simplicity. When analyzing larger quantities of data with an increasingly complex underlying structure, the efficiency of Hidden Markov Models is a limiting factor. This limiting factor is rooted in the algorithms used for evaluating, decoding and training the Hidden Markov Models. All algorithms except the Viterbi algorithm rely strongly on the efficiency of two main algorithms: the Forward and the Backward algorithms. In this report we propose a solution to the efficiency problem of Hidden Markov Models by optimizing the Forward and the Backward algorithms using linear algebra. Our solution is implemented in two libraries, one for the `C` programming language and one for the `Python` programming language utilizing the implementations in the `C` library.

The rest of this report is structured as follows: We will give an introduction to Hidden Markov Models and their associated algorithms in **section 2**. Then we will present our optimizations of the Forward and the Backward algorithms in **section 3**. Then we will present our library and its underlying architecture in **section 4**. Then we will present our results from conducting experiments with our library and see if we achieved a speed up compared to a conventional implementation in **section 5**. We will finish the report with a conclusion in **section 6**.

2 Hidden Markov Models

In the following we will introduce the Hidden Markov Model and the three basic problems of Hidden Markov Models: the evaluation problem, the decoding problem and the learning problem.

2.1 A Hidden Markov Model

A Hidden Markov Model (HMM) is a probabilistic model which in its simplest form consists of M observables:

$$O = \{o_1 \cdots o_M\}$$

N hidden states which are non observables:

$$H = \{h_1 \cdots h_N\}$$

N initial probabilities of starting in one of the N hidden states:

$$\pi = \{\pi_1 \cdots \pi_N\}$$

A $N \times N$ transition matrix φ where $\varphi_{i,j}$ is the probability of transitioning from state h_i to state h_j. A $N \times M$ emission matrix θ where $\theta_{i,j}$ is the probability to observe the i'th observable o_i at the state h_j.

2.2 Evaluation problem

The evaluation problem of an HMM is: Given an HMM λ and a sequence of observations Y, what is the probability of observing the sequence Y with respect to λ? This problem can be solved using the Forward and the Backward algorithms.

2.2.1 Forward algorithm

The Forward algorithm[2] given a sequence of observations Y of length K and an HMM λ creates an $N \times K$ matrix α. Here $\alpha_{i,j}$ is the probability of λ emitting the subsequence $Y_{1:i}$ of Y starting from the first observation, to the i'th observation and being in the j'th hidden state. α is calculated recursively in a forward manner starting from the first observation of Y. The initial step of the Forward algorithm is given by:

$$\alpha_{1,j} = \pi_j \cdot \theta_{Y_1,j}$$

The j'th index of the i'th observation of Y, Y_i, in α is defined as the following summation:

$$\alpha_{i,j} = \sum_{l=1}^{N} \theta_{Y_i,j} \cdot \varphi_{l,j} \cdot \alpha_{i-1,l}$$

The Forward algorithm does N operations for each index in α so the overall running time is $\mathcal{O}(N^2 \cdot K)$. It uses $\mathcal{O}(N \cdot K)$ space as it fills up the α matrix.

2.2.2 Backward algorithm

The Backward algorithm[2] given a sequence of observations Y of length K and an HMM λ, creates an $N \times K$ matrix β. Here $\beta_{i,j}$ is the probability of λ emitting the sub-sequence $Y_{i+1:K}$ starting in the j'th hidden state: $P(Y_{i+1:K}|X_i = x_j)$. β is calculated recursively in a backward manner starting with the last observation of Y where the first row of β has the value 1.

$$\beta_{i,j} = \sum_{l=1}^{N} \beta_{i+1,l} \cdot \varphi_{l,j} \cdot \theta_{Y_{i+1},j}$$

Like the forward algorithm, The Backward algorithm does N operations for each index in β so the overall running time is $\mathcal{O}(N^2 \cdot K)$. It uses $\mathcal{O}(N \cdot K)$ space as it fill up the β matrix.

2.3 Decoding problem

The decoding problem of an HMM is: Given an HMM λ and a sequence of observations Y, what is the most likely sequence of hidden states that produced Y with respect to λ?

2.3.1 Viterbi algorithm

The Viterbi algorithm[2] is a maximum-likelihood algorithm that, given an HMM λ and a sequence of observations Y of length K computes the most likely sequence of hidden states $X = x_1, \ldots, x_K$ that generates Y. The Viterbi algorithm fills out a $K \times N$ table V, where $V_{i,j}$ is the probability of the most likely path ending in the x_j at time j, having observed the subsequence $Y_{1:i}$.

$$V_{i,j} = \max_{x_{j-1}} P(Y_{1:i}, X_{j+1} = x_{j+1} | \lambda)$$

The most likely sequence of hidden states can then be calculated by backtracking through V starting in $\max(V_K)$.

Because the algorithm fills out a $K \times N$ table, where we for each cell have to iterate through all N states to find the transition with the highest probability, the running time is $\mathcal{O}(K \cdot N^2)$. The space consumption of the table V is $\mathcal{O}(K \cdot N)$.

2.3.2 Posterior decoding algorithm

The Posterior decoding algorithm[2], given an HMM λ and a sequence of observations Y of length K, calculates an array Z of size K which contains the highest a posteriori probability for each observation in Y. Z is calculated using the forward and backward probabilities. Here the j'th entrance in Z is given by:

$$Z_j = \max_{i \in H} \left(\frac{\alpha_{i,j} \cdot \beta_{i,j}}{P(Y|\lambda)} \right)$$

Because the Posterior decoding algorithm relies on the forward and backward probabilities, its time consumption for calculating these is $\mathcal{O}(N^2 \cdot K)$ and the space consumption for the α and β tables are $\mathcal{O}(N \cdot K)$. If we assume that both α and β are given, then the algorithm will use $\mathcal{O}(N \cdot K)$ time and $\mathcal{O}(K)$ space.

Posterior decoding and Viterbi is similar in that they, given a sequence of observables, both find a likely sequence of hidden states. The difference is that Viterbi finds the maximum likelihood sequence of hidden states over the full sequence Y, whereas Posterior decoding finds sequence of hidden states that gives the highest probability on the immediate observable y_i.

2.4 Learning problem

The learning problem is: Given an HMM λ and a sequence of observations Y, how should the φ, θ and π variables be adjusted so we maximize $P(Y|\lambda)$? This problem can be solved using the Baum-Welch algorithm.

2.4.1 Baum-Welch algorithm

The Baum-Welch [2] algorithm is an expectation maximization algorithm. Given a sequence of observations Y of length K and an HMM λ the Baum-Welch algorithm trains the φ, θ and π of the HMM so that the likelihood of $P(Y|\lambda)$ is a local maximum. This is done in an iterative procedure for a fixed amount of iterations or until the likelihood converges. In each iteration the algorithm updates the variables of the HMM using the forward and backward probability.
Because Baum-Welch uses the forward and backward probabilities it has to use the Forward and the Backward algorithms which both run in $\mathcal{O}(N^2 \cdot K)$ time. Updating the variables can be done with simple summations such that the overall running time is $\mathcal{O}(N^2 \cdot K)$ and uses $\mathcal{O}(N \cdot K)$ space.

3 Speeding up using matrix multiplications

Our main observation during this project is to what extend Baum-Welch and Posterior Decoding depend on the Forward and Backward algorithms. Looking closely at their structures they both run the Forward and the Backward algorithms, otherwise only doing one loop over the total length of the input without any computationally heavy operations. Our hypothesis is that if the efficiency of the Forward and Backward algorithms is increased, there will be an equal increase in the efficiency of the Baum-Welch and Posterior Decoding algorithms.

3.1 Linear algebra formulations of the Forward and Backward algorithms

The reason for formulating the Forward and the Backward algorithms using linear algebra is that it allows the implementation to take advantage of fast low level operations of the computer like SIMD.
Formulating the algorithms using linear algebra corresponds to generalizing the summations of the algorithms into matrix multiplications. While doing so we came up with a new matrix for replacing the emission and transition matrix in the context of the Forward and the Backward algorithms called the emission-transition matrix (ETM). In the following section we go through this generalization and show how the ETM is derived.

3.1.1 Forward algorithm

The following is the definition of $\alpha_{i,j}$ as described in **section 2.2.1**:

$$\alpha_{i,j} = \sum_{l=1}^{N} \theta_{Y_i,j} \cdot \varphi_{l,j} \cdot \alpha_{i-1,l} \tag{1}$$

Instead of having a definition for each index of α we made a formulation for each column of α. Let Θ_{Y_i} denote the diagonal matrix made from the i'th row in θ. We use that $\Theta_{Y_i} \cdot \varphi$ results in a matrix where each row contains the same values as each iteration of the corresponding summation from **(1)**. Using this observation we can now define the i'th column of α as:

$$\alpha_i = \Theta_{Y_i} \cdot \varphi \cdot \alpha_{i-1} \tag{2}$$

We notice that we in equation **(2)** do a matrix-matrix multiplication, $\Theta_{Y_i} \cdot \varphi$, for each observation in the sequence Y. This can be avoided by introducing the ETM as $\hat{\Theta}_{Y_i} = \Theta_{Y_i} \cdot \varphi$. We avoid the matrix matrix multiplication because we can compute all the M ETMs as a preprocessing step so that for any Y of length K where $K > M$ the ETMs reduce the amount of matrix matrix multiplications compared to **(2)**. Using the definition of the ETM we can now define the i'th column of α as:

$$\alpha_i = \hat{\Theta}_{Y_i} \cdot \alpha_{i-1} \tag{3}$$

It is not possible to use the ETM in the initial step of the forward algorithm but it can still be described using linear algebra using the Θ_{Y_1} from **(2)**:

$$\alpha_1 = \pi \cdot \Theta_{Y_1}$$

3.1.2 Backward algorithm

The initial step of the Backward algorithm cannot be meaningfully described with linear algebra since no operations are made. However the i'th step can and we will introduce the notation of ETM for the i'th step of the backward algorithm. The i,j'th index of β is defined as:

$$\beta_{i,j} = \sum_{l=1}^{N} \beta_{i+1,l} \cdot \varphi_{j,l} \cdot \theta_{Y_{i+1},j}$$

We now introduce $\Theta_{Y_{i+1}}$ from **(2)** and write i'th column of β as:

$$\beta_i = \beta_{i+1} \cdot \varphi \cdot \Theta_{Y_{i+1}}$$

Finally we introduce the ETM into the definition of the i'th column of β:

$$\beta_i = \beta_{i+1} \cdot \hat{\Theta}_{Y_{i+1}} \tag{4}$$

3.1.3 Increased space consumption

The ETM matrix has the dimension $N \times N$ and since there are M of them, we get a space consumption increase of $\mathcal{O}(M \cdot N^2)$.

3.1.4 Sparse matrix

Because we introduced the linear algebra formulation of the Forward and the Backward algorithms the instances where the emission or the transition matrix is sparse should be addressed. A sparse matrix is a matrix where most of the elements are zero, which means doing vector matrix multiplication most of the multiplications will have the form:

$$A \cdot x = a_{i,1} \cdot x_1 \cdots a_{i,n} \cdot x_n = 0 \cdot x_1 \cdots 0 \cdot x_n$$

It is clear that these multiplications are trivial. Our hypothesis is that in these instances it is possible to achieve an even greater speed up than expected for the straight forward linear algebra implementation. First of all we see that the ETMs are sparse when the transition matrix or the emission matrix is sparse. This means only one of the matrices have to be sparse before we consider our instance to be sparse. It is possible to represent a sparse matrix on a format where all zero values are omitted. If such a format allows for vector matrix multiplications it can be used for representing the ETM. There exist many different sparse matrix formats that support matrix vector multiplication. We choose to focus on the Compressed Sparse Row (CSR) format and the Recursive Sparse Block (RSB) format.

3.2 Implementation

In the following section we will go though the main points of our implementations of the Forward and the Backward algorithms with respect to the dense and sparse instance of the ETM.

3.2.1 BLAS

BLAS[3] stands for basic linear algebra subprograms. It is a specification that describes a set of routines for performing basic linear algebra operations. BLAS is typically implemented so that it is optimized for specific hardware. The major vendors like AMD[5] and Intel[6] have their own implementation optimized for their own CPUs. BLAS is the de facto specification for linear algebra operations which is the reason why we choose it for implementing our linear algebra formulation of the Forward and the Backward algorithms. We chose the ATLAS[9] distribution which complies so that it is optimized for the hardware it is compiled on.

The following code snippet performs the calculation of the i'th column of α as described in **equation (3)** using BLAS. In this step only one matrix vector multiplication is necessary. This is done by using the `cblas_dgemv` function. `cblas_dgemv` computes `y := alpha*A*x + beta*y` or where `A` is transposed depending on the input. In the following code we perform the transposed version. To achieve just the `y := A*x` part we set `alpha=1`, `beta=0` and `y` is a pointer to the beginning of the i'th column of α. We transpose the `A` (ETM) because it is stored in transpose order in the implementation and transposing it again moves it back.

```
cblas_dgemv(CblasRowMajor, CblasTrans, hmm->hiddenStates, hmm->hiddenStates, 1.0,
    new_emission_probs[Y_i], hmm->hiddenStates, alpha+hmm->hiddenStates*(i-1), 1, 0,
    alpha+hmm->hiddenStates*i, 1);
```

The following code snippet performs the calculation of the i'th column of β using BLAS. Here we do the same as previous code but where we use the first, untransposed, version. Because we want to perform a vector matrix multiplication as denoted in **(4)**, but `cblas_dgemv` only allows us to do a matrix vector multiplication, we benefit form the ETM being quadratic. By transposing the ETM and then doing a matrix vector multiplication makes it the same as doing a vector matrix multiplication. As the ETM is transposed by default we do nothing.

```
cblas_dgemv(CblasRowMajor, CblasNoTrans, hmm->hiddenStates, hmm->hiddenStates, 1.0,
    new_emission_probs[Y[T-i]], hmm->hiddenStates, beta+hmm->hiddenStates*T-i*hmm->
    hiddenStates, 1, 0, beta+hmm->hiddenStates*T-i*hmm->hiddenStates-hmm->hiddenStates
    , 1);
```

We can also calculate the ETM's using BLAS which we did.

3.2.2 CSR

The CSR format represent a sparse matrix using three arrays. One that contains all the non zero values of the matrix and two arrays for keeping track of the value indices in the original matrix. The format allows for vector matrix multiplication. The implementation utilizing the CSR format has a larger preprocessing as we have to calculate the ETMs and then turn them into the CSR format. The matrix vector multiplication using the CSR format consists of two nested loops that goes through the two index arrays. The amount of iterations in the loops and therefore the amount of multiplications are bounded by the amount of non-zero values in the ETM. The implementation uses BLAS for calculating the ETM's.

3.2.3 RSB

There is no straight forward implementation of sparse matrix vector multiplication utilizing low level operations of the computer like there is with BLAS. Instead we found libRSB[8], a library that implements the sparse BLAS specifications, which is a subset of the original BLAS operations for sparse matrix operations. The implementation takes advantage of the recursive sparse block matrix format that allows for cache efficient and multi threaded matrix operations. We use the libRSB as shown in the code snippet in **section (3.2.1)** the corresponding libRSB function is called `rsb_spmv`. The `rsb_spmv` function takes the same parameters however the input matrix must be in CSR format. Therefore the implementations of the Forward and the Backward algorithms for the sparse instance of an HMM utilizing RSB has the same preprocessing step as the CSR version described in **section (3.2.2)**. The implementation uses BLAS for calculating the ETM's.

3.3 Scaling

In order to avoid the numerical problems associated with multiplying probabilities (or numbers smaller than 1) we have to scale the results for each iteration of the Forward and the Backward algorithms. For the Forward algorithm we divide each cell in the current column with its sum. The sum was then saved in the `scaleFactor` variable to be used for scaling the Backward algorithm. To obtain compatible scaling, we use the same scale factor for the Forward algorithm when scaling the Backward algorithm, as in: we divide each cell with the sum of the corresponding column of the result of the Forward algorithm.

The scaling step can be done in two lines when using BLAS, which we did for both the BLAS, CSR and RSB implementations. The conventional implementation of the Forward and Backward algorithms is scaled using a single loop.

4 HMMM (Hidden Markov Model Matrix-*edition*)

In the following we will go through our main software architectural considerations about our `C` implementation of the HMM and its algorithms as well for the associated `Python` library. Both the `C` library and the `Python` library can be found on the Hidden Markov Model Matrix-edition github at: `https://github.com/Thornado-Carlkoder/hmmmlib`

4.1 C implementation

Our `C` implementation of the HMM and its associated algorithms are strongly influenced by our experimental implementations of the Forward and the Backward algorithms. We wanted our implementation to facilitate easy switches between implementations of the Forward and the Backward algorithms as well as ensuring that these implementations could be compared on the same basis.
To encapsulate the HMM we made a `struct` which contains the variables of the HMM as described in **section (2)** as well as the size of its variables. The struct is shown on **figure 1**.

```
1   struct HMM {
2       unsigned int hiddenStates;
3       unsigned int observations;
4       double * transitionProbs;
5       double * emissionProbs;
6       double * initProbs;
7       void (*forward)(struct HMM *hmm, const int *Y, const int T, double *
    scalingFactor, double * alpha);
8       void (*backward)(struct HMM *hmm, const int *Y, const int T, double *
    scalingFactor, double * beta);
9   };
```

Figure 1: Struct of the HMM in the `C` implementation

All the algorithms are implemented so the user is in charge of allocating and deallocating all the output variables. For each algorithm, a pointer to the output must be given as a parameter. This is done to give the user as much control as possible. To allow an easy way of changing implementations of the Forward and the Backward algorithms, and to avoid multiple implementations of the Baum-Welch and the Posterior decoding algorithms, we made an interface for both the Forward and the Backward algorithms. These are added to our HMM struct as two function pointers and are the last two variables of **figure 1**. This allows us to call the two algorithms through the HMM instance which is a required argument in the Baum-Welch and the Posterior decoding algorithms. This allows these algorithms to call the Forward algorithm like this:

```
1 F(hmm, Y, T, scaleFactor, alpha);
```

The Backward algorithm can be called in the same manner. This design made it very easy to construct a new version of the Forward and the Backward algorithms since they just need to follow the interface. To simplify this even more we made HMM constructors for the different versions of the

Forward and Backward algorithms so the different constructors set the function pointers to the wanted implementations of the Forward and the Backward algorithms.

The following code snippet is a usage example of the `C` library:

```
1   HMM * hmm = HMMBLAS(7, 4);
2   double transitionProbs[7][7] = {
3    {0.0, 0.0, 0.9, 0.1, 0.0, 0.0, 0.0},
4                         ...
5    {0.0, 0.0, 0.05, 0.9, 0.0, 0.05, 0.0},
6   };
7   double emissionProbs[4][4] = {
8    {0.3, 0.25, 0.25, 0.2},
9                ...
10   {0.25, 0.25, 0.25, 0.25},
11  };
12  double initProbs[7] = {0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0};
13  int input[1000] = {0, 2, 1, 1, ..., 3, 2, 2};
14  double * alpha = malloc(1000*hmm->hiddenStates*sizeof(double));
15  double * scaleFactor = malloc(1000*sizeof(double));
16  F(hmm, input, 1000, &scaleFactor, &alpha);
17  HMMDeallocate(hmm);
```

Figure 2: In the `C` library example we initialize the BLAS version of the HMM with 7 hidden states and a alphabet of size 4. We set its transition, emission and initial probabilities. When these a set the HMM instance is ready to be used. From line 13 to 16 we make a alpha pointer and a scale factor pointer and then call the Forward function of the HMM where the result is saved to the alpha pointer. At line 17 we end our example by deallocating the HMM struct and its pointers.

4.1.1 Implementation of Baum-Welch, Posterior decoding and Viterbi

To get a complete library with the HMM and it's associated algorithms and being able to test our hypothesis we implemented the Baum-Welch, Posterior decoding and Viterbi algorithms in a conventional manner without doing any optimization.

The Baum-Welch and Posterior Decoding algorithms both take advantage of the Forward and Backward algorithms. Thus they were implemented using the interface described in the previous section for calling the Forward and the Backward algorithms.

The Viterbi algorithm is scaled using logarithm in order to avoid numerical problems. We tested our

implementation of the Viterbi algorithm to validate that it has the expected running time, which it has. The result of the experiment can be seen on **figure 9** in the Appendix.
We also tested all the algorithms scale accordingly with respect to the state space, which they do. The results of theses experiments are shown on **figure 8** in the Appendix.

4.2 Python library

To make the HMM-library easily accessible, in order to quickly write tests, we decided to write a `Python` library which makes it possible to call all core methods and algorithms in the `C`-library.
The `Python` library is constructed by compiling the `C` library into a shared object which is then loaded into `Python` using the ctypes foreign function library[4] for `Python`. This is done in the `Python` file `HMMM.py`. In the main file of the `Python` library, the HMM-struct and all of it's functions and algorithms are defined. By importing this library, all the functions and algorithms in the `C`-library will be accessible in `Python`. Each time a new function or algorithm is added to the `C`-library, a corresponding definition must be made in `HMMM.py`.
In the `Python` library, we implemented the `__del__(self)` data model method[7], so when the reference count of the `Python` HMMM-class object reaches zero, all dynamically allocated memory in the `C`-library is deallocated. When using the python library no manual memory management is needed.
The following is an example of how to use the `Python` library:

```
1 from HMMM import *
2 hmmm = HMMM(3, 2, hmmType = "BLAS")
3 hmmm.set_random()
4 hmmm.setInitProbs([1,0,0])
5 hmmm.baumWelch(observations = [1,0,1,0,1,0,1,...,1,0,1,1,0,1,0,1], n_iterations = 5)
```

Figure 3: Example of the `Python` library. Setting up an HMM with 3 hidden states and an alphabet of size 2 using the BLAS-implementation. At line 3 the initialization, emission and transition matrices are set to random values. At line 4 the initialization probabilities is set to a specific list of values. At line 5 the Baum-Welch algorithm is used to train the transition and emission matrices using given data over 5 iterations.

4.2.1 Running time tests

We performed running time tests on all algorithms using the `Python` library. All the tests were made using uniform randomly generated strings made, over the alphabet, as input. We used the same input for every version of the different algorithms.

The advantage of using the `Python` library is that it is very easy to modify the tests and automate them. The disadvantage is that it introduces a level of abstraction, thereby possibly adding noise to the time measurements.

5 Experiments

To test or hypotheses we have conducted several experiments on the different implementations of the Forward and the Backward algorithms. All the experiments have been conducted on a ThinkPad W540 with 32GB RAM and an Intel I7-4700MQ running 64bit Debian 10. The `C` library was written in `C` version 11 and compiled using `gcc` version 8.3.0-6. We used `Python` version 3.7.3. All the experiments have 3 replicates in order to minimize random fluctuations due to other processes of the computer. All experiments were conducted through the `Python` library as described in **section 4.2**.

5.1 BLAS vs Conventional

To test our initial hypothesis, that a linear algebra implementation utilizing BLAS of the Forward and the Backward algorithms is faster than the conventional version, we conducted the following experiment. The experiment was conducted with an varying input size starting with $100'000$ going up to $1'000'000$, increasing $100'000$ each time. The state space was varied as well: from 10 to 400. Varying the state space allows us to test the consistency of the performance over all the influencing variables. The experiment is show in **figure 4**.

Figure 4 clearly shows that the BLAS version is faster than the conventional implementation in all experiments. It is also clear that the efficiency ratio between the BLAS version and the conventional implementation increases with the size of the state space. This supports our initial hypothesis.

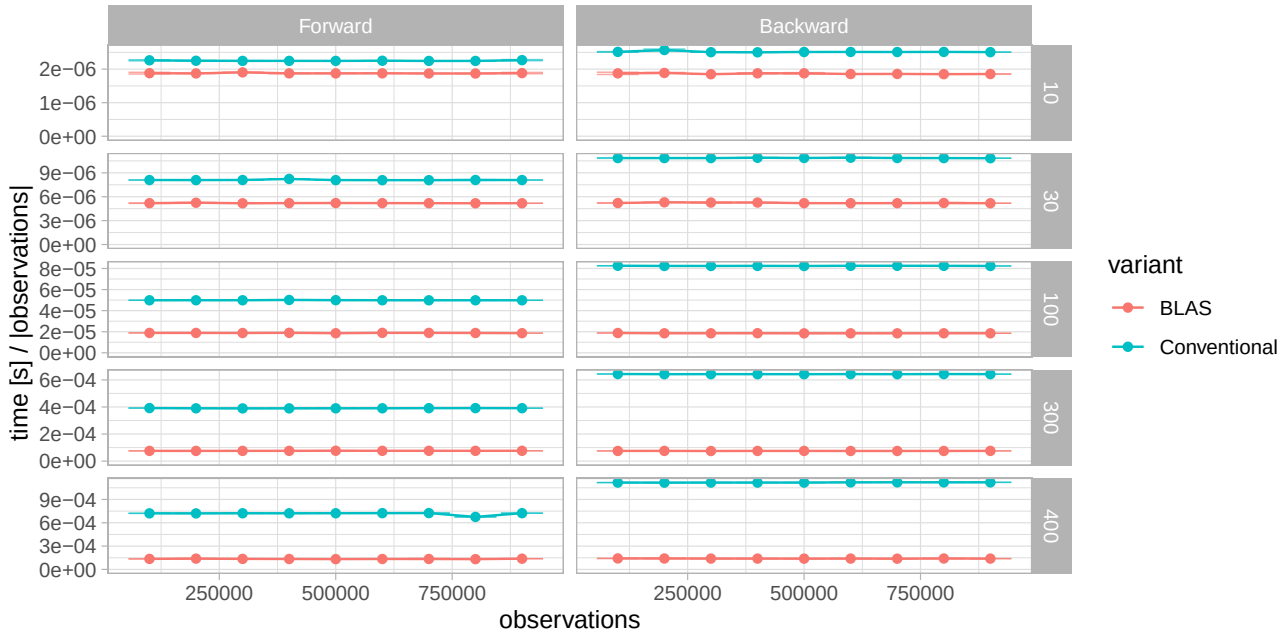

Figure 4: Forward and Backward running time experiment for the linear algebra and the conventional implementations. The vertical axis shows time divided by number of observations. Error bars denote standard deviation of 3 replicates. Alphabet size: 4.

To verify that the algorithms scale with their theoretical running times the measured time is divided with the input size on **figure 4**. It is clear that the algorithms follow their theoretical running times for both implementations.

Figure 4 implies that the BLAS implementation is faster than the conventional implementation supporting our hypothesis. This is especially clear when the statespace increases.

5.1.1 Baum-Welch and Posterior decoding

In the following experiment we want to test our hypothesis, that if we increase the speed of the Forward and the Backward algorithms we will see an equal speed up in the Baum-Welch and the Posterior decoding algorithms. **Figure 5** shows the experiment running the Baum-Welch and the Posterior decoding algorithms using the conventional and BLAS version of the Forward and the Backward algorithms.

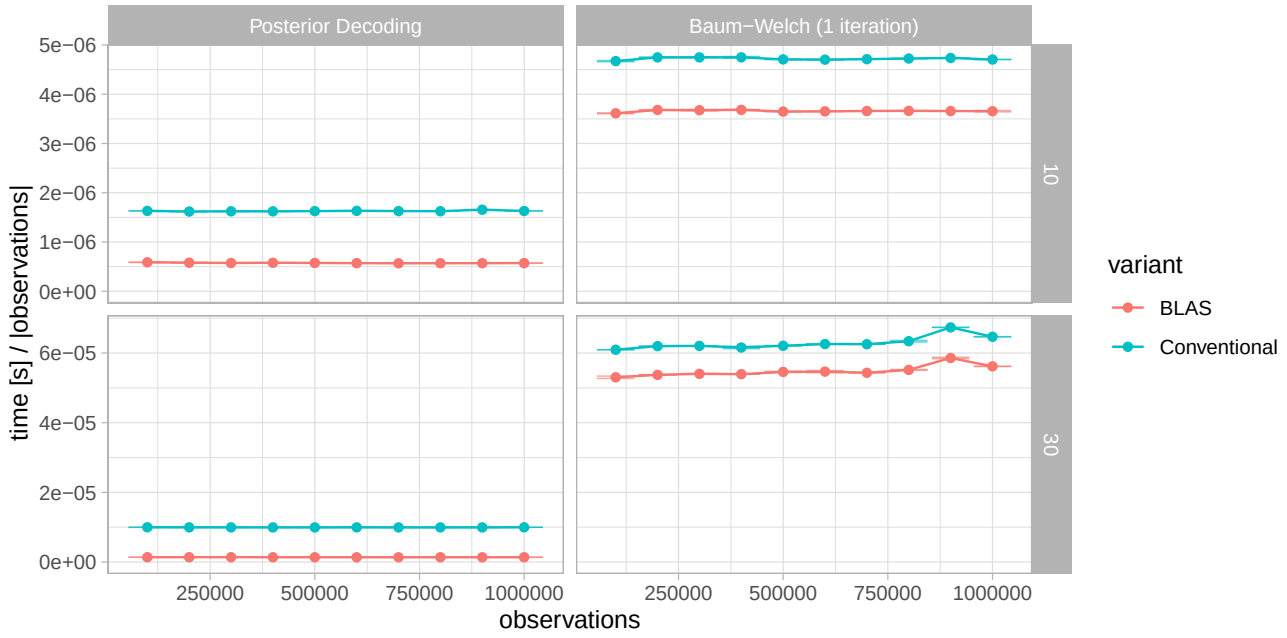

Figure 5: Posterior decoding and Baum-Welch running time experiment with the BLAS version and the conventional implementation. The vertical axis shows time divided by number of observations. Error bars denote standard deviation of 3 replicates. Alphabet size: 4.

To verify that the algorithms scale with their theoretical running times, we have divided the measured time with the input size on **figure 5**. It is clear that the algorithms follow their theoretical running times for both algorithms.

Figure 5 clearly shows that there is an equivalent increase in speed for both the Baum-Welch and Posterior decoding algorithms, supporting our hypothesis.

5.2 Sparse instance of HMM

To test our hypothesis for the sparse instance of an HMM, we implemented versions of the Forward and the Backward algorithm utilizing the CSR format and the RSB format as described in **section 3.2**. In the following two experiments we tested the two implementations against each other, and in the last experiment we tested the fastest of these (RSB) against the BLAS implementation.

We define density as a value between 0 and 1, where a density of 0 signifies the most sparse valid transition matrix where only one value per row is non-zero. When the density goes towards 1, the number of edges in the matrix goes from N to N^2.

All the experiments are conducted with an input size of $100'000$ observables, whilst varying the density of the transition matrix and the state space. For all the experiments we use a dense emission matrix

with 4 observables.

5.2.1 RSB vs. CSR

In the following experiment we want to compare the RSB and CSR implementations of the Forward and the Backward algorithms. The results of this experiment are shown on **figure 6**.

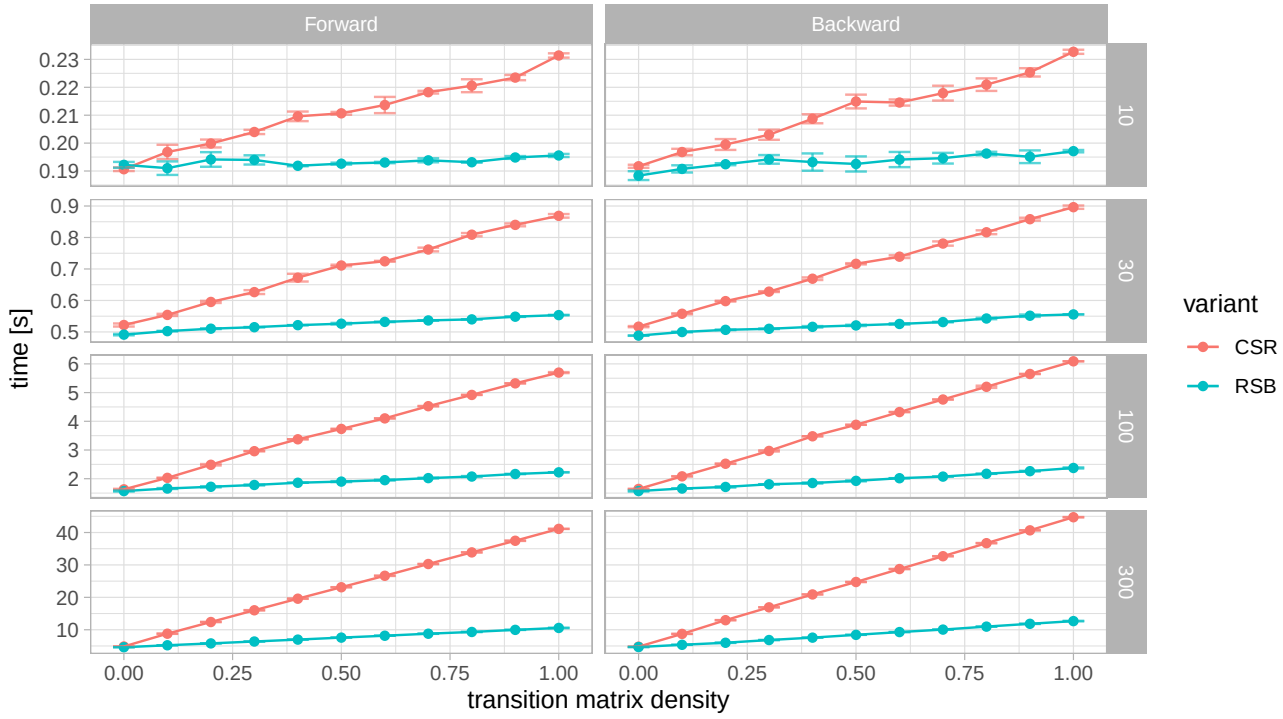

Figure 6: Comparison of the two sparse implementations: CSR and RSB. Forward and Backward algorithms with a fixed input of size 100′000 and varying state space and density. Alphabet size of 4.

From **figure 6** it is clear that the RSB implementation is faster than the CSR implementation for all combinations of state-space size and density.

5.2.2 BLAS vs RSB

In the following experiment we want to test our hypothesis, that the implementations of the Forward and the Backward algorithms optimized for sparse instance of an HMM utilizing libRSB yields a speed up compared to our density-ignorant BLAS version when the transition matrix is sparse.

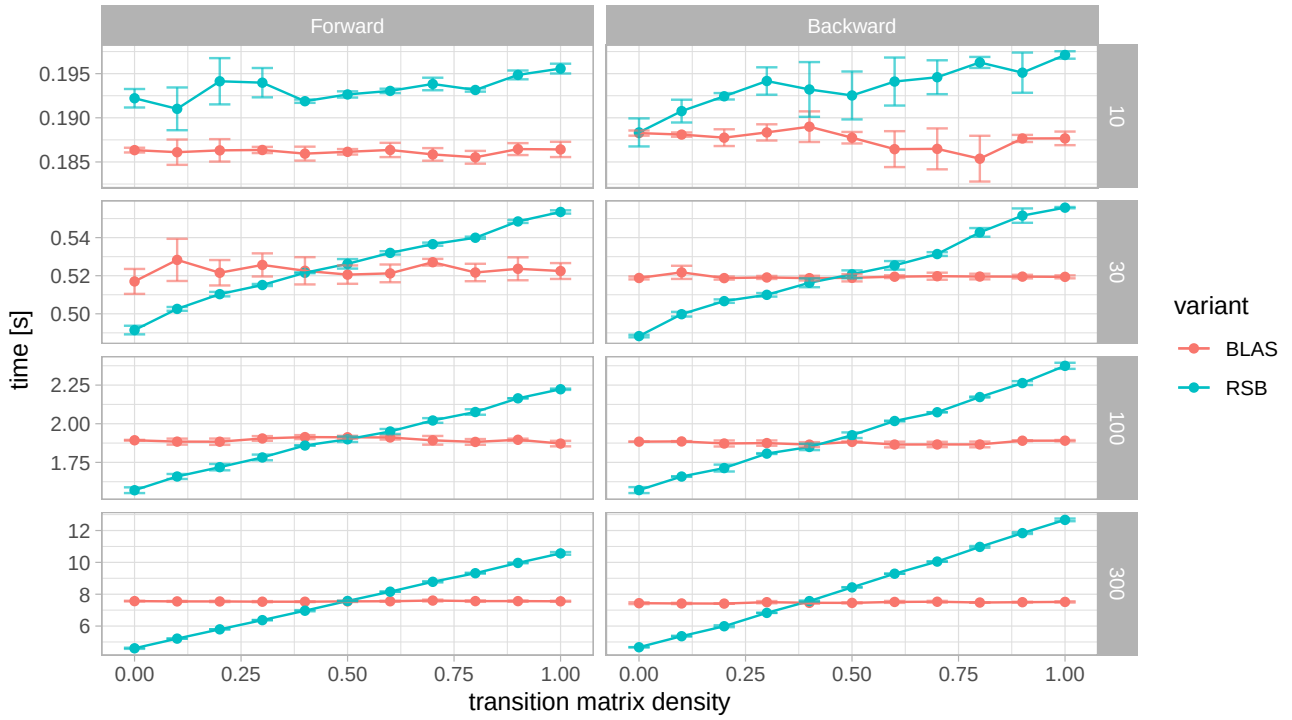

Figure 7: Experiment of the BLAS and RSB implementations of the Forward and Backward algorithms with a fixed input of size 100′000 and varying state space and density. Alphabet size of 4.

Figure 7 shows that for the state spaces in the range of 30 and up, the RSB implementation has a speed up relative to the BLAS version when the density is below 0.375. This speed up goes for both the Forward and the Backward algorithms. It seems like the advantage of the Forward algorithm is stabilizing at a density of 0.5 and for the Backward algorithm it is stabilizing at 0.375.
This supports or hypothesis, that it is possible to achieve a speed up for the sparse instances of the HMM when utilizing the sparseness. The increase in speed should also be present in the Baum-Welch and Posterior decoding algorithms as implied in the results of **section 5.1.1**.

5.3 Conclusion on experiments

Throughout all of our experiments we have managed to support our hypotheses, that utilizing linear algebra increases the efficiency of the Forward and Backward algorithms in **section 5.1** and that this increase is present in the Baum-Welch and Posterior Decoding algorithms in **section 5.1.1**. We have also shown, experimentally, that utilizing the sparse implementation of matrix multiplications leads to a further increase in speed for sparse instances of the HMM in **section 5.2.2**.

6 Conclusion

In this project report we have made a linear algebra formulation of the Forward and Backward algorithms with the hypothesis that this would increase the efficiency of these to algorithms, as well as the Baum-Welch and the Posterior decoding algorithms when implemented. We have implemented these algorithms in a conventional manner and the linear algebra formulation using BLAS. With the latter we have conducted experiments that supports our hypothesis. We have also made an implementation of the Forward and the Backward algorithms which takes advantage of the sparse instances of an HMM using RSB. We have made experiments with results supporting our hypothesis that the sparse instances of an HMM allows for an even greater speed up. From this project report and our experiments we can conclude that the linear algebra implementation of the Forward and Backward algorithms yields a significant increase in speed not only for the latter two but for all the algorithms, that uses them as an subroutine. We can also conclude that an even greater increase in speed can be achieved when the instance of the HMM is sparse.

References

[1] Leonard E. Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *Ann. Math. Statist.*, 37(6):1554–1563, 12 1966.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[3] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[4] ctypes Manual. https://docs.python.org/3/library/ctypes.html, 2019.

[5] AMD BLAS Library. https://developer.amd.com/amd-aocl/blas-library/, 2019.

[6] Intel® Math Kernel Library. https://software.intel.com/mkl, 2019.

[7] Python 3 Manual. https://docs.python.org/3/reference/, 2019.

[8] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations. *Online*, pages 300–305, 2010.

[9] R. Clint Whaley. *ATLAS (Automatically Tuned Linear Algebra Software)*, pages 95–101. Springer US, Boston, MA, 2011.

A Appendix

Overvej

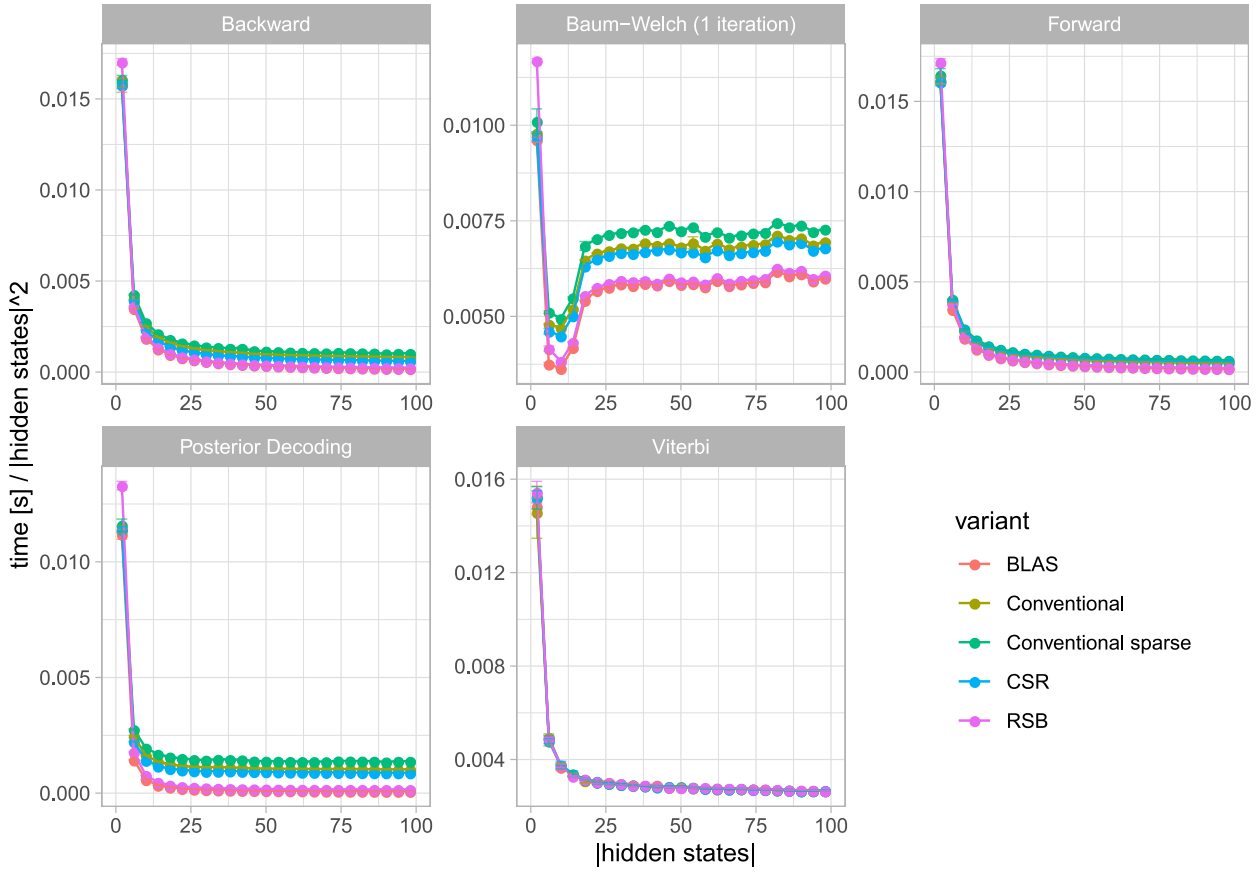

Figure 8: Running time of increasing number of hidden states for all algorithms and all implementations. Time measurement is averaged over 3 replicates. Error bars indicate standard deviation. Alphabet size = 4, input size = 100'000.

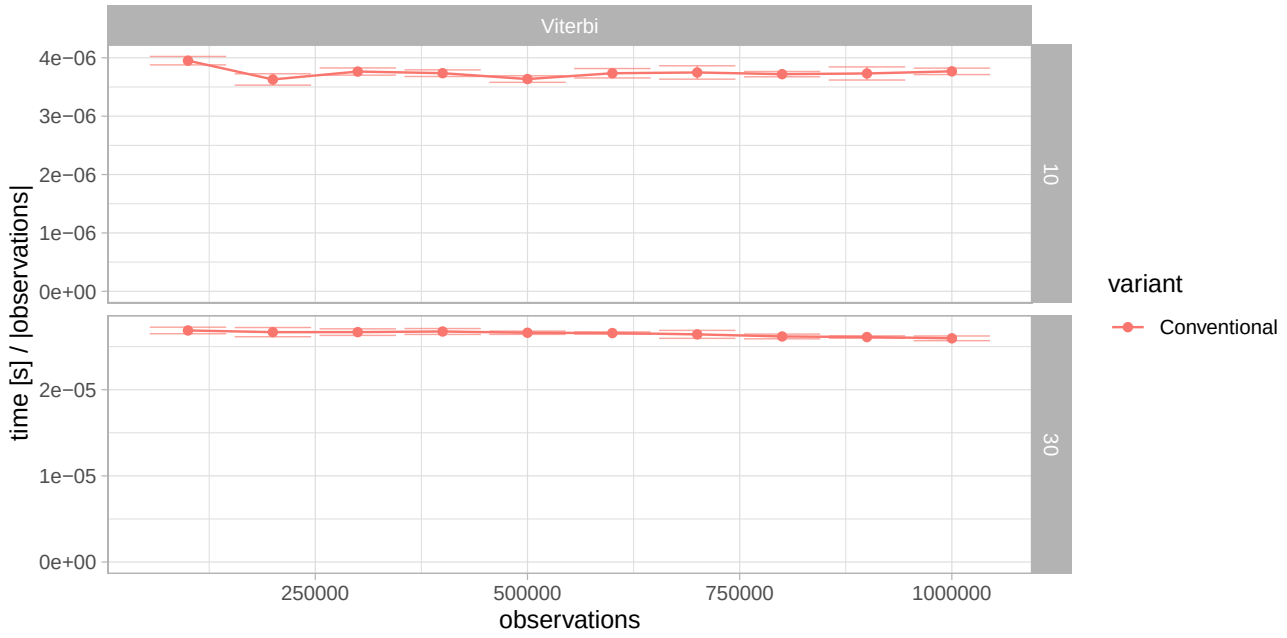

Figure 9: Running time of increasing number of observations in the Viterbi algorithm. For 10 and 30 hidden states. Time measurement is averaged over 3 replicates. Error bars indicate standard deviation. Alphabet size = 4.