Projet Modèles de Markov Cachés (HMM)

B.CALLARD
L.FOSSE

Contents

I Introduction . . . 2
II Modules utilisés . . . 2
III Modèle file Rouge . . . 2
IV Simulation d'un modèle markovien caché . . . 3
V Algorithme de Viterbi. Recherche d'un chemin dans un treillis . . . 4
V.1 une première fonction treillis (une approche naïve). . . . 4
V.2 Viterbi . . . 6
VI Fonction treillis générique . . . 10
VII Techniques supplémentaire pour trouver la chaine. . . . 12
VII.1 L'algorithme forward . . . 12
VII.2 Méthode de scaling. . . . 14
VII.3 Recherche par les quantités γ . . . 15
VIII Comparaison des methodes. . . . 17
IX Etude des méthodes en fonction du nombre d'observation . . . 20
X Choix des méthodes dans des cas concrets . . . 22
XI Conclusion . . . 22

I Introduction

Les processus Markoviens sont des processus dynamiques à évolution probabiliste, donc la dynamique suit le principe suivant : la connaissance de l'état actuel conditionne totalement l'état futur. Nous avons beaucoup étudié les processus Markoviens à espace d'état discrets, nous avons cherché à comprendre leur dynamique, à les simuler. Ces processus permettent de simuler de nombreux phénomènes connus allant de la simple problématique de gestion de stocks en entreprise au classement des pages web par google. Cependant le monde dans lequel nous vivons n'est pas discret et beaucoup de situations ne se résument pas à quelques états, mais plutôt à de nombreuses variables continues, pourtant leur dynamique est celle d'un processus Markovien. Dans ces processus nous observons des valeurs issues de mesures ou de simples relevés (ex : température, pression atmosphérique, ...) et derrière ces valeurs peuvent se cacher des états, des valeurs discrètes (ex : beau temps, mauvais temps, ...) on parle alors de processus Markoviens cachés. Ce sont ces processus que nous allons étudier à travers ce rapport en se concentrant sur la problématique suivante :

Comment retrouver les états de la chaîne de Markov, qui se cachent derrière une séquence d'observation ?

Nous verrons plusieurs méthodes d'estimation, allons les confronter voir leurs points forts et leurs défauts, et essayer de comprendre les situations les plus adaptées à chacune de ces méthodes.

II Modules utilisés

Pour ce projet, nous avons importé certains packages classique pour faciliter les manipulations graphiques et matricielles. Ils sont directement disponible si on utilise la distribution **Anaconda** de python.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random as rd
from random import randint
%matplotlib inline
```

III Modèle file Rouge

Afin de tester nos différentes fonctions, nous allons utiliser un modèle "jouet" sur lequel nous connaissons certains résultats théoriques. Un modèle de Markov est la donnée de :

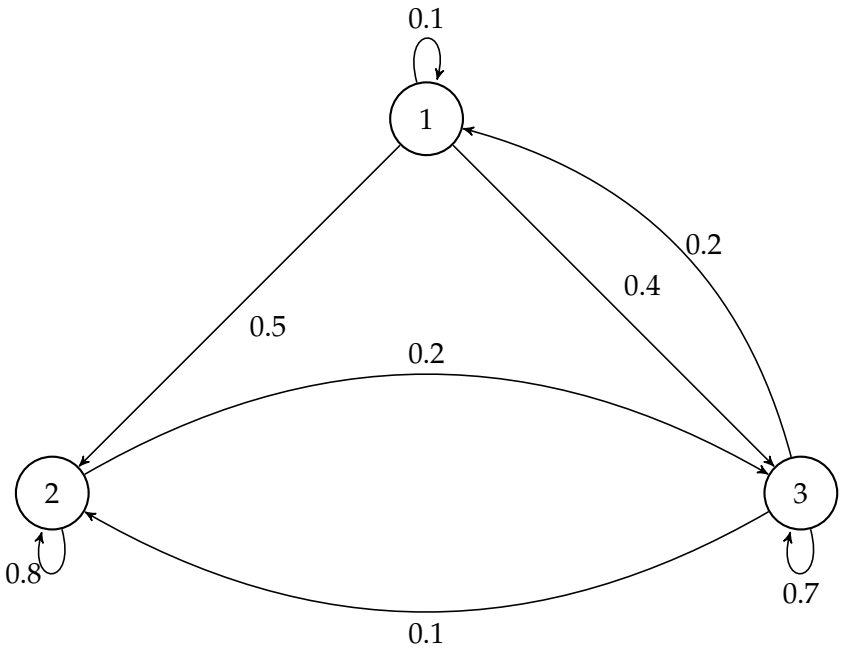
- Une mesure de probabilité π qui est la loi qui nous permet de simuler le premier état de la chaîne.
- Une matrice stochastique de transition A qui est la matrice qui définit la topologie de la chaîne de Markov.

  $A_{i,j} = \mathbb{P}\{X_{t+1} = j | X_t = i\} \ \forall t \in \mathbb{N}$
- Une matrice B, qui définit la dynamique des observations.

  $B_{i,j} = \mathbb{P}\{Y_t = y_j | X_t = i\} \ \forall t \in \mathbb{N}$

[2]:
```
A = np.array([[0.1,0.5,0.4],[0,0.8,0.2],[0.2,0.1,0.7]])
B = np.array([[0.3,0.7],[0.7,0.3],[0.5,0.5]])
pi = np.array([0.5,0.5,0])
y = [0,1,1,0]
```

La chaîne de Markov que nous utiliserons ici possède la topologie suivante :



Ce qui vient se rajouter en plus dans un modèle de Markov caché, c'est la séquence d'observation, cette séquence d'observation est dirigée par la matrice B :

$$B = \begin{pmatrix} 0.3 & 0.7 \\ 0.7 & 0.3 \\ 0.5 & 0.5 \end{pmatrix}$$

IV Simulation d'un modèle markovien caché

Pour simuler un modèle Markovien caché, il y a deux parties qui entrent en jeu.

- Dans un premier temps nous avons une partie simulation de chaîne de Markov. En effet nous allons simuler les états d'une chaîne de Markov dont la dynamique est donnée par le vecteur π et la matrice A.
- Une fois cette chaîne de Markov simulée, nous allons simuler les observations de la chaîne de Markov grâce à la matrice B, donc le vecteur y qui est la partie que l'on observe dans la pratique.
- Cette séparation des tâches est possible grâce aux hypothèses d'indépendance que nous faisons sur le modèle. En effet ici nous supposons que la simulation d'une observation à un instant t ne dépend que de l'état de la chaîne au même instant.

[3]:
```
def MM_sampling(pi,A,N,lt) :
    states = list(range(N)) # etats de la chaine de markov.
    X = [0]*lt
    X[0] = rd.choices(states,weights = pi, k = 1)[0]
    curr = X[0]

    for i in range(1,lt):
        w = A[curr]
        X[i] = rd.choices(states,weights = w, k = 1)[0]
        curr = X[i]
    return(X)
```

[4]:
```
def HMM_sampling (pi,A,B,N,obs,lt) :
    X = MM_sampling(pi,A,N,lt)
    Y = []
    for x in X :
        Y += rd.choices(obs,weights = B[x],k=1)[0]
    return({'obs' :Y ,'states' :X})
```

[5]:
```
HMM_sampling (pi,A,B,3,['a','b'],10)
```

[5]:
```
{'obs': ['b', 'a', 'a', 'a', 'b', 'a', 'b', 'a', 'a', 'a'],
 'states': [1, 1, 1, 1, 2, 2, 0, 1, 1, 2]}
```

V Algorithme de Viterbi. Recherche d'un chemin dans un treillis

Avant de commencer nous allons introduire brièvement la notion de treillis pour éviter tout malentendu. Dans ce projet, un treillis H représentera simplement une matrice.

Ici les colonnes de la matrice seront tous les pas de temps (la colonne 0 correspondant à l'observation pour t=0, et ainsi des suite) et les colonnes représenteront les différents états de la chaine de Markov. Dans les sections qui suivent nous allons proposer différentes méthodes pour remplir un treillis et trouver un chemin (donc une suite d'états de la chaîne de Markov) dans ce treillis.

V.1 une première fonction treillis (une approche naïve).

cette première fonction **highlight_cell** permet dans le treillis de venir entourer certaine case de façon à pouvoir observer un chemin. C'est un simple affichage graphique.

[6]:
```
def highlight_cell(x,y, ax=None, **kwargs):
    rect = plt.Rectangle((x-.5, y-.5), 1,1, fill=False, **kwargs)
    ax = ax or plt.gca()
    ax.add_patch(rect)
    return rect
```

fonction treillis Ici nous allons créer un treillis H tel que on ait : $H(i,t) = \mathbb{P}\{y_t|X_t = i\}$. Dans ce treillis nous allons chercher pour chaque colonne, la ligne i qui maximise la quantité $H(i,t)$. Cette première approche dite naïve nous donnera ainsi une première estimation de la séquence d'états.

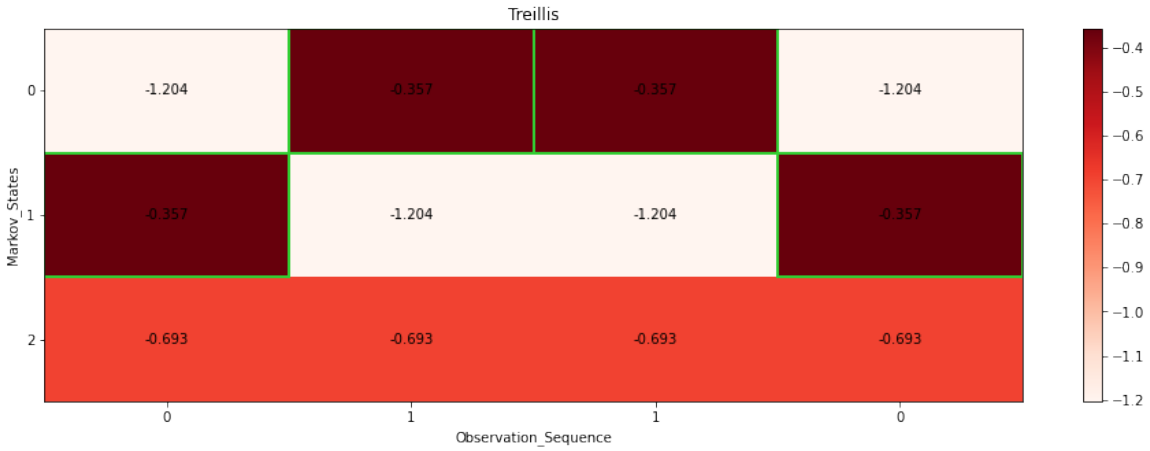
[7]:
```
def treillis (y,pi,b,A,N,Larg,Haut) :
    ### Matrice H ###
    H = np.zeros((N,len(y)))

    for j in range(len(y)):
        H[:,j] = np.log(B[:,y[j]])

    trace_1 = [0]*len(y)

    for i in range(len(trace_1)):
        trace_1[i] = int(np.argmax(H[:,i]))
```

```
    ### Paramètres graphiques ###
    fig = plt.figure(figsize=(Larg,Haut))
    plt.imshow(H,aspect = 'auto',cmap='Reds')
    plt.title("Treillis")

    x_label_list = y
    y_label_list = list(range(N))

    plt.xlabel('Observation_Sequence')
    plt.ylabel('Markov_States')

    ax = plt.gca()
    ax.set_xticks(range(len(y)))
    ax.set_yticks(range(N))
    ax.set_xticklabels(x_label_list)

    ### question 2 tracé du chemin naif

    for i in range(len(trace_1)):
        x_k = i
        y_k = trace_1[i]
        highlight_cell(x_k,y_k, color="limegreen", linewidth=2)



    ### Ajout de text sur le treillis ###
    for x_index in range(len(y)) :
        for y_index in range(N) :
            label = round(H[y_index,x_index],3)
            plt.text(x_index, y_index, label, color='black', ha='center',␣
↪va='center')



    ### Légende en colorbar pour le treillis ###
    plt.colorbar()
```

test fonction treillis

[8]:
```
print(y)
treillis(y,pi,B,A,3,16,5)
```

```
[0, 1, 1, 0]
```



Observations et remarques :

- Dans l'exemple choisi on peut voir ainsi que la séquence de Markov d'état choisi avec cette première technique est la suivante : 1,0,0,1. Cependant cette première technique ne permet pas de se rendre compte de la notion de chaînage entre les différents états de la chaîne de Markov. Nous avons simplement pris à chaque instant l'état qui maximisait la probabilité d'obtenir l'observation du temps de la chaîne d'observation. Nous n'avons en aucun cas vérifié si dans notre chaîne obtenue il était possible de faire des transitions entre les différents états, et si cette transition est possible, est-elle peu probable ou au contraire quasiment presque sur. Bref nous n'avons pas pris en compte le caractère Markovien du modèle. Nous avons simplement pris des décisions locales à chaque pas de temps sans regarder le passé ou l'avenir de la chaîne.
- Il nous faut ainsi améliorer cette première méthode en faisant apparaître la notion de chaînage dans notre calcul. L'algorithme de **Viterbi** est sûrement l'algorithme le plus connu qui permet de résoudre ce problème.

V.2 Viterbi

Algorithme de Viterbi : L'algorithme de Viterbi permet de venir chercher la séquence $\{X1, \ldots, X_n\}$, qui étant donné un modèle $\lambda_N := (\pi, A, B)$ maximise la quantité :

$$\boxed{\mathbb{P}_{\lambda_N}\{X_n, \ldots, X_1, Y_n, \ldots, Y_1\}}$$

On cherche la séquence d'états, qui est la plus **vraissemblante**, par rapport à un ensemble d'observations données. Cet méthode contrairement à la méthode précédente va venir prendre en compte le caractère **Markovien** de la chaîne et ainsi utiliser la topologie de la chaîne pour pouvoir faire des transitions entre les états.

Pour résoudre ce problème on utilise la programmation dynamique. La programmation dynamique, regroupe un ensemble de techniques algorithmiques visant à résoudre des problèmes d'optimisation, le principe est de résoudre le problème pour de petites instances et de propager ces solutions intermédiaires pour trouver des solutions correspondant à des instances plus grandes, à la manière de l'algorithme de **Bellman-ford** pour trouver le plus court chemin dans un graphe.

Ici on peut aisément se rendre compte qu'il s'agit d'un problème soluble à l'aide de programmation dynamique en exprimant le problème mais sous sa forme logarithmique. Composer par une fonction croissante ne change en rien le problème d'optimisation que nous avons et dans le cas des probabilités permet même de le simplifier car on a une transformation des produits en sommes. Nous avons :

$$ln(\mathbb{P}\{x_n,\dots,x_1,y_n,\dots,y_1\}) = ln(\pi(x_1)) + ln(B_{x_1,y_1}) + \sum_{t=2}^{n}(ln(A_{t-1,t}) + ln(B_{x_t,y_t})$$

On définit la quantité :

$$H(i,t) = \max_{x_1,\dots,x_n} ln(\mathbb{P}\{y_1,\dots,y_t,x_1,\dots,x_t = i\})$$

Cette expression va nous aider à exprimer la solution de notre problème, en effet cette expression respecte l'équation suivante :

$$\boxed{H(i,t) = \max_j(H(j,t-1) + ln(A_{j,i}) + ln(B_{i,y_t})}$$

Ici on obtient clairement une expression de type programmation dynamique, en ayant un problème qui s'exprime en fonction des solution des instances plus petites. Dans l'algorithme on aura ainsi deux parties.

- Une partie de **propagation** qui va ainsi nous servir à trouver la solution.
- Une partie de **backtracking** qui va nous permettre ainsi de retrouver le chemin qui nous a permis de trouver cette solution

[9]:
```
def viterbi(y,pi,b,A,N) :

    T = len(y) # c'est le temps final

    H = np.ones((N,T))
    B = np.zeros((N,T),dtype = int)

    ### Init ###

    for i in range(N):
        # ici on prend la valeur inf pour les log de 0
        H[i][0] = np.log(pi[i])+np.log(b[i][y[0]])


    ### Propagation ###
    for t in range(1,T):
        for i in range(N):
            row = [H[j][t-1]+np.log(A[j][i])+np.log(b[i][y[t]]) for j in range(N)]
            H[i][t] = max(row)
            B[i][t] = int(np.argmax(row))


    x = [0]*T
    cp =0

    col = [H[i][T-1] for i in range(N)]
    mv = max(col)
    x[T-1] = int(np.argmax(col))



    for t in range(T-2,-1,-1) :
```

```
        x[t] = int(B[x[t+1]][t+1])

    return({"H" : H ,"B" : B,"mv" : round(mv,1), "states":x})
```

Test de l'algorithme de Viterbi Dans un premier temps nous allons tester l'algorithme de Viterbi, avec l'exemple donnée en préambule. En effet pour cet exemple, théoriquement l'algorithme de Viterbi est censé retourner :

- la chaine [1,1,1,1]
- un score de Viterbi : $\max_i H(i,T) \approx -4.5$

[10]:
```
d = viterbi(y,pi,B,A,3)
print("états de la chaine (estimation par viterbi) : {}".format(d['states']))
print("log vraissemblance (score de viterbi) : {}".format(d['mv']))
```

```
états de la chaine (estimation par viterbi) : [1, 1, 1, 1]
log vraissemblance (score de viterbi) : -4.5
```

Ainsi avec notre fonction de Viterbi, on retrouve bien les résultats attendus pour cet exemple.

Comparaison entre la chaine trouvé par viterbi et la chaine réelle (sampling) Il peut être intéressant de voir les différences qu'il y a entre la chaine réelle qui se cache derrière une séquence d'observation et la chaîne la plus vraissemblante, calculé à partir de l'algorithme de Viterbi. C'est que nous allons essayer d'observer dans les séquences qui viennent.

[11]:
```
lt = 5
obs = ['a','b']
smp_1 = HMM_sampling (pi,A,B,3,obs,lt)
y_2 = [0 if x =='a' else 1 for x in smp_1['obs']]
d = viterbi(y_2,pi,B,A,3)
print(d['states'])
print(smp_1['states'])
```

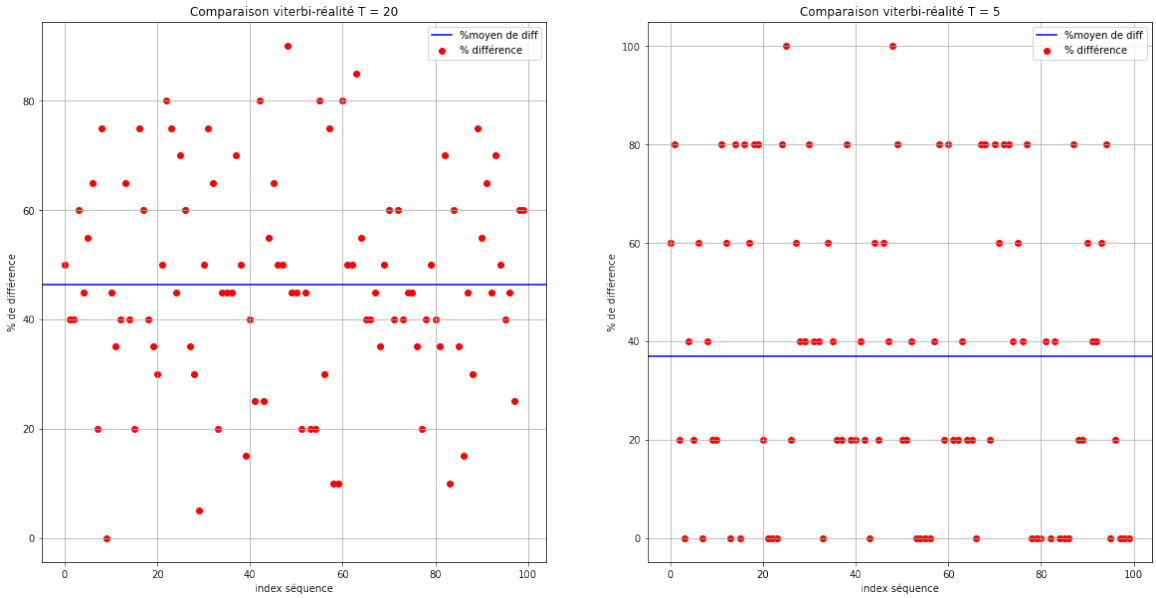
```
[0, 1, 1, 1, 1]
[0, 1, 1, 1, 1]
```

mise en forme graphique.

[12]:
```
T1 = 20
T2 = 5
L = 100 # nombre de comparaisons.

obs = ['a','b'] # espace d'observation

pr_dif1 = [0]*L # vecteur contenant les écarts entre les états.
pr_dif2 = [0]*L


for i in range(L) :

    d1 = HMM_sampling(pi,A,B,3,obs,T1)
    y_spl1 = [0 if x =='a' else 1 for x in d1['obs']]
    x_th1 = np.array(d1['states'])
    x_vit1 = np.array(viterbi(y_spl1,pi,B,A,3)['states'])
```

```
    s1 = [0 if x == 0 else 1 for x in x_th1-x_vit1]
    pr_dif1[i] = sum(s1)/T1*100


    d2 = HMM_sampling(pi,A,B,3,obs,T2)

    y_spl2 = [0 if x =='a' else 1 for x in d2['obs']]
    x_th2 = np.array(d2['states'])
    x_vit2 = np.array(viterbi(y_spl2,pi,B,A,3)['states'])


    s2 = [0 if x == 0 else 1 for x in x_th2-x_vit2]

    pr_dif2[i] = sum(s2)/T2*100
```

[13]:
```
fig = plt.figure(figsize=(20,10))


plt.subplot(1,2,1)
plt.scatter(list(range(L)), pr_dif1, c = 'red',label = '% différence')
plt.axhline(y=np.mean(pr_dif1),c='blue',label = '%moyen de diff')
plt.ylabel('% de différence')
plt.xlabel('index séquence')
plt.title('Comparaison viterbi-réalité T = 20')
plt.legend()
plt.grid()

plt.subplot(1,2,2)
plt.scatter(list(range(L)), pr_dif2, c = 'red',label = '% différence')
plt.axhline(y=np.mean(pr_dif2),c='blue',label = '%moyen de diff')
plt.ylabel('% de différence')
plt.xlabel('index séquence')
plt.title('Comparaison viterbi-réalité T = 5')
plt.legend()
plt.grid()
```



Observations et remarques :

- On peut voir que entre la chaîne réelle et la chaîne obtenue par l'algorithme de Viterbi on a des différences qui sont notables. En effet nous n'avons pas une tendance qui se dégage et qui montre que l'algorithme de Viterbi détermine une chaîne qui très proche de la chaîne réelle.
- Cette observation permet de prendre du recule sur la méthode de Viterbi, en effet même si la méthode trouve la chaîne la plus vraissemblante étant donnée une séquence d'observation, il ne faut pas en oublier le modèle pour autant. Ici nous avons des séquences d'observation de taille T=20 sur le premier graphique, ainsi le nombre de séquences d'états possible (probabilité non nulle) qui correspondent à chaque séquence d'observation est considérable. Ainsi même si on trouve la plus vraissemblante, sa probabilité sera très faible, autrement dit la probabilité que **ce ne soit pas cette chaîne** est très proche de 1. Il n'est ainsi pas choquant d'observer de grandes différences entre la chaîne de Viterbi et la chaîne réelle qui se cache derrière.

VI Fonction treillis générique

Nous allons dans cette partie écrire une fonction **treillis_gen** qui reprendre la structure de notre première fonction treillis mais qui sera cette fois-ci plus générique. En effet cette nouvelle fonction aura comme paramètre d'entré :

- un entier **N** qui permettra de représenter les états de la chaîne de Markov.
- une suite **obs** d'observation du modèle de Markov caché.
- une suite **trace** d'états de la chaîne de Markov pour tracer le chemin dans le treillis
- une matrice **mat** qui sera la matrice du treillis
- un paramètre **title** pour donner un titre à notre treillis
- deux paramètres graphiques **Larg** et **haut**, qui permettront de gérer la taille du treillis pour son affichage.

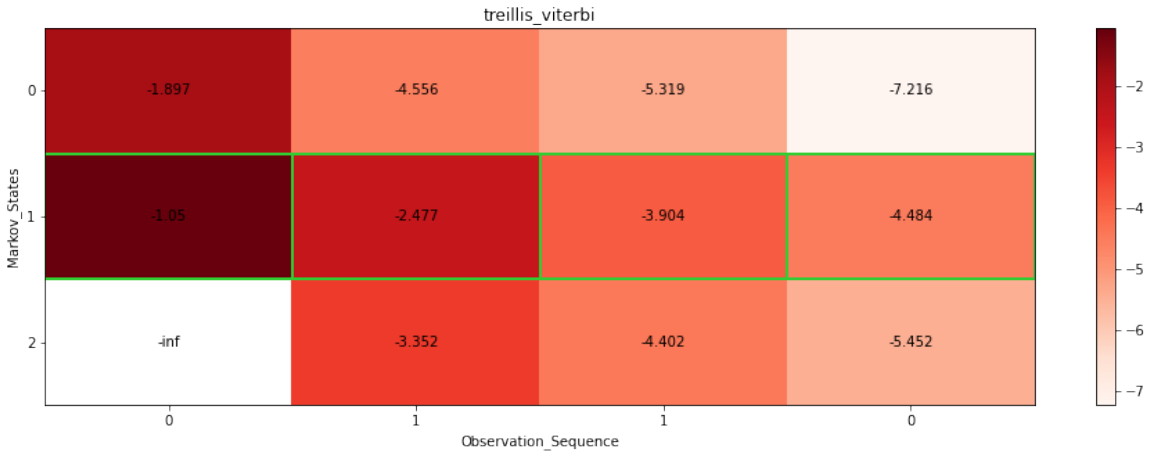
[14]:
```
def treillis_gen(N,obs,trace,mat,title,Larg,Haut) :

    ### Paramètres graphiques ###
    fig = plt.figure(figsize=(Larg,Haut))
    plt.imshow(mat,aspect = 'auto',cmap='Reds')
    txt = "treillis_"+title
    plt.title(txt)

    x_label_list = obs
    y_label_list = list(range(N))

    plt.xlabel('Observation_Sequence')
    plt.ylabel('Markov_States')

    ax = plt.gca()
    ax.set_xticks(range(len(obs)))
    ax.set_yticks(range(N))
    ax.set_xticklabels(x_label_list)

    ### question 2 tracé du chemin naif

    for i in range(len(trace)):
        x_k = i
```

```
        y_k = trace[i]
        highlight_cell(x_k,y_k, color="limegreen", linewidth=2)



    ### Ajout de text sur le treillis ###
    for x_index in range(len(obs)) :
        for y_index in range(N) :
            label = round(mat[y_index,x_index],3)
            plt.text(x_index, y_index, label, color= 'black', ha='center',␣
↪va='center')



    ### Légende en colorbar pour le treillis ###
    plt.colorbar()
```

Le but de cette fonction est en faite de pouvoir créer n'importe quel treillis à partir de résultats issues des différents algorithmes pour trouver une chemin dans le treillis. Par exemple on peut créer une fonction **treillis_viterbi**, qui va créer le treillis basé sur la matrice H qui contient les scores de l'algorithme de Viterbi.

[15]:
```
def treillis_viterbi(y,pi,b,A,N,Larg,Haut) :

    # scores obtenus avec Viterbi.
    d = viterbi(y,pi,b,A,N)

    mat = d['H']
    trace = d['states']
    treillis_gen(N,y,trace,mat,"viterbi",Larg,Haut)
```

[16]:
```
treillis_viterbi(y,pi,B,A,3,16,5)
```



VII Techniques supplémentaire pour trouver la chaine.

VII.1 L'algorithme forward

Remplisage de la matrice α $\alpha_{i,t} = \mathbb{P}\{y_1, \ldots, y_t, X_t = i\}$

[17]:
```
def forward(pi,A,B,y,N) :
    T = len(y)
    alpha = np.zeros((N,T))

    ### initialisation ###
    for i in range(N) :
        alpha[i,0] = pi[i]*B[i,y[0]]

    ### propagation ###
    for t in range(1,T) :
        for j in range(N) :
            row = [alpha[i,t-1]*A[i][j] for i in range(N)]
            alpha[j,t] = B[j][y[t]]*sum(row)

    return(alpha)
```

Recherche du chemin dans la matrice α

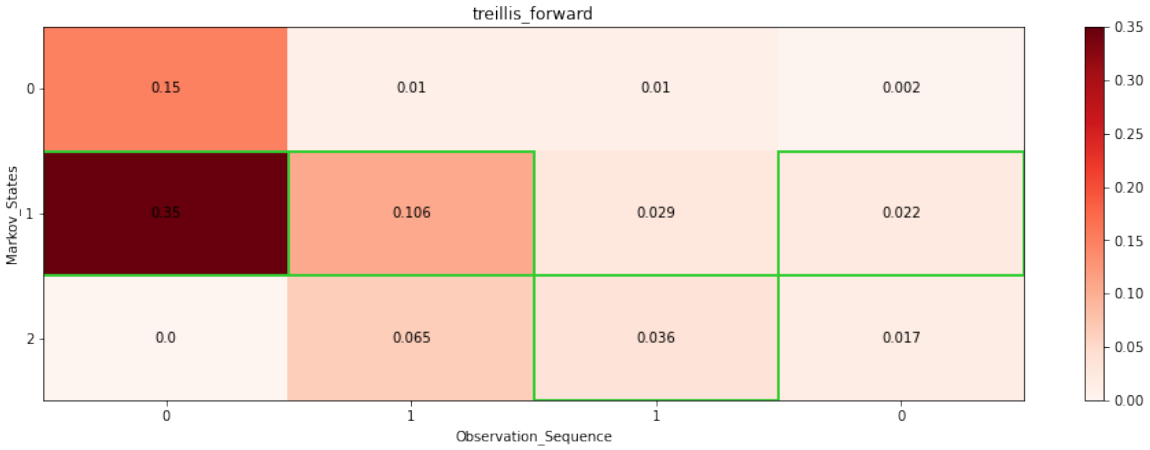
[18]:
```
def trace_forward(pi,A,B,y,N) :
    ### matrice alpha ###
    alpha = forward(pi,A,B,y,N)

    ### recherche du chemin dans la matrice alpha ###
    trace = [0]*len(y)

    for t in range(len(y)):
        trace[t] = int(np.argmax(alpha[:,t]))

    return(trace)
```

Treillis pour α

[19]:
```
def treillis_alpha(pi,A,B,y,N,Larg,Haut) :
    mat = forward(pi,A,B,y,N)
    trace = trace_forward(pi,A,B,y,N)

    ### tracé du treillis ###
    treillis_gen(N,y,trace,mat,"forward",Larg,Haut)
```

Test treillis α

[20]:
```
treillis_alpha(pi,A,B,y,3,16,5)
```



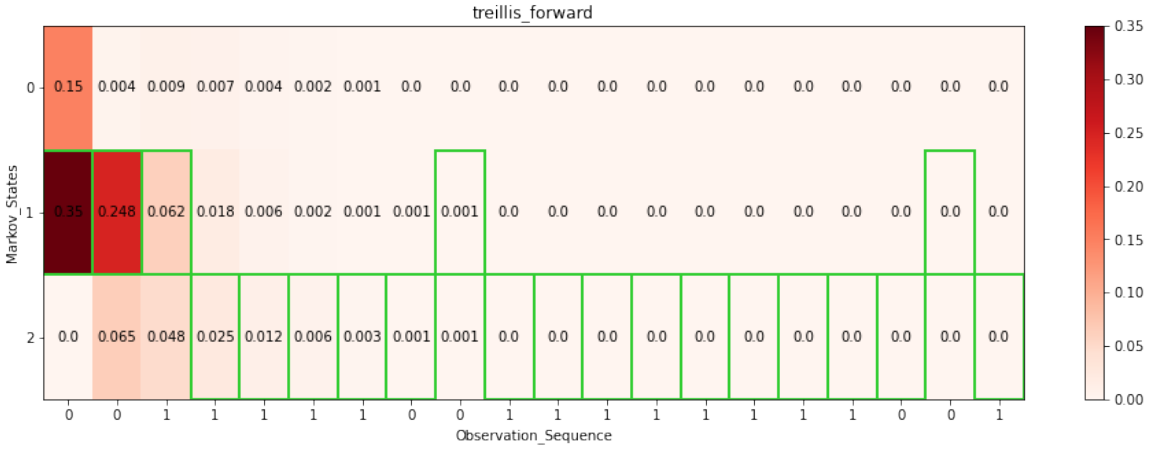
Remarques sur la méthode froward :

- La méthode forward est une méthode intéressante car elle est très rapide à écrire et sa complexité est assez faible dû à son caractère dynamique. En effet la complexité de l'algorithme forward est polynomiale et donnée par $\mathcal{O}(T \times N^2)$. Cependant elle ne prend en aucun cas le caractère Markovien de la chaine de markov. En effet cette méthode ne permet que de prendre des décisions qui sont dites locales, c'est à dire que pour un temps t donné on ne s'intéresse qu'à trouver l'état qui maximise la probabilité d'observer une observation particulière y_t. Ce genre de raisonnement se retrouve dans le domaine de la météorologie.
- La méthode forward sera ainsi une méthode qui sera plus utile dans des cas où seules les décisions locales sont importantes et ainsi ne pas respecter la topologie de la chaîne à certain moment n'importe pas réellement.

observation du comportement en temps long

[21]:
```
y_k = [randint(0,1) for i in range(20)]
```

[22]:
```
treillis_alpha(pi,A,B,y_k,3,16,5)
```



Observations et remarques :

- Ici on peut voir que pour une valeur plus grande de T (T = 20), à partir d'un certain rang, toutes les valeurs présentes dans le treillis sont égales à 0. Ainsi à partir d'un certain rang les

valeurs que nous calculons sont très proches de la précision machine. Il est très dangereux en informatique de travailler avec des valeurs proches du 0 machine, les résultats obtenus avec ces valeurs sont peut-être entachés d'erreurs et ainsi donnent des résultats faux. Ici les conséquences ne sont pas très importantes, car la séquence d'observation est encore relativement courte, il faut faire attention quand la séquence d'observations devient très grande ($T > 1000$), des erreurs peuvent apparaître dû à la proximité avec la précision machine.

- La méthode de **scaling** présentée dans la section suivante, permet de pallier ce problème.

VII.2 Méthode de scaling.

La méthode scaling va venir corriger à chaque passe les probabilités en divisant par un facteur de normalisation de façon à venir palier au faite que les probabilités soient très petites (proches du zéro machine). A chaque étape, ce facteur sera le coefficient c_t

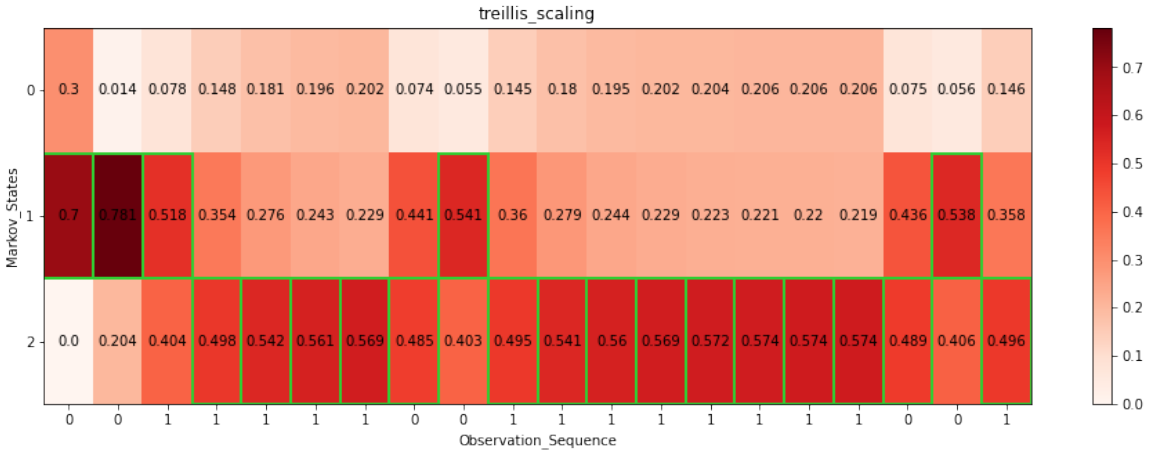
[23]:
```
def scaling_forward(pi,A,B,y,N):

    T = len(y)

    alpha_ch = np.zeros((N,T))
    alpha_p = np.zeros((N,T))

    c = [0]*T

    for i in range(N) :
        alpha_p[i,0]=pi[i]*B[i,y[0]]


    c[0] = 1/(sum(alpha_p[:,0]))

    for i in range(N) :
        alpha_ch[i,0]  = c[0]*alpha_p[i,0]

    # induction
    for t in range(1,T) :

        for i in range(N) :
            alpha_p[i,t]=sum([alpha_ch[j,t-1]*A[j,i]*B[i,y[t]] for j in range(N)])

        # scaling factor
        c[t]=1/(sum(alpha_p[:,t]))

        for i in range(N):
            alpha_ch[i,t]=c[t]*alpha_p[i,t]

    return(alpha_ch)
```

[24]:
```
def trace_scaling(pi,A,B,y,N) :
    ### matrice alpha_scaling ###
    alpha = scaling_forward(pi,A,B,y,N)

    ### recherche du chemin dans la matrice alpha ###
    trace = [0]*len(y)

    for t in range(len(y)):
```

```
        trace[t] = int(np.argmax(alpha[:,t]))

    return(trace)
```

[25]:
```
def treillis_scaling(pi,A,B,y,N,Larg,Haut):
    mat = scaling_forward(pi,A,B,y,N)
    trace = trace_scaling(pi,A,B,y,N)

    ### tracé du treillis ###
    treillis_gen(N,y,trace,mat,"scaling",Larg,Haut)
```

[26]:
```
treillis_scaling(pi,A,B,y_k,3,16,5)
```



Observations et remarques :

- On peut voir que la chaîne prédite par les quantités obtenues par la méthode scaling, sont exactement les mêmes que dans la chaîne précédente. Ceci est en accord avec la théorie, qui énonce que les deux estimations sont équivalentes, on a simplement un facteur de scaling qui vient augmenter les scores obtenus dans le treillis.

VII.3 Recherche par les quantités γ

Une autre technique pour trouver une chaine de markov correspondant à des observations **y** est de passer par le calcul des quantités γ, avec pour définition suivante :

$$\boxed{\gamma_{i,t} = \mathbb{P}\{X_t = i | y_1, \ldots, y_T\}}$$

Ces nombres s'obtiennent assez facilement à partir des quantités α calculées à partir de l'algorithme de **forward** et des quantités β calculé à partir de l'algorithme **backward**. Les quantités α ont été introduites dans la section précédente. Les quantités β quant à elle sont définit de la façon suivante :

$$\beta_{i,t} = \mathbb{P}\{y_{t+1}, \ldots, y_T | X_t = i\}$$

Une fois de plus, ces quantités β peuvent se calculer via de la programmation dynamique mais cette fois ci de façon inversée par rapport aux quantités α, on va faire une propagation dans les temps

décroissants. Les quantités β respectent l'équation suivante :

$$\beta_{i,t} = \sum_{j=1}^{N} A_{i,j} B_{j,y_{t+1}} \beta_{j,t+1} \text{ avec } \beta_{i,T} = 1 \ \forall i \in \{0, \dots, N\}$$

Tout ceci se met en forme dans l'algorithme backaward qui est présenté ci dessous.

[27]:
```
def backward(pi,A,B,y,N):
    T = len(y)
    beta = np.zeros((N,T)) # matrice beta à retourner à la fin de l'algorithme.

    ### init ###
    for i in range(N):
        beta[i,T-1] = 1

    ### propagation ###
    for t in range(T-2,-1,-1):
        for i in range(N):
            row = [A[i,j]*B[j,y[t+1]]*beta[j,t+1] for j in range(N)]
            beta[i,t] = sum(row)

    return(beta)
```

Une fois les quantités β et α calculées on peut calculer les quantités γ via l'équation suivante :

$$\boxed{\gamma_{i,t} = \frac{\alpha_{i,t}\beta_{i,t}}{\sum_{j=1}^{N} \alpha_{j,t}\beta_{j,t}}}$$

De plus on a le résultat suivant :

$$\forall t \in \{1, \dots, T\} \ : \ \sum_{j=1}^{N} \alpha_{j,t}\beta_{j,t} = \mathbb{P}\{y_1, \dots, y_T\}$$

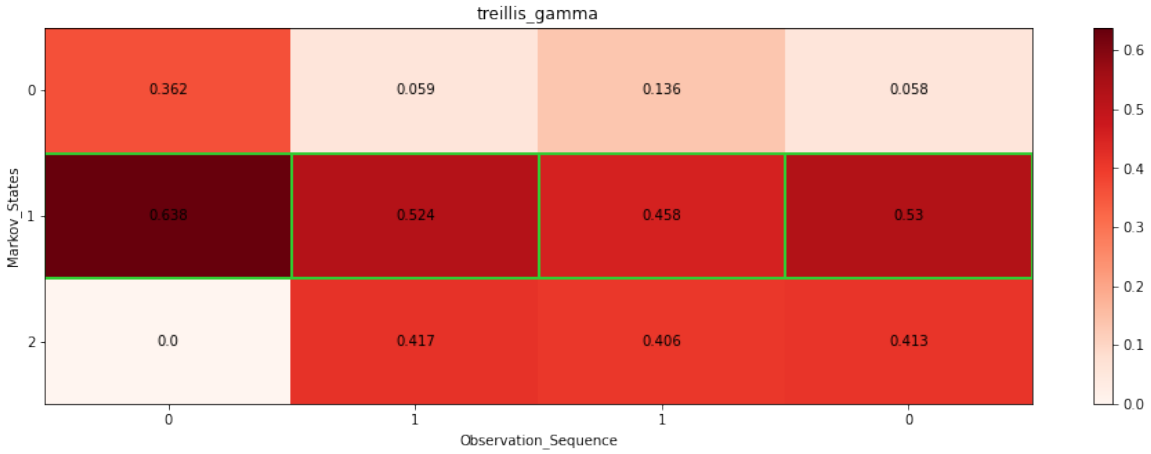
Ainsi dans les calculs des quantités γ, le dénominateur sera le même sur toutes les quantités que l'on calcul. Ceci sera ainsi intéressant en termes de programmation, en effet nous aurons simplement à former la matrice produit entre α et β, et ensuite venir diviser par le dénominateur que l'on aura calculé pour le temps initiale $t = 1$.

[28]:
```
def gamma (pi,A,B,y,N):
    T = len(y)
    alpha = forward(pi,A,B,y,N)
    beta = backward(pi,A,B,y,N)
    prod = alpha*beta

    # le dénominateur est le même quelque soit t ici on prend t = 0
    gamma = np.zeros((N,T))
    s = sum([prod[j,0] for j in range(N)])
    gamma = prod/s

    return(gamma)
```

[29]:
```
def trace_gamma(pi,A,B,y,N) :
    mat_gamma = gamma(pi,A,B,y,N)
    trace = [0]*len(y)
    for t in range(len(y)):
        trace[t] = int(np.argmax(mat_gamma[:,t]))
    return(trace)
```

[30]:
```
def treillis_gamma(pi,A,B,y,N,Larg,Haut) :
    mat = gamma(pi,A,B,y,N)
    trace = trace_gamma(pi,A,B,y,N)

    ### tracé du treillis ###
    treillis_gen(N,y,trace,mat,"gamma",Larg,Haut)
```

[31]:
```
treillis_gamma(pi,A,B,y,3,16,5)
```



On peut voir ici, que en passant par les quantités γ on obtient exactement la même chaine de Markov associée à notre séquence d'observation.

VIII Comparaison des methodes.

Nous avons maintenant plusieurs méthodes qui nous permettent d'estimer les états qui se cachent derrière une séquence d'observation.

Dans un premier temps, nous avons décidé de comparer les méthodes entre elles pour déceler des points communs ainsi que des différences lors de leur exécution pour des prédictions sur plusieurs chaînes de Markov cachées. Pour ce faire, nous avons créé N=100 instances d'observations aléatoires de longueur T=10. Puis nous avons exécuté les algorithmes de Viterbi, forward et gamma.

Ici on a à chaque étapes deux possibilités soit choisir que l'état sous-jacent serait 'a' ou 'b'. Nous avons à chaque étape comparer les algorithmes deux à deux. Pour cela nous avons regardé la trajectoire donnée par les algorithmes. En regardant le nombre d'état commun sur les trajectoires, nous avons pu regarder la ressemblance de deux méthodes. Ainsi sur le graphique ci-dessous, nous avons en abscisse nos 100 séquences. En ordonnée, nous avons pour chaque séquences le pourcentage de différence observé entre chaque séquence. Ainsi si le pourcentage de différence est faible alors on pourra conclure sur cette séquence que deux algorithmes ont donné les mêmes résultats.

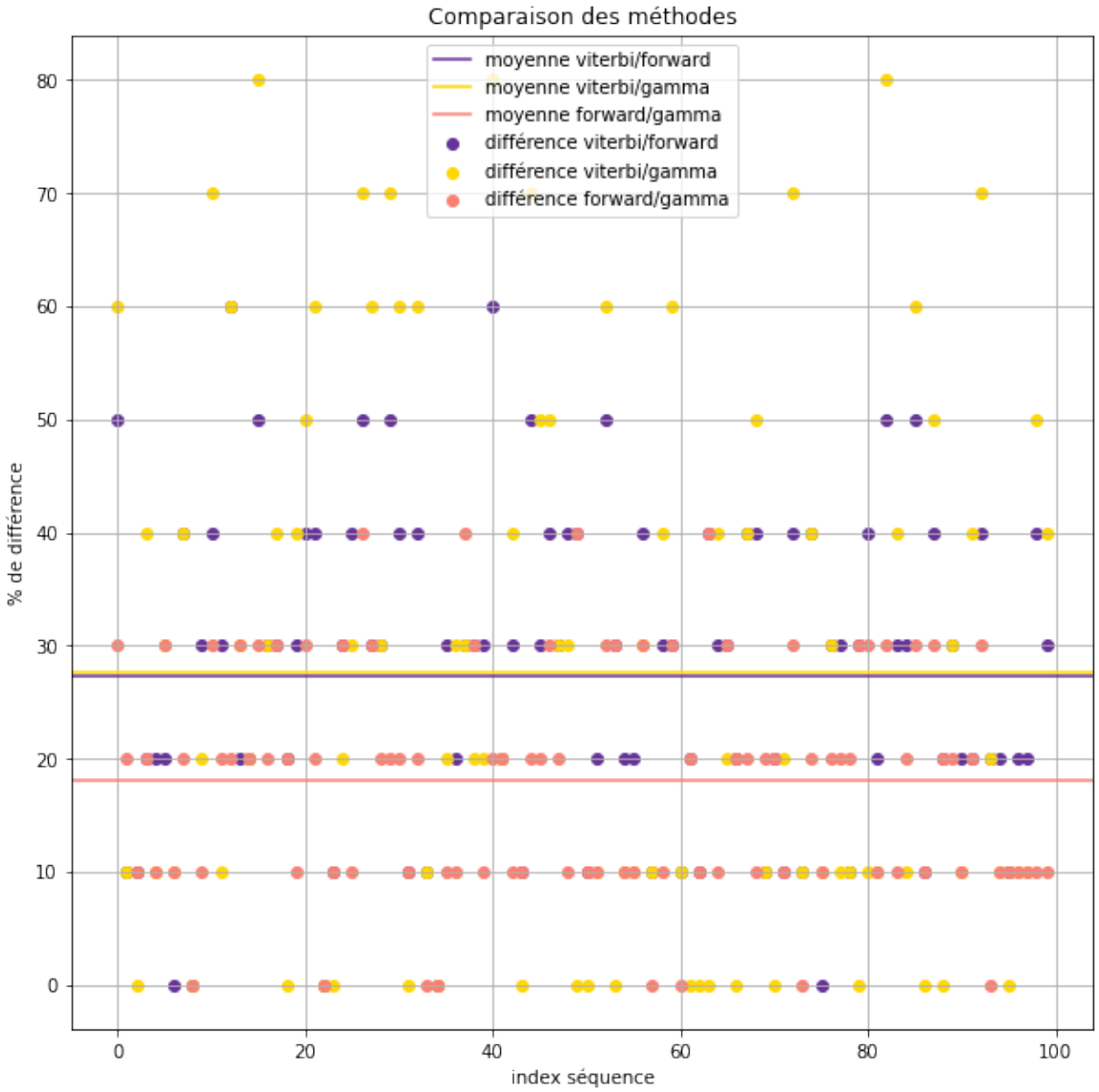
- En saumon, nous avons comparé Forward et Gamma.

- En jaune, nous avons comparé Viterbi et Gamma.
- En violet, nous avons comparé Viterbi et Forward.
- Pour se rendre compte du comportement global, nous avons aussi tracé la moyenne de ressemblance sur les 100 séquences pour chaque comparaison. En gardant le code couleur précedent.

Ici, il faut aussi comprendre que l'on ne s'intéresse pas à la véracité des résultats obtenus mais seulement de la ressemblance de deux algorithmes. Ainsi ce n'est pas parce que deux algorithmes sont proches qui seront mieux et inversement.

[32]:
```
# on simule des chaînes de
T = 10
L = 100

dif_v_a = np.array([0]*L)
dif_v_g = np.array([0]*L)
dif_a_g = np.array([0]*L)

for i in range(L):

    y_k = [randint(0,1) for i in range(T)] # sequence observation

    trace_v = np.array(viterbi(y_k,pi,B,A,3)['states'])
    trace_a = np.array(trace_forward(pi,A,B,y_k,3))
    trace_g = np.array(trace_gamma(pi,A,B,y_k,3))

    dif_v_a[i] = sum([0 if x ==0 else 1 for x in trace_v-trace_a])/T*100
    dif_v_g[i] = sum([0 if x ==0 else 1 for x in trace_v-trace_g])/T*100
    dif_a_g[i] = sum([0 if x ==0 else 1 for x in trace_a-trace_g])/T*100
```

[33]:
```
fig = plt.figure(figsize=(10,10))

plt.scatter(list(range(L)), dif_v_a, c = 'rebeccapurple',label = 'différence␣
 ↪viterbi/forward')
plt.scatter(list(range(L)), dif_v_g, c = 'gold',label = 'différence viterbi/gamma')
plt.scatter(list(range(L)), dif_a_g, c = 'salmon',label = 'différence forward/
 ↪gamma')
plt.axhline(y = np.mean(dif_v_a),c = 'rebeccapurple',label='moyenne viterbi/
 ↪forward')
plt.axhline(y = np.mean(dif_v_g),c = 'gold',label='moyenne viterbi/gamma')
plt.axhline(y = np.mean(dif_a_g),c = 'salmon',label = 'moyenne forward/gamma')
plt.ylabel('% de différence')
plt.xlabel('index séquence')
plt.title('Comparaison des méthodes')
plt.legend()
plt.grid()
```



Commentaires et observations :
Sur ce premier graphique, nous pouvons observer différents résultats :

- On peut dans un premier temps remarquer que l'estimation faite par la méthode forward et la méthode gamma, sont relativement proches. Ceci est assez cohérent, car en effet les deux méthodes se basent sur des calculs très similaires et la méthode gamma utilise les quantités forward.
- En opposition, on peut voir que la méthode de Viterbi fournit des estimations assez différentes des deux autres méthodes.
- Cependant de manière générale, sur ce graphique on ne peut pas observer de réel tendance, les points sont assez volatiles et occupent quasiment tout l'espace, preuve que globalement les méthodes fournissent des résultats différents les unes des autres.

Ceci vient une fois de plus appuyer quelque chose dont nous avons parlé. Les méthodes sont très différentes les unes des autres et fournissent des estimations différentes, mais qui suivant le contexte ont chacune leur intérêt propre.

IX Etude des méthodes en fonction du nombre d'observation

Nous avons ensuite décidé pour des tailles d'échantillons allant de 1 à 50, de prendre la moyenne (pour 100 séquences) du % de différence avec la chaîne simulée avec la chaine de Markov réellement suivie.

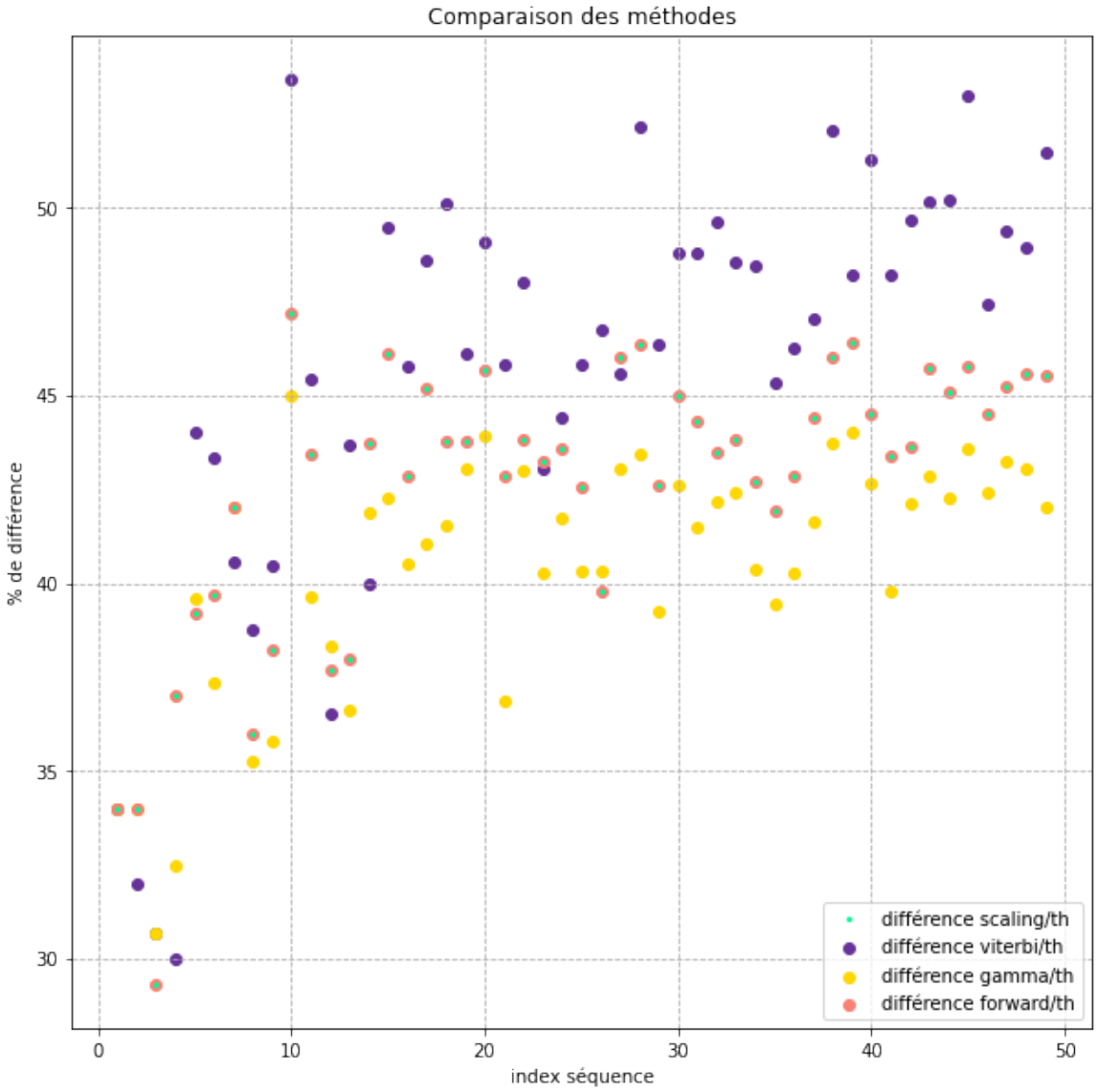
Nous avons adopté le code couleur suivant

- En saumon, nous avons comparé forward/théorique.
- En jaune, nous avons comparé gamma/théorique.
- En violet, nous avons comparé Viterbi/théorique.
- En vert, nous avons comparé scaling/théorique.

[34]:
```
list_T = np.arange(1,50,1)

v = []
g = []
a = []
s = []

obs = ['a','b']

for T in list_T :

    dif_v_th = []
    dif_g_th = []
    dif_a_th = []
    dif_s_th = []

    for j in range(50) :
        d1 = HMM_sampling(pi,A,B,3,obs,T)
        y_spl = [0 if x =='a' else 1 for x in d1['obs']]
        x_th = d1['states']

        trace_v = np.array(viterbi(y_spl,pi,B,A,3)['states'])
        trace_a = np.array(trace_forward(pi,A,B,y_spl,3))
        trace_g = np.array(trace_gamma(pi,A,B,y_spl,3))
        trace_s = np.array(trace_scaling(pi,A,B,y_spl,3))

        dif_v_th.append(sum([0 if x ==0 else 1 for x in trace_v-x_th])/T*100)
        dif_g_th.append(sum([0 if x ==0 else 1 for x in trace_g-x_th])/T*100)
        dif_a_th.append(sum([0 if x ==0 else 1 for x in trace_a-x_th])/T*100)
        dif_s_th.append(sum([0 if x ==0 else 1 for x in trace_s-x_th])/T*100)

    v.append(np.mean(dif_v_th))
    g.append(np.mean(dif_g_th))
    a.append(np.mean(dif_a_th))
    s.append(np.mean(dif_s_th))
```

[35]:
```
fig = plt.figure(figsize=(10,10))

plt.scatter(list_T, v, c = 'rebeccapurple',label = 'différence viterbi/th')
plt.scatter(list_T, g, c = 'gold',label = 'différence gamma/th')
```

```
plt.scatter(list_T, a, c = 'salmon',label = 'différence forward/th')
plt.plot(list_T, s,'ro', c = 'mediumspringgreen',label = 'différence scaling/th',ms␣
 ↪= 2)

plt.ylabel('% de différence')
plt.xlabel('index séquence')
plt.title('Comparaison des méthodes')
plt.legend()
plt.grid(True,which="both", linestyle='--')
```



Commentaires et observations :

- Ce graphique ne nous apprend pas grand-chose. En effet on peut voir que plus la chaîne d'observation grandit, plus l'estimation s'éloigne de la chaîne réelle, mais c'est un résultat déjà connu.
- Les points sont très volatiles et ne montrent pas une tendance qui se dégage, on ne voit pas de méthodes qui se rapproche plus de la théorie qu'une autre, et ceci vient une fois de plus appuyer la faite que le choix de la méthode dépend essentiellement du contexte d'étude dans lequel on se place.

X Choix des méthodes dans des cas concrets

Comme nous l'avons mentionné à plusieurs reprises le choix de la méthode dépend du contexte d'étude. Afin de choisir une méthode il faut se demander si les estimations que l'on doit faire doivent être :

- locales : C'est à dire que la topologie de la chaîne de Markov n'importe pas, seul l'instant t auquel on se situe à une importance. On retrouve ce genre de raisonnements en météorologie. En météorologie, nous avons une séquence d'observation, qui sont des vecteurs de paramètres atmosphérique (pression, température, ...) et seul le temps associé à ces observations nous importe. Le temps aux instants précédents n'a pas d'importance réel. Dans ce genre de contexte on va privilégier les algorithmes de scaling ou encore γ
- globales : Dans la prédiction de l'état suivant, les états le précédant sont importants dans la manière dont le phénomène étudié se déroule. Ainsi dans ce cas nous préférerons d'avantages l'utilisation de l'algorithme de Viterbi. Un exemple du cours qui illustre ce propos est dans l'utilisation des HMMs pour la classification et segmentation de vidéo dans le tennis. Dans ce cas, il semble intéressant d'assister le choix de la caméra. En effet, on ne peut pas diffuser les images dans n'importe quel ordre. Il y a une logique de dépendance avec le passé que l'on aimerait garder dans notre simulation.
  Dans la reconnaissance vocale, la topologie de la chaîne de Markov possède aussi un intérêt, car il y a des enchaînements de son qui sont impossibles suivants la langue dans laquelle on parle, ainsi la reconnaissance d'un son à l'instant $t-1$ va conditionner la reconnaissance à l'instant t.

XI Conclusion

A travers ce projet nous avons manipulé les chaînes de Markov cachées sous plusieurs angles. Nous avons dans un premier temps travaillé sur la simulation, et regardé comment les hypothèses d'indépendances faites sur le modèle permettent de faciliter la simulation. Le reste du projet nous a permis de comprendre comment estimer les états de la chaîne connaissant une certaine séquence d'observations. Cette opération est très intéressante et nous avons pu nous rendre compte de la difficulté à estimer proprement une chaîne de Markov. Nous avons utilisé des approches locales (forward, γ) et des approches utilisant la topologie de la chaîne (Viterbi), chacune des approches possédant son intérêt propre, qui dépend essentiellement du contexte d'étude.

Plusieurs questions restent ouvertes pour nous à l'issue de ce projet :

- Il nous reste maintenant à voir comment estimer les paramètres d'un modèle de Markov cachés, moyennant des séquences d'observations, afin de compléter nos outils sur ces modèles, qui ont certes aujourd'hui été remplacés par les réseaux de neurones récurrents, mais qui restent bien utiles dans de nombreux domaines (reconnaissance d'images, reconnaissance vocale, économétrie, ...).
- Ici nous sommes aussi limités aux HMMs discrets, mais que ce passe-t-il quand les observations ne suivent plus une loi discrète mais une loi continue comme un mélange de Gaussienne ? Comment utiliser la vraisemblance plutôt que la probabilité d'égalité ?

Répondre à ces questions nous permettrait de compléter nos connaissances sur ces modèles !