TME 7 Chaine de Markov Caché

Annotation de gènes par chaînes de Markov Caché

Les modeles de chaines de Markov caché sont tres utilisées notament dans les domaines de la reconnaisance de la parole, du traitement automatique du langage naturel, de la reconnaissance de l'écriture manuscrite et de la bioinformatique.

Les 3 problèmes de bases des HMM (*Hidden Markov Model*) sont :

1. Évaluation :
    - Problème : calculer la probabilité d'observation de la séquence d'observations étant donnée un HMM:
    - Solution : *Forward Algorithm*
2. Décodage :
    - Problème : trouver la séquence d'états qui maximise la séquence d'observations
    - Solution : *Viterbi Algorithm*
3. Entraînement :
    - Problème : ajuster les paramètres du modèle HMM afin de maximiser la probabilité de générer une séquence d'observations à partir de données d'entraînement
    - Solution : *Forward-Backward Algorithm*

Dans ce TME, nous allons appliquer l'algorithme Viterbi à des données biologiques.

Rappel de biologie

Dans ce TME, nous allons voir comment les modèles statistiques peuvent être utilisés pour extraire de l'information des données biologiques brutes. Le but sera de spécifier des modèles de Markov cachées qui permettent d'annoter les positions des gènes dans le génome.

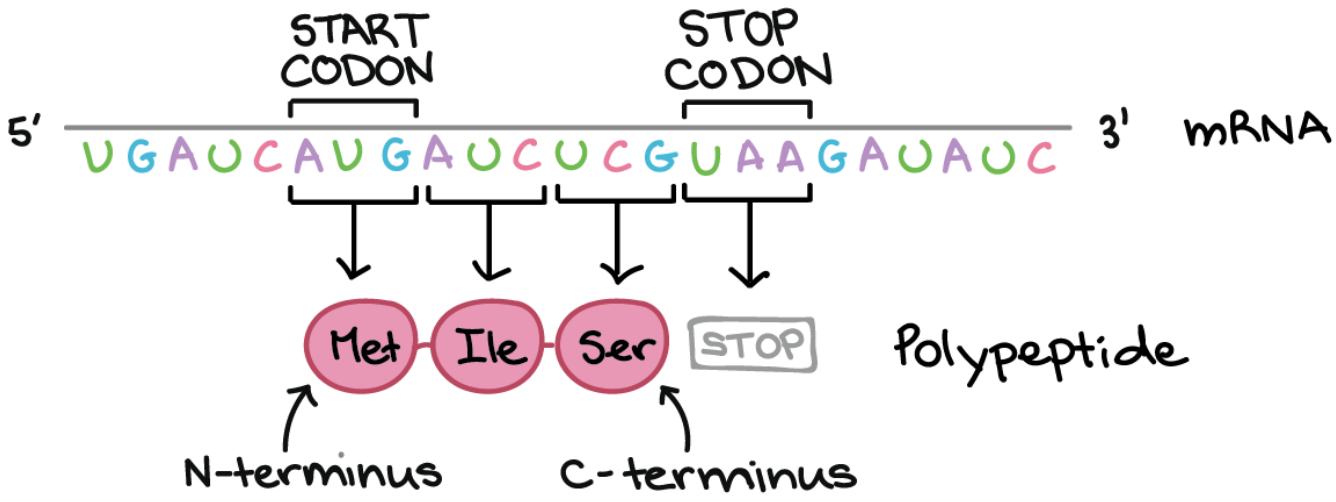
Le génome, support de l'information génétique, peut être vu comme une longue séquence de caractères écrite dans un alphabet à 4 lettres: A , C , G et T . Chaque lettre du génome est aussi appelée pair de base (ou bp). Il est maintenant relativement peu coûteux de séquencer un génome (quelques milliers d'euros pour un génome humain). Cependant on ne peut pas comprendre, simplement à partir de la suite de lettres, comment cette information est utilisée par la cellule (un peu comme avoir à disposition un manuel d'instructions écrit dans une langue inconnue).

Un élément essentiel est le gène, qui après transcription et traduction produira les protéines, les molécules responsables de la grande partie de l'activité biochimique des cellules.

La traduction en protéine est faite à l'aide du code génétique qui, à chaque groupe de 3 lettres (ou bp) transcrites fait correspondre un acide aminé. Ces groupes de 3 lettres sont appelés codon et il y en a 4^3, soit 64. Donc, en première approximation, un gène est défini par les propriétés suivantes (pour les organismes procaryotes):

- Le premier codon, appelé codon start est ATG,
- Il y a 61 codons qui codent pour la séquence d'acides aminés.
- Le dernier codon, appelé codon stop, marque la fin du gène et est l'une des trois séquences TAA , TAG ou TGA . Il n'apparaît pas dans le gène.

Nous allons intégrer ces différents éléments d'information pour prédire les positions des gènes. Notez que pour simplifier nous avons omis le fait que la molécule d'ADN est constituée de deux brins complémentaires, et donc que les gènes présents sur le brin complémentaire sont vus "à l'envers" sur notre séquence. Les régions entre les gènes sont appelées les régions intergéniques .



Chacune des séquences de gènes commence par un codon start et fini par un des codons stop.

Modélisation de gènes

Question 1 : Téléchargement des données

Nous travaillerons sur le premier million de bp du génome de E. coli (souche 042). Plutôt que de travailler avec les lettres A , C , G et T , nous allons les recoder avec des numéros ($A = 0$, $C = 1$, $G = 2$, $T = 3$).

Les annotations fournies sont également codées de 0 à 3 :

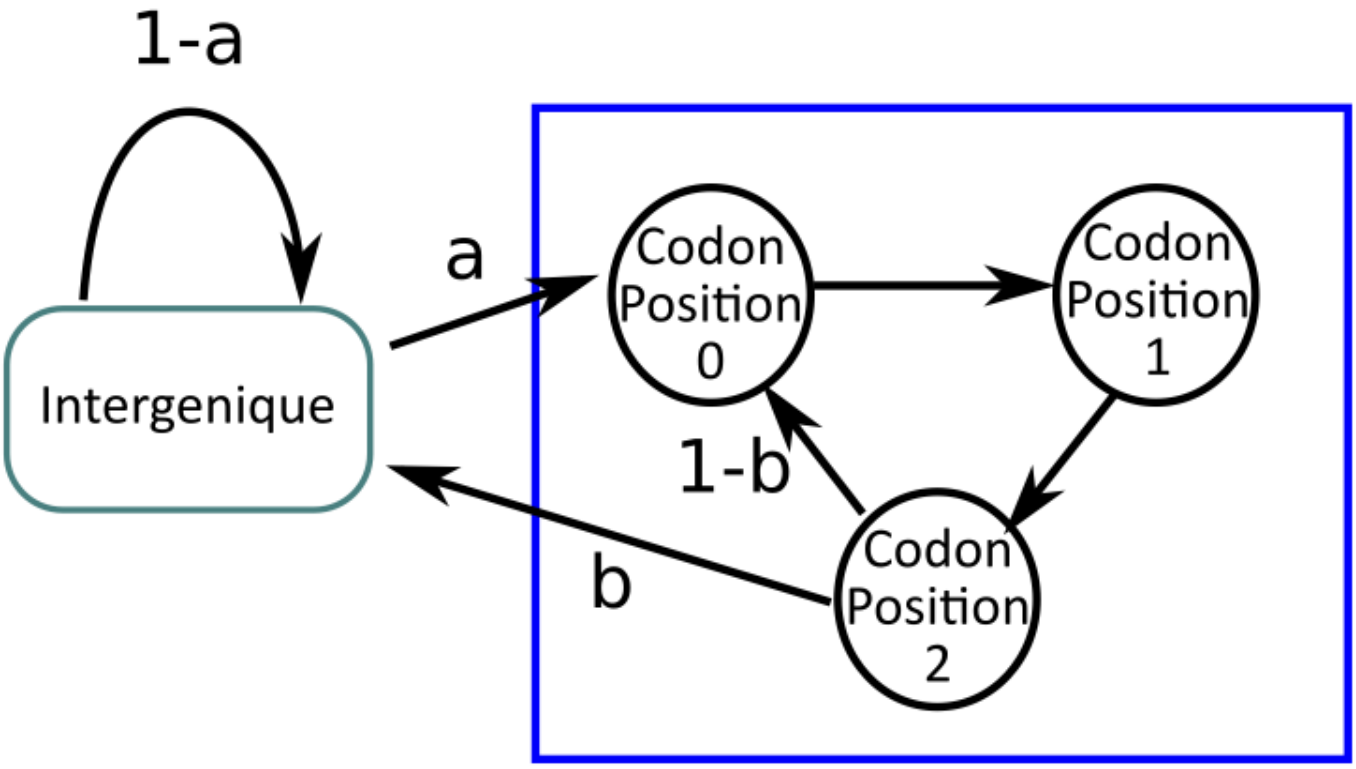
- 0 si la position est dans une region non codante = region intergenique
- 1 si la position correspond a la position 0 d'un codon
- 2 si la position correspond a la position 1 d'un codon
- 3 si la position correspond a la position 2 d'un codon

In [38]:

```
# Telechargez le fichier et ouvrez le avec pickle
import numpy as np
import pickle as pkl

Genome=np.load('genome.npy') # Le premier million de bp de E. coli
Annotation=np.load('annotation.npy')# L'annotation sur le genome

## On divise nos donnees, la moitie va nous sevir pour l'apprentissage du modèle
## l'autre partie pour son evaluation

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

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

Question 2 : Apprentissage

Comme modèle le plus simple pour séparer les séquences de codons des séquences intergéniques, on va définir la chaîne de Markov caché dont le graphe de transition est donné ci dessous.



Un tel modèle se défini de la manière suivante : nous considerons qu'il existe 4 états cachés possibles (intergénique, condon 0, codon 1, condon 2).

On peut rester dans les régions intergéniques, et quand on démarre un gène, la composition de chaque base du codon est différente. Il va falloir, afin de pouvoir utiliser ce modèle pour classifier, connaître les paramètres pour la matrice de transition (donc ici uniquement les probas a et b), et les lois $(b_i, i = 0, \ldots, 3)$ des observations pour les quatre états.

```
Pi = np.array([1, 0, 0, 0])  ##on commence dans l'intergenique
A =  np.array([[1-a, a  , 0, 0],
               [0  , 0  , 1, 0],
               [0  , 0  , 0, 1],
               [b  , 1-b, 0, 0 ]])
B = ...
```

Étant donnée la structure d'un HMM (Hidden Markov Chain):

- les observations n'influencent pas les états: les matrices Π (distribution de probabilité initiale), A (matrice de transition) s'obtiennent comme dans un modèle de Markov simple (cf semaine 6)
- chaque observation ne dépend que de l'état courant

La nature des données nous pousse à considérer des lois de probabilités discrètes quelconques pour les émissions. L'idée est donc de procéder par comptage en définissant la matrice B (matrice de probabilités des émissions) comme suit:

- K colonnes (nombre d'observations), N lignes (nombre d'états)
- Chaque ligne correspond à une loi d'émission pour un état (ie, chaque ligne somme à 1)

Ce qui donne l'algorithme:

1. b_{ij} = comptage des émissions depuis l'état s_i vers l'observation x_j
2. normalisation des lignes de B

Donner le code de la fonction `def learnHMM(allX, allS, N, K):` qui apprend un modèle à partir d'un ensemble de couples (seq. d'observations, seq. d'états)

In [39]:

```
def learnHMM(allx, allq, N, K):
    """ apprend un modèle à partir
    d'un ensemble de couples (seq. d'observations, seq. d'états)
    retourne les matrices A  B """
    A = np.zeros((N, N))
    B = np.zeros((N, K))
    # allx = un génome = observation
    # allq = annotation = états
    print(allx.shape)
    print(allq.shape)
    print(allq)
    for i in range(allx.shape[0]):
        B[allq[i], allx[i]]+=1
    for i in range( allq.shape[0]-1):
        A[allq[i], allq[i+1]]+=1
    #normaliser
    B/=np.maximum(B.sum(1).reshape(N,1), 1)
    A/=np.maximum(A.sum(1).reshape(N,1), 1)
    ### votre code
    return A, B
    pass
```

In [40]:

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

```
(500000,)
(500000,)
[0 0 0 ... 0 0 0]
[[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]]
```

Vous devez trouver

$A =$

```
[[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]]
```

Notez que ce sont des matrices de probabilites, la somme de chaque ligne donne 1.

Question 3 : Estimation la séquence d'états par Viterbi

Il n'est pas toujours évident de trouver les régions codante et non codante d'un genome. Nous souhaiterions annoter automatiquement le génome, c'est-à-dire retrouver **la séquence d'états cachés la plus probable** ayant permis de générer la séquence d'observation.

Rappels sur l'algorithme Viterbi (1967):

- Il sert à estimer la séquence d'états la plus probable étant donnés les observations et le modèle.
- Il peut servir à approximer la probabilité de la séquence d'observation étant donné le modèle.

1. Initialisation (avec les indices à 0 en python):

$$\begin{aligned} \delta_0(i) &= \log \pi_i + \log b_i(x_0) \\ \Psi_0(i) &= -1 \end{aligned}$$

Note: L'initialisation de $\Psi_0(i)$ à -1 car -1 n'est pas utilisé normalement (n'est pas un état valide).

2. Récursion:

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

3. Terminaison (indices à $\{T-1\}$ en python)

$$S^\star = \max_i \delta_{T-1}(i)$$

4. Chemin

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

L'estimation de $\log p(x_0^{T-1}|\lambda)$ est obtenue en cherchant la plus grande probabilité dans la dernière colonne de δ. Donner le code de la méthode `viterbi(x,Pi,A,B)`:

In [92]:

```
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 #colonne 0 pour tous les états
    delta = np.zeros((len(A), len(allx)))  # initialisation en dimension mais pa
s en contenu !
    # init pour chaque état
    delta[:,0]=np.log(Pi)+ np.log(B[:,allx[0]])

    # debugging

    delta1= np.zeros((len(A), len(allx)))
    #delta1[:,0]=np.log(Pi)+ np.log(B[:,allx[0]])
    for i in range(len(A)):
        delta1[i,0]=np.log(Pi[i])+np.log(B[i, allx[0]])
    psi1=np.zeros((len(A), len(allx)))
    psi1[:,0]=-1
    print(delta[:,0])
    print(delta1[:,0])
    ## recursion ...  (votre code )
    for t in range(1,len(allx)):
        # je remplis tous les j d'un seul coup en vectorisant
        # j correspond au numéro d'état
        # t correspond à une rang d'observation parmi toute la séquence d'observ
ations
        # delta= meilleur score pour un chemin au temps T se terminant à l'état
 j

        delta[:, t]=np.transpose(np.max(np.reshape(delta[:, t-1], (len(A),1))+ n
p.log(A[:,:]), axis=0))+np.log(B[:,allx[t]])
        psi[:, t]=np.transpose(np.argmax(np.reshape(delta[:,t-1], (len(A), 1))+n
p.log(A), axis=0))
        '''for j in range(len(A)):
            delta1[j, t]=np.max(delta1[:, t-1]+np.log(A[:,j]))+np.log(B[j, allx
[t]])

            psi1[j,t]=np.argmax(delta1[:, t-1]+np.log(A[:,j]))
        if(t<10):
            print(delta[:,t])
            print(delta1[:,t])'''

    # parcourir à l'envers pour trouver le chemin optimal
    # initialisation dans le chemin inverse
    T=(len(allx))
    '''print(T)
    print("delta1 dernier état T-1")
```

```
    print(delta1[:,T-1 ])
    print("psi1 dernier etat T-1")
    print(psi1[:,T-1])'''
    S=np.zeros(allx.shape)
    # pour debuggage
    '''S1=np.zeros(allx.shape)
    S1[T-1]=np.argmax(delta1[:,T-1])
    for i in range(T-2,-1, -1):
        S1[i]=psi1[np.int(S1[i+1]),i+1]
    print("delta dernier état T-1")
    print(delta[:,T-1 ])
    print("psi dernier etat T-1")
    print(psi[:,T-1])
    print("---------------")
    for i in range(4):
        print(delta[:,i])
        print(delta1[:,i])'''
    S[T-1]=np.argmax(delta[:,T-1 ])
    for i in range(T-2,-1, -1):
        #print(S[i+1])
        S[i]=psi[np.int(S[i+1]),i+1]
    return S
    pass
```

In [94]:

```
etat_predits=viterbi(genome_test,Pi,A,B)
print("états prédits")
print(etat_predits)
print(" annotations données")
print(annotation_test)
```

```
/usr/lib/python3/dist-packages/ipykernel_launcher.py:20: RuntimeWarn
ing: divide by zero encountered in log
/usr/lib/python3/dist-packages/ipykernel_launcher.py:27: RuntimeWarn
ing: divide by zero encountered in log
/usr/lib/python3/dist-packages/ipykernel_launcher.py:39: RuntimeWarn
ing: divide by zero encountered in log
/usr/lib/python3/dist-packages/ipykernel_launcher.py:40: RuntimeWarn
ing: divide by zero encountered in log

[-1.41273607        -inf        -inf        -inf]
[-1.41273607        -inf        -inf        -inf]
états prédits
[0. 0. 0. ... 1. 2. 3.]
 annotations données
[0 0 0 ... 0 0 0]
```

In [95]:

```
print(etat_predits==annotation_test)
print(np.sum(etat_predits1==annotation_test))
print ( np.sum(etat_predits==annotation_test))
print(np.sum(etat_predits))
print(len(etat_predits))
```

```
[ True  True  True ... False False False]
265721
180635
711036.0
0.0
500000
```

In [96]:

```
I=np.array([[1,2], [3, 4]])
print(np.max(I, axis=0))
S=np.array([[1],[2]])
print(S+I)
```

```
[3 4]
[[2 3]
 [5 6]]
```

In [85]:

```
I[:,0]=np.transpose(np.array([5,5]))
print(I)
```

```
[[5 2]
 [5 4]]
```

Affichage

On met les états cachés soit à 0 (**non codant**) soit à 1 (**codant**).

```
etat_predits[etat_predits!=0]=1
annotation_test[annotation_test!=0]=1
```

puis on affiche pour position du génome si c'est une position codante ou non en utilisant les vrais annotations, puis on affiche pour chaque position si elle est predite comme codante ou non.

```
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", ls="--")
plt.legend(loc="best")
plt.show()
```

In [61]:

```
import matplotlib.pyplot as plt
'''etat_predits[etat_predits!=0]=1
annotation_test[annotation_test!=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", ls="--")
plt.legend(loc="best")
plt.show()'''
```

Out[61]:

```
'etat_predits[etat_predits!=0]=1 \nannotation_test[annotation_test!=
0]=1\n\nfig, ax = plt.subplots(figsize=(15,2))\nax.plot(annotation_t
est, label="annotation", lw=3, color="black", alpha=.4)\nax.plot(eta
t_predits, label="prediction", ls="--")\nplt.legend(loc="best")\npl
t.show()'
```

Vous pouvez considérer une sous partie du génome, par exemple entre 100000 et 200000. Commentez vos observations sur la qualité de la prédiction.

Question 4 : Evaluation des performances

À partir des prédictions et des vrai annotations du génome, dessiner la matrice de confusion.

Avec :

- TP = True Positives, les régions codantes correctement prédictes,
- FP = False Positives, les régions intergénique prédites comme des régions codantes,
- TN = True Negatives, les régions intergeniques prédites correctement,
- FN = False Negatives, les régions codantes prédites comme non codantes.

L'état **non codant** est l'état 0, les autres $(1, 2, 3)$ sont les états **codants**.

In [103]:

```
def create_confusion_matrix(true_sequence, predicted_sequence):
    ## votre code
    etat_predits[predicted_sequence!=0]=1
    #etat_predits[predicted_sequence==0]=1
    annotation_test[true_sequence!=0]=1
    #annotation_test[true_sequence==0]=1
    conf=np.zeros((2, 2))
    for i in range(len(etat_predits)):
        conf[1-np.int(etat_predits[i]), 1-np.int(annotation_test[i]) ]+=1
    return conf
    pass
```

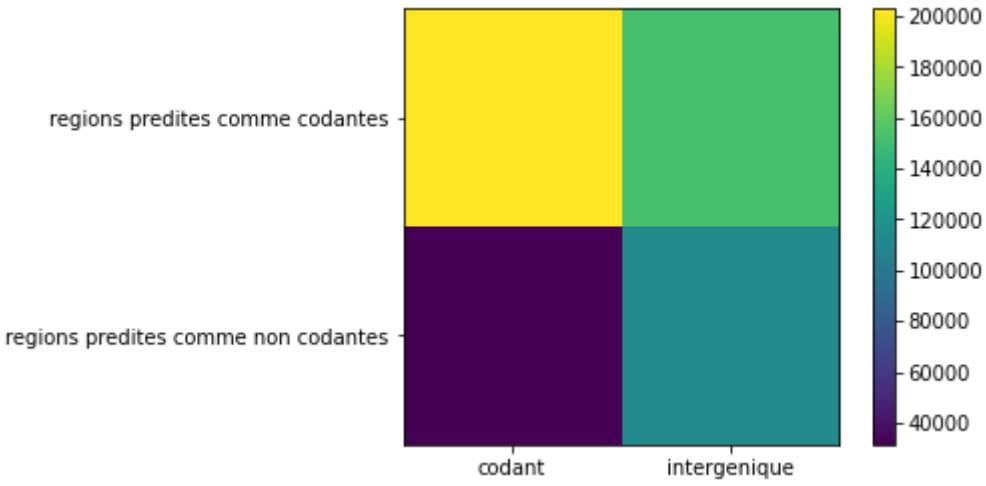
Après avoir créé la matrice de confusion, vous pouvez l'afficher en utilisant :

```
mat_conf=create_confusion_matrix(annotation_test, etat_predits)
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(['codant','intergenique']);
ax.set_yticklabels(['regions predites comme codantes','regions predites c
omme non codantes']);

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

In [104]:

```
import matplotlib.pyplot as plt

mat_conf=create_confusion_matrix(annotation_test, etat_predits)
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(['codant','intergenique']);
ax.set_yticklabels(['regions predites comme codantes','regions predites comme no
n codantes']);

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

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



Donner une interprétation. Peut-on utiliser ce modèle pour prédire la position des gènes dans le génome ?

In [105]:

```
TP,FP=mat_conf[0]
FN,TN=mat_conf[1]
```

on ne peut pas utiliser ce modèle pour prédire les gènes dans le génomes car le nombre de faux positifs est beaucoup trop grand

on veut que la précision et le recall soit grand pour dire qu'un modèle est bon

In [107]:

```
precision=TP/(TP+FP)
recall=TP/(TP+FN)
accuracy=(TP+TN)/len(annotation_test)
print(precision)
print(recall)
print(accuracy)
```

```
0.5704886953684483
0.8657156638025602
0.631682
```

On observe bien que la précision n'est pas très bonne, elle est de 57%

Question 5 : Génération de nouvelles séquences

En utilisant le modèle $\lambda = \{Pi, A, B\}$, créer `create_seq(N,Pi,A,B)` une fonction permettant de générer :

- une séquence d'états cachés
- une sequence d'observations.

In [111]:

```
# on remet la fonction de tirage aléatoire d'états du tme précédent
import random
def tirage( sc):
    #somme cumulée
    cum_sum=sc.cumsum()
    val=random.random()
    i=0
    while(val>cum_sum[i]):
        i+=1
    return i
```

In [114]:

```
def create_seq(N,Pi,A,B):
    '''
    Produire N états cachés en utilisant Pi et A

    et pour chaque état cachés produire une observation en utilisant B
    '''
    ## votre code
    # S: séquence d'états cachés
    S=np.zeros((N,1))
    # X sequence d'observations
    X=np.zeros((N,1))
    # on tire d'abord l'état initial
    S[0]=tirage(Pi)
    for i in range(N):
        S[i]=tirage(A[np.int(S[0]),:])
        X[i]=tirage(B[np.int(S[i]),:])
    return S, X
    pass
```

In [116]:

```
S,X=create_seq(4,Pi,A,B)
print(S)
print(X)
```

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

Question 6 : Construction d'un nouveau modèle

Évaluons maintenant si cela s'améliore en prenant en compte les frontières des gènes en construisant un modèle avec codon start et codon stop. On veut maintenant d'intégrer l'information complémentaire qui dit qu'un gène commence "toujours" par un codon start et finit "toujours" par un codon stop avec le graphe de transition ci-dessous.

On considère donc maintenant un modèle à 12 états cachés.

- Écrivez la matrice de transition correspondante, en mettant les probabilités de transition entre lettres pour les codons stop à 0.5.
- Adaptez la matrice des émissions pour tous les états du modèle. Vous pouvez réutiliser la matrice B, calculée précédement. Les états correspondant au codons stop n'émettrons qu'une seule lettre avec une probabilité 1. Pour le codon start, on sait que les proportions sont les suivantes:
  - ATG : 83%,
  - GTG: 14%,
  - TTG: 3%

```
Pi2 = np.array(   [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ])  ##on commence e
ncore dans l'intergenique
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 = ...
```

Évaluez les performances du nouveau modèle en faisant de nouvelles predictions d'annotation pour genome_test, et comparez les avec le modèle précédent.

```
etat_predits2=viterbi(genome_test,Pi2,A2,B2)
etat_predits2[etat_predits2!=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()
```

Calculer la matrice de confusion avec les nouvelles prédictions.

In [ ]: