Fiche recette du projet d'intégration de NiSQA

Auteurs :
Séverin CHEVALIER
Liam COURSODON
Jules DECAESTECKER
Zaccarie KANIT
Clémence MARINIER
Léo STENGEL
Elias TRANCHANT

Décembre 2022

1 Présentation

Ce projet a été réalisé par 7 étudiants d'IMT Atlantique Brest avec Vincent BARRIAC, Yvain JORIGNY et Nicolas PENNANEACH de Orange (Innovations/Networks). NiSQA est un algorithme évaluant la qualité d'une communication audio développé par des chercheurs de l'Université Technique de Berlin. Ce projet visait à remplacer l'algorithme PESQ anciennement utilisé. Pour cela, nous devions créer un logiciel debian avec un serveur qui nous donne le résultat de la qualité quand on transmettait un fichier audio, créer une image docker avec la même focntionnalité et créer une interface en ligne de commande de cet algorithme NiSQA.
Cette fiche récapitule les travaux effectués, les résultats obtenus de certaines mesures et des conseils pour l'amélioration de nos travaux.

2 Livrables rendus

2.1 Logiciel Debian

Le logiciel Debian consiste en la création d'un serveur similaire à celui de la solution précédente (PESQ), sur lequel un utilisateur est créé et peut utiliser l'algorithme NISQA pour mesurer la qualité audio d'un fichier .wav .
Nous avons rendu au sein d'un gitlab un ensemble de fichiers dont un dossier à empaqueter ainsi que des fichiers d'installation pour les dépendances. Un élément important est la readme qui donne toutes les informations essentielles à l'utilisateur de l'installation à l'utilisation.
De plus, il est possible d'effectuer de l'intégration continue sur gitlab ce qui facilitera la continuité du projet par d'autres équipes.
Enfin, un userguide est joint pour expliquer en détail le fonctionnement du serveur et assurer la modularité du projet.

2.2 Docker

Le Docker simplifie le déploiement du service. En effet, le docker se lance sur n'importe quelle machine et ne nécessite pas d'installation. Le docker est autonome dans son fonctionnement.
Nous avons fourni des fichiers qui permettent une intégration continue du projet pour de futurs travaux. Comme pour le logiciel debian, un userguide est joint au projet gitlab.

2.3 Ligne de Commande

La ligne de commande simplifie beaucoup plus l'architecture puisqu'elle n'implique pas la création d'un serveur. L'ensemble du travail s'effectue sur le compte d'un utilisateur NISQA qui s'utilise sans mot de passe pour éviter les restrictions et être employable par des scripts d'autres utilisateurs. Ainsi, l'exécution peut se faire sans être root.
Ce service est aussi fourni sur un gitlab avec des explications d'installation et d'utilisation et surtout du développement continu.
Un userguide est aussi joint à ce projet.

3 Résultats obtenus

Dans un premier temps, nous avons testé l'algorithme NISQA sur un ensemble de fichiers audio provenant de l'ITU. On peut les retrouver sur `https://www.itu.int/rec/T-REC-P.862-200102-I/fr` dans le répertoire **T-REC-P.862-200102-I!!SOFT-ZST-S/P862/Software/Conform/** .
Ces fichiers sont au format .wav et échantillonés à 8kHz. Cela peut expliquer en partie les différences de valeur entre la note MOS attribuée par l'alogrithme PESQ et celle renvoyée par NISQA. En effet NISQA est conçu pour fonctionner avec des fréquences d'échantillonnage du signal plus élevées que PESQ. Un autre facteur important est leur mode de fonctionnement. PESQ calcule cette note MOS en comparant le fichier dégradé au fichier de référence, alors que NISQA ne prend en compte que le fichier dégradé. L'échelle des notes MOS n'est donc pas la même.

Reference file	Degraded file	PESQ MOS	Reference NISQA MOS	Degraded NISQA MOS	Difference NISQA MOS Reference-Degraded	Difference MOS PESQ-NiSQA
or105.wav	dg105.wav	1.84394	3.450691	1.495162	1.95553	0.34878
or109.wav	dg109.wav	3.09075	3.417776	1.828903	1.58887	1.26185
or114.wav	dg114.wav	1.75777	3.495709	1.357655	2.13805	0.40011
or129.wav	dg129.wav	2.36606	3.545906	1.521932	2.02397	0.84413
or134.wav	dg134.wav	1.97845	3.593792	1.44308	2.15071	0.53537
or137.wav	dg137.wav	3.77944	3.547771	2.858148	0.68962	0.92129
or145.wav	dg145.wav	2.84601	3.483449	2.534688	0.94876	0.31132
or149.wav	dg149.wav	2.20702	2.941261	1.341206	1.60006	0.86581
or152.wav	dg152.wav	2.48674	3.127168	2.060962	1.06621	0.42578
or154.wav	dg154.wav	2.38495	3.616052	1.189305	2.42675	1.19564
or155.wav	dg155.wav	2.26835	3.030383	1.417973	1.61241	0.85038
or161.wav	dg161.wav	2.27094	2.861657	1.505695	1.35596	0.76524
or164.wav	dg164.wav	2.60293	3.113923	1.505193	1.60873	1.09774
or166.wav	dg166.wav	2.16832	3.516995	1.407979	2.10902	0.76034

Dans un second temps nous avons testé avec NISQA plusieurs fichiers audios dégradés de façons différentes. Ceux-ci sont au format .wav et sont échantillonnés à 48kHz. On voit que les notes MOS varient généralement entre 1 et 5, comme prévu. Cependant, certaines notes dépassent légèrement de l'intervalle. On peut peut-être expliquer cela par le fait que la fonction d'activation du modèle NISQA doit certainement retourner des notes légèrement hors de l'intervalle [1,5] dans les cas limites.

audio	MOS	Noisiness	Coloration	Discontinuity	Loudness	Description
F1S1.C01	5.150409	4.817146	4.87177	4.702571	4.831268	référence haute, spectre complet sans codage
F1S2.C02	2.764512	2.513579	4.70238	4.164081	3.836166	ajout de bruit de fond
F1S2.C08	1.672902	4.20311	1.292345	3.776736	4.116467	20% de paquets IP perdus
F1S2.C26	4.649351	4.513099	4.741065	4.046252	4.496233	codage de la voix HD en 3G
F1S3.C15	3.80061	4.142939	4.202243	2.986036	3.579714	codage du GSM
F1S4.C04	3.486153	4.722647	4.715476	4.24715	1.821873	signal atténué de 20 dB
F1S5.C05	3.193204	4.001684	4.506642	1.973342	3.656659	spectre ramené à 500-2500 Hz

4 Suite du projet

4.1 Algorithme Nisqa

Le modèle d'évaluation de la qualité audio que nous utilisons est nommé NiSQA pour Non-intrusive Speech Quality Assessment. C'est un réseau de neurones convolutif basé donc sur des méthodes de deep-learning codé en python. Il a été entraîné sur différents ensembles de données et est utilisable tel quel, notre projet ne nécessite donc pas que nous optimisions le modèle. Pour l'instant, l'algorithme fonctionne sans fichier de référence. Cela signifie que le fichier audio dégradé après transmission n'est pas comparé avec celui enregistré avant la transmission. Les sources sont hébergées sur le dépôt Github de G. Mittag :

https://github.com/gabrielmittag/NISQA. Nous avons réutilisé tel quel le répertoire Github et la modification de NISQA ne rentre pas dans le cadre de ce projet.

4.2 Pistes d'amélioration

4.2.1 Amélioration de la CI

Nous avons suivi des auto formations au sein du groupe pour effectuer le projet. Certaines pratiques ne sont donc peut être pas les plus optimales. De plus nous n'avons pas d'idée sur la façon dont se font les choses au sein de chez Orange. Nous avons donc vu des pistes d'amélioration pour le développement du service.

- Sécuriser la conteneurisation du Docker : nous avons effectué un Docker in Docker pour permettre le développement continu sans maîtriser tous les enjeux de sécurité que cela implique. Il pourrait être judicieux de revoir cette partie pour créer un utilisateur et revoir les tests réalisés.
- Homogénéiser notre travail avec celui de Orange : nous avons proposé un service sous forme de CLI et des fichiers d'intégration continu. Pour permettre une meilleure lisibilité par un employé de Orange, il pourrait être utile de les réécrire de façon similaire à ce qui est fait au sein de Orange car nous n'avons pas suivi de standard particulier.

4.2.2 Réduction de la taille des fichiers

L'algorithme d'estimation des notes se base sur un modèle de Machine Learning qui utilise la librairie Pytorch en Python. Malheureusement cette librairie est extrêmement lourde et a donc grandement augmenté la taille des livrables. Nous ne possedons pas les compétences nécessaires pour résoudre ce problème en n'utilisant que l'architecture et les poids du modèle afin d'effectuer l'inférence. Voici quelques propositions pour un futur projet.

- Garder une image Python sur Docker : il existe une librairie Onnx intégrée à Pytorch qui permet l'exportation du modèle pour seulement faire l'inférence. Malheureusement il manque quelques fonctions à cette librairie qu'il faudrait réécrire et nous n'avons pas pu nous en occuper.
- Refonte de l'architecture sur Tensorflow lite : il faudrait cette fois ci utiliser une image Tensorflow lite sur Docker et refaire l'architecture dans ce langage. Cela permettrait d'éviter d'avoir des librairies qui ne servent pas au projet en plus d'une bonne rapidité et légèreté.
- Refonte de l'architecture en C++ : il est aussi possible d'utiliser du C++ dans notre Docker, ici pour descendre en abstraction et gagner en légèreté et rapidité. En effet, il existe la librairie Darknet en C++ qui permet de faire du Machine Learning. Dans ce cadre là, il est possible encore une fois de traduire l'architecture et d'utiliser les poids fournis pour faire le même service.