

A comparison of algorithms for determining gene structure

Viterbi and Posterior decoding

Adrián Gómez Repollés

April 19, 2019

Abstract

The increase in the amount of genetic data have derive in the necessity of using reliable approaches to extract all possible information in a reduce amount of time. The reduction of the processing time could be reach by the use of computers, more exactly, algorithms. Such an example of 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 is done by estimating how good the prediction is in relation to the original gene structure. The results are divided in untrained and trained model, showing a better performance for Viterbi decoding in the fist model whereas it does not show a preference decoding algorithm for the second model. Finally, taking into account several factor, 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 to increase 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 to each other but this would break the true relationship between the nucleotides. However, the natural complexity of the sequence is unknown and could be computationally 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 prediction, 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 (for now on, HMM). 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.

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 length of 1.8Mb and the second one is from *Streptococcus agalactiae* with length of 2.2Mb. The genomes were in FASTA format so we extracted the data using an 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 make 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, \dots, X_N) and a sequence of discrete hidden states (Z_1, Z_2, \dots, Z_N) . Also, Hidden Markov model consist 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 the 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 in pursue of 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 each character. Also, the states coding and reverse coding can only be reached going through non coding state so the genes can not be overlapped.

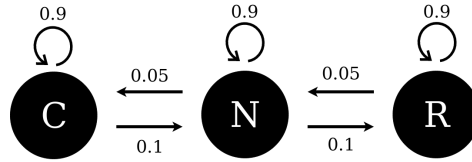


Figure 1: Three states model

Second, an slightly complex model where coding and reverse-coding regions should be at least 3 nucleotide 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 do not permit state coding and reverse to be reached without going through non-coding state, making the genes not to overlap in the sequence.

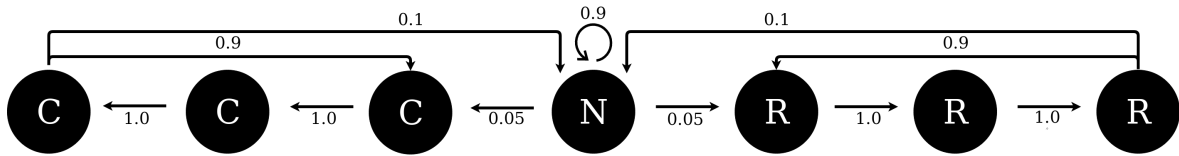


Figure 2: Seven states model

Even though these two models seems 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 a better maintain of the reading frame (David Kulp, 1996)

2.3 Probabilities

The Hidden Markov model contains three different probabilities:

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

$$\pi = P(X_1|Z_k) \quad (2)$$

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

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

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

$$\phi = P(X_n|Z_k) \quad (4)$$

2.4 Decoding algorithms

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

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

Moreover, if we want to build a model for sequences that is not limited by the Markov assumptions (?) we can use a limited numbers 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:

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

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 $N \times K$ 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 a plausible computational running time. The algorithm is divided in two parts:

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

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 in two parts:

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

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 it 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.

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

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

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

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.

$$\beta = P(X_n, \dots, X_1 | Z_n, \theta) \quad (11)$$

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.

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

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 it 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:

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

2.5 Computational corrections

Hidden Markov model implementation for genome sequences, among other inputs, have a main drawback that make 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 solve with two different corrections:

Scaling factor

Logarithm

Because of the need of 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.

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

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

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.

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

The computation of the scaled matrices lead us 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:

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

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

Alpha table scaled:

```
1 def compute_alpha_scale(model, x):
2     k = len(model.init_probs)
3     n = len(x)
4
5     init = model.init_probs
6     trans = model.trans_probs
7     emission = model.emission_probs
8
9     alpha = make_table(n, k)
10
11     # Base case
12     cn = []
13     factor = 0
14     for i in range(0, k):
15         alpha[0][i] = init[i] * emission[i][x[0]]
16         factor += alpha[0][i]
17
18     cn.append(factor)
19     for h in range(0, k):
20         alpha[0][h] /= factor
21
22     # Inductive case
23     for j in range(0, n-1):
24         factor = 0
25         for i in range(0, k):
26             for l in range(0, k):
27                 alpha[j+1][i] += alpha[j][l] * trans[l][i]
28                 alpha[j+1][i] *= emission[i][x[j+1]]
29                 factor += alpha[j+1][i]
30             cn.append(factor)
31         for i in range(0, k):
32             alpha[j+1][i] /= factor
33
34     return(alpha, cn)
```

Beta table scaled:

```
1 def compute_beta_scale(model, x, cn):
2     k = len(model.init_probs)
3     n = len(x)
4
5     beta = make_table(n, k)
6
7     init = model.init_probs
8     trans = model.trans_probs
9     emission = model.emission_probs
10
11     # Base case
12
13     for i in range(0, k):
14         beta[n-1][i] = 1
15
16     # Inductive case
17
18     for j in range(n-2, -1, -1):
19         for i in range(0, k):
20             val = 0
21             for l in range(0, k):
22                 beta[j][i] += beta[j+1][l] * emission[l][x[j+1]] \
23                     * trans[i][l]
24             beta[j][i] /= cn[j+1]
25
26     return beta
```

Posterior decoding scaled:

```
1 def posterior_decoding(model, x):
2     alpha_var = compute_alpha_scale(model, x)
3     alpha = alpha_var[0]
4     beta = compute_beta_scale(model, x, alpha_var[1])
5
6     k = len(model.init_probs)
7     n = len(x)
8
9     post = []
10    for j in range(0, n):
11        post.append([alpha[j][i] * beta[j][i] for i in range(0, k)])
12
13    z = []
14    for j in range(0, n):
15        count = 0
16        for i in range(0, k):
17            if post[j][i] == max(post[j]) and count == 0:
18                z.append(i)
19                count += 1
20
21    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.

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

The following code shows the implementation of Viterbi decoding logarithm, the code is divided in three functions.

Viterbi decoding algorithm:

```
1 def compute_w_log(model, x):
2     k = len(model.init_probs)
3     n = len(x)
4
5     w = make_table(k, n)
6
7     init = model.init_probs
8     trans = model.trans_probs
9     emission = model.emission_probs
10
11     # Base case
12     for i in range(0, k):
13         w[i][0] = log(init[i]) + log(emission[i][x[0]])
14
15     # Inductive case
16     for j in range(0, n-1):
17         for i in range(0, k):
18             w[i][j+1] = log(0)
19             for l in range(0, k):
20                 w[i][j+1] = max(w[i][j+1], w[l][j] +
21                                 log(emission[i][x[j+1]]) + log(trans[l][i]))
22
23     return w
```

Bactracking algorithm:

```
1 def backtrack_log(model, w, x):
2     trans = model.trans_probs
3     emission = model.emission_probs
4     z = []
5     n = len(w[0])
6     k = len(w)
7
8     # Base case
9     for i in range(0, k):
10         if w[i][n-1] == opt_path_prob_log(w):
11             z.append(i)
12
13     # Inductive case
14     t = 0
15     for j in range(n-1, 0, -1):
16         counter = 1
17         for i in range(0, k):
18             if w[z[t]][j] == w[i][j-1] + log(emission[z[t]][x[j]])
19                 + log(trans[i][z[t]]) and counter == 1:
20                 z.append(i)
21                 counter += 1
22         t += 1
23
24     return (z[::-1])
```

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

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

where the log of the sum result 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:

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

Alpha table algorithm:

```

1 def compute_alpha_log(model, x):
2     k = len(model.init_probs)
3     n = len(x)
4
5     init = model.init_probs
6     trans = model.trans_probs
7     emission = model.emission_probs
8
9     alpha_log = make_table_log(k, n)
10
11     # Base case
12     for i in range(0, k):
13         alpha_log[i][0] = elnproduct(eln(init[i]), eln(emission[i][x[0]]))
14
15     # Inductive case
16     for j in range(0, n-1):
17         for i in range(0, k):
18             for l in range(0, k):
19                 alpha_log[i][j+1] = elnsum(alpha_log[i][j+1],
20                 elnproduct(alpha_log[l][j], eln(trans[l][i])))
21                 alpha_log[i][j+1] = elnproduct(
22                     alpha_log[i][j+1],
23                     eln(emission[i][x[j+1]]))
24
25     return(alpha_log)

```

Beta table algorithm:

```
1 def compute_beta_log(model, x):
2     k = len(model.init_probs)
3     n = len(x)
4
5     beta_log = make_table_log(k, n)
6
7     init = model.init_probs
8     trans = model.trans_probs
9     emission = model.emission_probs
10
11     # Base case
12     for i in range(0, k):
13         beta_log[i][n-1] = 0
14
15     # Inductive case
16     for j in range(n-1, 0, -1):
17         for i in range(0, k):
18             for l in range(0, k):
19                 beta_log[i][j-1] = elnsum(beta_log[i][j-1],
20                 elnproduct(
21                     eln(trans[l][i]),
22                     elnproduct(eln(emission[l][x[j]]), beta_log[l][j])
23                 ))
24
25     return beta_log
```

Posterior decoding algorithm:

```
1 def posterior_decoding_log(model, x):
2     alpha_log = compute_alpha_log(model, x)
3     beta_log = compute_beta_log(model, x)
4
5     k = len(model.init_probs)
6     n = len(x)
7
8     z = []
9     for j in range(0, n):
10         post = []
11         for i in range(0, k):
12             val = alpha_log[i][j] * beta_log[i][j]
13             post.append(val)
14
15         count = 0
16         for i in range(0, k):
17             if max(post) == post[i] and count == 0:
18                 z.append(i)
19                 count += 1
20
21     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 bias). 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 with genome sequences, can not be ascertain 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.

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, other parameters were also used to compare, such as, the running time or the structure of the input.

3 Results

3.1 Comparison of accuracy with untrained model

The results 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	0.334633	0.337630	0.389478	0.397537

Table 1: Accuracy results

In the previous table there is represented the accuracy of the different decoding algorithms divided by states and genome. As the table shows, Viterbi decoding increase in accuracy as the number of states increase, as the other algorithms does, but the linearity of its accuracy seems to relate with a better approach. In other words, the discrepancy between the values of the Posterior decoding gives the impression of an approach more influence by the variability of the input. Moreover, the outputs were checked and the approach of the logarithm in Posterior decoding returned a path of most probable state were the state non-coding (N) was not present, so the approach seems not to work for this specific case and the accuracy result in a matter of coincidence between around 2 million characters. Taking the last point into account and comparing Viterbi decoding with the scaled approach of Posterior decoding, the table shows that the values of the first one are slightly better. The maximum difference between values is around 9 percent with an average difference of around 3-4 percent.

Another important point that lead to compare 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 more time consuming 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 increase.

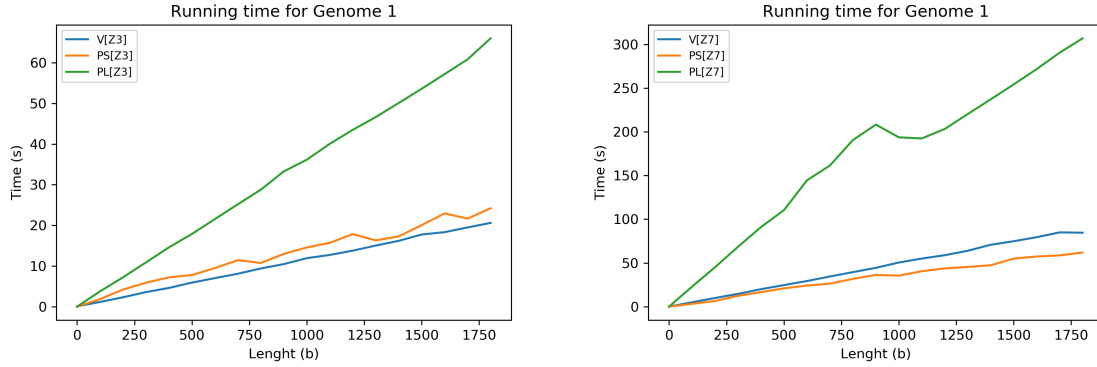


Figure 3: Running time for genome 1 with 3 states model (left) and 7 states model (right)

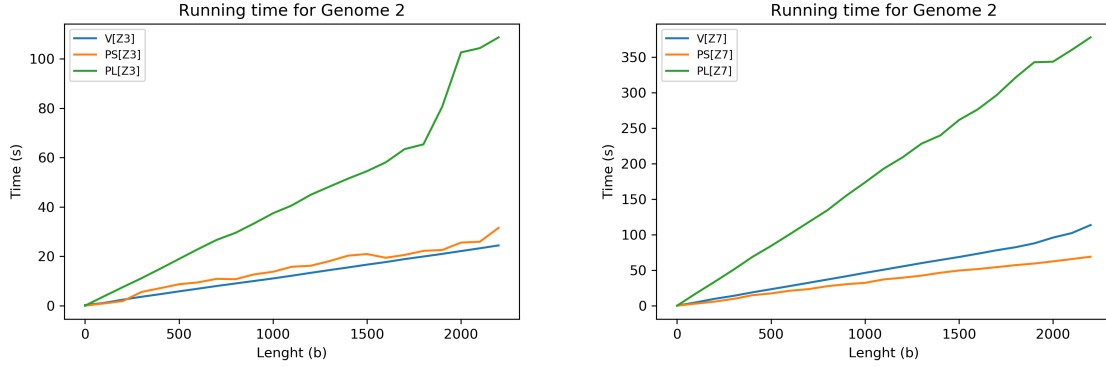


Figure 4: Running time for genome 2 with 3 states model (left) and 7 states model (right)

Moreover, we have to remark that Posterior decoding do not extract the most likely explanation of the observation but just the explanation with the highest probability computed. Because the method is not maximizing at each step, such as Viterbi does, the explanation can not be trusted as the best explanation so the decrease in accuracy was expected.

3.2 Comparison of accuracy with trained model

The results 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	0.117257	0.353025	0.097701	0.368005

Table 2: Accuracy results with trained model

The comparison of both tables. table 1 and table 2, shows a significant increase in accuracy for Viterbi decoding and Posterior decoding scaled. Also, both have values close to each other showing almost an equal accuracy. On the other hand, the logarithm approach of Posterior decoding does not shows a better accuracy, actually, the results are worst than for the untrained model.

The running time with the training model did not show a better performance, indeed, it was equal to the performance of section 3.1.

4 Conclusion

After analysing the results obtained from the experimentation, I conclude that the algorithm correction of Posterior decoding is not a plausible approach to solve this specific problem. The main reason is that the sequence of states returned as a prediction is not defining properly a gene structure. Also, the obtained results of the accuracy evince that chance is acting in the accuracy calculation and the running time is much larger than the other approaches. Rely on Posterior decoding algorithm is not possible and there is a need of more experimentation to test if it can be trust. Otherwise, both Viterbi and Posterior decoding gave plausible results that could be improved. Moreover, Viterbi decoding performance better with an untrained model whereas with a trained model both performed alike. The comparison of the running time of both algorithms does not show a preference for any of them.

Finally, because the prediction of Viterbi decoding is more precise and trusty than the one from Posterior decoding and both solve the problem in a similar running time, it is rather preferable to use Viterbi as the best approach to get the gene structure. Definitely, the interest of this method is to return the most likely path of states, such as Viterbi does, and not the highest probability of a given sequence, such as Posterior does. Finally, in my opinion, the implementation of Viterbi decoding is easier to code and understand than the scale factor solution of Posterior decoding.

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 improves 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.