Acoustic

Groupe 17 :
Juliette JACQUOT (EPITA)
Guillaume LALIRE (EPITA)
Vincent LE DUIGOU (Albert School)
Djunice LUMBAN (EPITA)

Avril 2024

Table des matières

1 Introduction

Le projet Acoustic présenté dans ce hackathon repose sur le traitement de signaux sonores afin de localiser une personne dans une pièce, qui est une salle dans notre cas. Lors de ce projet, notre objectif principal est d'explorer l'efficacité de différents modèles d'apprentissage automatique pour effectuer cette tâche. Les modèles choisits sont présentés dans l'article "SOUNDCAM: A Dataset for Finding Humans Using Room Acoustics".

2 Jour 1

- Outils de collaboration et environnement de travail

Tout d'abord, nous avons réfléchi aux outils à utiliser durant ce Hackathon, afin de communiquer entre nous et de se partager les travaux. Nous avons créé un groupe Discord essentiellement pour le partage de liens (sites internets liés au projet, recherches sur différentes méthodes, etc.).

Nous avons également créé un dépôt Github pour travailler ensemble sur le code source du projet. De plus, nous avons choisi de travailler sur des instances de la plate-forme Onyxia pour éviter de stocker trop de données en local, ainsi que pour avoir un accès à un GPU.

- En quoi consiste le dataset fourni ?

Suite au téléchargement du dataset, nous avons pu accéder et étudier les données à utiliser. Celles-ci sont réparties dans trois dossiers :

— 'Empty' : les données pour une pièce vide,

— 'Human1' : les données pour une pièce avec un humain de présent

— 'Human2' : les données pour une pièce avec un différent humain de présent

Le dossier 'Empty' contient seulement un fichier 'deconvolved.npy', qui stocke les signaux sonores mesurés dans cette pièce.

Les dossiers 'Human1' et 'Human2' comprennent trois fichiers distincts :

— 'deconvoled_trim.npy' : contient les RIRs qui représentent les signaux sonores,

— 'centroid.npy' : contient les localisations x et y de la personne dans la pièce,

— 'skeletons.npy' : contient les poses et les emplacements des articulations de la personne capturés par 3 caméras.

Le sujet implique la présence d'une personne dans la pièce, donc nous n'aurons pas besoin d'utiliser les mesures dans la pièce vide. De même, la pose de la personne n'est pas mentionnée dans le sujet. Nous utiliserons donc uniquement les fichiers 'deconvolved_trim.npy' et 'centroid.npy' pour déterminer sa localisation dans la pièce.

- Quels sont les différents modèles présentés dans l'article scientifique ?

Les modèles d'apprentissage automatique présentés sont :

— kNN : k-Nearest Neighbors

— Régression Linéaire

— VGGish (pré-entrainé)
— VGGish (multichannel)

Cet article présente aussi un algorithme nommé “Time of Arrival”, mais celui-ci ne fait pas partie de notre étude car il n’est pas un modèle d’apprentissage automatique.

	Number of Microphones			
Treated Room	10	4	2	1
kNN on Levels	133.5 (51.0)	133.5 (48.4)	135.8 (51.9)	133.3 (49.2)
Linear Regression on Levels	133.7 (46.9)	133.9 (48.8)	133.8 (47.4)	133.7 (47.3)
VGGish (pre-trained)	164.8 (79.0)	148.0 (82.0)	147.5 (79.4)	155.8 (79.4)
VGGish (multichannel)	**31.4** (23.9)	**44.6** (34.0)	**48.6** (39.2)	**106.3** (79.6)
Time of Arrival	232.4 (95.7)	232.2 (96.9)	219.8 (95.9)	232.0 (97.7)
Living Room				
kNN on Levels	167.2 (56.5)	170.0 (59.8)	169.3 (65.7)	168.7 (54.3)
Linear Regression on Levels	125.0 (62.7)	154.5 (54.7)	165.3 (54.2)	168.4 (53.5)
VGGish (pre-trained)	186.6 (89.4)	191.4 (99.9)	199.7 (96.8)	189.9 (100.9)
VGGish (multichannel)	**25.6** (18.4)	**40.3** (38.6)	**43.1** (41.6)	**82.2** (53.8)
Time of Arrival	297.0 (133.0)	298.1 (131.0)	281.5 (127.1)	301.7 (132.6)

FIGURE 1 – Erreur de localisation de chaque modèle en utilisant des RIR (Réponses Impulsionnelles de la Réverbération) basées sur la musique provenant de différents nombres de microphones dans différents environnements.

Durant ce hackathon, on se concentrera sur le kNN, la régression linéaire et le VGGish multichannel vu que ce sont les 3 modèles ayant les meilleurs résultats selon l’article.

- Choix des training et test sets

Au début, on s’est décidé d’utiliser les données de ‘Human1’ comme training set et les données de ‘Human2’ pour la validation set. Mais on a remarqué qu’en répartissant les sets de cette façon, on obtenait des résultats insatisfaisants. On s’est donc décidé de faire un mélange des données des deux dossiers, et d’utiliser 80% de ces données pour l’entrainement. Les 20% restants sont utilisés pour tester nos modèles.

- A quoi sert la RMS et comment la calculer ?

La RMS, aussi appelée “Root Mean Square”, est une méthode permettant de représenter un grand nombre de données en une seule valeur de façon similaire à la moyenne. Elle est calculée à l’aide de la formule suivante :

$$RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

Cette valeur est utilisée pour faciliter le fonctionnement des modèles kNN et de Régression Linéaire, qui ont de moins bons résultats lorsqu’ils ont trop d’informations en entrée.

- Comment implémenter la régression linéaire ?

Notre implémentation de la régression linéaire est faite à l’aide de la librairie *sklearn*. Le modèle de régression linéaire que l’on utilise prend en entrée une valeur de RMS pour

chaque micro, et renvoie des coordonées.

Ce modèle n'a pas de paramètre à configurer, et donne des résultats avec une erreur moyenne de 149.5cm par rapport aux positions attendues.

- Comment implémenter le modèle kNN ?

Le modèle des k-Nearest Neighbors est aussi implémenté à l'aide de la librairie *sklearn*, et a les mêmes entrées et sorties que le modèle de régression linéaire.

Ce modèle a un paramètre à préciser : le nombre de voisins à prendre en compte pour faire une prédiction de coordonnées. Ce choix de paramètre à été fait suite à des tests sur différentes valeurs, allant de 2 à 9.

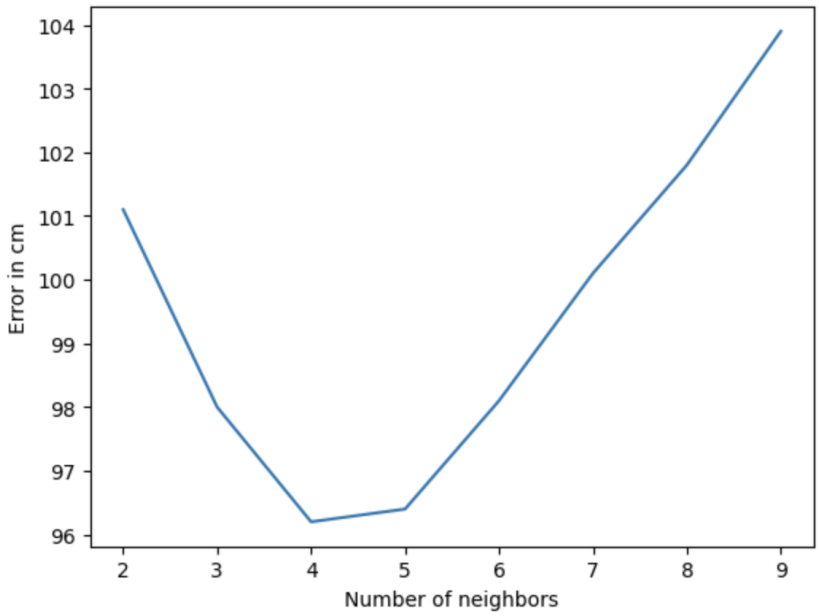

FIGURE 2 – Erreur de localisation en fonction du nombre de voisins utilisés

Dans notre cas, l'erreur est minimale lorsque 4 voisins sont utilisés. Ce nombre est donc le paramètre que l'on a précisé dans notre version finale de ce modèle, qui a une erreur moyenne de 96.2cm.

3 Jour 2

- En quoi consiste le modèle VGGish ?

Le modèle VGGish est un réseau de neurones convolutionnel conçu pour traiter des sons. Cela fait contraste avec les CNN traditionnels, qui sont surtout utilisés pour le traitement d'image.

Ce modèle prend donc des signaux sonores en entrée, et utilise ses couches de neurones pour extraire des caractéristiques importantes de ces sons. Ces caractéristiques sont ensuite utilisées pour l'analyse des sons et la prédiction de la localisation de la personne dans la pièce.

- Comment faut-il formater les données pour les entrer dans un CNN ?

Pour formater les données d'entrée pour un réseau de neurones convolutionnel, il faut convertir les signaux audio en une image. Dans notre cas, nous avons choisi de transformer les signaux audios en un spectrogramme.

En effet, un spectrogramme est une représentation temps-fréquence du signal obtenue en calculant la transformée de Fourier à court terme (STFT), et peut être représenté sous forme d'image.

On utilise donc les spectrogrammes comme représentation d'entrée pour les réseaux de neurones convolutionnels dédiés au traitement audio car ils fournissent une vue temps-fréquence compacte.

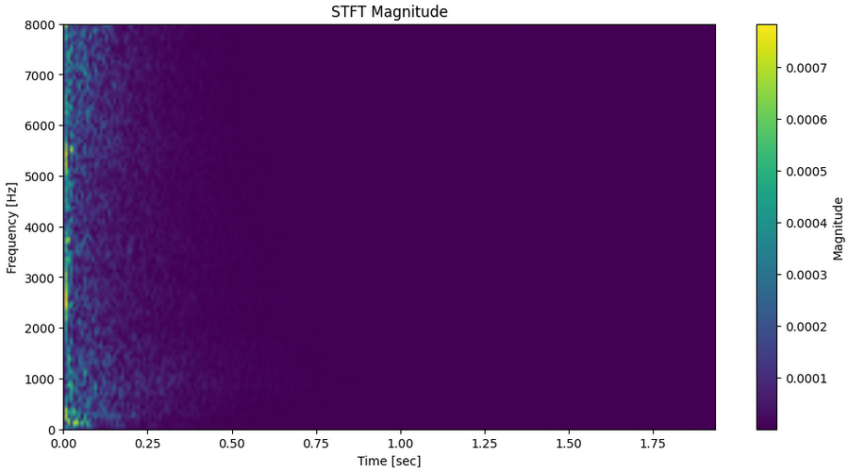

FIGURE 3 – Spectrogramme représentant la magnitude de la transformée de Fourier à court terme (STFT) d'un signal audio en fonction du temps et de la fréquence

- Quels sont les défis rencontrés durant l'implémentation de VGGish ?

On a d'abord rencontré des problèmes au niveau des GPU. Au début, il y en avait seulement 27, on était dans le file d'attente mais en vain. Finalement, le problème s'est résolu avec une augmentation de GPU mis à disposition.

La première approche pour l'implémentation de VGGish a été d'essayer d'expérimenter avec le VGGish implémenté dans l'article scientifique, disponible sur GitHub. Leur implémentation utilise Pytorch. Cependant, nous avons rencontré beaucoup de difficultés dans l'utilisation de cette implémentation avec notre jeu de données :

— Problème avec l'import de "torchvggish.vggish" : L'importation du module "torchvggish.vggish" échouait, car "vggish" n'est pas un sous-module du module "torchvggish" dans le package installé avec pip. Nous avons essayé deux versions différentes du package "torchvggish" (0.1 et 0.2) mais sans succès. Finalement, nous avons réussi en utilisant le code source du dépôt "torchvggish" sur GitHub.

— Problème avec l'utilisation du dataset : Le modèle VGGish utilisé dans l'implémentation de l'article supposait que l'ensemble de données contenait des données provenant de 10 micros, alors que notre ensemble de données n'en comporte que 4, ce qui entraînait des erreurs dans le code. Nous avons résolu ce problème en modifiant manuellement les indices dans le fichier de l'implémentation de l'article.

— Problème avec une fonction du module "vggish" : Nous avons ensuite rencontré une erreur de type indiquant qu'une fonction "make_layers()" était appelée dans le fichier de l'implémentation de l'article avec un paramètre inattendu 'in_channels', ce qui implique une version différente du module utilisé dans l'implémentation de l'article. L'implémentation du dépôt GitHub étant la seule à avoir fonctionné, nous n'avions pas d'autres versions fonctionnelles à tester.

- Retour sur RMS : limiter à 2s

On a remarqué que l'article n'utilisait que les données des 2 premières secondes pour son calcul des RMS. On a donc décidé d'étudier l'impact de cette décision sur les modèles kNN et de régression linéaire.

Dans le cas de la régression linéaire, ce changement permet de passer d'une erreur moyenne de 149.5cm à une de 149.4cm, ce qui est une amélioration de 0.1cm. Le modèle kNN lui passe d'une erreur de 96.2cm à une de 95.9cm, et a donc une amélioration de 0.3cm.

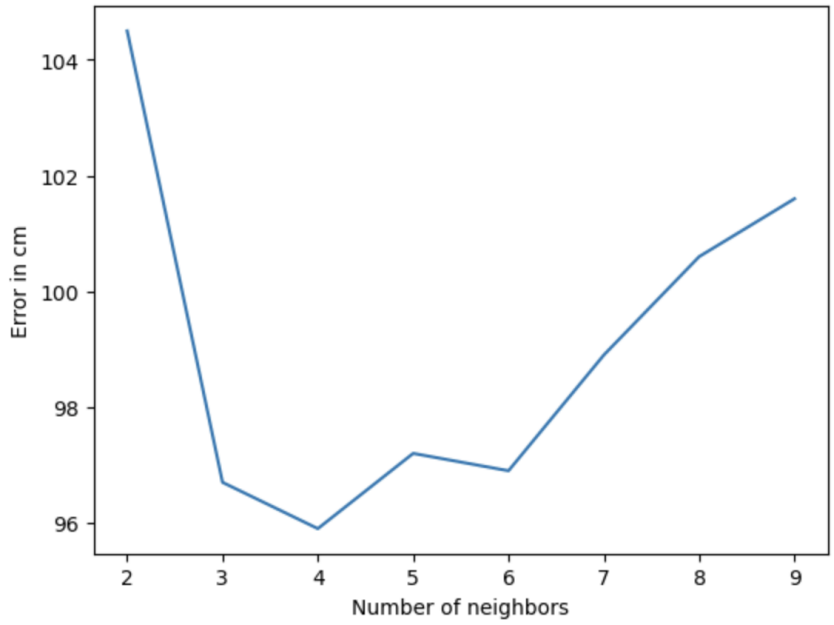

FIGURE 4 – Erreur de localisation en fonction du nombre de voisins utilisés avec 2 secondes de données

Une observation intéressante à faire se trouve dans l'amélioration du modèle kNN en fonction du nombre de voisins. En effet, l'erreur moyenne en utilisant deux voisins seulement est la plus grande, mais celle-ci se réduit grandement dès l'ajout d'un voisin supplémentaire.

On peut aussi remarquer que les erreurs se stabilisent plus longtemps aux alentours de l'erreur minimale que lorsque l'on prend toutes les données disponibles, avec des erreurs assez proches entre 3 et 6 voisins.

4 Jour 3

- Retour sur VGGish : expérimentation avec un modèle pré-entrainé

Suite à notre échec de l'utilisation du VGGish utilisé dans le modèle, nous avons recherché des implémentations de VGGish différentes qui pourraient fonctionner. Durant ces recherches, nous avons trouvé un modèle VGGish pré-entrainé par Google disponible sur Kaggle.

Il a ensuite fallu préparé le format des signaux. Nous avons donc fusionné les signaux des 4 micros en un seul signal, puisqu'il ne s'agit pas d'une implémentation à canaux multiples.

Le modèle pré-entrainé ne prend en compte que 16 000 échantillons. Nous avons donc réduit la fréquence de 48 à 16 Hz, puis nous avons sélectionné la première seconde seulement, ce qui représente 16 000 échantillons. Le fait de se limiter au début du signal était aussi une contrainte évoquée dans l'article, où ils se limitaient à 1.93s. Nous avons ensuite pu convertir ces signaux en des vecteurs de taille 128, en les soumettant au modèle.

Ensuite, il a fallu créer un second réseau de neurones pour passer de 128 à 2 valeurs (x et y). Pour cela, nous nous sommes inspiré du réseau de neurones présent dans l'implémentation VGGish de l'article qui compte 2 couches denses de taille 256, 2 dropouts et une couche dense de sortie de taille 2.

Nous avons pu générer des graphes afin d'observer l'évolution de l'erreur et de la précision.

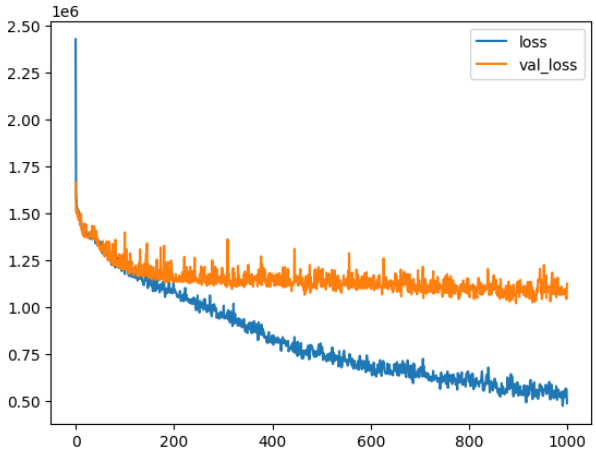

FIGURE 5 – Evolution de l'erreur lors de l'entrainement du réseau de neurones avec 1000 epochs et un taux d'apprentissage de 10^{-2}

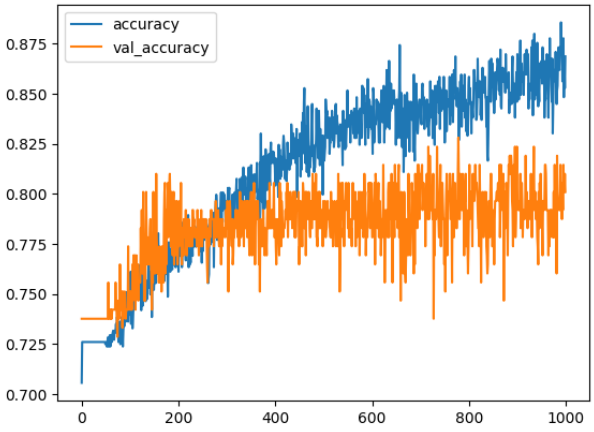

FIGURE 6 – Evolution de la précision lors de l'entrainement du réseau de neurones avec 1000 epochs de 10^{-2}

Nous avons tout d'abord repris les hyperparamètres présentés dans l'article mais nous avons également testé différentes valeurs afin de déterminer si d'autres valeurs pouvaient donner de meilleurs résultats.

— Le batch size de 32 est le même que celui utilisé dans l'article.

— Le nombre d'epochs de 1000 a été également reprise de l'article, mais il nous semble pertinent aux vus des graphes, car l'erreur sur les données de validation continue de décroître légèrement et n'augmente pas (ce qui serait le signe d'un overfitting).

— Le taux d'apprentissage de 10^{-2} a été sélectionné en comparant l'exécution avec des valeurs différentes, regroupés dans le tableau suivant :

Taux d'apprentissage	10^{-4}	10^{-3}	10^{-2}
Distance moyenne (cm)	152.1	137.0	128.8

Ainsi, en utilisant ce modèle pré-entrainé de VGGish avec les meilleurs hyperparamètres que nous avons trouvé, on obtient une distance moyenne de 128.8 cm.

- Le nombre de micros utilisés influence-t-il les résultats ?

Nous avons également commencé une comparaison des distances moyennes obtnues en sélectionnant une partie des micros seulement, en utilisant le même modèle VGGish. Voici les échantillons que nous avons récoltés :

Micro 1	Micro 2	Micro 3	Micro 4	Distance moyenne (cm)
X	X	X	X	128.8
X	X	X		131.8
	X	X	X	143.1
X	X			130.2
	X	X		118.7
		X	X	133.8
X				120.5
	X			134.3
		X		132.9
			X	137.2

Nos données récoltées peuvent être résumées en faisant la moyenne des échantillons avec un même nombre de micros :

Nombre de micros	1	2	3	4
Distance moyenne (cm)	131.2	127.6	137.5	128.8

Les résultats obtenus semblent cependant incohérents, puisque l'utilisation d'un seul micro donne des résultats plus satisfaisants en moyenne mieux que l'utilisation des 4 micros. Cela s'oppose également aux résultats obtenus dans l'article, qui montraient que plus on utilise de micros meilleurs sont les résultats.

Par ailleurs, ces données sont loin d'être complètes puisqu'il y a énormément de sous-ensembles de micros possibles et nous n'avons pas eu le temps de tous les récolter, ce qui peut influencer la qualité des résultats.

- Serait-il possible de ne pas fusionner les canaux pour utiliser ce VGGish pré-entrainé ?

Une piste que nous avons exploré est d'envoyer le contenu de tous les canaux un à un au modèle pré-entrainé. Les groupes de vecteurs sont ensuite soumis au deuxième modèle. Celui-ci est similaire au modèle utilisé pour la gestion d'un canal simple, avec l'ajout d'un applatissement des données.

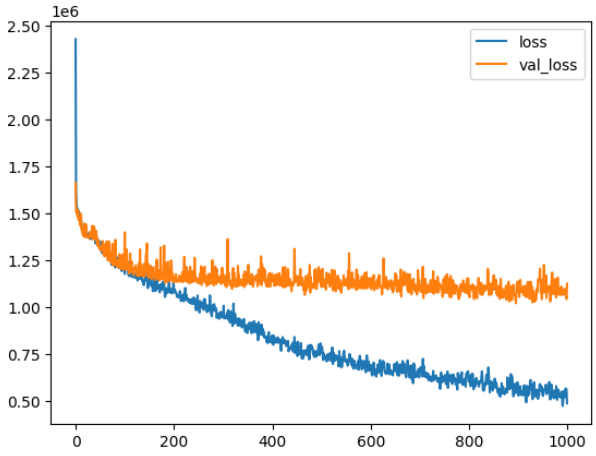

FIGURE 7 – Evolution de l'erreur lors de l'entrainement du réseau de neurones avec 1000 epochs et un taux d'apprentissage de 10^{-3}

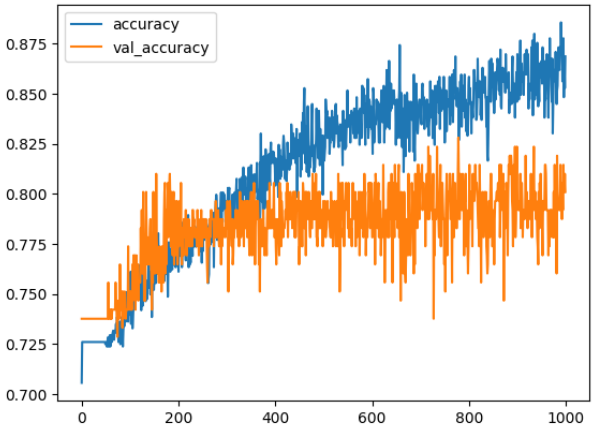

FIGURE 8 – Evolution de la précision lors de l'entrainement du réseau de neurones avec 1000 epochs de 10^{-3}

Les hyperparamètres ont étés choisit d'une manière similaire que pour l'apprentissage avec un seul canal.

— Le batch size est de 32, comme dans l'article et le VGGish précédent

— Le nombre d'epochs est aussi de 1000

— Le taux d'apprentissage sélectionné suite à une comparaison des exécutions avec des valeurs différentes ci-dessous :

Taux d'apprentissage	10^{-4}	10^{-3}	10^{-2}
Distance moyenne (cm)	118.6	105.4	112.4

A l'aide de ce modèle, on obtient donc une distance moyenne de 105.4cm.

Ce résultat est très grand comparé à ceux que l'on attendait suite à la lecture de l'article. En effet, la prise en compte et fusion des différents canaux effectuée dans l'article était présente dès l'entrée dans le modèle VGGish. Notre implémentation actuelle effectue cette fusion après le passage dans VGGish, et a donc des résultats différents.

Une façon de retrouver des résultats similaires à ceux attendus serait donc de recréer notre propre implémentation de VGGish, ainsi que tout le prétraitement des signaux nécessaire.

5 Récapitulatif des résultats

Modèle	Notre distance moyenne (cm)	Distance moyenne dans l'article (cm)
Régression linéaire	149.5	154.5
kNN	96.2	93.2
VGGish (pré-entrainé)	128.8	147.2
VGGish (multi-canaux)	103.3	23.6

6 Ressources

GitHub du groupe 17
Présentation pitch1
Présentation pitch2