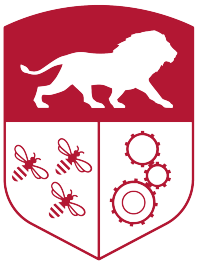


CENTRALE LYON

École Centrale Lyon

MOD Traitement et analyse des données visuelles et sonores
Rapport

BE2 : Systèmes simples de reconnaissance de la parole

Élèves :
Rémy Gasmi
Simon Vincent

Enseignant :
Emmanuel Dellandréa

8 janvier 2025

Table des matières

1 Introduction

2 Présentation des données fournies

Nos données sont plus de deux mille enregistrements sous format .wav

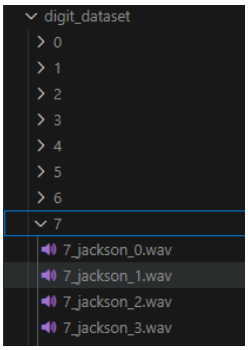


FIGURE 1 – Organisation des données fournies

Il y a donc plus de 2000 enregistrements, de quelques secondes, de plusieurs personnes qui disent un chiffre entre 0 et 9. Notre objectif sera d'identifier quel nombre a été dit en comparant aux données d'entraînement.

Pour cela, nous devons charger les données d'une manière qui soit ensuite traitable sous Python. Nous créons pour cela une fonction, nommée `load_data(input_folder, Standart_FE=8000)` .

```
# Initialiser les listes pour stocker les données extraites, les fréquences d'échantillonnage et les étiquettes
data_Fe = []
labels = []
data_audio = []

# Parcourir chaque répertoire dans le dossier d'entrée
for dirname in os.listdir(input_folder):
    subfolder = os.path.join(input_folder, dirname)

    # Passer si ce n'est pas un répertoire
    if not os.path.isdir(subfolder):
        continue

    # Parcourir chaque fichier .wav dans le sous-dossier
    for filename in [x for x in os.listdir(subfolder) if x.endswith('.wav')]:
        filepath = os.path.join(subfolder, filename)

        # Charger le fichier audio avec la fréquence d'échantillonnage spécifiée
        audio, Fe = librosa.load(filepath, sr=Standart_FE)

        # Normaliser l'audio
        audio = audio / np.max(np.abs(audio))

        # Extraire les frames de l'audio
        frames = auto_get_frames(audio, Fe)

        # Ajouter les frames extraites, la fréquence d'échantillonnage et l'étiquette aux listes respectives
        data_audio.append(frames)
        data_Fe.append(Fe)
        labels.append(dirname)

# Retourner les listes contenant les données extraites, les fréquences d'échantillonnage et les étiquettes
return data_audio, data_Fe, labels
```

FIGURE 2 – Fonction load_data

La fonction `load_data` fonctionne de la manière suivante :

— Elle prend en entrée un dossier contenant des sous-dossiers, chacun représentant une classe différente (un chiffre entre 0 et 9).

— Pour chaque sous-dossier, elle parcourt les fichiers audio au format .wav.

— Chaque fichier audio est chargé avec une fréquence d'échantillonnage spécifiée (par défaut 8000 Hz) en utilisant la librairie `librosa`.
— L'audio est normalisé pour que ses valeurs soient comprises entre -1 et 1.
— Les frames de l'audio sont extraites à l'aide de la fonction `auto_get_frames`.
— Les frames extraites, la fréquence d'échantillonnage et le nom de la classe de l'enregistrement (nom du sous-dossier) sont ajoutées à des listes respectives.
— Enfin, la fonction retourne trois listes : data_audio (les frames extraites), data_Fe (les fréquences d'échantillonnage) et labels (les étiquettes).

3 Utilisation coefficients LPC, de la classification des k plus proches voisins avec la distance élastique

3.1 Formatage des données

Après avoir utilisé la fonction `load_data(input_folder, Standart_FE=8000)` présentée partie 2, nous avons alors nos données sous la forme de trois listes : trois listes : data_audio (les frames extraites), data_Fe (les fréquences d'échantillonnage) et labels (les étiquettes).

Pour séparer nos données entre les données d'entraînement et de test, nous devons donc créer une fonction qui sera capable de prendre séparer plusieurs listes de la même facon. en effet, actuellement, un enregistrement dans data_audio à la position i aura pour classe l'enregistrement de même position i dans la liste labels.
Pour cela, la fonction `split_train_test(lists, train_ratio=0.8, randomize=True)` génère une liste des indices de même taille que toutes les listes contenues dans lists. C'est ensuite cette liste des indices que l'on va mélanger puis séparer entre les données de test et d'entraînement.

3.2 Calcul des coefficients LPC

Les coefficients LPC (Linear Predictive Coding) modélisent la production de la parole en s'appuyant sur le modèle source-filtre, où un signal de parole est considéré comme un son produit par une source originale, modulé par les résonances du conduit vocal appelées les formants. On considère de plus que le son est modifié par la langue, les lèvres ou la gorge.
LPC décompose le signal en estimant les formants (les résonances du conduit vocal), la source et les modifications dans la bouche. Cette analyse se fait sur des trames courtes du signal, 30 à 50 par seconde, pour capturer les variations temporelles du signal de parole.

```
data_audio,data_Fe,labels = load_data(input_folder)

used_data, discarded_data = split_train_test([data_audio,data_Fe,labels], train_ratio=1)
train_data, test_data = split_train_test(used_data, train_ratio=0.92)

(data_audio_train, data_Fe_train, original_labels_train) = train_data
(data_audio_test, data_Fe_test, original_labels_test) = test_data
```

FIGURE 3 – Séparation des données d'entrainement et de test

```python
def split_train_test(lists, train_ratio=0.8, randomize=True):
    """
    lists: liste de listes, par ex. [data_audio, data_Fe, labels].
    On suppose que toutes ces listes ont la même longueur.
    """
    # Longueur commune à toutes les listes
    n = len(lists[0])

    # Vérifier qu'elles ont toutes la même taille
    for lst in lists:
        if len(lst) != n:
            raise ValueError("Toutes les listes doivent avoir la même longueur.")

    # Générer les indices
    indices = list(range(n))

    # Mélanger les indices une seule fois
    if randomize:
        random.shuffle(indices)

    # Déterminer l'indice de coupure
    split_index = int(n * train_ratio)

    # Construire train_lists et test_lists en appliquant la même permutation
    train_lists = []
    test_lists = []

    for lst in lists:
        # lst_shuffled = [lst[i] for i in indices] si on veut vraiment garder la liste initiale intacte
        # ou bien on peut l'appliquer directement en compréhension de liste
        train_lists.append([lst[i] for i in indices[:split_index]])
        test_lists.append([lst[i] for i in indices[split_index:]])

    return train_lists, test_lists
```

FIGURE 4 – fonction `split_train_test`

Pour calculer ces coefficients, on utilise l'équation de Yule-Walker exprimant la relation entre les coefficients LPC et les covariances du signal considéré.

$$\begin{bmatrix} R(0) & R(1) & \cdots & R(N) \\ R(-1) & R(0) & \ddots & \vdots \\ \vdots & \ddots & \ddots & R(1) \\ R(-N) & \cdots & R(-1) & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} \sigma^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

FIGURE 5 – Equation de YuleWalker

les a_i sont les coefficients LPC d'un modèle d'ordre N, 2 est la puissance de l'erreur de prédiction et R(i) est la fonction d'autocovariance définie par R(i) = E(s(t+i)s(t)).

On calcule donc les R(i) avec l'autocovariance, avec des slices des valeurs de f judicieusement choisit pour correspondre à la corrélation entre les signaux décalés de i, et `np.mean`. On arrange ensuite les R(i) dans la forme de la matrice de la figure 5 avec `scipy.linalg.toeplitz`.

```python
R = np.zeros((ordre_modele+1,1))
for k in range(ordre_modele+1):
    R[k] = np.mean(f[0:t_fenetre-1-k]*f[k:t_fenetre-1])

m_R = toeplitz(R)
```

FIGURE 6 – Calcul de la matrice des R(i)

Il ne reste plus qu'à inverser l'équation de Yule-Walker. Comme on ne connaît pas σ^2, on met 1 à la place dans le vecteur de droite et on normalisera par la première valeur du vecteur obtenue, qui sera σ^{-2}

```python
def process_audio_to_lpc(data_audio, labels, ordre_modele, batch_size=50, prefix=""):
    """
    Convertit les données audio en coefficients LPC normalisés.

    Args:
        data_audio: Liste des données audio à traiter
        labels: Liste des labels correspondants
        ordre_modele: Ordre du modèle LPC
        batch_size: Taille du lot pour les messages de progression
        prefix: Préfixe pour les messages (ex: "train" ou "test")

    Returns:
        tuple: (lpc_arrays, processed_labels)
        - lpc_arrays: Liste de tableaux numpy contenant les coefficients LPC
        - processed_labels: Labels correspondant aux données traitées
    """
    index_to_pop = []
    lpc_arrays = []

    for i in range(len(data_audio)):

        if  i%batch_size == 0:
            print(f"Calcul des LPC des données {prefix} : {i} / {len(data_audio)}" )
        try:
            # Pré-allouer un tableau pour les coefficients LPC
            lpc_frames = np.zeros((len(data_audio[i]), ordre_modele ))
            for j, frame in enumerate(data_audio[i]):
                lpc_coeffs = normalize_lpc_coefficients(calcul_lpc_frame(frame, ordre_modele))
                lpc_frames[j] = np.squeeze(lpc_coeffs)
            lpc_arrays.append(lpc_frames)


        except:
            index_to_pop.append(i)
            print(f"Un audio n'est pas pris en compte: le numéro {i} appartenant à la classe {labels[i]}")

    # Convertir et nettoyer les labels
    processed_labels = np.array(labels)
    processed_labels = np.delete(processed_labels, index_to_pop)

    return lpc_arrays, processed_labels
```

FIGURE 8 – Fonction `process_audio_to_lpc`

```python
v = np.zeros((ordre_modele+1,1))
v[0] = 1

lpc = np.dot(inv(m_R),v)
lpc = lpc/lpc[0]

return lpc[1:]
```

FIGURE 7 – Obtention des coefficients lpcs

On a donc détaillé la fonction `calcul_lpc_frame` qui calcule les coefficients lpc d'une frame simple. Pour calculer tous les coefficients lpc d'un audio sous la forme de liste de frames, on utilise `process_audio_to_lpc`. Cette fonction à pour particularités de normaliser les coefficients lpc calculées, de gérer les erreurs en supprimant un audio si celui-ci n'est pas traité correctement (garder en mémoire son index, puis le supprimer à la toute fin), et de renvoyer ses résultats sous forme d'array NumPy, ce qui sera pratique pour la suite.

3.3 Classification des k plus proches voisins utilisant la distance élastique

3.3.1 Distance élastique

A ce stade, nous devons calculer les distances entre les différents signaux audios calculées, caractérisés par leur coefficients lpc à chaque trame. Cependant, en fonction de la prononciation, certaines syllabes peuvent être élongés ou raccourcis, il nous faut donc une mesure de la distance qui permette de comparer des enregistrements de longueur variable. Pour résoudre ce problème, nous utilisons la distance élastique. Contrairement à une mesure classique de distance, qui suppose un alignement direct des points correspondants des deux séquences, la distance élastique peut "étirer" ou "compresser" certaines parties des séquences pour minimiser l'écart total.

Le principe repose sur la construction d'une matrice de coûts accumulés. Chaque élément de cette matrice représente le coût cumulé minimal pour aligner une sous-séquence de la première séquence à une sous-séquence de la seconde. Ce coût est calculé récursivement en combinant les distances locales entre les points des deux séquences (ici, les coefficients LPC de chaque trame) et en tenant compte des chemins d'alignement possibles : insertion, suppression ou substitution.

Ces chemins peuvent être pondérés par des coefficients, mais içi on garde 1 partout :

```
def distance_elastique(a,b):
    tableau_res = np.zeros((len(a),len(b)) )
    wv = 1.0
    wd = 1.0
    wh = 1.0
```

FIGURE 9 – Poids des chemins

En particulier, le tableau tableau_res dans notre implémentation conserve les coûts accumulés pour chaque combinaison de points des deux séquences. La distance locale d_{ij} entre deux points est calculée comme la somme des carrés des différences entre leurs dimensions respectives (LPC). L'algorithme explore ensuite les trois chemins possibles pour atteindre chaque point (i,j) : par le haut (insertion), par la gauche (suppression), ou en diagonale (substitution).

```
    for i in range(len(a)):
        for j in range(len(b)):

            # dij est calculé dans une boucle pour être compris par numba
            dij = 0.0
            for k in range(len(a[i])):
                dij += (a[i][k] - b[j][k]) ** 2

            if j==0 and i == 0:
                tableau_res[i,j] = dij
            elif i ==0 :
                tableau_res[i,j] = tableau_res[i,j-1] + dij*wh
            elif j == 0:
                tableau_res[i,j] = tableau_res[i-1,j] + dij*wv
            else:
                terme1 = tableau_res[i-1,j] + wv*dij
                terme2 = tableau_res[i-1,j-1] + wd*dij
                terme3 = tableau_res[i,j-1] + wh*dij
                tableau_res[i,j] = min(terme1,terme2,terme3)
```

FIGURE 10 – Calcul de meilleur chemin à d_{ij}

La valeur minimale parmi ces trois options est retenue pour garantir le coût total le plus faible à chaque étape.

La distance élastique finale correspond à la valeur accumulée au coin inférieur droit de la matrice, qui représente le coût minimal pour aligner toute la première séquence sur la seconde.

```
    return tableau_res[len(a)-1,len(b)-1]
```

FIGURE 11 – Résultat final de la distance élastique

3.3.2 K plus proches voisins

Pour comparer les enregistrements de test avec ceux d'entraînement, nous devons calculer une matrice de distances. Chaque élément de cette matrice représente la distance élastique entre un enregistrement de test et un enregistrement d'entraînement. La fonction `calculate_distance_matrix` réalise cette tâche.

```
@njit
def calculate_distance_matrix(train,test,batch_size):

    matrix = np.zeros(( len(test),len(train)))

    for i, lpc_one_audio_train in enumerate(train):
        if i%batch_size == 0:
            print(f"Calcul des distances avec la piste audio d'entrainement n°{i} sur {len(train)}")
        for j, lpc_one_audio_test in enumerate(test):
            matrix[j, i] = distance_elastique(lpc_one_audio_train, lpc_one_audio_test)
    return matrix
```

FIGURE 12 – Fonction calculate_distance_matrix

La fonction `calculate_distance_matrix` fonctionne de la manière suivante :

- Elle prend en entrée les coefficients LPC des enregistrements d'entraînement et de test, ainsi qu'une taille de lot pour afficher les messages de progression.
- Elle initialise une matrice de zéros de dimensions (nombre de tests, nombre d'entraînements).
- Pour chaque enregistrement d'entraînement, elle calcule la distance élastique avec chaque enregistrement de test en utilisant la fonction `distance_elastique`.
- Elle affiche un message de progression tous les *batch_size* enregistrements d'entraînement.
- Elle retourne la matrice des distances.

Une fois la matrice des distances calculée, nous utilisons l'algorithme des k plus proches voisins (k-NN) pour prédire les étiquettes des enregistrements de test. La fonction `knn_predict` réalise cette tâche.

```python
def knn_predict(dists, labels_train , k):
    assert len(labels_train)== dists.shape[1]
    predicted_labels= []

    for test_dists in dists:
        nearest_neighbors = k_min_args(test_dists,k)

        #a dictionary for counting the labels of the neighbors
        neighbors_labels = {}
        for neighbor in nearest_neighbors:
            neighbors_labels[labels_train[neighbor]]= neighbors_labels.get(labels_train[neighbor],0)+1

        most_neighbor_label = max(neighbors_labels, key=neighbors_labels.get)
        predicted_labels.append(most_neighbor_label)
    return predicted_labels
```

FIGURE 13 – Fonction calculate_distance_matrix

La fonction `knn_predict` fonctionne de la manière suivante :

— Elle prend en entrée la matrice des distances, les étiquettes des enregistrements d'entraînement et le nombre de voisins k.
— Pour chaque enregistrement de test, elle trouve les indices des k plus proches voisins dans les enregistrements d'entraînement en utilisant la fonction `k_min_args`.
— Elle compte les étiquettes des voisins les plus proches dans un dictionnaire.
— Elle détermine l'étiquette la plus fréquente parmi les voisins et l'ajoute à la liste des étiquettes prédites.
— Elle retourne la liste des étiquettes prédites.

3.3.3 Accélération des temps de calcul avec numba

Comme on a plusieurs milliers d'enregistrements, certaines fonctions peuvent être longues à exécuter, comme le calcul des coefficients LPC pour chaque trame de chaque audio.

C'est cependant le calcul des distances qui est le plus lent, car il requiert de remplir la matrice des distances (de dimensions n_trames_audio_train * n_trames_audio_test). Pour chaque case, il faut calculer la distance L2 entre deux coefficients LPC, de taille ordre_modele, et tout ce calcul doit être fait nb_audio_train * nb_audio_test fois!!!

Pour accélérer le processus, une première approche est de ne garder qu'une partie des données.

```python
used_data, discarded_data = split_train_test([data_audio,data_Fe,labels], train_ratio=1)
train_data, test_data = split_train_test(used_data, train_ratio=0.92)
```

FIGURE 14 – Suppression d'une Partie des données avec la fonction split_train_test

Cependant, cette approche réduit alors nos performances.

Une meilleure approche consiste à utiliser numba.

Numba est un compilateur juste-à-temps (JIT) pour Python qui permet d'accélérer les fonctions en les compilant en code machine. En utilisant le décorateur `@njit`, nous pouvons compiler nos fonctions critiques pour améliorer considérablement les performances.

```python
from numba import njit
from sklearn.metrics import ConfusionMatrixDisplay


def calcul_lpc_frame(f, ordre_modele):···

def normalize_lpc_coefficients(lpc_coeffs):···

def process_audio_to_lpc(data_audio, labels, ordre_mode

@njit
def distance_elastique(a, b):···

@njit
def calculate_distance_matrix(train,test,batch_size):···
```

FIGURE 15 – Utilisation du décorateur njit

Dans notre cas, nous avons utilisé numba pour accélérer les fonctions `distance_elastique` et `calculate_distance_matrix`. Pour que le code soit compatible avec numba, nous avons dû modifier certaines parties pour fonctionner avec numpy et des boucles for explicites. Cela permet de réduire significativement le temps de calcul de la matrice des distances, rendant ainsi notre algorithme plus efficace et plus rapide.

3.4 Intégration des éléments dans le script principal

```python
if __name__ == "__main__":
    input_folder = './digit_dataset'

    # La sélection de paramètres a été réalisée au préalable
    ordre_modele = 10
    k = 5

    # Charger les données audio
    data_audio, data_Fe, labels = load_data(input_folder)

    # Diviser les données en ensembles d'entraînement et de test
    used_data, discarded_data = split_train_test([data_audio, data_Fe, labels], train_ratio=1)
    train_data, test_data = split_train_test(used_data, train_ratio=0.92)

    # Extraire les données d'entraînement et de test
    (data_audio_train, data_Fe_train, original_labels_train) = train_data
    (data_audio_test, data_Fe_test, original_labels_test) = test_data
    accuracy = []

    # Convertir les données audio en coefficients LPC
    list_of_audio_as_lpc_list_train, labels_train = process_audio_to_lpc(data_audio_train, original_labels_train, ordre_modele, 200, "train")
    list_of_audio_as_lpc_list_test, labels_test = process_audio_to_lpc(data_audio_test, original_labels_test, ordre_modele, 200, "test")

    # Calculer la matrice des distances
    matrix = calculate_distance_matrix(list_of_audio_as_lpc_list_train, list_of_audio_as_lpc_list_test, 200)

    # Prédire les étiquettes avec k-NN
    predicted_labels = knn_predict(matrix, labels_train, k)
    print(f"longueur des predictions: {len(predicted_labels)}, longueur des data_train : {len(list_of_audio_as_lpc_list_train)}, longueur des test : {len(list_of_audio_as_lpc_list_test)} ")

    # Calculer la précision
    accuracy = sum([(predicted_labels[i] == labels_test[i]) for i in range(min(len(labels_test), len(predicted_labels)))]) / min(len(labels_test), len(predicted_labels))
    print(f"On obtient un taux de classification correcte de {accuracy} %")
```

FIGURE 16 – Script principal

3.5 Résultats

Notre modèle possède deux hyperparamètres : k et ordre_modèle. Pour les comparer, nous avons généré les graphes suivants :

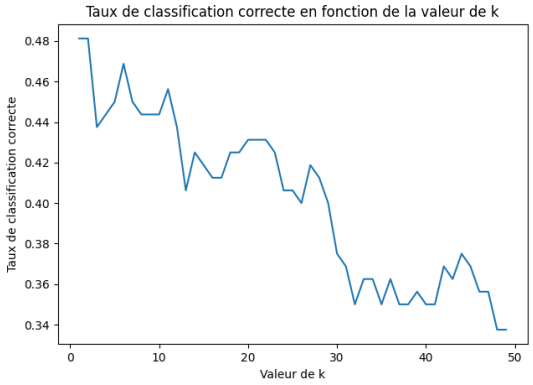


FIGURE 17 – Précision en fonction de k (ordre_modèle = 25)

On remarque que la précision semble baisser à mesure que k augmente. Cependant, ces mesures sont à relativiser : les données d'entraînement et de test ne contiennent que quatre personnes qui disent chaque chiffre des dizaines de fois. Il semble logique qu'au moins un fichier d'entraînement par la même personne soit proche et suffise pour deviner le chiffre énoncé. Pour cette raison, nous gardons k=5. Il faudrait idéalement tester avec des voix non présentes dans les fichiers d'entraînement, ce qui sera fait en deuxième partie.

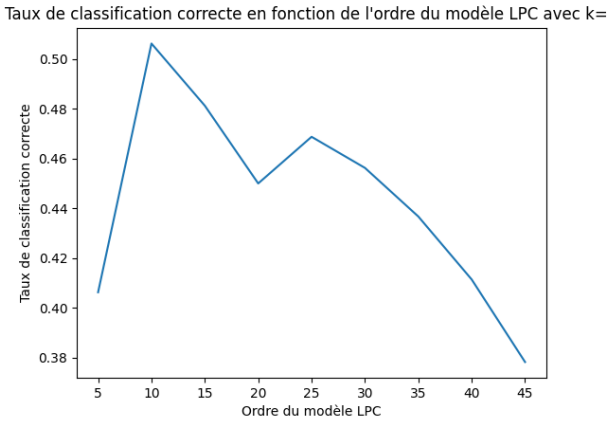


FIGURE 18 – Précision en fonction de l'ordre du modèle (k=5)

Le meilleur résultat est obtenu pour un ordre_modèle d'environ 15, après quoi la qualité commence à baisser. Nous obtenons donc k = 5 et ordre_modèle = 15. Les résultats pour ces paramètres sont ensuite consignés :

```
longueur des predictions: 160, longueur des data_train : 1834, longueur des test : 160
On obtient un taux de classification correcte de 0.5 %

 Matrice de confusion:
[[14.  1.  2.  2.  0.  1.  0.  0.  1.  1.]
 [ 3.  8.  1.  1.  1.  1.  0.  0.  0.  0.]
 [ 1.  0. 10.  2.  0.  0.  1.  0.  1.  0.]
 [ 1.  0.  3.  5.  0.  1.  1.  0.  1.  1.]
 [ 1.  0.  1.  2.  6.  0.  1.  1.  0.  0.]
 [ 1.  4.  0.  0.  1.  4.  0.  1.  0.  1.]
 [ 3.  0.  0.  0.  0.  1. 14.  0.  2.  3.]
 [ 1.  1.  2.  1.  1.  0.  0.  6.  0.  1.]
 [ 4.  2.  0.  2.  2.  0.  1.  0.  6.  2.]
 [ 2.  1.  2.  1.  0.  1.  1.  1.  0.  7.]]
```

FIGURE 19 – Précision du modèle

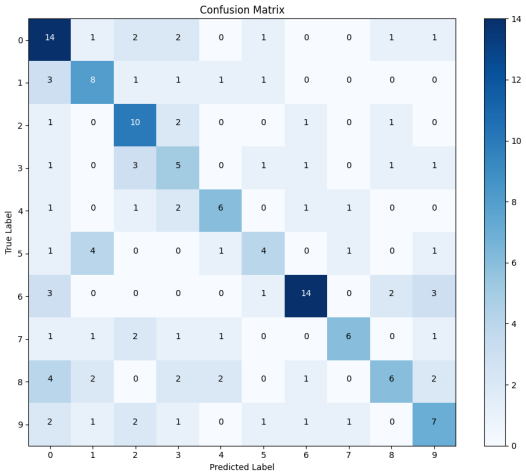


FIGURE 20 – Matrice de confusion

Les classes 0 et 6 sont les mieux classées, tandis que la 4 sur 12 des samples de la classe "5" ont été classés en "1". Cela peut être du à une prononciation relativement similaire de "five" et "one"

Le taux d'instance correctement classé est de 50%, ce qui est significativement mieux que le hasard. Il reste tout de même une grande marge d'amélioration, c'est pourquoi on implémente en partie une autre architecture basé sur les coefficients MFCC et les Modèles de Markov Cachés.

4 Développement d'un système de reconnaissance de la parole basé sur les coefficients MFCC et les Modèles de Markov Cachés.

4.1 Révisions à `load_data`

On n'utilise pas la même fonction que dans la partie 1, car on a repéré un souci méthodologique : les données d'entraînement du modèle contiendraient des samples de la même personne qui dit les samples de test.

Nous avons donc ajouté deux paramètres, include_substring et exclude_substring, à la fonction `load_data`. Ces paramètres permettent de filtrer les fichiers audio à charger en fonction de leur nom. Par exemple, nous avons choisi un des locuteurs, "theo", dont nous ne prendrons pas en compte les audios dans les données d'entraînement, et dont les audios constitueront les données de test. Ainsi, nous mesurons la capacité du système à s'adapter à de nouveaux locuteurs.

```
        for filename in [x for x in os.listdir(subfolder) if x.endswith('.wav')]:
            # Appliquer les filtres include_substring et exclude_substring
            if include_substring and include_substring not in filename:
                continue
            if exclude_substring and exclude_substring in filename:
                continue
            filepath = os.path.join(subfolder, filename)
```

FIGURE 21 – Filtrage des données d'entrées

En outre, la nouvelle fonction `load_data` retourne un dictionnaire où les clés sont les étiquettes et les valeurs sont des listes de tuples contenant les données audio et les fréquences d'échantillonnage. Cela diffère de la fonction de la partie 1, qui retournais trois listes distinctes : data_audio, data_Fe et labels. Cette différence de format des sorties sera utile pour faire un autre ajustement méthodologique dans la fonction `split_train_test_by_label`. Elle sera également plus simple d'utilisation lors de l'entraînement des modèles de Markov cachés sur chacun des labels.

4.2 Révisions de split_train_test_by_label

Cette fois, nous effectuons le split en gardant les proportions pour chaque classe. Concrètement, comme nos données sont sous la forme d'un dictionnaire, nous itérons sur les labels (qui sont les clés du dictionnaire) et nous effectuons un split classique pour chaque label.

```
    label_list = data_dict.keys()
    train_dict = dict()
    test_dict = dict()

    # Itérer sur chaque label (clé du dictionnaire)
    for label in label_list:
        # Séparer les données pour ce label en ensembles d'entraînement et de test
        data_train, data_test = split_train_test(data_dict[label], train_ratio, randomize)
        train_dict[label] = data_train
        test_dict[label] = data_test

    return train_dict, test_dict
```

FIGURE 22 – Modifications de split_train_test_by_label pour garder les proportions

Cette fonction n'est pas utilisée lors du test final, car la séparation est déjà faite en incluant et excluant les substrings, mais elle est utilisée pour la validation sur les données en excluant "theo".

```
if __name__ == "__main__":

    input_folder = './digit_dataset'
    sr = 16000

    n_mfcc = 15
    win_length=512
    hop_length=512//2

    # 1) Chargement des données
    data_audio = load_data(input_folder, sr=sr, exclude_substring='theo')

    # 2) Split data into train and valid sets by label
    data_audio_Fe_train, data_audio_Fe_valid = split_train_test_by_label(data_audio, train_ratio=0.8, randomize=True)
```

FIGURE 23 – Utilisation de split_train_test_by_label pour les données de validation

Idéalement, nous aurions aussi exclu les locuteurs de validation des données d'entraînement, mais les données ne comportant que 4 locuteurs différents, nous avons préféré faire ainsi.

4.3 Calcul des coefficients MFCC

Les coefficients cepstraux de fréquences Mel (MFCC) sont des caractéristiques spectrales cruciales dans l'analyse audio, car ils modélisent la perception humaine des sons. Leur calcul passe par plusieurs étapes : le signal est divisé en fenêtres temporelles, une transformée de Fourier est appliquée pour obtenir le spectre, les puissances spectrales sont ensuite mappées sur une échelle Mel (adaptée à la perception humaine). Enfin, après avoir pris le logarithme des puissances pour simuler la dynamique auditive, une transformée en cosinus discrète (DCT) condense les informations pertinentes.

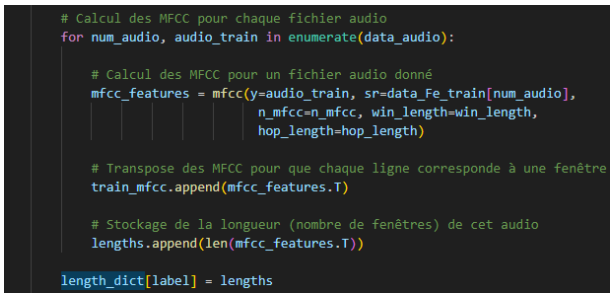
La fonction `librosa.feature.mfcc` est utilisée pour calculer les MFCC pour un signal donné. Cependant, le traitement de nos données nécessite une manipulation plus complexe, car elles sont organisées sous forme de dictionnaire où chaque clé correspond à un label. De plus, la fonction `hmmlearn.hmm.GaussianHMM.fit(X,length)`, prend en entrée des donnés qui doivent être formatées de manière spécifique. Le vecteur **X** contient toutes les fenêtres MFCC (chaque ligne correspond aux coefficients calculés pour une fenêtre temporelle), tandis que le vecteur **lengths** enregistre le nombre de fenêtres dans chaque fichier audio. On utilise donc une fonction `calculate_mfcc_and_length_for_dict` pour gérer les formatages des données.

Cette fonction traite chaque label indépendamment en récupérant les paires (audio, fréquence d'échantillonnage) associées dans le dictionnaire d'entrée. Pour chaque fichier audio, elle calcule les MFCC à l'aide des paramètres spécifiés (*n_mfcc*, *win_length*, *hop_length*).

```
mfcc_dict = {}
length_dict = {}

# Parcours des labels du dictionnaire d'entrée
for label in data_audio_Fe_dict.keys():

    # Extraction des données audio et fréquences d'échantillonnage associées
    data_audio = []  # Liste pour les fichiers audio de ce label
    data_Fe_train = []  # Liste pour les fréquences d'échantillonnage correspondantes
    for audio, Fe in data_audio_Fe_dict[label]:
        data_audio.append(audio)
        data_Fe_train.append(Fe)

    train_mfcc = []
    lengths = []

    # Calcul des MFCC pour chaque fichier audio
    for num_audio, audio_train in enumerate(data_audio):

        # Calcul des MFCC pour un fichier audio donné
        mfcc_features = mfcc(y=audio_train, sr=data_Fe_train[num_audio],
                             n_mfcc=n_mfcc, win_length=win_length,
                             hop_length=hop_length)
```

FIGURE 24 – Calcul des coefficients mfcc

Le vecteur **lengths**, qui enregistre le nombre de fenêtres temporelles calculées pour chaque fichier audio. Ce vecteur est nécessaire pour `hmmlearn.hmm.GaussianHMM.fit`, car il permet de reconstituer les segments d'origine dans **X**.



```
# Calcul des MFCC pour chaque fichier audio
for num_audio, audio_train in enumerate(data_audio):

    # Calcul des MFCC pour un fichier audio donné
    mfcc_features = mfcc(y=audio_train, sr=data_Fe_train[num_audio],
                         n_mfcc=n_mfcc, win_length=win_length,
                         hop_length=hop_length)

    # Transpose des MFCC pour que chaque ligne corresponde à une fenêtre
    train_mfcc.append(mfcc_features.T)

    # Stockage de la longueur (nombre de fenêtres) de cet audio
    lengths.append(len(mfcc_features.T))

length_dict[label] = lengths
```

FIGURE 25 – Calcul du vecteur length dans `calculate_mfcc_and_length_for_dict`

Enfin, ces données sont concaténées pour produire un vecteur **X** contenant toutes les fenêtres MFCC associées à un label. `calculate_mfcc_and_length_for_dict` retourne ensuite deux dictionnaires : **mfcc_dict**, contenant les coefficients MFCC concaténés pour chaque label, et **length_dict**, qui conserve les longueurs associées.

```
    # Calcul des MFCC pour chaque fichier audio
    for num_audio, audio_train in enumerate(data_audio):

        # Calcul des MFCC pour un fichier audio donné
        mfcc_features = mfcc(y=audio_train, sr=data_Fe_train[num_audio],
                             n_mfcc=n_mfcc, win_length=win_length,
                             hop_length=hop_length)

        # Transpose des MFCC pour que chaque ligne corresponde à une fenêtre
        train_mfcc.append(mfcc_features.T)

        # Stockage de la longueur (nombre de fenêtres) de cet audio
        lengths.append(len(mfcc_features.T))

    length_dict[label] = lengths

    # Concaténation des MFCC pour tous les fichiers audio d'un même label
    mfcc_dict[label] = np.concatenate(train_mfcc, axis=0)

return mfcc_dict, length_dict
```

FIGURE 26 – Calcul et formatage des coefficients mfcc et des vecteurs lengths

Le code complet de cette fonction est donné en annexe. 6

4.4 Création et test du model de Markov caché Gaussien

4.4.1 Principes des Modèles de Markov caché Gaussien

Les modèles de Markov cachés (HMM) sont des outils probabilistes utilisés pour modéliser des séquences où les états sous jacent sont invisibles (cachés) mais influencent les observations visibles. Ces modèles supposent que les transitions entre les états suivent un processus aléatoire, chaque état ayant une probabilité d'évoluer vers un autre. De plus, chaque état génère une observation visible (comme des caractéristiques audio) de manière aléatoire, selon une distribution propre à cet état.

L'entraînement du modèle consiste alors à estimer les paramètres, C'est-à-dire les différentes probabilités de changement d'état et d'émission, à partir des observations. Cette

étape est réalisée grâce à l'algorithme de Baum-Welch, qui permet une optimisation progressive des paramètres en exploitant les calculs intermédiaires, comme les probabilités "Forward" et "Backward"

Plus précisément, le code implémente Modèle de Markov caché Gaussien. Ce type de modèle est particulièrement adapté aux données audio, car il suppose que les observations associées à chaque état sont distribuées selon une gaussienne multivariée.

4.4.2 Implémentation

Pour implémenter ce modèle, nous utilisons la bibliothèque `hmmlearn`, et plus précisément un modèle HMM gaussien. Les états cachés représentent les différents phonèmes associés aux mots prononcés, tandis que les observations générées par ces états sont les coefficients MFCC.

Le format des données renvoyé par `calculate_mfcc_and_length_for_dict` étant compatible avec la fonction `hmmlearn.hmm.GaussianHMM.fit`, nous pouvons simplement entraîner un modèle par label.

```
n_components = 10
n_iter = 5000

model_dict = {}
for label in label_list:
    model = hmm.GaussianHMM(n_components=n_components, n_iter=n_iter, covariance_type='diag')
    model.fit(train_mfcc_dict[label], lengths_train[label])
    model_dict[label] = model
```

FIGURE 27 – Entraînement des modèles HMM

La phase de test est plus complexe. En effet, nos coefficients MFCC des fichiers tests sont concaténés dans une matrice numpy, avec les longueurs des différents échantillons stockées séparément. Pour effectuer une prédiction par échantillon, nous devons :

Calculer la somme cumulée des longueurs précédentes pour localiser chaque échantillon dans la matrice concaténée Extraire chaque échantillon en utilisant les indices calculés et sa longueur

```
    # Somme cumulée des longueurs des échantillons pour chaque label:
    # chaque sample mfcc commence à partir de la somme cumulée des longueurs des samples précédents
    cumsum = np.cumsum([0] + lengths_test[true_label][:-1])

    # Séparer les échantillons individuels pour chaque label
    mfcc_individual_samples = [
        test_mfcc_dict[true_label][start:start + length]
        for start, length in zip(cumsum, lengths_test[true_label])
    ]
```

FIGURE 28 – Extraction des échantillons de test à partir de la matrice concaténée

Pour la prédiction, nous calculons le score de chaque modèle HMM sur l'échantillon de test considéré. Le modèle donnant le meilleur score détermine la classe prédite. Ces prédictions sont stockées dans une matrice de confusion, dont la somme de la diagonale donne le nombre d'instances classés correctement. Le taux d'instance correcte est donc la somme de la diagonale sur la somme du total.

```
    # Pour chaque échantillon extrait
    for mfcc_sample in mfcc_individual_samples:

        # Initialiser le meilleur score et l'indice du meilleur label
        best_score = -np.inf
        best_label_idx = None

        # Pour chaque label candidat
        for candidate_label_idx, candidate_label in enumerate(label_list):
            model = model_dict[candidate_label]
            if model is None:
                continue
            # Calculer le score du modèle sur l'échantillon
            score = model.score(mfcc_sample)
            # Mettre à jour le meilleur score si nécessaire
            if score > best_score:
                best_score = score
                best_label_idx = candidate_label_idx

        # Incrémenter la matrice de confusion
        if best_label_idx is not None:
            confusion_matrix[true_label_idx, best_label_idx] += 1

# Calculer l'accuracy comme le rapport entre les prédictions correctes et le total
accuracy = np.sum(np.diag(confusion_matrix)) / np.sum(confusion_matrix)

return accuracy, confusion_matrix
```

FIGURE 29 – Calcul des prédictions du modèle

```
if __name__ == "__main__":

    input_folder = './digit_dataset'
    sr = 16000

    n_mfcc = 15
    win_length=512
    hop_length=512//2

    # 1) Chargement des données
    data_audio = load_data(input_folder, sr=sr, exclude_substring='theo')

    # 2) Séparation des données en ensembles d'entraînement et de validation
    data_audio_Fe_train, data_audio_Fe_valid = split_train_test_by_label(data_audio, train_ratio=0.8, randomize=True)

    # 3) Calcul des MFCC et longueurs pour les données d'entraînement et de validation
    train_mfcc_dict, lengths_train = calculate_mfcc_and_length_for_dict(data_audio_Fe_train, n_mfcc, win_length, hop_length)
    valid_mfcc_dict, lengths_valid = calculate_mfcc_and_length_for_dict(data_audio_Fe_valid, n_mfcc, win_length, hop_length)


    # Liste de tous les labels (ex: ['0', '1', '2', ...])
    label_list = list(data_audio.keys())

    # 4) Entraînement et test des modèles HMM de validation

#    n_components_list = [2,4,6,8]
#    n_iters = range(500, 5001, 500)
#    accuracies = np.zeros((len(n_components_list), len(n_iters)))
#    for id_component,n_components in enumerate(n_components_list):
#        for id_iter,n_iter in enumerate(n_iters):
#
#            model_dict = {}
#            for label in label_list:
#                model = hmm.GaussianHMM(n_components=n_components, n_iter=n_iter, covariance_type='diag')
#                model.fit(train_mfcc_dict[label], lengths_train[label])
#                model_dict[label] = model

#            num_labels = len(label_list)
#            confusion_matrix = np.zeros((num_labels, num_labels), dtype=int)

#            accuracy, confusion_matrix = test_model(model_dict, valid_mfcc_dict, lengths_valid)
#            print(f"n_components={n_components}, n_iter={n_iter}, accuracy={accuracy}")
#            accuracies[id_component, id_iter] = accuracy
```

FIGURE 30 – Script pour la validation des paramètres

4.5 Validation des hyperparamètres

La première partie du script principal est partiellement commentée, et correspond au code utilisé pour trouver les hyper-paramètres n_iter et n_components.

n_components correspond aux nombres d'états cachés du modèle, et n_iter correspond au nombre d'itération de l'algorithme de Baum-Welch

Sur la plage recherchée (2,4,6,8) plus n_components augmente, plus la précision est bonne. de plus , (2,4,6,8) plus n_components augmente, plus le modèle prend d'itérations à converger, mais converge plus haut. on est limité par notre puissance de calcul pour voir le comportement quand les paramètres augmentes encore plus : pour le test final, on se contentera de n_components = 10,n_iter = 5 000

4.6 Test et résultats finaux

Test et résultats finaux Nous testons le modèle avec deux configurations :

— Configuration 1 : n_components = 10, n_iter = 5 000, les meilleurs paramètres

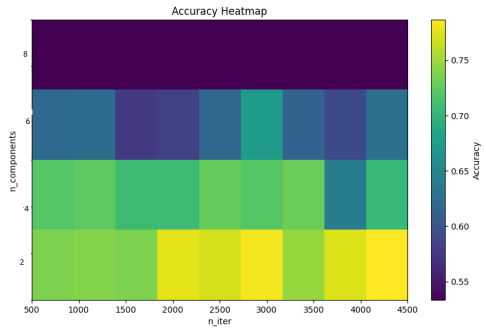


FIGURE 31 – Taux d'instance correctement classé en fonction de n_iter et de n_components

obtenus par la validation
— Configuration 2 : n_components = 20, n_iter = 10 000, les paramètres limitant notre puissance de calcul

Pour évaluer la capacité de généralisation du modèle à de nouveaux locuteurs, nous excluons les enregistrements de "theo" de l'entraînement et les utilisons comme données de test.

```
input_folder = './digit_dataset'
sr = 16000

n_mfcc = 15
win_length=512
hop_length=512//2

data_audio = load_data(input_folder, sr=sr, exclude_substring='theo')
train_mfcc_dict, lengths_train = calculate_mfcc_and_length_for_dict(data_audio, n_mfcc, win_length, hop_length)


data_audio_Fe_test = load_data(input_folder, sr=sr, include_substring='theo')
test_mfcc_dict, lengths_test = calculate_mfcc_and_length_for_dict(data_audio_Fe_test,n_mfcc, win_length, hop_length)

n_components = 20
n_iter = 10000
```

FIGURE 32 – Préparation des données de test sur les audios de theo

L'entraînement et le test des modèles se font via les fonctions précédemment définies :

```
# Paramètres du modèle HMM
n_components = 10
n_iter = 5000

# Entraînement d'un modèle HMM pour chaque label
model_dict = {}
for label in label_list:
    # Création et entraînement du modèle HMM gaussien
    model = hmm.GaussianHMM(n_components=n_components, n_iter=n_iter, covariance_type='diag')
    model.fit(train_mfcc_dict[label], lengths_train[label])
    model_dict[label] = model

# Initialisation de la matrice de confusion
num_labels = len(label_list)
confusion_matrix = np.zeros((num_labels, num_labels), dtype=int)

# Test du modèle et calcul de la précision
accuracy, confusion_matrix = test_model(model_dict, valid_mfcc_dict, lengths_valid)
```

FIGURE 33 – Processus d'entraînement et de test

Configuration 1 ($\mathbf{n}_{components} = 10, n_iter = 5000$)

Précision : 76.3

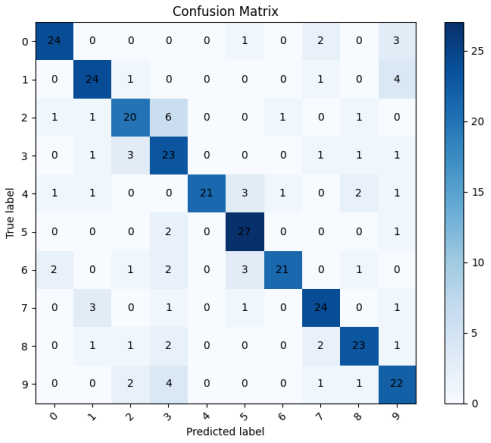


FIGURE 34 – Matrice de confusion - Configuration 1

Configuration 2 (n$_{c}omponents = 20, n_{i}ter = 10000$)
Précision : 85.7

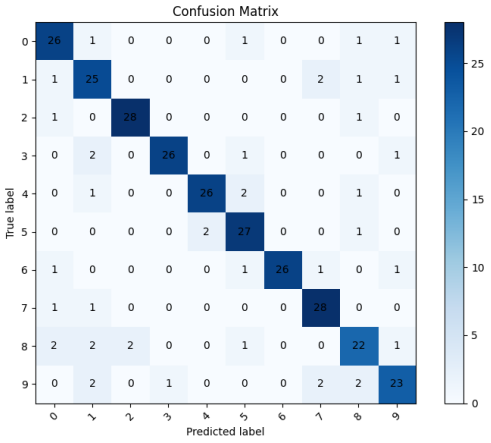


FIGURE 35 – Matrice de confusion - Configuration 2

Ces résultats sont remarquables considérant que :

Les enregistrements de test proviennent d'un locuteur absent des données d'entraînement, et les données d'entraînement ne contiennent que 3 locuteurs De plus L'amélioration significative entre les deux configurations suggère qu'augmenter les paramètres pourrait encore améliorer les performances. L'optimisation des paramètres MFCC offre également une piste d'amélioration.

Les matrices de confusion montrent une amélioration notable : la confusion sur la classification des samples de la classe 2 présente dans la première configuration disparaît dans la seconde. (de 20 instances classé correctement à 28 sur 30)

5 Conclusion

Nous avons implémenté deux approches pour la reconnaissance vocale : LPC + k-NN et MFCC + HMM. La combinaison LPC + k-NN a donné des résultats significatifs avec un taux de classification correcte de 50%. Cependant, les résultats obtenus avec la combinaison MFCC + HMM sont bien meilleurs, avec des précision 76.3% dans la première Configuration et 85.7% dans la seconde. Ces algorithmes sont plus optimisés et offrent encore une marge d'amélioration des paramètres.

La prochaine étape consisterait à utiliser une base de données de mots beaucoup plus grande et à intégrer des modèles statistiques de langage pour améliorer la pondération. La combinaison de MFCC avec des modèles de langage semble très prometteuse pour de futures améliorations.

6 Annexe

6.1 Partie 2

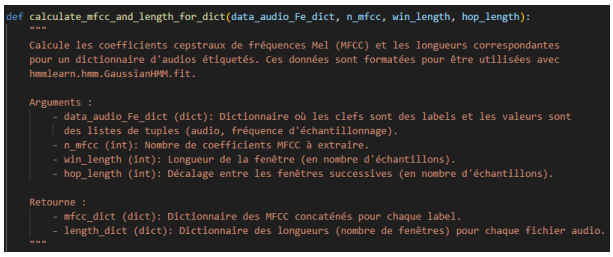


FIGURE 36 – Docstring de calculate_mfcc_and_length_for_dict

```python
def calculate_mfcc_and_length_for_dict(data_audio_Fe_dict, n_mfcc, win_length, hop_length):
    """la docstring, trop longue, n'a pas été inclu dans ce sreenshot. voir le code source pour la docstring complète"""
    mfcc_dict = {}
    length_dict = {}

    # Parcours des labels du dictionnaire d'entrée
    for label in data_audio_Fe_dict.keys():

        # Extraction des données audio et fréquences d'échantillonnage associées
        data_audio = []  # Liste pour les fichiers audio de ce label
        data_Fe_train = []  # Liste pour les fréquences d'échantillonnage correspondantes
        for audio, Fe in data_audio_Fe_dict[label]:
            data_audio.append(audio)
            data_Fe_train.append(Fe)

        train_mfcc = []
        lengths = []

        # Calcul des MFCC pour chaque fichier audio
        for num_audio, audio_train in enumerate(data_audio):

            # Calcul des MFCC pour un fichier audio donné
            mfcc_features = mfcc(y=audio_train, sr=data_Fe_train[num_audio],
                                 n_mfcc=n_mfcc, win_length=win_length,
                                 hop_length=hop_length)

            # Transpose des MFCC pour que chaque ligne corresponde à une fenêtre
            train_mfcc.append(mfcc_features.T)

            # Stockage de la longueur (nombre de fenêtres) de cet audio
            lengths.append(len(mfcc_features.T))

        length_dict[label] = lengths

        # Concaténation des MFCC pour tous les fichiers audio d'un même label
        mfcc_dict[label] = np.concatenate(train_mfcc, axis=0)

    return mfcc_dict, length_dict
```

FIGURE 37 – `calculate_mfcc_and_length_for_dict` , sans la docstring