TP - Modèles de Mélanges Gaussiens

Rodrigo César COELHO FERNANDES

Expliquer et justifier les étapes de calcul de la méthode logsumexp()

La fonction reçoit un vecteur de logarithmes de probabilités et calcule leur somme de manière paramétrée.

Le paramétrage se fait en soustrayant le X_max dans le calcul au sein de la fonction exponentielle, puis renvoie la valeur de la somme des logarithmes en inversant le paramétrage.

La principale raison d'utiliser des logarithmes dans le contexte des probabilités est d'éviter de calculer avec des valeurs extrêmement petites, ce qui pourrait entraîner une perte de précision, car la multiplication ou l'addition de plusieurs petites probabilités peut donner lieu à des nombres que l'ordinateur ne peut pas représenter. À l'aide de logarithmes, nous transformons les multiplications en sommes et maintenons les nombres sur une échelle numériquement plus stable.

```
def logsumexp(X):

    X_max = max(X) # identifie la plus grande valeur dans le vecteur X
pour utiliser cette valeur comme référence pour le paramétrage
    if math.isinf(X_max):
        return -float('inf') # renvoie directement -inf, car cela ne
sert à rien de poursuivre le calcul dans ce cas

    acc = 0
    for i in range(X.shape[0]):
        acc += math.exp(X[i] - X_max) # somme des exponentielles sous
une forme paramétrée

    return math.log(acc) + X_max # applique le logarithme à la valeur
accumulée et additionne X_max à "inverser" pour ajuster les log-cotes
d'origine
```

Déduire le calcule réalisé par la méthode LogSumExp()

La fonction reçoit une matrice logarithmique de la vraisemblance, où chaque ligne représente la probabilité logarithmique qu'un échantillon soit généré par l'un des clusters. Et il renvoie un vecteur avec la somme des logarithmes des probabilités pour chaque élément.

```
def LogSumExp(Log_Vrais_Gauss):

    K,N = np.shape(Log_Vrais_Gauss) # le nombre d'éléments dans
l'échantillon

    logsomme = np.zeros(N)
    for n in range(N):
        logsomme[n] = logsumexp(Log_Vrais_Gauss[:,n]) # les logarithmes
de la vraissambance s'additionnent

    return logsomme # renvoie la somme logarithmique des probabilités
pour chaque échantillon
```

Déduire le calcule réalisé par la méthode my_GMM_predict()

La fonction effectue une prédiction de cluster basée sur le modèle de mélange gaussien (GMM), fournissant les cluster prédites et la log-vraisemblance totale.

En support, la fonction my_GMM_p_a_postiori() calcule les probabilités a posteriori de chaque point appartenant à chaque cluster grâce à la fonction.

$$P_{(C_k|x_n)} = \frac{P_{(C_k)} * P_{(X_n|C_k)}}{\sum P_{(C_j)} * P_{\left(X_n|C_j\right)}}$$

Après avoir obtenu les probabilités a posteriori, np.argmax() est utilisé pour identifier le cluster qui a la probabilité a posteriori la plus élevée pour chaque échantillon.

```
def my_GMM_predict(X,K,P,Mean,Cov):

    Proba_Clusters, LogVrais = my_GMM_p_a_posteriori(X,K,P,Mean,Cov) #
Calcule les probabilités a posteriori des données appartenant à chaque
cluster

    y = np.argmax(Proba_Clusters,axis=0) # Identifie le cluster le plus
probable pour chaque point de données en fonction des probabilités a
posteriori

    return y,LogVrais # Renvoie les étiquettes prévues et la
vraisemblance totale du journal
```

Resultados

Dans l'image (a), nous voyons les 1000 points générés aléatoirement, sur la base des matrices de probabilité, de centroïde et de covariance définies précédemment. La matrice de probabilité définit le nombre de points dans chaque cluster, tandis que la matrice de covariance influence la répartition de ces points dans l'espace autour de leur centroïde.

De même, dans l'image (b) nous avons ces mêmes 1000 coordonnées générées, mais cette fois les clusters ont été définis par la méthode GMM. Il est possible de constater que la méthode a classifié efficacement les points, en maintenant une répartition spatiale et un nombre de points proche de ce qui a été généré artificiellement. Cependant, il existe une division claire entre les deux groupes centraux, sans chevauchement entre les deux, car la forte vraisemblance du groupe ayant la probabilité la plus élevée finit par remporter les points les plus proches.

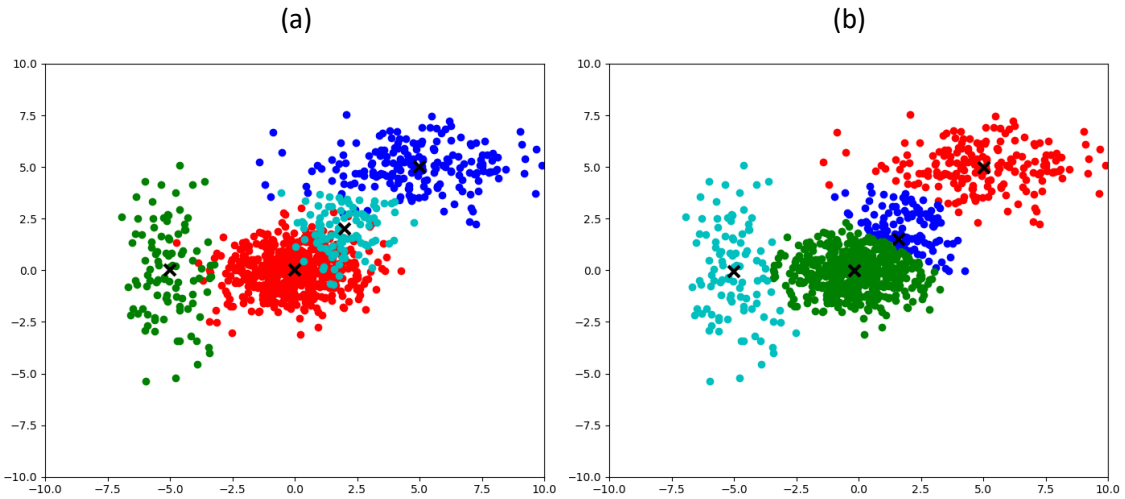

Figure 1 – Distribution des clusters générés (à gauche) et prédits (à droite).

Le critère de qualité du modèle est donné par la vraisemblance, c'est-à-dire que plus la vraisemblance du modèle est élevée, meilleure est sa précision. Sur l'image, nous pouvons voir que la courbe de vraisemblance (vraisemblance) se stabilise rapidement pendant les itérations de l'algorithme à une valeur de vraisemblance élevée, puis atteint notre critère d'arrêt avant le nombre maximum d'itérations, ce qui signifie que la précision souhaitée a été atteinte.

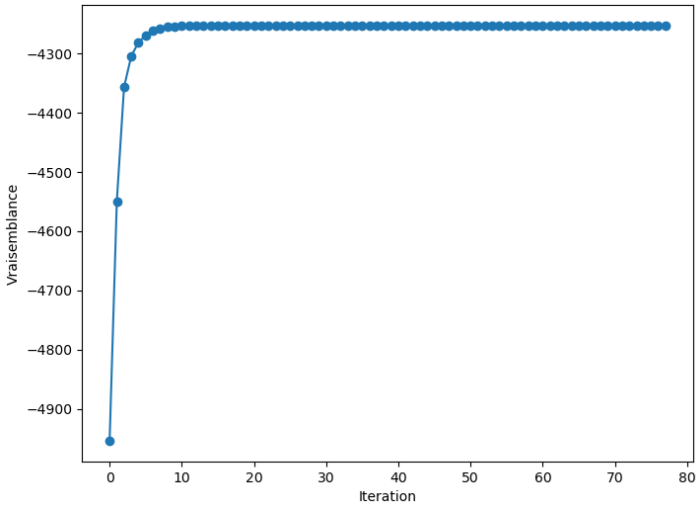

Figure 2 – Vraisemblance du modèle par itération.

Un problème majeur avec les méthodes de clustering telles que les k-means et GMM est qu'il n'existe pas de moyen définitif de choisir le nombre idéal de clusters. Cependant, les critères AIC (Akaike Information Criterion) et BIC (Bayesian Information Criterion) sont utilisés pour comparer différents modèles. et sélectionnez celui qui correspond le mieux aux données.

L'image montre les valeurs AIC et BIC pour différentes valeurs de k dans l'ensemble de données en question. En comparant les différents modèles GMM fournis pour chaque valeur de k, le meilleur d'entre eux est celui avec la valeur AIC ou BIC la plus basse. Dans ce cas, on voit que les deux critères s'accordent pour dire que la valeur de k=4 est la meilleure pour représenter la distribution des clusters.

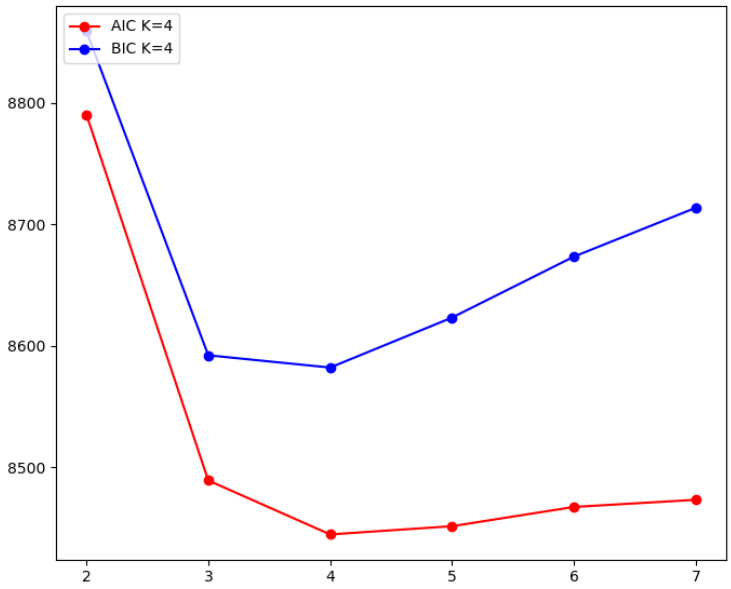

Figure 3 – AIC et BIC par nombre de clusters.

D:\TP-3-GMM Etudiant\TP-3-GMM Etudiant\your-GMM.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# M1 Science et Ingénieurie des données
# Université de Rouen Normandie
# T. Paquet
import matplotlib
import matplotlib.pyplot as plt
from sklearn import datasets
import numpy as np
import math
from numpy.linalg import norm

colors =['r','b','g','c','m','o']
n_colors = 6

############################################################################

def logsumexp(X):

    X_max = max(X) # identifie la plus grande valeur dans le vecteur X pour utiliser cette valeur comme référence pour le paramétrage
    if math.isinf(X_max):
        return -float('inf') # renvoie directement -inf, car cela ne sert à rien de poursuivre le calcul dans ce cas

    acc = 0
    for i in range(X.shape[0]):
        acc += math.exp(X[i] - X_max) # somme des exponentielles sous une forme paramétrée

    return math.log(acc) + X_max # applique le logarithme à la valeur accumulée et additionne X_max à "inverser" pour ajuster les log-cotes d'origine

############################################################################
#
def LogSumExp(Log_Vrais_Gauss):

    K,N = np.shape(Log_Vrais_Gauss) # le nombre d'éléments dans l'échantillon

    logsomme = np.zeros(N)
    for n in range(N):
        logsomme[n] = logsumexp(Log_Vrais_Gauss[:,n]) # les logarithmes de la vraissambance s'additionnent

    return logsomme # renvoie la somme logarithmique des probabilités pour chaque échantillon

############################################################################
#
#            Generation aléatoire d'un ensemble de N échantillons
#            conforme à la loi du mélange
def my_GMM_generate(P,Mean,Cov,N,Visualisation=False):
```

```
48
49      K,p = np.shape(Mean)
50      # insérer votre code ici
51
52      eff = np.asarray(N*P,dtype=int)
53      X = np.random.multivariate_normal(Mean[0,:],Cov[0,:,:],eff[0])
54      y = [0 for i in range(eff[0])]
55
56      for k in range(1,K):
57          #eff = int(N*P[k])
58          Xk = np.random.multivariate_normal(Mean[k,:],Cov[k,:,:],eff[k])
59          X = np.concatenate((X,Xk),axis=0)
60          yk = [k for i in range(eff[k])]
61          y = np.concatenate((y,yk),axis=0)
62
63      #
64      if Visualisation: #on visualise les deux premières coordonnées
65          plt.figure(figsize=(8,8))
66          debut=0
67          for k in range(K):
68              fin=debut+eff[k]
69              plt.plot(X[debut:fin,0],
70                       X[debut:fin,1],
71                       colors[k]+'o',markersize=4,markeredgewidth=3)
72              plt.plot(Mean[k,0],Mean[k,1],'kx',markersize=10,markeredgewidth=3)
73              debut=fin
74          plt.xlim(-10, 10)
75          plt.ylim(-10,10)
76          plt.show()
77
78      return X, y
79
80  ###################################################################
81  def my_G_LogVraisemblance(X,mean,cov):
82
83      N,p = np.shape(X)
84
85      covinv = np.linalg.inv(cov)
86      det = np.linalg.det(cov)
87      log_factor = np.log((2*np.pi)**(p/2) * math.sqrt(det))
88      Res = X - mean
89      Ex = -np.diag(Res @ covinv @ Res.T)/2
90      logvrais = Ex - log_factor
91
92      return logvrais
93
94  ###################################################################
95  def my_GMM_init(X,K):
96      N,p = np.shape(X)
97
98      # intialisation des proba a priori
99      P = np.random.random_sample(K)
100     P = P / np.sum(P)
101
```

```
102     # Initialisation des centroide
103     # par tirage de K exemples, pour tomber dans les données
104     Index_init = np.random.choice(N, K,replace = False)
105     Mean = np.zeros((K,p))
106     for k in range(K):
107         Mean[k,:] = X[Index_init[k],:]
108 
109     # intitialisation des matrices de covariance
110     # par affectation des données au plus proche centroide
111     # puis calcule de la matrice de covariance par cluster
112     Dist=np.zeros((K,N))
113     for k in range(K):
114         Dist[k,:] = np.square(norm(X - Mean[k,:],axis=1))
115     y = np.argmin(Dist,axis=0)
116 
117     Cov = np.zeros((K,p,p))
118 
119     for k in range(K):
120         Cluster = X[y==k,:]
121         Nk = np.shape(Cluster)[0]
122         Res = Cluster - Mean[k,:]
123         Cov[k,:,:] = Res.T @ Res / Nk
124 
125     return P, Mean, Cov
126 
127 ###########################################################################
128 def my_GMM_p_a_posteriori(X,K,P,Mean,Cov):
129 
130     N, p = np.shape(X)
131     Log_Vrais_Gauss = np.zeros((K,N))
132 
133     for k in range(K): #Soma, porque se trabalha com Logs
134         Log_Vrais_Gauss[k,:] = math.log(P[k]) + my_G_LogVraisemblanc↩
    e(X,Mean[k,:],Cov[k,:,:])
135 
136     LogDen = LogSumExp(Log_Vrais_Gauss)
137     Proba_Clusters = np.exp(Log_Vrais_Gauss - LogDen)
138     LogVrais = np.sum(LogDen)
139 
140     return Proba_Clusters,LogVrais
141 
142 ###########################################################################
143 def my_GMM_predict(X,K,P,Mean,Cov):
144 
145     Proba_Clusters, LogVrais = my_GMM_p_a_posteriori(X,K,P,Mean,Cov) # Calcule les
    probabilités a posteriori des données appartenant à chaque cluster
146 
147     y = np.argmax(Proba_Clusters,axis=0) # Identifie le cluster le plus probable pour chaque
    point de données en fonction des probabilités a posteriori
148 
149     return y,LogVrais # Renvoie les klusters prédites et la vraisemblance totale
150 
151 ###########################################################################
152 def my_GMM_fit(X,K,Visualisation,Seuil=0.0000001,Max_iterations = 100):
```

```
153
154     N,p = np.shape(X)
155
156     # INITIALISATION D'UN PREMIER MODÈLE
157     P, Mean, Cov = my_GMM_init(X,K)
158
159     if Visualisation :
160         print("P init = ",P)
161         print("Mean init = ",Mean)
162         print("Cov init = ",Cov)
163
164     iteration = 0
165     Log_Vrais_Gauss = np.zeros((K,N))
166     Nk = np.zeros(K)
167     New_Mean = np.zeros((K,p))
168     New_Cov = np.zeros((K,p,p))
169     New_P = np.zeros(K)
170
171     LOGVRAIS=np.zeros(Max_iterations+1)
172     LOGVRAIS[0] = -100000
173
174     while iteration < Max_iterations:
175         iteration +=1
176         ######################################################################
177         # E step : estimation des données manquantes
178         #          affectation des données aux clusters les plus proches
179         Proba_Clusters, LOGVRAIS[iteration] = my_GMM_p_a_posteriori(X,K,P,Mean,Cov)
180
181         if np.abs(LOGVRAIS[iteration] - LOGVRAIS[iteration-1]) / np.abs(LOGVRAIS[iteration])
    < Seuil:
182             print("itération =",iteration,"BREAK")
183             break
184
185         ######################################################################
186         # M Step : calcul du nouveau GMM
187
188         # les centroïdes
189         for k in range(K):
190             Nk[k] = np.sum(Proba_Clusters[k,:])
191             New_Mean[k,:] = np.sum(X.T * Proba_Clusters[k,:], axis=1) / Nk[k]
192
193         # les matrices de covariance
194         for k in range(K):
195             Res_gauche = (X[:,:] - Mean[k,:]).T * Proba_Clusters[k,:]
196             Res_droite = X[:,:] - Mean[k,:]
197             New_Cov[k,:,:] = (Res_gauche @ Res_droite) * np.identity(p) / Nk[k]
198
199         # les proba des clusters
200         New_P = Nk/N
201
202         Mean = New_Mean
203         P = New_P
204         Cov = New_Cov
205
```

```
206         # if Visualisation:
207         #     print("LOGVRAIS = ",LOGVRAIS[iteration])
208         #     print("P = ",P)
209         #     print("Mean = ",Mean)
210         #     print("Cov = ",Cov)
211 
212     if Visualisation:
213         fig = plt.figure(figsize=(8, 6))
214         plt.plot(LOGVRAIS[1:iteration], 'o-')
215         plt.xlabel('Iteration')
216         plt.ylabel('Vraisemblance')
217         plt.show()
218 
219     return P, Mean, Cov, LOGVRAIS[1:iteration]
220 
221 ################################################################################
222 if __name__ == '__main__':
223 
224     ######################################################################
225     #
226     #              Génération de données multivariées Gaussiennes (Etape 0)
227     PROB = np.array([0.6,0.2,0.1,0.1])
228     MEAN = np.array([[0,0],[5,5],[-5,0],[2,2]])
229     COV = np.array([[[2,0],[0,1]],[[5,0],[0,1]],[[1,0],[0,5]],[[1,0],[0,1]]])
230 
231     K,p = np.shape(MEAN)
232     N = 1000
233 
234     X,y = my_GMM_generate(PROB,MEAN,COV,N,Visualisation=True)
235 
236     P, Mean, Cov, LOGVRAIS = my_GMM_fit(X,K,True)
237 
238     y, LV = my_GMM_predict(X,K,P,Mean,Cov)
239 
240     plt.figure(figsize=(8,8))
241     for k in range(K):
242         plt.plot(X[y==k,0],X[y==k,1],colors[k]+'o',markersize=4,markeredgewidth=3)
243         plt.plot(Mean[k,0],Mean[k,1],'kx',markersize=10,markeredgewidth=3)
244     plt.xlim(-10, 10)
245     plt.ylim(-10,10)
246     plt.show()
247 
248 
249 BIC = []
250 AIC = []
251 
252 for KK in range(2, 2*K):
253     P, Mean, Cov, LOGVRAIS = my_GMM_fit(X, KK,False)
254     y, LV = my_GMM_predict(X, KK, P, Mean, Cov)
255 
256     bic = KK * (1 + p + p**2) * np.log(N) - 2 * LV
257     aic = KK * (1 + p + p**2) * 2 - 2 * LV
258 
259     BIC = BIC + [bic]
```

```
260     AIC = AIC + [aic]
261
262 K_AIC = AIC.index(min(AIC)) + 2
263 K_BIC = BIC.index(min(BIC)) + 2
264
265 plt.figure(figsize=(8,8))
266 plt.plot(range(2,2*K),AIC, 'ro-', label='AIC K='+str(K_AIC))
267 plt.plot(range(2,2*K),BIC, 'bo-', label='BIC K='+str(K_BIC))
268 plt.legend(loc="upper left")
269 plt.show()
270
271
```