NOM:OUNADI

Prénom: IKram

spécialité :IA

PROJET 2: Reconnaissance de la parole par distance DTW

Partie théorique :

explication du principe du fonctionnement de l'algo

Notre approche fonctionne à partir de comparaison entre cepstres MFCC. Plusieurs étapes sont nécessaires pour transformer un fichier audio en cepstre MFCC. Le principal intérêt des MFCC est d'extraire des informations pertinentes et en nombres limités en s'appuyant à la fois sur la production théorique (théorie cepstrale) et à la fois sur la perception de la parole (échelle des Mels).

DTW, Dynamic Time Warping, est une méthode fondée sur un principe de comparaison d'un signal à analyser avec un ensemble de signaux stockés dans une base de référence. Le signal à analyser est comparé avec chacune des références et est classé en fonction de sa proximité avec une des références stockées.

Partie Implementation:

calculer le taux de reconnaissance en utilisant une méthode de reconnaissance de forme DTW avec les coefficients MFCC extraits des fichiers wav du Dataset , pour la reconnaissance des chiffres

In [11]:

```
import numpy as np
from scipy.fftpack import dct, idct
import matplotlib.pyplot as plt
from IPython.display import Audio,display
from fastdtw import fastdtw
from scipy.io import wavfile
from scipy.stats import mode
from scipy.spatial.distance import euclidean
```

In [12]:

```
#convertir une fréquence en mel vers une fréquence en Hz
def Mel2Hz(mel):
    return 700 * (np.power(10,mel/2595)-1)
#convertir une fréquence en Hz vers une fréquence en Mel
def Hz2Mel(freq):
    return 2595 * np.log10(1+freq/700)
#convertir une fréquence Hz en indice de matrice (sachant que fs --> T)
def Freq2Ind(freq,fs,T):
    return (freq*T/fs).astype(int)
#Fenêtre de hamming
def hamming(T):
    t=np.arange(T)
    return 0.54-0.46*np.cos(2*np.pi*t/(T-1))
#Filtre de préaccentuation
def preaccentuation(x):
    return x[:-1]-0.97*x[1:]
#Calcul du spectre amplitude et phase du signal x, taille de la fenetre=T, pas=p
#window est une fenêtre de type hamming, laissez vide pour ne pas utiliser
#fftsz=taille de la fft, si > T le zéro padding sera activé(voir cours acoustique)
def spectrogram(x,T,p,window=None,fftsz=None):
    n=len(x) #taille de x
    m=len(np.arange(0,n-T,p)) #estimation du nombre de fenêtres
    S=np.zeros((m,T))
```

```
    if fftsz is None: fftsz=T #si vide on prend la valeure de T
    if window is None: window=np.ones(T) #si vide on prend des 1
    #début fenêtrage


    for i in range(m):
        S[i,:]=x[i*p:i*p+T]*window

    #Transformée de Fourier
    S=np.fft.fft(S,fftsz)
    amp=np.abs(S) #spectre d'amplitude
    phase=np.angle(S) #spectre de phase
    return amp,phase

#Créer des filtres proches de ceux de l'oreille, nf: nombre de filtres à créer
def filtresOreille(nf,T,fs):
    freqMin=50 #Hz On définit la fréquence minimum que perçoit l'oreille
    freqMax=min(10000, fs/2) #Hz On définit la fréquence minimal que perçoit l'oreille
#convertir en mel
    melMin=Hz2Mel(freqMin) # 50Hz = 77mel
    melMax=Hz2Mel(freqMax) # 10000Hz = 3073mel

#calcul du début et la fin de chaque filtre en Mel
    pas=(melMax-melMin)/(nf+1) #diviser l'espace de perception de l'oreille sur le nombre de filtre
s
    debut=np.arange(melMin,melMax-2*pas+1,pas) #réalisation de filtres qui se chevauchent de moitie
r
    fin=debut+2*pas

    #convertir les filtres dans les fréquences (Hz)
    debuthz=Mel2Hz(debut)
    finhz=Mel2Hz(fin)

    #convertir les fréquences en indices
    debutI=Freq2Ind(debuthz,fs,T)
    finI=Freq2Ind(finhz,fs,T)

    #construie les filtres
    filtres=np.zeros((int(T/2),nf))
    for i in range(len(debutI)):
        filtres[debutI[i]:finI[i],i]=hamming(finI[i]-debutI[i])
    return filtres
#Calcul des coefficients MFCC, data: le vecteur des données (domaine temporel), T: taille de la fe
nêtre
# p: le pas , filtres: matrice des filtres de l'oreille, nc: le nombre de coefficients à garder
def mfcc(data,filtres,T=1024,p=32,nc=13):
    amp,phase=spectrogram(preaccentuation(data),T,p,window=hamming(T)) #calcul du spectrogramme
    amp=amp[:,:int(T/2)] #On coupe le spectre d'amplitude en 2 à cause de l'effet miroir
    amp=np.dot(amp,filtres) #application des filtres de l'oreille
    amp=np.log(amp) #calcul du log du spectre (convertir la multiplication en addition -
voircours)
    amp=idct(amp, norm = 'ortho') #calcul de l'idct du spectre filtré en log
    amp=amp[:,:nc] #ne choisir que les "nc" premiers (réduction des données + lissage ceptral)
    return amp
```

In [8]:

```
######################Chargement du
Dataset#############################################################
chemin='/home/ikram/Downloads/projet num2/free-spoken-digit-dataset-master/recordings/'
locuteurs=['jackson','nicolas','theo','yweweler']
datamfcc=[] #on va stocker les mfcc de chaque fichier audio dans cette liste
labels=[] #on va stocker la classe réelle [0-9] de chaque fichier audio dans cette liste

fs=22050#on suppose qu'on va travailler sur une fréquence d'échantillonnage de 22050

T=1024 #taille de la fenêtre
p=32 #la valeur du pas
nf=36 #nombre de filtres de l'oreille à utiliser
nc=13 #nombre de coefficients mfcc à garder


filtres=filtresOreille(nf,T,fs)
for ch in range(10): #pour les chiffres de 0-9
    for loc in locuteurs: #pour chaque locuteur
        for i in range(50): #chaque locuteur a répété 50 fois le même chiffre
```

```
        for i in range(50): #chaque locuteur a répété 50 fois le même chiffre
            fichier="%d_%s_%d.wav"%(ch,loc,i) #le nom du fichier à charger
            fs, data = wavfile.read(chemin+fichier) #lire un fichier audio
            data=data/max(abs(data)) #le normaliser entre -1 et 1
            fichier_mfcc=mfcc(data,filtres,T=1024,p=32,nc=13) #calculer les mfccs
            datamfcc.append(fichier_mfcc) #ajouter les mfcc du fichier à la liste
            labels.append(ch) #ajouter la classe du fichier aux labels (s'il s'agit d'un 0 ou 1 ou2
...)

#mélange aléatoire du dataset et division 50% train / 50% test
n=len(labels) #le nombre de fichiers
indices=np.arange(n) #mélange aléatoire du dataset
np.random.shuffle(indices)
datamfcc=np.array(datamfcc)[indices]
labels=np.array(labels)[indices]

m=int(n*0.5) #prendre 50% comme train et 50% comme test
trainmfcc=datamfcc[:m] #données d'apprentissage
trainlabels=labels[:m] #sortie désirée pour chaque donnée d'apprentissage
testmfcc=datamfcc[m:] #données de test, ne pas utiliser qu'à la fin pour tester (calcul du taux)
testlabels=labels[m:] #sortie désirée (réelle) de chaque donnée de test

######################Réalisation du
modèle#################################################################



def reconnaissance(fichier_mfcc):
    #la methode dtw
    #appliquer l'algorithme DTW qui fournit des alignements optimaux ou quasi-optimaux avec un tem
ps O (N) et une complexité de mémoire.
    distance=[] # tableau pour les distances calculées
    for tr in trainmfcc: #pour données d'apprentissage
        dist,path=fastdtw(fichier_mfcc,tr,dist=euclidean) #appliquer le principe de l'algo et
calculer la distance euclidienne
        distance.append(dist) #ajouter les distances calculés à la liste distance
    listeDistance = np.array(distance) # create an array
    ind = np.argmin(listeDistance )# Renvoie les indices de l'élément min du tableau
    DTWlabel=trainlabels[ind]
    return DTWlabel

###################Test du modèle avec (testmfcc et
testlabel)#############################################
reponse=[] #initialiser la liste qui va contenir la réponse de votre méthode pour chaque fichier t
est
for f_mfcc in testmfcc:
    reponse.append(reconnaissance(f_mfcc))
correct=sum(reponse==testlabels) #calcul du nombre de réponses correctes
print('Réponses correctes: %d / %d'%(correct,len(testlabels)))
taux=np.mean(reponse==testlabels) #calcul en pourcentage (taux de reconnaissance)
print('Taux de reconnaissance: %.2f%%'%(taux*100))
```

```
Réponses correctes: 970 / 1000
Taux de reconnaissance: 97.00%
```

conclusion:

apres l'application de l'algorithme on a reussit à avoir 970 reponses correctes a partie de 1000 reponse ca veut dire un taux de reconnaissance de 97% on peut alors déduire que L'algorithme DTW est un très bon outil capable de comparer deux spectres audio ayant des durées différentes, un débit, une intensité de la voix différente et cela de façon optimale en recherchant le meilleur chemin pour passer d'un spectre à l'autre.

In [ ]: