Gene Prediction with Hidden Markov Models

Hidden Markov Models (HMM) are widely used in various fields of research: speech recognition, automatic natural language processing, handwriting recognition, and bioinformatics.

The 3 main problems associated to HMMs are:

1. Evaluation :
    - Problem: Compute the probability of observing the sequence given an HMM:
    - Solution: **Forward Algorithm**
2. Decoding:
    - Problem: find the sequence of states that maximizes the probability of observing the sequences.
    - Solution: **Viterbi Algorithm**
3. Training/Estimation:
    - Problem: Adjust the parameters of the HMM model to maximize the probability of generating the sequence of observations from the training data
    - Solution: **Forward-Backward Algorithm**

In this TME, we will apply the Viterbi algorithm to molecular biology data, in particular for the problem of gene prediction.

Some Biological background

In this small project, we will see how statistical models can be used to extract information from raw biological data. The goal will be to specify Hidden Markov Models which will allow to annotate the positions of the genes in the genome.

The genome, the carrier of genetic information, can be thought of as a long sequence of characters written in a 4-letter alphabet: `A`, `C`, `G` and `T`. Each letter of the genome is also called a base pair (or bp). It is now relatively inexpensive to sequence a genome (some direct to consumer company have offers as low as a few hundred euros for a human genome). However, we cannot understand, simply from the series of letters, how this information is used by the cell (a bit like having an instruction manual written in an unknown language, or a compiled code with no information on the machine).

An essential element is the gene, which after transcription and translation will produce proteins, the molecules responsible for much of the biochemical activity of cells.

if you do not know about transcription and translation, a short video about the basics: https://youtube.com/shorts/mSMjwxNK2EU?feature=share

Ameoba sisters page https://www.youtube.com/c/AmoebaSisters

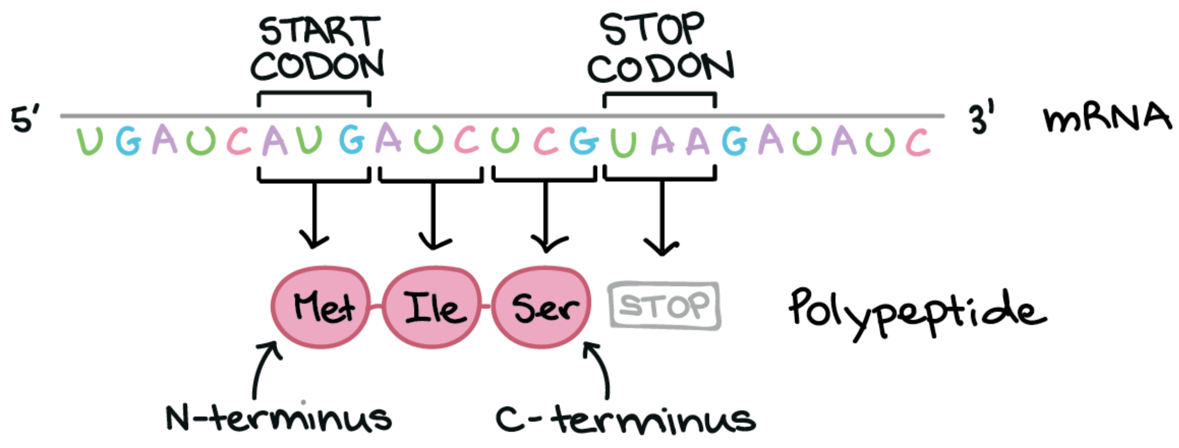
Two paragraphs summary for the impatient:

The translation into protein is done using the genetic code which, for each group of 3 letters (or bp) transcribed, matches an amino acid. These groups of 3 letters are called codons and there are 4^3, or 64. So, as a first approximation, a gene is defined by the following properties (for prokaryotic organisms):

- The first codon, called start codon is `ATG`,
- There are 61 codons which code for Amino Acids.
- The last codon, called the stop codon, marks the end of the gene and is one of the three sequences `TAA`, `TAG` or `TGA`. It does not appear in the gene.

We will integrate these different pieces of information to predict the positions of genes. Note that this figure is for the moment simplified, as we have omitted the fact that the DNA molecule consists of two complementary strands, and therefore that the genes present on the complementary strand are seen "upside down" on our sequence.

The regions between genes are simply called *intergenic regions*.



Important information to remember for the following: Each gene sequence starts with a start codon and ends with a stop codon

A few more biological information for the uninitiated

What is a chromosome? https://www.youtube.com/watch?v=IePMXxQ-KWY

Some more information about DNA and RNA (6min) https://youtu.be/JQByjprj_mA

And if you like to know how protein synthesis works (9min):
https://youtu.be/oefAI2x2CQM

And other video on the subject (3min): https://www.youtube.com/watch?v=gG7uCskUOrA

What is a gene (5min)?
https://www.youtube.com/watch?v=5MQdXjRPHmQ&list=PLInNVsmlBUlQT_peuWctrmGMiLngK-6fb&index=7

Here is a short (6min) video explaining the basis of gene regulation. Note that the part about operon is not important. https://youtu.be/h_1QLdtF8d0

Gene Modeling

Question 1: data download

We will be working on the first million bp of the E. coli genome (strain 042). Rather than working with the letters A, C, G, and T, we'll recode them with numbers ($A = 0$, $C = 1$, $G = 2$, $T = 3$).

The annotations provided are also encoded with integer values from 0 to 3:

- 0: the position is in a non-coding region = intergenic region
- 1: the position corresponds to a codon in phase 0
- 2: the position corresponds to a codon in phase 1
- 3: the position corresponds to a codon in phase 2

For instance for the drawing above we have the following values:

- `32031032031312300203031` for the sequence
- `00000123123123123000000` for the annotation

In [1]:
```
# Download pickle files for the genome sequences and
# its annotation
import numpy as np
import pickle as pkl

Genome=np.load('genome.npy') # the first Mio bp of E. coli
Annotation=np.load('annotation.npy')# gene annotation

## Let's split the data in two, one half for training and
## the second half for testing.

genome_train=Genome[:500000]
genome_test=Genome[500000:]

annotation_train=Annotation[:500000]
annotation_test=Annotation[500000:]
```

In [2]:
```
print(genome_train[0:5])
print(annotation_train[0:5])
```

```
[0 2 1 3 3]
[0 0 0 0 0]
```

Question 2: Parameters Estimation / Learning

As the simplest model for separating codon sequences from intergenic sequences, we will define the hidden Markov chain whose transition graph is given below.

modele1

Such a model is defined as follows: we consider that there are 4 possible hidden states (intergenic, codon phase 0, codon phase 1, codon phase 2).

You can stay in the intergenic regions, and when you start a gene, the composition of each base of the codon is different. In order to be able to use this model to classify, it will be necessary to know the parameters for the transition matrix (so here only the probas a and b), and the distribution $(b_i, i = 0, \ldots, 3)$ of the nucleotides given the four states.

```
Pi = np.array ([1, 0, 0, 0]) ## we start in intergenic regions
A = np.array ([[1-a, a, 0, 0],
               [0, 0, 1, 0],
               [0, 0, 0, 1],
               [b, 1-b, 0, 0]])
B = ...
```

Given the structure of an HMM:

- The initial distribution Pi and the transition matrix A are estimated in the same way as for a simple Markov model (see lecture 4). In other words, the observations have no influence on the hidden states when they are known.
- The distribution of each observation only depends on the current state.

Given the nature of the data we use Multinoulli for the emissions. As a convenience we will store all the distributions b_i in a matrix B (emission probability matrix) structured as follows:

- K columns (number of possible states), N rows (number of states)
- Each row corresponds to an emission law for a state (ie, each row sums to 1)

We can now simply learn the parameters b_i with the two following steps:

1. for each state $i \in \Sigma$ store in cell $b_{i,j}$ the number of times the letter j was observed with state i.
2. Normalize the rows of B to sum to one.

Write the code of the function `def learnHMM (allX, allS, N, K):` which learns a model from the combined sequence of observations and sequence of states.

In [3]:
```
def learnHMM(allx, allq, N, K):
    """ Learn an HMM given a pair of observation and states
    np.array[int] * np.array[int] * int * int ->
            (np.array[double,double], np.array[double,double])
    return transition matrices A and B"""
    A = np.zeros((N, N))
    B = np.zeros((N, K))

    ### Your code here
    A_temp = np.zeros((N, N))
    for i in range(len(allq)-1):
        A_temp[allq[i], allq[i+1]] += 1
    A = normalize_matrix(A_temp)

    B_count = np.zeros((N, K))
    for i, val_i in enumerate(allx):
        val_j = allq[i]
        B_count[val_j, val_i] += 1 #K is the ACTG protein (columns). N is the codon di

    B = normalize_matrix(B_count)

    return A,B

def normalize_matrix(matrix):
    norm_matrix = np.zeros((matrix.shape[0], matrix.shape[1]))
    for i, row_i in enumerate(matrix):
        if np.sum(row_i) != 0:
            norm_matrix[i] = row_i / np.sum(row_i)
        else:
            norm_matrix[i] = 0.0
    return norm_matrix
```

In [4]:

```
Pi = np.array([1, 0, 0, 0])
nb_states= 4 ## (intergenic, codon 0, codon 1, codon 2)
nb_observation = 4 ## (A,C,G,T)
A,B =learnHMM(genome_train, annotation_train, nb_states, nb_observation)
print(A)
print(B)
```

```
[[0.99899016 0.00100984 0.         0.        ]
 [0.         0.         1.         0.        ]
 [0.         0.         0.         1.        ]
 [0.00272284 0.99727716 0.         0.        ]]
[[0.2434762  0.25247178 0.24800145 0.25605057]
 [0.24727716 0.23681872 0.34909315 0.16681097]
 [0.28462222 0.23058695 0.20782446 0.27696637]
 [0.1857911  0.26246354 0.29707437 0.25467098]]
```

You should find:

$A =$

```
[[0.99899016 0.00100984 0.         0.        ]
 [0.         0.         1.         0.        ]
 [0.         0.         0.         1.        ]
 [0.00272284 0.99727716 0.         0.        ]]
```

$B =$

```
[[0.2434762  0.25247178 0.24800145 0.25605057]
 [0.24727716 0.23681872 0.34909315 0.16681097]
 [0.28462222 0.23058695 0.20782446 0.27696637]
 [0.1857911  0.26246354 0.29707437 0.25467098]]
```

Note that each row sums to 1

Question 3: Decoding using Viterbi algorithm

It is not always easy to find the coding and non-coding regions of a genome. We would like to automatically annotate the genome, that is to say to find **the most probable sequence of hidden states** which made it possible to generate the observation sequence.

Reminders on the Viterbi algorithm (1967):

- It is used to estimate the most probable sequence of states given the observations and the model.
- It can be used to approximate the probability of observing the sequence given the model.

It uses two recursion variables:

probability: $\delta_i(t) = \log \max_{s_1^{t-1}} P(s_1^{t-1}, s_t = i, y_1^t)$

backtrack: $\Psi_j(t) = \arg\max_{i\in\Sigma} \delta_i(t-1)a_{ij}$

1. Initialization (indices starting at 0):

$$\begin{aligned}\delta_i(0) &= \log \pi_i + \log b_i(x_0) \\ \Psi_i(0) &= -1\end{aligned}$$

Note: We initialize the first bactracking variable $\Psi_i(0)$ to -1 as this variable should not be used (-1 does not correspond to a state).

2. Recursion:

$$\begin{aligned}\delta_j(t) &= \left[\max_i \delta_i(t-1) + \log a_{ij}\right] + \log b_j(x_t) \\ \Psi_j(t) &= \arg\max_{i\in[1,\,N]} \delta_i(t-1) + \log a_{ij}\end{aligned}$$

3. Terminaison (with indices at {$T-1$} in python)

$$S^\star = \max_i \delta_i(T-1)$$

4. Path

$$\begin{aligned}s^\star_{T-1} &= \arg\max_i \delta_i(T-1) \\ s^\star_t &= \Psi_{t+1}(s^\star_{t+1})\end{aligned}$$

The estimate of $\log p(x_0^{T-1} \mid \lambda)$ is obtained by finding the greatest probability in the last column of δ.

Write down the algorithm of the `viterbi (x, Pi, A, B)` method:

Note: if you encounter problem with 0 cells giving infinite log values, you can try to add a very low value ϵ to all the cells of the transition matrix a_{ij} and for the emission probabilities b_j (something like $\epsilon = 10^{-10}$). Be carefull to renormalize the rows of a and each of the b_j to sum to 1 afterward.

In [5]:
```python
def viterbi(allx,Pi,A,B):
    """
    Parameters
    ----------
    allx : array (T,)
        Sequence d'observations.
    Pi: array, (K,)
        Distribution de probabilite initiale
    A : array (K, K)
        Matrice de transition
    B : array (K, M)
        Matrice d'emission matrix

    """

    ## initialisation
    psi = np.zeros((len(A), len(allx))) # A = N
    psi[:,0]= -1
    delta = np.zeros((len(A), len(allx))) #Initializing delta

    epsilon = 10**-10
    A_eps = np.copy(A)
    A_eps[A_eps == 0] = epsilon

    B_eps = np.copy(B)
    B_eps[B_eps == 0] = epsilon

    for x in range(len(allx)):
        if x == 0:
            #print(Pi_eps, np.log(Pi_eps))
            first_run = Pi * B[:,allx[x]]
            delta[:,x] = Pi
            first_run[first_run == 0] = epsilon
            first_run = np.log(first_run)
        elif x==1:
            last_max_val = first_run
            second_run = np.max(first_run, axis = 0) + np.log(A_eps[np.argmax(first_ru
            delta[:, x] = np.max(first_run, axis = 0)
            index = np.argmax(first_run, axis = 0)
            psi[:, x] = index
            delta[:, x] = second_run
        else:
            temp = np.zeros((len(A), len(A)), dtype = float)
            last_max_val = delta[:, x-1]
            for i, col in enumerate(last_max_val):
                for j in range(len(A)):
                    temp[j, i] = last_max_val[i] + np.log(A_eps[i,j] * B_eps[j, allx[x
            index = np.argmax(temp, axis = 1)
            #print(index)
            psi[:, x] = index
            delta[:, x] = np.take_along_axis(temp.T, index[np.newaxis, :], axis = 0)

    begin_q = np.argmax(delta[:,len(allx)-1])
    path = get_path(psi,begin_q)

    return path

def get_path(psi, begin_q):
    """
    From the matrix of psi values and the value S*, get the path of maximal values
    Parameters
    ----------
    psi: array (K,T)
    begin_q = int - value between 0 and K-1
    """
    ##Your code here
    path = np.zeros(psi.shape[1])
    path[-1] = begin_q
    for i in range(psi.shape[1]-2, -1, -1):
        path[i] = psi[path[i+1].astype(int), i+1]

    return path
```

Here is a small sequence if you want to test your Viterbi code:

In [6]:
```python
##Small code to test viterbi result
test_seq = "CGTGATATCATCAGGGCAGACCGGTTACATCCCCCTAACAAGCTGTTTAAAGAGAAATACTATCATGACGGACA
dDNA = {"A": 0 , "C": 1 , "G": 2, "T": 3}
test_seqi = np.array([dDNA[c] for c in test_seq])

path = viterbi(test_seqi, Pi, A, B)

print(path)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.
 3. 1. 2. 3. 1. 2. 3. 1. 2. 3. 1. 2.]
```

You should find the following state sequence after running Viterbi:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2,
       3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3,
       1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1,
       2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2,
       3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3,
       1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1,
       2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2,
       3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3,
       1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1,
       2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2,
       3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3,
       1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2])
```

You can now predict the states on your test sequences

In [7]:
```
predicted_states = viterbi(genome_test, Pi, A, B)

test_annotation = annotation_test
```

Display

Lets simplify the annotation to two categories:

- **coding** (1)
- and **non coding** (0).

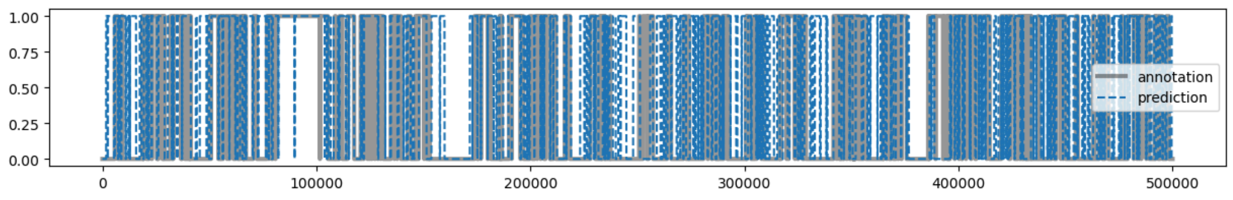
We can simply do that with a reallocation of the matrix of predictions (note that intergenic is state 0).

```
predicted_states[predicted_states != 0]=1
test_annotation[test_annotation != 0]=1
```

Then we will print for each genomic position if it is a coding or a non coding position using the true annotations and add the prediction on that.

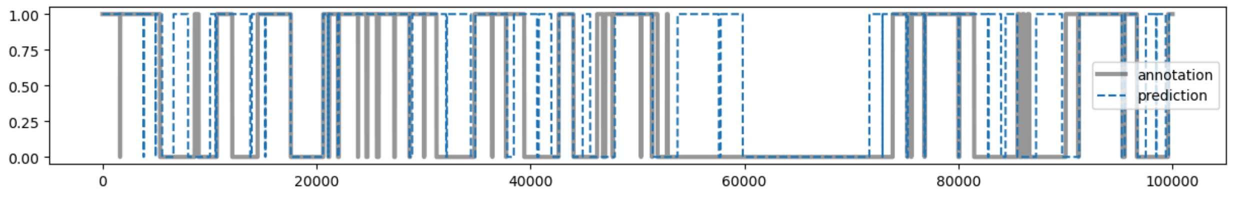
```
fig, ax = plt.subplots(figsize=(15,2))
ax.plot(test_annotation, label="annotation", lw=3, color="black", alpha=.4)
ax.plot(predicted_states, label="prediction", ls="--")
plt.legend(loc="best")
plt.show()
```

In [8]:
```
##Display here
import matplotlib.pyplot as plt

predicted_states[predicted_states != 0]=1
test_annotation[test_annotation != 0]=1

fig, ax = plt.subplots(figsize=(15,2))
ax.plot(test_annotation, label="annotation", lw=3, color="black", alpha=.4)
ax.plot(predicted_states, label="prediction", ls="--")
plt.legend(loc="best")
plt.show()
```



You can consider a part of the genome, plot the values between the positions 100,000 and 200,000. Comment on the quality of the prediction.

In [9]:
```python
fig, ax = plt.subplots(figsize=(15,2))
ax.plot(test_annotation[100000:200000], label="annotation", lw=3, color="black", alpha
ax.plot(predicted_states[100000:200000], label="prediction", ls="--")
plt.legend(loc="best")
plt.show()
```



your comment here, which state are well predicted? Do we overpredict sometimes? Are there non coding regions predicted as coding? Why would the model predict that? Conversely, are there coding regions that are predicted as intergenic?

The model predict better for non-coding area. The graph shown a big portion that is actually non-coding area, but predicted as coding (between position 55k to 60k) and predicted as non-coding while it is actually coding area. From the visualization, it doesn't look like it has a good result

Question 4 : Performance Evaluation

Using predictions and annotations of the genome, compute the confusion matrix.

In other words, we have:

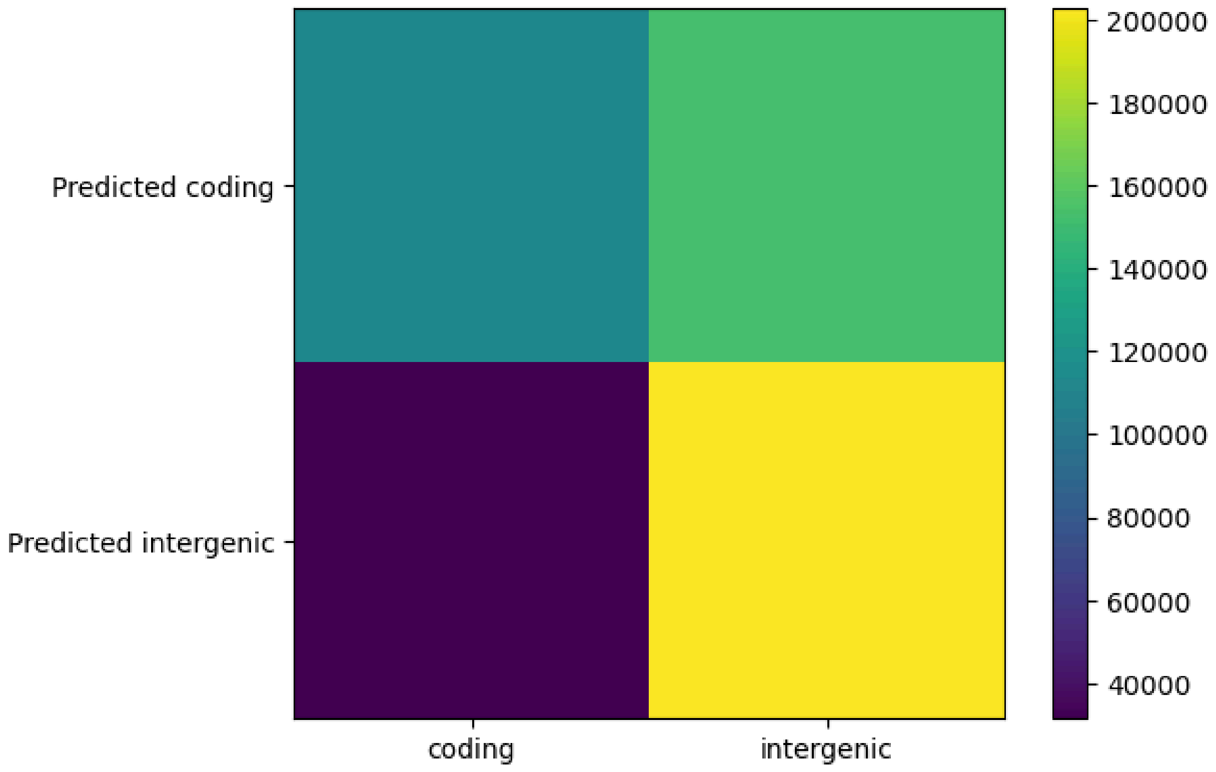
- TP = True Positives, coding regions that are correctly predicted,
- FP = False Positives, intergenic regions predicted as coding regions,
- TN = True Negatives, intergenic regions correctly predicted,
- FN = False Negatives, coding regions predicted as intergenic.

non coding state has index 0, the other states $(1, 2, 3)$ are **coding** states.

In [10]:
```python
from sklearn.metrics import confusion_matrix

def create_confusion_matrix(true_sequence, predicted_sequence):
    ## your code here
    mat_conf = confusion_matrix(true_sequence, predicted_sequence)
    return mat_conf

###Display the confusion matrix
import matplotlib.pyplot as plt
```

```
mat_conf=create_confusion_matrix(annotation_test, predicted_states)
plt.imshow(mat_conf)
plt.colorbar()
ax = plt.gca();

# Major ticks
ax.set_xticks(np.arange(0, 2, 1));
ax.set_yticks(np.arange(0, 2, 1));

# Labels for major ticks
ax.set_xticklabels(['coding','intergenic']);
ax.set_yticklabels(['Predicted coding','Predicted intergenic']);

print(mat_conf)
plt.show()
```

```
[[113022 152699]
 [ 31460 202819]]
```



Give an interpretation of the results, can we use this model to predict the position of the genes in the genome?

The TN and TP are high, but the FP and FN are also high. Meaning the model predict a large part of intergenic area as a coding area and smaller part of coding area predicted as intergenic area. If we are doing prediction using this model, then we will have a low accuracy score. The model is definitely need to be improved.

Question 5: Generating new sequences

Using the model estimated $\Theta = \{Pi, A, B\}$, specify a function `create_seq(N,Pi,A,B)` that, given a sequence length `N` would return:

- a sequence of hidden states
- a sequence of observations.

In [11]:
```
def create_seq(N,Pi,A,B):
    '''
    Return a sequence of N hidden states using Pi and A
    and for each hidden state return an observation using B
    '''
    ## your code here
    new_seq = np.zeros(N, dtype = int)
    new_obs = np.zeros(N, dtype = int)
    prob = np.zeros((N,4), dtype = float)
    for i in range(N):
        if i == 0:
            temp = np.matmul(Pi, A)
            new_seq[i] = np.argmax(temp, axis = 0)
            new_obs[i] = np.argmax(B[:,new_seq[i]], axis = 0)
            prob[i] = temp[new_seq[i]]
        else:
            temp = np.matmul(prob[i-1], A)
            new_seq[i] = np.argmax(temp, axis = 0)
            new_obs[i] = np.argmax(B[:,new_seq[i]], axis = 0)
            prob[i] = temp
    return new_obs, new_seq
```

In [12]:
```
new_observation, new_hidden_states = create_seq(500, Pi, A, B)
print(new_observation)
print(new_hidden_states)
```

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Question 6: Improving the model

Now let's assess if we can improve our prediction by incorporating an addtional layer of information in the model. We will take into account the gene boundaries by building a model that explicitly detects start codon and stop codon. We now want to integrate the additional information that says that a gene "always" begins with a start codon and "always" ends with a stop codon with the transition graph below.

The model now has 12 hidden states.

- Write the corresponding transition matrix, setting the transition probabilities between letters for stop codons to 0.5.
- Adapt the emissions matrix for all states of the model. You can reuse matrix B, calculated previously. The states corresponding to the stop codons will emit only one letter with a probability 1. For the start codon, we know that the proportions are as follows:
  - ATG : 83%,
  - GTG: 14%,
  - TTG: 3%

```
Pi2 = np.array(   [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ])  ##again, we start
in an intergenic region
A2 =  np.array([[1-a, a, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
                [0  , 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
                  ... ])
B2 = ...
```

Assess the performances of the new model by comparing it to the first model on `genome_test`.

```
predicted_states2=viterbi(genome_test,Pi2,A2,B2)
predicted_states2[predicted_states2!=0]=1
```

```
fig, ax = plt.subplots(figsize=(15,2))
ax.plot(annotation_test, label="annotation", lw=3, color="black", alpha=.4)
ax.plot(etat_predits, label="prediction model1", ls="--")
ax.plot(etat_predits2, label="prediction model2", ls="--")

plt.legend(loc="best")
plt.show()
```

Compute the confusion matrix with those new predictions and comment the results. Is it better than the previous one?

In [13]:
```
#Matrix B: how many A-C-T-G in start codon, C0, C1, C2, and stop codon (on each hidden
#(A=0, C=1, G=2, T=3)
def create_matrix(allx, allq, N, K):
    A_temp = np.zeros((N, N))
    B_temp = np.zeros((N, K))
```

```
for i in range(len(allq)-1):
    if allq[i] == 0: #from intergene
        if allq[i+1] == 0: #to intergene
            A_temp[0,0] += 1 #intergene to itself
        else: #to other than intergene
            A_temp[0,1] += 1 # intergene to start codon A T G
            A_temp[1,2] += 1 # start codon A T G to T
            A_temp[2,3] += 1 # start codon T to G
            A_temp[3,4] += 1 # G to C0
    else: #not from intergene
        if allq[i+1] != 0: #destination other than intergene
            A_temp[allq[i]+3, allq[i+1]+3] += 1 #C0 to C1, C1 to C2, C2 to C0
        else:  #C2 to Stop codon - Intergene section
            A_temp[6,7] += 1 #C2 to stop codon T
            if allq[i+1] == 2: #check the next observation
                A_temp[10,0] += 1 #stop codon A to intergene
            else:
                if allq[i+2] <= len(allq)-2 and allq[i+2] == 0:
                    A_temp[10,0] += 1 #stop codon A to intergene
                else:
                    A_temp[11,0] += 1 #stop codon G to intergene
A_temp[7,8] = 0.5
A_temp[8,10] = 1

A_temp[7,9] = 0.5
A_temp[9,10] = 0.5
A_temp[9,11] = 0.5


#(A=0, C=1, G=2, T=3)
B_count = np.zeros((N, K))
for i, val_i in enumerate(allx):
    val_j = allq[i]
    B_count[val_j, val_i] += 1

B_count = normalize_matrix(B_count)

B_temp[0,:] = B_count[0,:] #intergene
B_temp[4,:] = B_count[1,:] #C0
B_temp[5,:] = B_count[2,:] #C1
B_temp[6,:] = B_count[3,:] #C2

B_temp[1,0] = 0.83 #start codon A
B_temp[1,2] = 0.14 #start codon G
B_temp[1,3] = 0.03 #start codon T

B_temp[2,3] = 1 #start codon T
B_temp[3,2] = 1 #start codon G

B_temp[7,3] = 1 #stop codon T

B_temp[8,2] = 1 #stop codon G
B_temp[9,0] = 1 #stop codon A

B_temp[10,0] = 1 #stop codon A
B_temp[11,2] = 1 #stop codon G

#print(B_temp)
```

```
    B = B_temp

    A = normalize_matrix(A_temp)
    #B = normalize_matrix(B_temp.T)
    return A, B
```

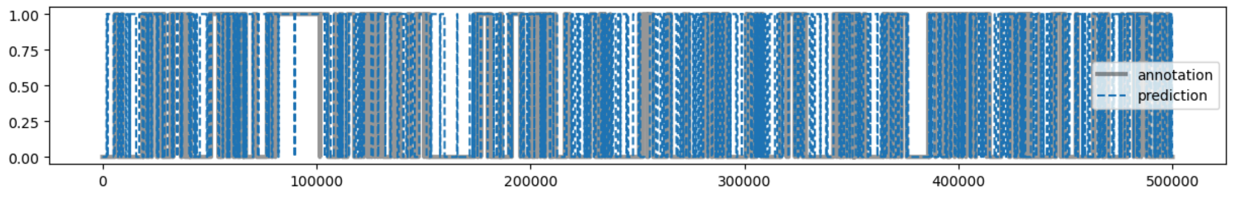
In [14]:
```
#from intergene to start
#print(genome_train[180:196])
#print(annotation_train[180:196])

#from codon to stop
#print(genome_train[250:265])
#print(annotation_train[250:265])
```

In [15]:
```
Pi2 = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ])
A2, B2 = create_matrix(genome_train, annotation_train, 12, 4)
print(A2)
print(B2)
```

```
[[0.99899016 0.00100984 0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.        ]
 [0.         0.         1.         0.         0.         0.
  0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         1.         0.         0.
  0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         1.         0.
  0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         1.
  0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  1.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.99727716 0.
  0.         0.00272284 0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.5        0.5        0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         1.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.5        0.5       ]
 [1.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.        ]
 [1.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.        ]]
[[0.2434762  0.25247178 0.24800145 0.25605057]
 [0.83       0.         0.14       0.03      ]
 [0.         0.         0.         1.        ]
 [0.         0.         1.         0.        ]
 [0.24727716 0.23681872 0.34909315 0.16681097]
 [0.28462222 0.23058695 0.20782446 0.27696637]
 [0.1857911  0.26246354 0.29707437 0.25467098]
 [0.         0.         0.         1.        ]
 [0.         0.         1.         0.        ]
 [1.         0.         0.         0.        ]
 [1.         0.         0.         0.        ]
 [0.         0.         1.         0.        ]]
```

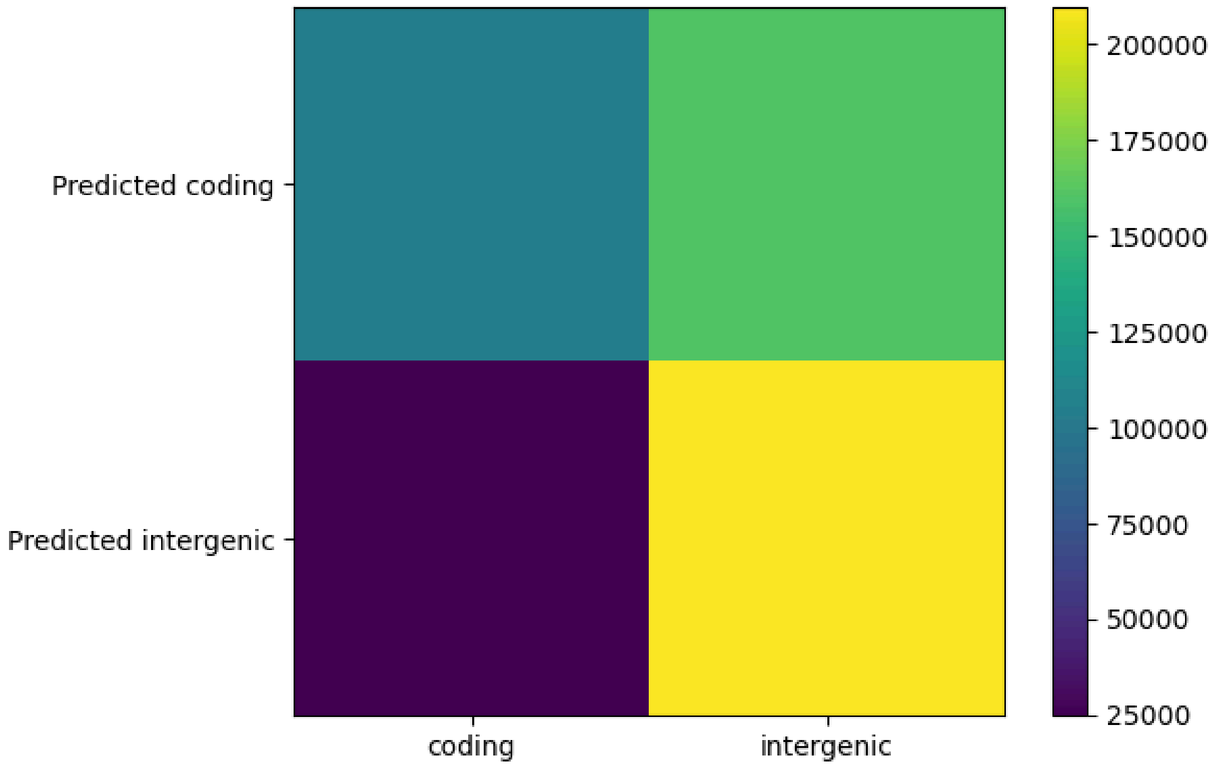
In [16]:
```
predicted_states2 = viterbi(genome_test, Pi2, A2, B2)
test_annotation2 = annotation_test
```

In [17]:
```
predicted_states2[predicted_states2 != 0]=1
test_annotation2[test_annotation2 != 0]=1

fig, ax = plt.subplots(figsize=(15,2))
ax.plot(test_annotation2, label="annotation", lw=3, color="black", alpha=.4)
ax.plot(predicted_states2, label="prediction", ls="--")
plt.legend(loc="best")
plt.show()
```



In [18]:
```
mat_conf=create_confusion_matrix(test_annotation2, predicted_states2)
plt.imshow(mat_conf)
plt.colorbar()
ax = plt.gca();

# Major ticks
ax.set_xticks(np.arange(0, 2, 1));
ax.set_yticks(np.arange(0, 2, 1));

# Labels for major ticks
ax.set_xticklabels(['coding','intergenic']);
ax.set_yticklabels(['Predicted coding','Predicted intergenic']);

print(mat_conf)
plt.show()
```

```
[[105835 159886]
 [ 24661 209618]]
```



Based on my result, it almost looks like there is no improvement on the model. If I have to calculate the accuracy score, it is more or less the same.

Question 7: Integrating reverse strand

We now want to add the information about the genes that could come from the complementary strand.

If you are unsure about what the forward and complementary strands means, you can check the videos above and the following video that summarizes it (the end about gene order is not important): https://youtu.be/JC6ew2xnJBA

In summary for a sequence of DNA that is written as `GCGATGCGTTGTAAACGCGATCAGCGCAT`, we have in fact two sequences, one for each strand:

```
                 x--------->
          5'  GCGATGCGTTGTAAACGCGATCAGCGCATGGG  3'   forward (or
plus) strand
              ||||||||||||||||||||||||||||||||
          3'  CGCTACGCAACATTTGCGCTAGTCGCGTACCC  5'   complementary
(or minus) strand
                                   <-------x
```

On the example above, there are two genes, one on the forward strand and one on the complementary strand. Because the genes are annotated using only forward strand, we will need to detect the gene information using the *reverse complementary sequence*.

In other words, we will add states for genes on the minus strand in the following way:

- we enter a minus gene with either `TCA`, `CTA`, `TTA` (reverse complementary of the stop codons: `TGA`, `TAG`, `TAA`)
- we leave a gene with `[C]A[TCA]` (reverse complementary of a start codon)

The corresponding transition graph is as follow (You can see that this model has 22 states, numbered in orange):

A perfect annotation for our small example sequence woul be:

```
                 x---------->
          5'  GCGATGCGTTGATAAACGCGATCAGCGCATGGG  3'   forward (or
plus) strand
              |||||||||||||||||||||||||||||||||
          3'  CGCTACGCAACTATTTGCGCTAGTCGCGTACCC  5'   complementary
(or minus) strand
                                    <-------x
```

```
                     1      111111222          state number
0001234564567900000002367890120000
```

To implement such a model, you will have to

- deduce from the observation matrices for codons a second one for the codons that are seen on the reverse strand. You can make the hypothesis that the codon distribution is the same on both strands.
- encode the transition matrix for observing genes on the reverse strand, starting from a reverse codon stop and ending with a codon start.

Implement a third model `Pi3, A3, B3` that would take into account the reverse strand and evaluate its performances with respect to model 1 and model 2.

Note: Be careful with the evaluation, the annotation given only provides the genes that are on the forward strand (the genes on the reverse strand are annotated as intergenic). If we use the numbering provided in the figure, we would do something like:

```
predicted_states[predicted_states > 11]=0 #reverse strand genes as negatives
predicted_states[predicted_states != 0]=1 #forward strand genes as positives
```

```
In [ ]:
```

```
In [ ]:
```

Question 8: Implementing the forward-backward algorithm (optional)

Using the information presented in the lecture, implement an EM estimation of the parameter based on the forward-backward algorithm.

1. Write a function for the forward algorithm
2. Write a function for the backward algorithm
3. Deduce a function to compute the smoothing probabilities
4. Write the EM algorithm and compare the estimation results with the viterbi based method.

Did you encountered any problem while implementing this algorithm? Detail and comment each step of your analysis

Note: The forward and backward values decrease exponentially fast with the length of the sequence, leading to numerical issues. **You will need to integrate rescaling factors for the probabilities (as described in the lecture)**.

```
In [ ]:
```