Compte rendu TP d'initiation à la recherche

auteur : Yann ROBLIN

Introduction

Dans le cadre du cours à l'initiation à la recherche nous avons été emmené à observer les erreurs chronologique des instruments de mesure. Les objectif de ce cours sont d'observer puis corrigé ces erreurs pour ensuite pouvoir les utiliser. Dans une première partie nous allons observer puis corrigée l'erreur. En seconde partie nous utiliserons les données corrigés pour en tirer de l'information

/!\ Attention

Comme mon ordinateur personnelle ne me permet de pouvoir exécuter le travail demandé, les programmes ainsi que se rendu sont effectué sur Google Colab. Je ne peux donc pas garantir le bon fonctionnement du projet le fichier Python rendu avec celui-ci.

Partie 1 Observation de l'erreur

Dans cette partie nous allons observer l'erreur

```
import os

# Vérifie si le code est exécuté sur Google Colab
if 'COLAB_GPU' in os.environ:
    # Commandes à exécuter uniquement sur Google Colab
    !git clone https://github.com/grifire/erreur_signal.git
    %cd erreur_signal
else:
    # Commandes à exécuter si ce n'est pas sur Google Colab
    print("Pas sur Google Colab, ces commandes ne seront pas exécutées.")
```

```
fatal: destination path 'erreur_signal' already exists and is not an empty directory.
/content/erreur_signal
```

```
from copy import deepcopy
import soundfile as sf
import numpy as np
import scipy as sp
from sklearn.manifold import TSNE
from matplotlib import pyplot as plt
```

```
data1, Fe1 = sf.read("/content/erreur_signal/ONECAT_20200114_150201_247.wav")
data2, Fe2 = sf.read("/content/erreur_signal/S70_20200114_150211_982.wav")

```

Trouver les PPS

Les PPS sont des impulsion envoyés par satélite chaque seconde. C'est sur ces PPS que nous baserons notre mesure du temps. La fonction GetPPS nous permet de récupérer le premier PPS d'une piste

```
"""
GetPPS permet de trouver le début et la fin du signal PPS
Entrée :
  - data : données du signal
Sortie :
  - startPPS : début du signal PPS
  - endPPS : fin du signal PPS
"""
def GetPPS(data) :
  startPPS = 0
  while(data[startPPS][4]) < 0.5 :
    startPPS+=1
  endPPS = startPPS
  while data[endPPS][4] > -0.5 :
    endPPS+=1
  return startPPS, endPPS

print(f"Premier PPS de data1 : {GetPPS(data1)[0]}")
```

```
Premier PPS de data1 : 256111
```

Observation de l'erreur

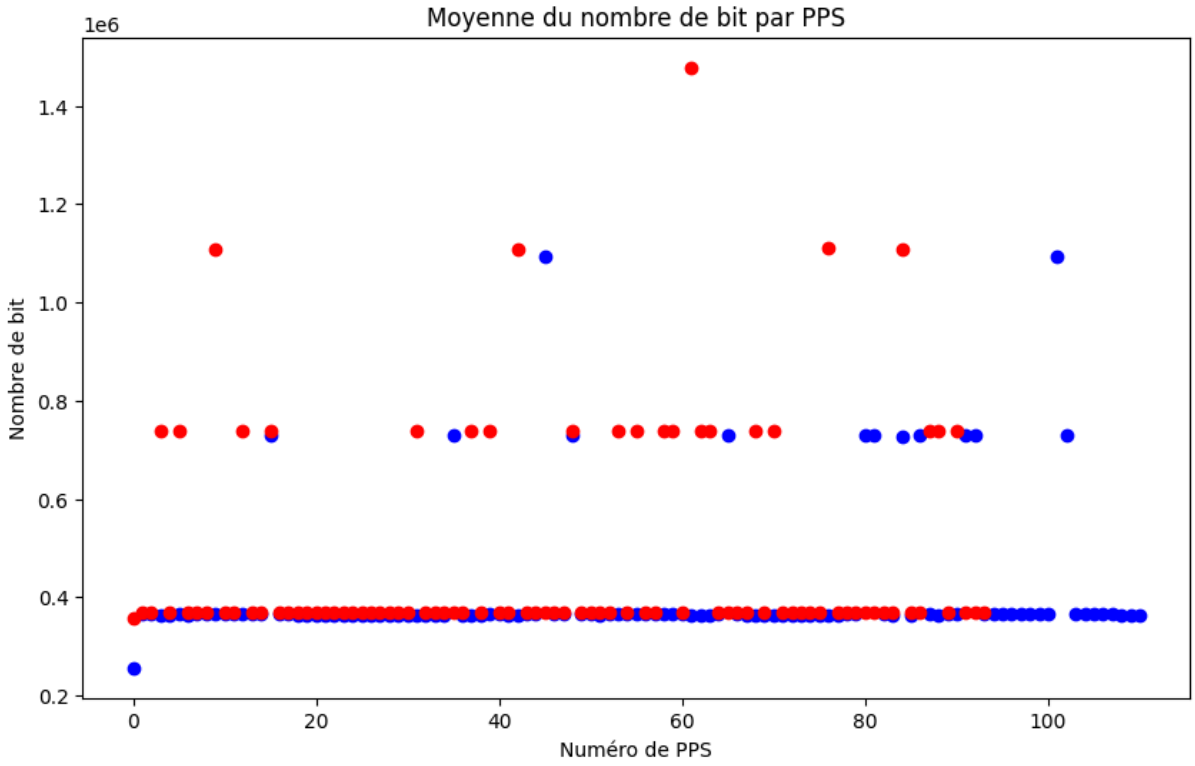
Nous allons observer l'erreur des deux données à l'aide d'un graphique graphique

```
"""
AveragePPS permet de calculer la moyenne du signal PPS
Entrée :
  - data : données du signal
  - no_extrem : si vrai garde les élément au extrimité qui sont imcomplètes
Sortie :
  - average_bit : tableau du nombre de bit par seconde
  - average_data : tableau de data séparé par seconde
"""
def AveragePPS(data, no_extrem=False) :
  average_bit = []
  tmp = 0
  pre = data[0][4]
  average_data = []
  i1p = 0
  for k in range(0,len(data)) :
      if data[k][4] > 0.6 and pre < 0.5 :
          average_bit +=[tmp]
          tmp = 0
          average_data.append(deepcopy(data[i1p:k]))
          i1p = k
      else :
          tmp += 1
      pre = data[k][4]
  if no_extrem :
    return average_data, average_bit
```

```
    # Ignore le premier et dernier élément qui sont probablement incomplé
    return average_data[1:-1], average_bit[1:-1]
```

```
#Bit moyen par PPS
_, av_bit_data1 = AveragePPS(data1, True)
_, av_bit_data2 = AveragePPS(data2, True)
plt.figure(figsize=(10, 6))
plt.title("Moyenne du nombre de bit par PPS")
plt.xlabel("Numéro de PPS")
plt.ylabel("Nombre de bit")
plt.scatter(range(len(av_bit_data1)), av_bit_data1, marker='o', color='blue', )
plt.scatter(range(len(av_bit_data2)), av_bit_data2, marker='o', color='red')
#plt.scatter(av_bit_data2,len(av_bit_data2))
print(f"Fréquence moyenne data1 obesrver / Fréquence indiqué: {sum(av_bit_data1) //
print(f"Fréquence moyenne data2 obesrver/ Fréquence indiqué: {sum(av_bit_data2) //
plt.show()
```

```
Fréquence moyenne data1 obesrver / Fréquence indiqué: 412811 / 384000
Fréquence moyenne data2 obesrver/ Fréquence indiqué: 487511 / 384000
```



Observations

Nous pouvons observer dans un premier temps qu'il y a une variation dans le nombre de donnée entre chaque PPS. La fréquence réél est différente de la fréquence annoncé. Nous observons aussi une différence de fréquence moyenne entre les deux drones.

Reconstruction du signal

Maintenant que nous connaissons l'erreur nous allons reconstruire les données. Pour ce faire il suffie de rééchantillonné. Les données seront rééchantillonnée sur la moyenne des deux fréquences réél. De cette façon les donnée seront de taille identique.

```
"""
ReconstructSignal permet de reconstruire le signal.
Il rééchantillonne les données et les concatène.
Entrée :
  - average_data : tableau de data séparé par seconde
  - nb_bit : nombre de bit par seconde
Sortie :
  - data_final : signal reconstruit
"""
def ReconstructSignal(average_data, nb_bit) :
  resamp_data = []
  # Rééchantillonnage du signal
  for m in average_data :
      resamp_data.append(sp.signal.resample(m, int(nb_bit)))
  # Concatenate la data pour pouvoir l'exploiter
  data_final = np.concatenate(resamp_data, axis=0)
  return data_final
```

Synchronisation

Une fois le signal rééchantillionné, faut maintenant synchroniser les deux signaux. Nous connaisons la date de début des signaux et savons qu'il dur 2 minutes chacun. Comme S70 commence avec 2 10 seconde après ONECAT nous allons observer les signaux sur 100 secondes.

```
def Traitement(file1, start1, end1, file2, start2, end2) :
  # Vérification de la cohérence des tailles d'échantillon
  if end1-start1 != end2-start2 :
    print("Erreur dans la fonciton Traitement:\n\tLa taille des données de sortie n
    return -1
  print("Loading files...")
  data1, Fe1 = sf.read(file1)
  data2, Fe2 = sf.read(file2)

  print("Reconstructing data...")
  startPPS1, endPPS1 = GetPPS(data1) # Récupération du premier PPS de data1
  startPPS2, endPPS2 = GetPPS(data2) # Récupération du premier PPS de data2

  # Calcul des moyennes de bit par PPS
  average_data1, average_bit1 = AveragePPS(data1)
  average_data2, average_bit2 = AveragePPS(data2)

  # Début et fin de chaque signal
  average_data1, average_bit1 = average_data1[start1:end1], average_bit1[start1:end
  average_data2, average_bit2 = average_data2[start2:end2], average_bit2[start2:end

  # Reconstruction du signal avec concatennation basé sur la moyenne du nombre de b
  Fe = (sum(average_bit1) // len(average_bit1) + sum(average_bit2) // len(average_b
  data1 = ReconstructSignal(average_data1, Fe)
  data2 = ReconstructSignal(average_data2, Fe)
  return data1, data2, Fe

```

Construction de la donnée

Nous allons travailler sur les distances. Nous utiliserons la fonction BuildDeltas. Cette fonction fera :

- Repérer les n plus grand peaks d'énergie pour chaque piste
- Pour chaque peak, la faire corréler avec les autres pistes afin de trouver un écart de temps.
- Fournir une liste de l'ensemble des écart de temps (deltas)

Nous prendrons 500 peaks par pistes. Ce n'est pas temps que ça mais cela permet de réaliser le travaille dans un temps résonnable.

```
"""
CalcEnergie calcul l'énergie du signal et retourne les nb_peaks peaks les plus
élevés.
Entrée :
  - data : données du signal
  - piste : numéro de la piste
  - Freq : fréquence du signal
  - ratio : facteur de conversion de fréquence
  - window_size : taille de la fenêtre de calcul de l'énergie
  - nb_peaks : nombre de peaks à retourner
Sortie :
  - peaks : tableau des nb_peaks peaks les plus élevés
  - index : tableau des index des nb_peaks peaks les plus élevés
"""
def CalcEnergie(data, piste, Freq, ratio, window_size, nb_peaks) :
  interval = Freq // ratio
  window = interval * window_size
  peaks = []
  for i in range(0,len(data),interval) :
    peaks += [sum(data[i:i+window,piste]**2)]
  peaks = np.array(peaks)
  index = np.argsort(peaks)[-nb_peaks:]
  return peaks[index], index, interval, window
```

```
"""
PeaksSimilitude permet de calculer les distances entre les peaks de deux pistes.
Entrée :
  - data1 : données du signal 1
  - piste1 : numéro de la piste
  - data2 : données du signal 2
  - piste2 : numéro de la piste
  - index : tableau des index des nb_peaks peaks les plus élevés
  - interval : intervalle entre deux fenêtres
  - window : taille de la fenêtre de calcul de l'énergie
  - marge : marge de recherche
  Sortie :
  - deltas : tableau des distances entre les peaks
"""
def PeaksSimilitude(data1, piste1, data2, piste2, index, interval, window, marge) :
  deltas = [] #Liste de distances
  for i in index :
    max_corr = 0
    max_corr_idx = 0
    idx = i * interval
    for j in range(max(0, idx - marge),min(idx + marge, len(data2)), interval) :
      corr = sum(sp.signal.correlate(data1[idx:idx + window, piste1],data2[j:j + wi
      # print(idx, j)
      if max_corr < corr :
        max_corr = corr
```

```
        max_corr_idx = j
    deltas += [max_corr_idx - idx]

  return deltas
```

```
"""
BuildDeltas permet de construire la liste des deltas.
"""
def BuildDeltas(data1, data2, Freq, ratio, window_size, nb_peaks) :
  delta = []
  for i in range(4) :
    peaks1, index1, interval1, window1 = CalcEnergie(data1, i, Freq, ratio, window_s
    peaks2, index2, interval2, window2 = CalcEnergie(data1, i, Freq, ratio, window_s
    for j in range(4) :
      print(f"{i} -- {j}")
      if i != j :
        delta += [PeaksSimilitude(data1, i, data1, j, index1, interval1, window1, 10
        delta += [PeaksSimilitude(data2, i, data2, j, index2, interval2, window2, 10
      delta += [PeaksSimilitude(data1, i, data2, j, index1, interval1, window1, 1000
      delta += [PeaksSimilitude(data2, i, data1, j, index2, interval2, window2, 1000
  return delta
```

```
data1, data2, Fe = Traitement("/content/erreur_signal/ONECAT_20200114_150201_247.wav

print(data1.shape)
print(data2.shape)
print(Fe)
```

```
Loading files...
Reconstructing data...
(28629600, 5)
(28629600, 5)
286296
```

```
delta = BuildDeltas(data1, data2, Fe, 5, 5, 500)
```

Afficher la sortie masquée

```
import numpy as np

# Define a filename for saving the delta data
output_filename = 'delta_data1.npy'

# Save the delta_np array to the file
np.save(output_filename, delta)

print(f"Delta data saved to {output_filename}")
```

```
Delta data saved to delta_data1.npy
```

```
delta=np.load(output_filename)
print(delta.shape)

```

```
(56, 500)
```

Remarque

Le jeux de donnée est un peu faible en raison du temps de compilation. Cependant je pense qu'il est suffisant pour la suite.

Analyse des Données

A partir des tableaux de distances récupérés nous allons maintenant chercher à reconnaitre et isoler les mammifère marin. Je me suis basée sur les TP réalisé l'année dernière.

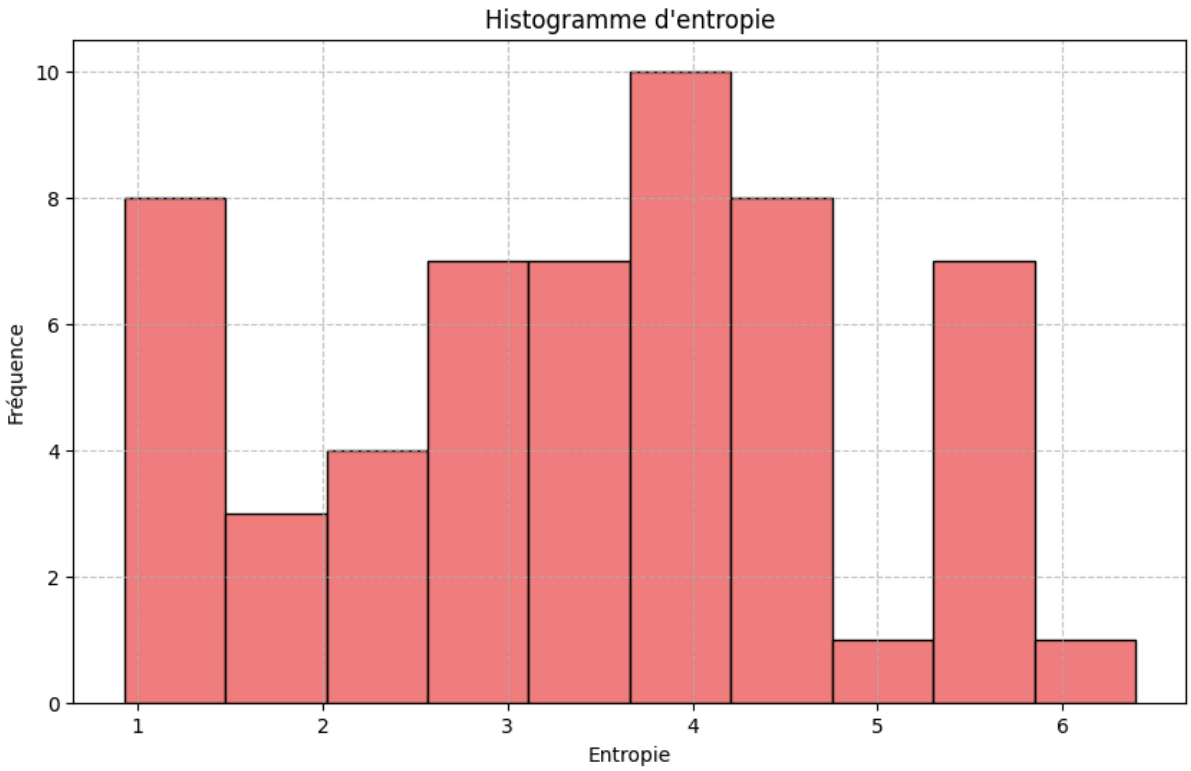
Histogramme d'entropie

Nous allons commencer par l'histogramme d'entropie. Pour voir si l'on peut récupérer de l'information

```
"""
Fonction pour calculer l'entropie
"""
def entropie(p) :
  return - np.sum(p * np.log2(p))


```

```
row_entropies = []
for d in delta :
  hist_counts = np.histogram(d, bins=200, density=False)[0]
  total_elements_in_row = hist_counts.sum()
  probabilities = hist_counts / total_elements_in_row
  row_entropies.append(entropie(probabilities))
plt.figure(figsize=(10, 6))
plt.hist(row_entropies, bins=10, edgecolor='black', color='lightcoral')
plt.title("Histogramme d'entropie")
plt.xlabel("Entropie")
plt.ylabel("Fréquence")
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



Observation

On remarque remarque une vague vers 3.5. Cependant nous travaillerons sur l'ensemble du jeu de donnée en raison de sa faible quantité

Clusters

Nous allons maintenant générer les clusters. Pour se faire, nous allons utiliser TSNE et kmean. Comme nous savons que sur les enrigistrement nous devrions entrendre deux cachalots, nous allons définir 3 clusters. Un cluster par cachalot et un cluster pour le bruit.

```
#TSNE
tsne = TSNE(n_components=2, random_state=0, perplexity=30)
tsne = tsne.set_params()
delta_2d = tsne.fit_transform(delta)

k = 3 #Nombre de clusters
codebook, _ = sp.cluster.vq.kmeans(delta_2d, k)
labels, _ = sp.cluster.vq.vq(delta_2d, codebook)

codebook_np = np.array(codebook)
unique_labels, counts = np.unique(labels, return_counts=True)
# Nombre d'élément par cluster
for label, count in zip(unique_labels, counts):
    print(f"  Cluster {label}: {count} elements")
plt.figure(figsize=(10, 8))
plt.title('Visualisation des clusters')
plt.scatter(delta_2d[:, 0], delta_2d[:, 1], c=labels, cmap='viridis')
plt.scatter(codebook_np[:,0], codebook_np[:,1], marker='x', color='red')
# Legende des cluster
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label=f'Cluster {i}'
```

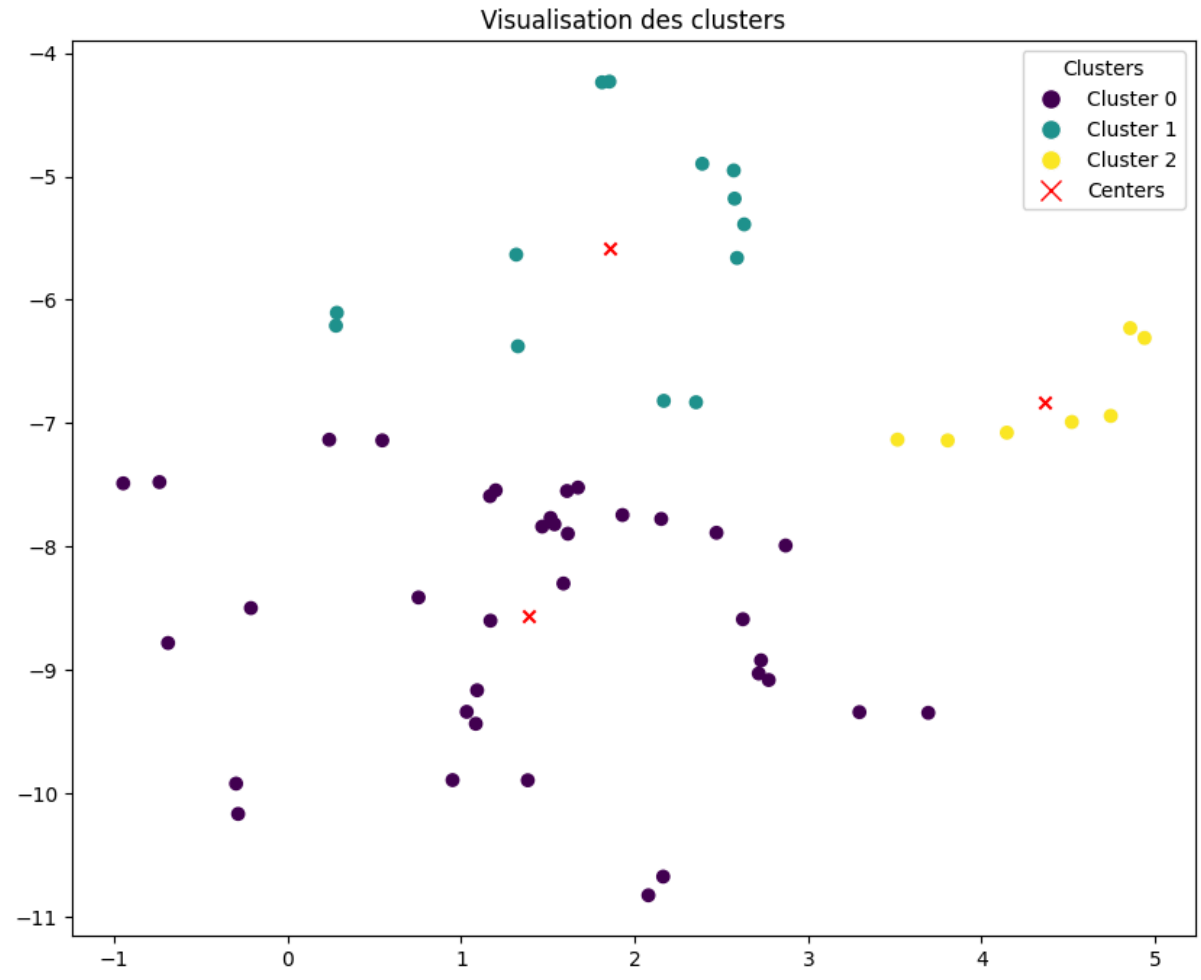
```
                                markerfacecolor=scatter.cmap(scatter.norm(i)), marke
                  for i in range(len(set(labels)))]

# Add an element for the cluster centers to the legend
legend_elements.append(plt.Line2D([0], [0], marker='x', color='red', label='Centers
                                  linestyle='None', markersize=10))

plt.legend(handles=legend_elements, title='Clusters')

```

```
Cluster 0: 36 elements
Cluster 1: 13 elements
Cluster 2: 7 elements
<matplotlib.legend.Legend at 0x7d9d509945c0>
```



Observation

Comme attendu, on observe deux clusters avec peu d'élément que l'on peut considérer comme deux cachalot distincte. Le troisième clusters est probablement que du bruit.

Conclusion

A partir des donnée de deux observateur, nous avons pus isoler la présence de deux cachalot distincte. On peut en conclure que la reconstruction de la donnée ainsi que sont traitement a été assez efficace.