TME sur la classification de lettres manuscrites

Format des données

Nous travaillerons sur des lettres manuscrites. Les données sont fournies au format pickle (le standard de sérialisation python, particulièrement convivial). Pour les charger :

In [1]:

```
import numpy as np
import pickle as pkl
import matplotlib.pyplot as plt

with open('ressources/lettres.pkl', 'rb') as f:
    data = pkl.load(f, encoding='latin1')
X = np.array(data.get('letters')) # récupération des données sur les lettres
Y = np.array(data.get('labels')) # récupération des étiquettes associées
```

```
/usr/lib/python3/dist-packages/ipykernel_launcher.py:7: VisibleDepre
cationWarning: Creating an ndarray from ragged nested sequences (whi
ch is a list-or-tuple of lists-or-tuples-or ndarrays with different
lengths or shapes) is deprecated. If you meant to do this, you must
specify 'dtype=object' when creating the ndarray
  import sys
```

Les données sont dans un format original: une lettre est en fait une série d'angles (exprimés en degrés). Un exemple:

In [2]:

```
X[0]
```

Out[2]:

```
array([ 36.214493, 347.719116, 322.088898, 312.230957, 314.851013,
       315.487213, 313.556702, 326.534973, 141.288971, 167.606689,
       199.321594, 217.911087, 226.443298, 235.002472, 252.354492,
       270.045654, 291.665161, 350.934723,  17.892815,  20.281025,
        28.207161,  43.883423,  53.459026])
```

Lors de l'acquisition, un stylo intelligent a pris des mesures régulièrement dans le temps: chaque période correspond à un segment de droite et le stylo a calculé l'angle entre deux segments consécutifs... C'est l'information qui vous est fournie.

Pour afficher une lettre, il faut reconstruire la trajectoire enregistrée... C'est ce que fait la méthode ci-dessous:

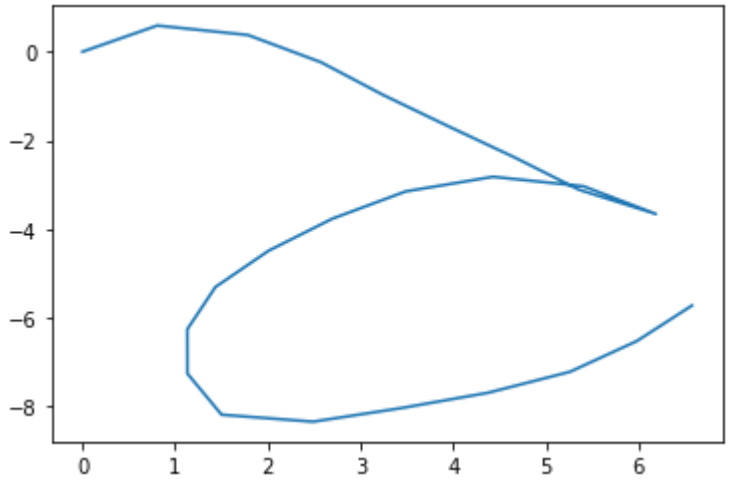
In [3]:

```
# affichage d'une lettre
def tracerLettre(let):
    a = -let*np.pi/180; # conversion en rad
    coord = np.array([[0, 0]]); # point initial
    for i in range(len(a)):
        x = np.array([[1, 0]]);
        rot = np.array([[np.cos(a[i]), -np.sin(a[i])],[ np.sin(a[i]),np.cos(a[i
])]])
        xr = x.dot(rot) # application de la rotation
        coord = np.vstack((coord,xr+coord[-1,:]))
    plt.figure()
    plt.plot(coord[:,0],coord[:,1])
    #plt.savefig("exlettre.png")
    return
```

In [4]:

```
# il s'agit d'un 'a'
tracerLettre(X[0])
print(Y[0]) # vérification de l'étiquette
```

a



A. Apprentissage d'un modèle CM (max de vraisemblance)

A1. Discrétisation

1 état = 1 angle

Il est nécessaire de regrouper les angles en un nombre fini d'états (par exemple 20)

- définir un `intervalle = 360 / n_etats` (18° si on choisit 20 états)
- discrétiser tous les signaux à l'aide de la formule `np.floor(x / intervalle)`
  - si n_etats = 20 alors [0, 18[=> 0, [18, 36[=> 1, etc...

Donner le code de la méthode `discretise(x, d)` qui prend la base des signaux et retourne une base de signaux discrétisés.

In [20]:

```
def discretise(x, d):

    # Votre code :
    intervalle=360/d
    #print(x)
    a=x/intervalle
    #print(a)

    return np.floor(a)
```

In [21]:

```
### test
discretise(X[0], 3)
```

Out[21]:

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

VALIDATION : code du premier signal avec une discrétisation sur 3 états:

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

A2. Regrouper les indices des signaux par classe (pour faciliter l'apprentissage)

In [22]:

```
def groupByLabel(y):
    index = []
    for i in np.unique(y): # pour toutes les classes
        ind, = np.where(y == i)
        index.append(ind)
    return index
```

In [23]:

```
index=groupByLabel(Y)
print(index)
```

```
[array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]), array([11, 12,
13, 14, 15, 16, 17, 18, 19, 20, 21]), array([22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32]), array([33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43]), array([44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54]), array([5
5, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65]), array([66, 67, 68, 69,
70, 71, 72, 73, 74, 75, 76]), array([77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87]), array([88, 89, 90, 91, 92, 93, 94, 95, 96, 97]), array
([ 98,  99, 100, 101, 102, 103, 104, 105, 106, 107]), array([108, 10
9, 110, 111, 112, 113, 114, 115, 116, 117]), array([118, 119, 120, 1
21, 122, 123, 124, 125, 126, 127]), array([128, 129, 130, 131, 132,
133, 134, 135, 136, 137]), array([138, 139, 140, 141, 142, 143, 144,
145, 146, 147]), array([148, 149, 150, 151, 152, 153, 154, 155, 156,
157]), array([158, 159, 160, 161, 162, 163, 164, 165, 166, 167]), ar
ray([168, 169, 170, 171, 172, 173, 174, 175, 176, 177]), array([178,
179, 180, 181, 182, 183, 184, 185, 186, 187]), array([188, 189, 190,
191, 192, 193, 194, 195, 196, 197]), array([198, 199, 200, 201, 202,
203, 204, 205, 206, 207]), array([208, 209, 210, 211, 212, 213, 214,
215, 216, 217]), array([218, 219, 220, 221, 222, 223, 224, 225, 226,
227]), array([228, 229, 230, 231, 232, 233, 234, 235, 236, 237]), ar
ray([238, 239, 240, 241, 242, 243, 244, 245, 246, 247]), array([248,
249, 250, 251, 252, 253, 254, 255, 256, 257]), array([258, 259, 260,
261, 262, 263, 264, 265, 266, 267])]
```

Cette méthode produit simplement une structure type:

```
[array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
 array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]),
 array([22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]),
 array([33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43]),
 array([44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54]),
 array([55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65]),
 ...
```

Chaque ligne regroupe les indices de signaux correspondant à une classe. Il y a donc 11 exemples de 'a'.

A3. Apprendre les modèles CM

Soit $\{X_C\}$ la base de signaux discrétisés correspondant à une classe $\{C\}$ et $\{d\}$ le nombre d'états. Donner le code de la fonction `learnMarkovModel(Xc, d)` qui retourne un tuple contenant Pi et A.

Rappel:

- Initialisation de

```
A = np.zeros((d, d))
Pi = np.zeros(d)
```

- Parcours de tous les signaux et incréments de A et Pi
- Normalisation (un peu réfléchie pour éviter les divisions par 0)

```
A = A / np.maximum(A.sum(1).reshape(d, 1), 1) # normalisation
Pi = Pi / Pi.sum()
```

Note : la solution proposée pour gérer le cas des lignes entièrement à 0 est naïve et n'est pas totalement satisfaisante. Comprendre pourquoi. On proposera une solution améliorée plus loin dans le TME.

In [59]:

```
#retourne tuple pi, A
def learnMarkovModel(Xc, d):
    #Xc: signaux d'une classe donnée
    # votre code
    # init
    A=np.zeros((d,d))
    Pi=np.zeros(d)
    for i in range(Xc.shape[0]):
        # pour pi, on regarde la premiere valeur du signal X[i,0]
        #donne l'état initial pour la séquence i
        indice=Xc[i][0]
        #on rajoute 1 à celui ci
        Pi[np.int(indice)]+=1
        for j in range(Xc[i].shape[0]-1):
            A[np.int(Xc[i][j]),np.int( Xc[i][j+1])]+=1

    # on normalise pi sur le nombre de séquences
    #faire attention aux divisions par 0 ici
    Pi=Pi/Pi.sum()
    A=A/np.maximum(A.sum(1).reshape(d,1), 1)
    return Pi, A
```

In [62]:

```
# test
Disc=np.array([discretise(X[i], 3) for i in range(X.shape[0])])
#print(Disc)

index=groupByLabel(Y)
# regrouper les signaux par classes
#print(Disc[index[0]])
learnMarkovModel(Disc[index[0]] ,3)
```

```
/usr/lib/python3/dist-packages/ipykernel_launcher.py:2: VisibleDepre
cationWarning: Creating an ndarray from ragged nested sequences (whi
ch is a list-or-tuple of lists-or-tuples-or ndarrays with different
lengths or shapes) is deprecated. If you meant to do this, you must
specify 'dtype=object' when creating the ndarray
```

Out[62]:

```
(array([0.36363636, 0.        , 0.63636364]),
 array([[0.84444444, 0.06666667, 0.08888889],
        [0.        , 0.83333333, 0.16666667],
        [0.11382114, 0.06504065, 0.82113821]]))
```

Validation : premier modèle avec une discrétisation sur 3 états :

```
(array([ 0.36363636,  0.        ,  0.63636364]),
 array([[ 0.84444444,  0.06666667,  0.08888889],
        [ 0.        ,  0.83333333,  0.16666667],
        [ 0.11382114,  0.06504065,  0.82113821]]))
```

A4. Stocker les modèles dans une liste

Pour un usage ultérieur plus facile, on utilise le code suivant :

In [63]:

```
d = 20                    # paramètre de discrétisation
#d = discretise(X, d)     # application de la discrétisation
Xd=np.array([discretise(X[i], d) for i in range(X.shape[0])])
index = groupByLabel(Y)  # groupement des signaux par classe
models = []
for cl in range(len(np.unique(Y))): # parcours de toutes les classes et optimisa
tion des modèles
    models.append(learnMarkovModel(Xd[index[cl]], d))
```

```
/usr/lib/python3/dist-packages/ipykernel_launcher.py:3: VisibleDepre
cationWarning: Creating an ndarray from ragged nested sequences (whi
ch is a list-or-tuple of lists-or-tuples-or ndarrays with different
lengths or shapes) is deprecated. If you meant to do this, you must
specify 'dtype=object' when creating the ndarray
  This is separate from the ipykernel package so we can avoid doing
imports until
```

A5. Distribution stationnaire

La distribution stationnaire μ, pour un système ergodique, correspond à:

- la distribution des états observés dans l'ensemble des données
- la distribution obtenu après un grand nombre de passage dans A à partir de n'importe quelle distribution ou état d'origine (une matrice de transition ergodique va nous permettre de converger)
- la solution de l'équation de stabilité: $\mu = \mu A$

Dans le cas de la lettre 'a' et d'une discrétisation en 3 états, calculer la distribution stationnaire par les deux première méthode et vérifier que vous obtenez des résultats similaires.

Note: pour la marche aléatoire, vous construirez une boucle sur 100 iérations (borne sup) et vous sortirez dès que la somme des écarts (en absolu) entre μ_t et μ_{t+1} est inférieure à 10^{-3}. Vous vérifierez ensuite que le système converge quelque soit l'état de départ (3 appel àla fonction) ou avec une initialisation aléatoire.

In [ ]:

```
```

B. Test (affectation dans les classes sur critère MV)

B1. (log)Probabilité d'une séquence dans un modèle

Donner le code de la méthode `probaSequence(s,Pi,A)` qui retourne la log-probabilité d'une séquence `s` dans le modèle $\{\lambda = \{Pi, A\}\}$

In [75]:

```
def probaSequence(s, Pi, A):

    # Votre code
    res=0
    # retourne la log proba dans un modèle
    for i in range(s.shape[0]-1):
        # = somme des log de chaque proba
        res+=np.log(A[np.int(s[i]), np.int(s[i+1])])
    return res

    return
```

In [82]:

```
# pour les 26 modèle
Res=np.zeros( np.size(np.unique(Y)))
for i in range( np.size(np.unique(Y))):
    Pi,A=learnMarkovModel(Disc[index[i]] ,3)
    a=Disc[0]
    Res[i]=probaSequence(a,Pi,A )

print(Res)
```

```
26
(26,)
[-12.47948509         -inf         -inf         -inf         -inf
         -inf         -inf         -inf         -inf         -inf
         -inf         -inf         -inf         -inf         -inf
         -inf         -inf         -inf -13.18651935         -inf
         -inf         -inf         -inf         -inf         -inf
 -12.48285678]

/usr/lib/python3/dist-packages/ipykernel_launcher.py:8: RuntimeWarni
ng: divide by zero encountered in log
```

VALIDATION : probabilité du premier signal dans les 26 modèles avec une discrétisation sur 3 états :

```
array([-13.491086  ,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf,         -inf,         -inf,         -inf,
               -inf, -12.48285678])
```

- Ce signal est-il bien classé ?
- D'où viennent tous les `-inf` ?

B2. Application de la méthode précédente pour tous les signaux et tous les modèles de lettres

L'application se fait en une ligne de code si vous avez respecté les spécifications précédentes :

In [ ]:

```
proba = np.array([[probaSequence(Xd[i], models[cl][0], models[cl][1]) for i in r
ange(len(Xd))]
                  for cl in range(len(np.unique(Y)))])
```

B3. Evaluation des performances

Pour l'évaluation, nous proposons l'approche suivante:

In [ ]:

```
# calcul d'une version numérique des Y :
Ynum = np.zeros(Y.shape)
for num, char in enumerate(np.unique(Y)):
    Ynum[Y == char] = num

# Calcul de la classe la plus probable :
pred = proba.argmax(0) # max colonne par colonne

# Calcul d'un pourcentage de bonne classification :
np.where(pred != Ynum, 0.,1.).mean()
```

INDICE DE PERFORMANCE : 91% de bonne classification avec 20 états, 69% avec 3 états

C. Biais d'évaluation, notion de sur-apprentissage

Dans le protocole précédent, nous avons triché:

- les données servent d'abord à apprendre les modèles...
- puis nous nous servons des mêmes données pour tester les modèles ! Les performances sont forcément bonnes !

Afin de palier le problème, nous allons diviser en deux la base de données: une partie servira à l'apprentissage des modèles, l'autre à leur évaluation. Pour effectuer la division, nous fournissons le code suivant:

In [ ]:

```
# separation app/test, pc=ratio de points en apprentissage
def separeTrainTest(y, pc):
    indTrain = []
    indTest = []
    for i in np.unique(y): # pour toutes les classes
        ind, = np.where(y == i)
        n = len(ind)
        indTrain.append(ind[np.random.permutation(n)][:int(np.floor(pc * n))])
        indTest.append(np.setdiff1d(ind, indTrain[-1]))
    return indTrain, indTest

# exemple d'utilisation
itrain, itest = separeTrainTest(Y, 0.8)
```

dans `itrain` , nous obtenons les indices des signaux qui doivent servir en apprentissage pour chaque classe :

In [ ]:

```
itrain
```

Note : pour faciliter l'évaluation des modèles, vous aurez besoin de re-fusionner tous les indices d'apprentissage et de test. Cela se fait avec les lignes de code suivantes :

In [ ]:

```
ia = []
for i in itrain:
    ia += i.tolist()
it = []
for i in itest:
    it += i.tolist()
```

Note 2 : Du fait de la permutation aléatoire, les résultats vont bouger (un peu) à chaque execution du programme.

C1. Questions importantes

- Ré-utiliser les fonctions précédemment définies pour apprendre des modèles et les évaluer sans biais.
- Calculer et analyser les résultats obtenus en apprentissage et en test
- Etudier l'évolution des performances en fonction de la discrétisation

In [ ]:

```
# Votre code
```

C2. Lutter contre le sur-apprentissage

Cette base de données met en lumière le phénomène de sur-apprentissage : il y a peu de données et dès que le nombre d'états augmente, il y a trop peu d'exemple pour estimer correctement les matrices $\{A, \pi\}$. De nombreuses cases sont donc à 0, voire des lignes entières (d'où la sécurisation du code pour la normalisation des matrices stochastiques).

Ces 0 sont particulièrement discriminants: considérant la classe $\{c\}$, ils permettent d'éliminer de cette classe tout signal présentant cette caractéristique. Cette règle est trop forte compte tenu de la taille de la base d'apprentissage. Nous proposons une astuce pour palier cette faiblesse : lors du comptage, initialiser les matrices $\{A, \pi\}$ avec ones au lieu de zeros . On fait semblant d'avoir observer une transition de chaque type avant même le début du comptage.

Comparer les performances en test.

In [ ]:

```
# Votre code
```

D. Evaluation qualitative

Nous nous demandons maintenant où se trouvent les erreurs que nous avons commises...

Calcul de la matrice de confusion: pour chaque échantillon de test, nous avons une prédiction (issue du modèle) et une vérité terrain (la vraie étiquette). En posant Nc le nombre de classes, la matrice de confusion est une matrice (Nc x Nc) où nous comptons le nombre d'échantillon de test dans chaque catégorie :

- Initialisation à 0 :

In [ ]:

```
conf = np.zeros((26,26))
```

- Pour chaque échantillon, incrément de la case (prediction, vérité)

In [ ]:

```
# Votre code
```

- Tracé de la matrice :

In [ ]:

```
plt.figure()
plt.imshow(conf, interpolation = 'nearest')
plt.colorbar()
plt.xticks(np.arange(26), np.unique(Y))
plt.yticks(np.arange(26), np.unique(Y))
plt.xlabel(u'Vérité terrain')
plt.ylabel(u'Prédiction')
#plt.savefig("mat_conf_lettres.png")
```

E. Modèle génératif

Utiliser les modèles appris pour générer de nouvelles lettres manuscrites.

E1. Tirage selon une loi de probabilité discrète

- faire la somme cumulée de la loi $\{sc\}$
- tirer un nombre aléatoire $\{t\}$ entre 0 et 1
- trouver la première valeur de $\{sc\}$ qui est supérieure à $\{t\}$
- retourner cet état

Note : comme vu en cours, tout repose sur la somme cumulée (notée ici `sc$` , calculable en appelant `np.cumsum` . Sur un exemple: la loi `V = [0.2, 0.4, 0.3, 0.1]` a pour somme cumulée `V.cumsum() == [0.2,  0.6,  0.9,  1.0]`

E2. Génération d'une séquence de longueur N

- tirer un état $\{s_0\}$ selon Pi
- tant que la longueur n'est pas atteinte :
  - tirer un état $\{s_{t+1}\}$ selon $\{A[s_t]\}$

In [ ]:

```
# Votre code
```

E3. Affichage du résultat

In [ ]:

```
newa = generate(models[0][0], models[0][1], 20)       # generation d'une séquenc
e d'états
intervalle = 360. / d                                 # pour passer des états =>
valeur d'angles
newa_continu = np.array([i * intervalle for i in newa]) # conv int => double
tracerLettre(newa_continu)
```