A comparison of algorithms for determining gene structure

Viterbi and Posterior decoding

Adrián Gómez Repollés

May, 2019

Abstract

The increase in the amount of genetic data has derived in the necessity of using reliable approaches to extract all possible information in a reduced amount of time. The reduction of the processing time could be reached by the use of computers, more exactly, algorithms. Such an example of the extraction of information from genetic data is to estimate the gene structure hide in the genetic sequence by the use of a gene finder. This project compares the different decoding methods implemented in gene finders trying to determine which of them would return a better solution. The comparisons of the decoding methods are done by estimating how good the prediction is in relation to the original gene structure. The results are divided into untrained and trained models, showing better performance for Viterbi decoding in the first model whereas it does not show a preference decoding algorithm for the second model. Finally, taking into account several factors, Viterbi decoding seems to be a better solution for the specific problem that it is trying to be solved.

1 Introduction

During the last decades, the development of technology has promoted the parallel advance of different fields of biology, in addition to other sciences. An example of this is the improvement in sequencing techniques. This has allowed increasing the fidelity and size of the sequences obtained from the DNA / RNA extracted from the different species. At the same time, the amount of information collected has grown exponentially, surpassing the natural capacity of the human being to handle the data, Kulp. D and associates in 1996 and Korf I. in 2004 commented on this increase. Therefore, the implementation of new methods of data management has been necessary. Along with the advancement of technology, the improvement of computer systems has developed rapidly, generating the most effective solution for this problem. In the same way, it has opened a range of possibilities for the study of genetic material and the development of methods to infer the relationship between the observed and its function.

If we simplify the DNA to a sequence where there are regions that encode proteins and regions that do not encode proteins since these proteins are responsible for the functioning of the organism, the interest of biology would be to know precisely where these zones are located. There are different methods to identify these regions, however, in this project, we will use a 'gene finder'. The main idea of a gene finder is to find the structure of the genome through the use of probabilities. The estimation of the genetic structure must take into account different assumptions, the more flexible the assumptions, the more complex the model is. Therefore, there is a balance between the complexity of the model, the computational cost and the veracity of the estimate.

Genetic data, more exactly DNA, is basically represented by four different nucleotides (A, C, G and T) as a pair of sequences of variable length. Each sequence would be read independently and the genes inside of the sequence will be translated into proteins. A genetic sequence could be seen as a group of observations that are independent of each other but this would break the true relationship between the nucleotides. However, the natural complexity of the sequence is unknown and could be computational overwhelming trying to define an explanatory model with that many parameters, to infer estimations from genetic data we need to make some assumptions. For example, each nucleotide of the sequence is independent to the rest except for the one before. We expect that recent observations are likely to be more informative than more historical observations in predicting future values (Bishop, 2006). This type of sequence is known as a *first-order Markov chain* and it could be used to make predictions, such as gene structure.

In this project, we are going to use two different genes finders to infer the gene structure of two genomes using two implementations of Hidden Markov Models (HMM as its contracted form). The difference between the HMM is the algorithm implemented, which will approach in a different way the optimization of the gene structure. Due to the fact of this contrast, we are going to compare how well they predict by comparing with the original gene structure of the genomes. Finally, we will train the model and repeat the comparison to determine how the model affects the accuracy of the algorithm.

2 Methods

2.1 Genetic data

For our studies we used two different complete genomes, both were given. The first one is from *Streptococcus pyogenes* with a length of 1.8Mb and the second one is from *Streptococcus agalactiae* with a length of 2.2Mb. The genomes were in FASTA format so we extracted the data using a small parser coded in Python. Also, we used the real pattern of genes coded by each genome to compare with the output we got from the implemented gene finder and calculated its accuracy, these files are sequences made of three different state characters:

C: Coding

N: Non Coding

R: Reverse Coding

Once we had both data, we translate it to numbers because it makes it easier for us to implement the code. For the genome, we change each nucleotide for a number between 0 and 3 given the alphabetical order of the nucleotide. Also, we got the output of the decoding using indexes, so is made out of numbers, that we need to translate into the three original latent states to be able to compare with the true annotation of genes.

2.2 Hidden Markov Model

2.2.1 Annotation

Let N be the length of observations in the sequence and K the number of states of the model, where observations are defined by X and hidden states by Z. Further, let n be the n^{th} observation and k the k^{th} state of a given point of the computed matrix. Moreover, let M be the total number of observation types.

2.2.2 Model

Hidden Markov Models consist of a sequence of observations $(X_1, X_2, \ldots, X_N)$ and a sequence of discrete hidden states $(Z_1, Z_2, \ldots, Z_N)$. Also, Hidden Markov model consists of a model defined by three different sets of probabilities, we can define the model as:

$$\theta = (\pi, A, \phi) \tag{1}$$

Given the sequence of observations and the sequence of states, all three probabilities are calculated using the counting method. Once the different possible outcomes have been calculated the result values are normalized. This approach is also used to train the model to pursue finding a better explanatory model.

For this project, we used 2 different models depending on the number of states, both of them were given:

First, the model that explains the simplest approach for the type of data that we got from the genomes. Due to the fact that the true pattern of genes was made of 3 different characters, we implemented one state for each character. Also, the states coding and reverse coding can only be reached going through the non-coding state so the genes can not be overlapped.

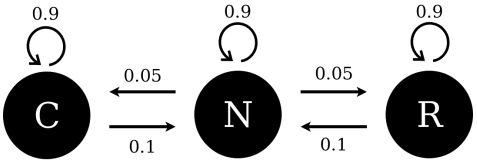

Figure 1: Three states model

Second, a slightly complex model where coding and reverse-coding regions should be at least 3 nucleotides long. In that case, we have 3 different states for both coding or reverse-coding, plus an extra state for non-coding. In total, a 7 states model. However, this model does not permit state coding and reverse to be reached without going through the non-coding state, making the genes not to overlap in the sequence.

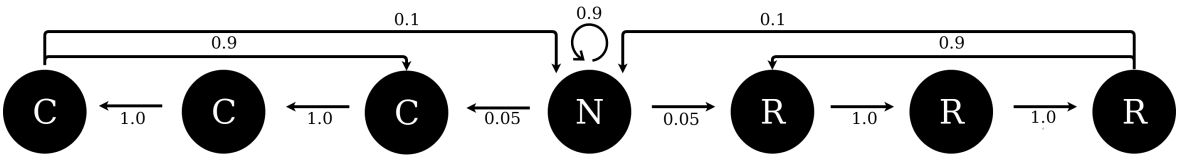

Figure 2: Seven states model

Even though these two models seem to be simplistic the approach is not meaningful and the computational cost is low. The more states the models have the more complex they are and the tougher they are to implement and execute. However, the increase in states could lead to better maintenance of the reading frame (David Kulp, 1996)

2.3 Probabilities

The Hidden Markov model contains three different probabilities:

Initial probabilities: It is a vector of size K. It defines how probable is the first observation likely to be seen. It is represented by the letter π.

$$\pi = P(X_1|Z_k) \tag{2}$$

Transition probabilities: It is a matrix of size K x K. It defines how likely is to change from one state to another. It is represented by the letter A.

$$A = P(Z_{n+1}|Z_n) \tag{3}$$

Emission probabilities: It is a matrix of size M x K. It defines how likely it is to emit an observation given the state. It is represented by the letter ϕ.

$$\phi = P(X_n|Z_k) \tag{4}$$

2.4 Decoding algorithms

Assuming a *first-order Markov chain*, the observations of the sequences can be explained by the joint distribution:

$$P(X_1, ..., X_N) = P(X_1) \prod_{n=2}^{N} P(X_n|X_{n-1}) \tag{5}$$

Moreover, if we want to build a model for sequences that are not limited by the Markov assumptions we can use a limited number of free parameters (Bishop, 2006). The parameters are latent states so that they are associated with each observation individually. Latent states are related as observations are so there is a continuity between them which can lead to a chain of states. The joint distribution for observations and states is:

$$P(X_1, ..., X_N, Z_1, ..., Z_N) = P(Z_1|\pi)[\prod_{n=2}^{N} (Z_n|Z_{n-1}, A)] \prod_{n=1}^{N} P(X_n|Z_n, \phi) \tag{6}$$

2.4.1 Viterbi decoding

Viterbi decoding, also known as the max-sum algorithm, finds the path through the model which has the maximal probability (Fariselli, 2005). It finds the best explanation for the observation sequence which is, in fact, the more likely sequence of states.

To compute this we should maximize the joint probability of a single path for each step so we will maximize the complete path in the end. The main idea of this approach is to compute a matrix of size *N x K* by maximizing through the joint probabilities of each observation. Because the number of paths is determined by the number of states and, mostly, by the length of the sequence (Bishop, 2006) the idea of computing all possible paths and choose the one with the highest probability is not an option. Moreover, the maximization will find the correct path in plausible computational running time. The algorithm is divided into two parts:

$$
\begin{aligned}
&Base\,case : \omega(Z_1) = P(X_1, Z_1) = P(Z_1|A)P(X_1|Z_1, \phi) \\
&Recursion : \omega(Z_n) = P(X_n|Z_n, \phi) \max_{Z_{n-1}} \omega(Z_{n-1})P(Z_n|Z_{n-1}, A)
\end{aligned}
\tag{7}
$$

Once the table (ω) is filled with the joint probabilities we need to select the most likely sequence of states given the observations. To do that in an efficient way, we need to select from end to beginning the path using the backtracking procedure. Because, for each state of a given variable, there is a unique state of the previous variable that maximizes the probability. Once we know the most probable value of the final node X_N, we can then simply follow the link back to find the most probable state of node X_{N-1} and so on back to the initial node X_1 (Bishop, 2006). We will start with the highest joint probability value of the last observation and continue until the beginning selecting the values that maximized the probability. The algorithm is also divided into two parts:

$$
\begin{aligned}
&Base\,case : Z_n^* = \max_{Z_n} \omega(Z_N) \\
&Recursion : Z_{n-1}^* = \max_{Z_n} (\omega(Z_{n-1})P(X_n|Z_n^*, \phi)P(Z_n^*|Z_{n-1}, A))
\end{aligned}
\tag{8}
$$

The running time of each algorithm is $O(NK^2)$ as it goes through the columns ones but twice through the rows. Also, the memory space that it requires is $O(NK)$ which is the size of the matrix.

2.4.2 Posterior decoding

The posterior decoding finds the path which maximizes the product of the posterior probability of the states (Fariselli, 2005).

First of all, we need to calculate the matrix of forward probabilities to get the probability of the observed sequence. In other words, each value of the matrix is the probability of the observation in a given state regarding all the possible paths that lead to that exact point on the sequence.

$$
\alpha = P(X_1, ..., X_n, Z_n|\theta) \tag{9}
$$

The computation of the alpha matrix is divided into two parts:

$$
\begin{aligned}
&Base\,case : \alpha(Z_1) = P(Z_1|\pi)P(X_1|Z_1, \phi) \\
&Recursion : \alpha(Z_n) = [\sum_{n=2}^{N} \alpha(Z_{n-1})P(Z_n|Z_{n-1}, A)]P(X_n|Z_n, \phi)
\end{aligned}
\tag{10}
$$

As the formula shows, we are summing all the probabilities of the previous observation and multiplying to the transition probabilities (A) between states. Then, we got the joint probability of the new observation for a given state multiplying to the emission probability (ϕ).

Second, we need to calculate the matrix of backwards probabilities to get the other possible path that explains the observed sequence. In this case, we will compute as the matrix alpha but starting in the end and going backward.

$$\beta = P(X_n, ..., X_1 | Z_n, \theta) \tag{11}$$

In this case, the model cannot be applied as the initial probabilities. To solve that issue, we fill each row of the last columns with values. Because the values of alpha are normalized between 0 and 1, the value to choose is 1.

$$\begin{aligned} &Base\,case : \beta(Z_N) = 1 \\ &Recursion : \beta(Z_n) = \sum_{n=N-1}^{1} \beta(Z_{n+1}) P(Z_n | Z_{n+1}, A) P(X_n | Z_n, \phi) \end{aligned} \tag{12}$$

The running time of each algorithm is $O(NK^2)$ as it goes through the columns ones but twice through the rows. Also, the memory space that it requires is $O(NK)$ which is the size of the matrix.

Once we have α and β matrices we can compute the highest local likelihood of each observation, and because of the horizontal relation of observations, we can estimate the most likely sequence of states regarding the most probable state per observation. We will end up having the most likely explanation for the sequence of observations similar to Viterbi decoding.

The local maximization is described by the following equation:

$$Z_n^* = \max_{Z_n} \frac{\alpha(Z_n)\beta(Z_n)}{P(X)} \tag{13}$$

2.5 Computational corrections

Hidden Markov model implementation for genome sequences, among other inputs, has the main drawback that makes the information retain during the computation to lose their meaning. This disadvantage is called *Underflow* and it is caused by numerical instabilities in the computation that make the values to reach the unrepresentable range of computers. This problem is due to the cumulative multiplication of small values, between 0 and 1, and it can be solved with two different corrections:

Scaling factor

Logarithm

Because of the need for the computational correction in the gene finder only the corrected decoding algorithms were implemented. For the implementation the programming language used was Python, more exactly, version 3 and the IDE used was Anaconda Jupyter Notebook. The complete code is available in Github.

2.5.1 Scaling factor

The implementation of the scaling factor was done for the Forward and Backward algorithms in Posterior decoding such as Bishop (2006) specifies in his book. The scaling factor used was computed as the sum of the α values for a given observation of the α matrix. Then, the α values of each observation were normalized using their own scaling factor. This approach satisfies two assumptions: the range of the normalized α values is from 0 to 1 and the sum of the normalized α values for a given observation is equal to 1.

$$c_n = \sum_{k=1}^{K} \alpha(Z_k) \tag{14}$$

$$\hat{\alpha}(Z_n) = \frac{\alpha(Z_n)}{c_n} \tag{15}$$

The scaling factor of the α matrix was storage and used in the β matrix. The values applied for each observation of β were the same as the ones computed for the α observation in the previous observation.

$$\hat{\beta}(Z_n) = \frac{\beta(Z_n)}{c_{n+1}} \tag{16}$$

The computation of the scaled matrices leads us to compute the most likely explanation of the sequence of observations but with a small change. Defining the $P(X)$ as the scaling factor of each observation, the product of these factors gives us $P(X)$. Because α and β are defined as $\hat{\alpha}$ and $\hat{\beta}$ multiplied by the scaling factor, the explanation could be calculated as:

$$Z_n^* = \max_{Z_n} \hat{\alpha}(Z_n)\hat{\beta}(Z_n) \tag{17}$$

The following code shows the implementation of Posterior decoding scaled, the code is divided into three functions.

Alpha table scaled:

```
def compute_alpha_scale(model, x):
    k = len(model.init_probs)
    n = len(x)

    init = model.init_probs
    trans = model.trans_probs
    emission = model.emission_probs

    alpha = make_table(n,k)

    # Base case
    cn = []
    factor = 0
    for i in range(0,k):
        alpha[0][i] = init[i] * emission[i][x[0]]
        factor += alpha[0][i]

    cn.append(factor)
    for h in range(0,k):
        alpha[0][h] /= factor

    # Inductive case
    for j in range(0,n-1):
        factor = 0
        for i in range(0,k):
            for l in range(0,k):
                alpha[j+1][i] += alpha[j][l] * trans[l][i]
            alpha[j+1][i] *= emission[i][x[j+1]]
            factor += alpha[j+1][i]
        cn.append(factor)
        for i in range(0,k):
            alpha[j+1][i] /= factor

    return(alpha,cn)
```

Beta table scaled:

```
def compute_beta_scale(model, x, cn):
    k = len(model.init_probs)
    n = len(x)

    beta = make_table(n, k)

    init = model.init_probs
    trans = model.trans_probs
    emission = model.emission_probs

    # Base case

    for i in range(0,k):
        beta[n-1][i] = 1

    # Inductive case

    for j in range(n-2,-1,-1):
        for i in range(0,k):
            val = 0
            for l in range(0,k):
                beta[j][i] += beta[j+1][l] * emission[l][x[j+1]] \
                              * trans[i][l]
            beta[j][i] /= cn[j+1]

    return beta
```

Posterior decoding scaled:

```
def posterior_decoding(model, x):
    alpha_var = compute_alpha_scale(model,x)
    alpha = alpha_var[0]
    beta = compute_beta_scale(model,x,alpha_var[1])

    k = len(model.init_probs)
    n = len(x)

    post = []
    for j in range(0,n):
        post.append([alpha[j][i] * beta[j][i] for i in range(0,k)])

    z = []
    for j in range(0,n):
        count = 0
        for i in range(0,k):
            if post[j][i] == max(post[j]) and count == 0:
                z.append(i)
                count += 1

    return z
```

2.5.2 Logarithm

Logarithms are a directly mathematical approach that make underflow be easily corrected. Nonetheless, it depends on the type of computation that the algorithm does. Viterbi decoding was easily corrected using the logarithm.

$$\begin{aligned} &Base\,case : \hat{\omega}(Z_1) = log(P(Z_1|\pi)) + log(P(X_1|Z_1,\phi)) \\ &Recursion : \hat{\omega}(Z_n) = log(P(X_n|Z_n,\phi)) + \max_{Z_{n-1}}(\hat{\omega}(Z_{n-1})) + log(P(Z_n|Z_{n-1},A)) \end{aligned} \tag{18}$$

The following code shows the implementation of Viterbi decoding logarithm, the code is divided into two functions.

Viterbi decoding algorithm:

```
def compute_w_log(model, x):
    k = len(model.init_probs)
    n = len(x)

    w = make_table(k, n)

    init = model.init_probs
    trans = model.trans_probs
    emission = model.emission_probs

    # Base case
    for i in range(0,k):
        w[i][0] = log(init[i]) + log(emission[i][x[0]])

    # Inductive case
    for j in range(0,n-1):
        for i in range(0,k):
            w[i][j+1] = log(0)
            for l in range(0,k):
                w[i][j+1] = max(w[i][j+1], w[l][j] +
                            log(emission[i][x[j+1]]) + log(trans[l][i]))

    return w
```

Bactracking algorithm:

```
def backtrack_log(model,w,x):
    trans = model.trans_probs
    emission = model.emission_probs
    z = []
    n = len(w[0])
    k = len(w)

    # Base case
    for i in range(0,k):
        if w[i][n-1] == opt_path_prob_log(w):
            z.append(i)

    # Inductive case
    t = 0
    for j in range(n-1,0,-1):
        counter = 1
        for i in range(0,k):
            if w[z[t]][j] == w[i][j-1] + log(emission[z[t]][x[j]])
                             + log(trans[i][z[t]]) and counter == 1:
                z.append(i)
                counter += 1
        t += 1

    return(z[::-1])
```

Furthermore, we also implemented the logarithm correction in Posterior decoding such as Mann (2006) suggested in his paper.

$$
\begin{aligned}
&Base\,case : \hat{\alpha}(Z_1) = log(P(Z_1, \pi)) + log(P(X_1|Z_1, A)) \\
&Recursion : \hat{\alpha}(Z_n) = log(\sum_{n=2}^{N} \alpha(Z_{n-1}) P(Z_n|Z_{n-1}, A)) + log(P(X_n|Z_n, \phi))
\end{aligned}
\tag{19}
$$

where the log of the sum results in:

$$
log(x + y) = log(x) + log(1 + 2^{log(y)-log(x)}) \tag{20}
$$

The computation of the most likely explanation of the sequence is given by:

$$
Z_n^* = \max_{Z_n}(log(\hat{\alpha}(Z_n)) + log(\hat{\beta}(Z_n))) \tag{21}
$$

The code is divided, the same as posterior decoding scaled, into three functions:

Alpha table algorithm:

```
def compute_alpha_log(model, x):
    k = len(model.init_probs)
    n = len(x)

    init = model.init_probs
    trans = model.trans_probs
    emission = model.emission_probs

    alpha_log = make_table_log(n,k)

    # Base case
    for i in range(0,k):
        alpha_log[0][i] = elnproduct(eln(init[i]),eln(emission[i][x[0]]))

    # Inductive case
    for j in range(0,n-1):
        for i in range(0,k):
            for l in range(0,k):
                alpha_log[j+1][i] = elnsum(alpha_log[j+1][i],
                    elnproduct(alpha_log[j][l], eln(trans[l][i])))
            alpha_log[j+1][i] = elnproduct(
                                alpha_log[j+1][i],
                                eln(emission[i][x[j+1]]))

    return(alpha_log)
```

Beta table algorithm:

```
def compute_beta_log(model, x):
    k = len(model.init_probs)
    n = len(x)

    beta_log = make_table_log(n, k)

    init = model.init_probs
    trans = model.trans_probs
    emission = model.emission_probs

    # Base case
    for i in range(0,k):
        beta_log[n-1][i] = 0

    # Inductive case
    for j in range(n-1,0,-1):
        for i in range(0,k):
            for l in range(0,k):
                beta_log[j-1][i] = elnsum(beta_log[j-1][i],
                                          elnproduct(eln(trans[i][l]),
                                          elnproduct(eln(emission[l][x[j]]),
                                          beta_log[j][l])))

    return beta_log
```

Posterior decoding algorithm:

```
def posterior_decoding_log(model, x):
    alpha_log = compute_alpha_log(model,x)
    beta_log = compute_beta_log(model,x)

    k = len(model.init_probs)
    n = len(x)

    z = []
    for j in range(0,n):
        post = []
        for i in range(0,k):
            val = alpha_log[j][i] + beta_log[j][i]
            post.append(val)

        count = 0
        for i in range(0,k):
            if max(post) == post[i] and count == 0:
                z.append(i)
                count += 1

    return z
```

2.6 Training

Once the accuracy was computed for the different approaches using the given model, the estimates were computed again using a trained model. The main reason for this idea is to use a different model to discard that our first groups of probabilities were incorrect *'per se'* and that it explains the results more than the algorithm does (they were biased). The models were created using the counting method explained in section 2.2.2 applied to both genomes so the result was two different models. Then, the computations were calculated again for each genome and each decoding algorithm using the model from the other genome.

2.7 Post-conditions

The computation of large matrices, such as the ones related to genome sequences, can not be ascertained manually. To verify that the computed result is correct, post conditions must be used. For example, checking that the assumptions of the calculations are maintained or that the intermediate values do not exceed the theoretical range. In the case of the implementation of Posterior decoding scale, the scaling factor was verified to maintain in the range between 0 and 1. Also, for all the corrected algorithms the final results were tested to follow a range so the values did not lose their meaning. Moreover, direct comparison with other implementation was done, as it is explained in the following subsection.

2.7.1 Code verification

Although the code is based on the work of Rabiner (1989) and Bishop (2006), the verification of the algorithms implemented has been done by comparing the results with the code provided by my tutor, Christian Storm. Obviously, the work of my tutor is established as a correct approach to the desired solution and therefore as a truth. Then, the results derived from the analyses carried out in this project must be the same as those obtained through the given algorithms. Finally, all the analyzes coincided for the different decoding genomes and algorithms, making the exposed results real and correct.

2.8 Comparison approach

The comparison method was based on the equality between states of prediction and the true annotation for each of the positions of the sequence of states. The accuracy was calculated with the total count of coincidences divided by the length of the sequence. Further, the running time was used to compare the different algorithms. In addition, the *True Positive rate* or *Sensitivity*, *False Positive rate* or *Specificity* and the *Precision* were computed for each prediction using the following formulas:

$$Sensitivity = \frac{TP}{TP + FN} \quad Specificity = 1 - \frac{FP}{FP + TN} \quad Precision = \frac{TP}{TP + FP}$$

3 Results

3.1 Comparison of accuracy with untrained model

The results, which are identical to the ones computed with my tutor's python scripts, are shown in the following table:

	Genome 1		**Genome 2**	
	3 states	**7 states**	**3 states**	**7 states**
Viterbi	0.318733	0.391936	0.350883	0.371922
Posterior scaled	0.302050	0.356339	0.297226	0.337900
Posterior log	0.302050	0.356339	0.297226	0.337900

Table 1: Accurracy results

In the previous table, there is represented the accuracy of the different decoding algorithms divided by states and genome. As we can see Viterbi decoding have better accuracy in all the cases with a maximum difference of around 5 percent. Also, the values for both the results shown for both posterior decoding algorithms are the same, as it should be because both maintain the same information in relation with the values given the same prediction per observation. However, the difference between values is not big enough to determine that Viterbi decoding is better than Posterior decoding approaches. On the other hand, the results show than the increase in the number of states could be related to an increase in accuracy. Due to the last point, it can be observed that the model given to the algorithm can have an important role in the generation of the prediction. That is the reason why the training of the model is carried out using the opposite genome to predict and to infer which model has a better behavior. The results are shown in the following section.

In addition, the following table collects data on the sensitivity, specificity and precision of the predictions of the different algorithms on both genomes. Only one value for Posterior decoding is shown because both approximations, scaled and logarithm, generate the same sequence of states and therefore the same values for the estimates. Because the values of sensitivity and specificity are not explanatory by themselves, precision is used as the comparison value of the predictions. In general, better precision for posterior decoding values is appreciated because although the sensitivity is better for Viterbi it also has a worse range of specificity that makes the precision decrease.

	Genome 1				Genome 2			
	3 states		7 states		3 states		7 states	
	Viterbi	**Posterior**	**Viterbi**	**Posterior**	**Viterbi**	**Posterior**	**Viterbi**	**Posterior**
Sensitivity	0.2769	0.2012	0.4959	0.3955	0.3895	0.2605	0.4938	0.4075
Specificity	0.7583	0.8059	0.5022	0.5634	0.6839	0.7672	0.5003	0.5460
Precision	0.3737	0.4035	0.2968	0.3216	0.4829	0.4210	0.2560	0.2988

Table 2: Sensitivity, Specificity and Precision for the untrained model

Another important point that leads to comparing between different approaches is the real running time of computation. In theory, all of them have the same running time (NK^2), however, the running time of the logarithmic computational solution for Posterior decoding was significantly higher than the two other approaches. If we compare the running time of Viterbi decoding and Posterior decoding scaled it can be seen that both are close to the same running time but Viterbi is less efficient when the number of states increases. The differences are not large enough to determine which approach runs in a better time.

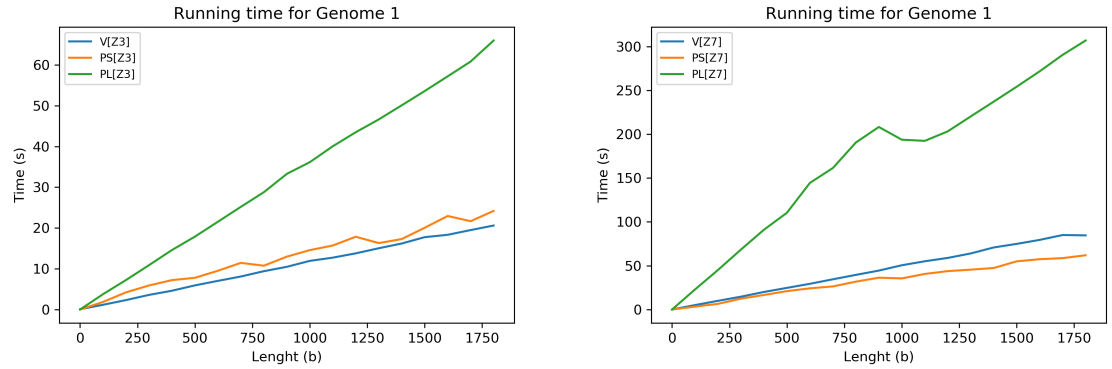

Figure 3: Running time for genome 1 with 3 states model (left) and 7 states model (right)

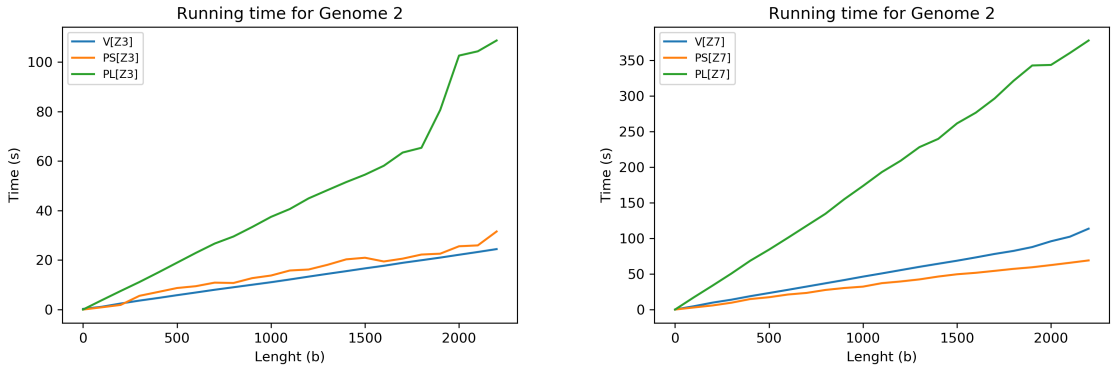

Figure 4: Running time for genome 2 with 3 states model (left) and 7 states model (right)

Even though the differences between the observed results and the running time do not show a preferred decoding algorithm, it has to be remark that Posterior decoding does not extract the most likely explanation of the observation but just the explanation with the highest probability computed. Because the method is not maximizing the computed value at each step (per nucleotide of the sequence), such as Viterbi does, the explanation can not be trusted as the best explanation (sequence of states) so the decrease in accuracy was expected.

3.2 Comparison of accuracy with trained model

The results for accuracy are shown in the following table:

	Genome 1		**Genome 2**	
	3 states	**7 states**	**3 states**	**7 states**
Viterbi	0.589919	0.764550	0.569582	0.782264
Posterior scaled	0.606373	0.765597	0.566750	0.793535
Posterior log	0.606373	0.765597	0.566750	0.793535

Table 3: Accuracy results with trained model

The comparison of both tables, table 1 and table 3, shows a significant increase in accuracy for Viterbi decoding and Posterior decoding. Also, both have values close to each other showing almost an equal accuracy but slightly better for posterior decoding approaches. The running time with the training model did not show a better performance, indeed, it was equal to the performance of section 3.1 so the Posterior logarithm decoding should not be used instead of Posterior decoding scaled.

The results for sensitivity, specificity and precision are shown in the following table:

	Genome 1				**Genome 2**			
	3 states		**7 states**		**3 states**		**7 states**	
	Viterbi	**Posterior**	**Viterbi**	**Posterior**	**Viterbi**	**Posterior**	**Viterbi**	**Posterior**
Sensitivity	0.8071	0.7931	0.8641	0.8668	0.6193	0.5842	0.8625	0.8795
Specificty	0.6833	0.7210	0.8868	0.8810	0.8455	0.8862	0.9025	0.8982
Precision	0.5941	0.6209	0.8156	0.8081	0.7158	0.7623	0.8477	0.8445

Table 4: Sensitivity, Specificity and Precision with training model

As expected, with increasing values of accuracy, values for sensitivity, specificity and precision increase too. As in table 2, the values of precision are slightly better for Posterior decoding and also, such as the values for accuracy show there is a tendency to return higher values with an increasing number of states.

4 Conclusion

Taking into account the results of the comparisons of the different algorithms, it seems that there is no clearly better method than the others. However, we can discard the implementation of the subsequent decoding logarithm because its execution time is several times higher than the scaled version of the subsequent decoding and increases as the number of states increases.

If we compare Viterbi with the subsequent decoding, we find slight differences, for example, the precision calculated for the predictions of the subsequent decoding is superior in all cases to Viterbi, and therefore, we estimate that it is a better prediction method. On the other hand, if we compare the precision values between both, there is no clear winner.

Finally, one of the possible keys to choosing the best method to implement is based on the performance of the methods. In other words, the way they get the prediction. In this case, the decoding of Viterbi should be more reliable because, as mentioned above, subsequent decoding does not guarantee the prediction of the most likely sequence of states, if not the most probable one.

5 References

Christopher M. Bishop. Pattern recognition and machine learning. Springer, 13: 605-635, 2006.

David kulp, David Haussler, Martin G. Reese and Frank H. Eeckman. A generalized Hidden Markov model for the recognition of human genes in DNA. ISMB-96 Proceedings. *University of California* and *Lawrence Berkeley National Laboratory*, 1996.

Ian Korf. Gene finding in novel genomes. BMC Bioinformatics, 5. *Welcome Trust Sanger Institute*, 2004.

Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE, 77(2):257-286,1989.

Piero Fariselli, Pier Luigi Martelli and Rita Casadio. A new decoding algorithm for hidden markov models imporves the prediction of the topology of all-beta membrane proteins. BMC bioinformatics, 6 .*University of Bologna*, 2005.

Tobias P. Mann. Numerically stable hidden markov model implementation. 2006.