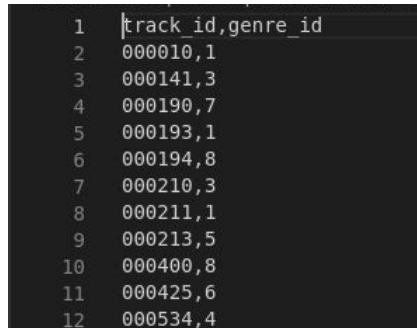


# Rendu Final Projet TSMA

Piet Thomas  
Heurtel Thomas

## 1 Solution Aléatoire

Pour ce premier rendu, nous avons commencé par associer des genres aléatoires. Pour ça, nous avons créé un fichier `random_genre.py`, qui copie les lignes du fichier `test.csv` (sauf le header) et qui rajoute à chaque ligne un chiffre random entre 1 et 8, correspondant à un genre. On copie ensuite cette liste dans un nouveau fichier `random.csv`, après avoir ajouté le header. On obtient donc dans le fichier `random.csv`, le format attendu, la première colonne étant les `track_id` et la seconde les `genre_id` générés aléatoirement.



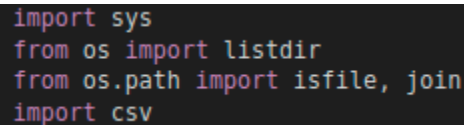
```
1 track_id,genre_id
2 000010,1
3 000141,3
4 000190,7
5 000193,1
6 000194,8
7 000210,3
8 000211,1
9 000213,5
10 000400,8
11 000425,6
12 000534,4
```

Figure 1: Début du fichier `random.csv`

Nous avons obtenu un score de 0.11876 avec cette solution.

## 2 Classification à partir de MFCCs

Dans `main.py`, on utilise l'algorithme NN. On aura besoin des bibliothèques suivantes :



```
import sys
from os import listdir
from os.path import isfile, join
import csv
```

Pour exécuter le programme, il faut utiliser la syntaxe suivante : `"python3 main.py <test folder> <train folder> <test csv> <train csv> <out csv>"`

On a écrit plusieurs fonctions :

- `computeScore()` permet de récupérer la différence entre deux mfcc (plus elle est petite, plus c'est intéressant)

- `mfcc_read()` retourne les valeurs d'un fichier mfcc sous la forme d'une liste de flottants
- `eligible_files()` parcourt le répertoire d'entrée et retient tous ses fichiers mfcc avant de les retourner sous forme de liste de strings
- `mfcc_files()` parcourt cette liste et retourne le contenu de chaque fichier mfcc
- `nearest()` prend en entrée un fichier et un répertoire, et retourne le fichier du répertoire le plus proche de celui d'entrée
- `id_from_filename()` retourne l'id d'un fichier selon son nom et son chemin d'accès (fonctionne dans le cas où le nom du fichier est de la forme "chemin/dacces/id.extension")

Puis grâce à la bibliothèque csv, on parcourt les lignes du fichier csv d'entraînement et on stocke chaque id (élément 0 de la ligne) et le genre correspondant (élément 1) dans un dictionnaire afin d'y accéder facilement par la suite.

On ouvre ensuite le fichier de sortie en écriture et on écrit la ligne d'en-tête. On ouvre le fichier csv de test, contenant tous les id nécessaires, et pour chaque id on calcule son plus proche voisin dans le répertoire train grâce à notre fonction `nearest()`. Important : le nom du fichier doit être "id.wav.mfcc" - on peut cependant changer l'extension en modifiant la variable extension définie plus haut.

Puis on écrit une nouvelle ligne dans le fichier de sortie, avec notre id courant en 0 et en 1 le genre de son plus proche voisin, que l'on pioche dans notre dictionnaire.

Pour utiliser le programme, il faut avoir déjà nos mfcc dans un répertoire. On peut utiliser le code fourni dans `main.c` pour cela, sachant qu'il ne fonctionne qu'avec des fichiers wav (le fichier `mp3_to_wav.py` permet de convertir si besoin). Quelques-uns des fichiers mp3 fournis étant corrompus (moins d'une dizaine), le programme C n'a pas pu créer les mfcc correspondants. Dans ces cas là nous avons créé nous même les fichiers et copié dedans les valeurs d'un mfcc au hasard. Les valeurs ne sont donc pas bonnes sur ces quelques mfcc mais cela permet au programme de tourner.

Pour le moment nous avons obtenu le score de 0.27744.

### 3 Classification à l'aide de catboost

[Lien colab](#)

On commence par récupérer les features grâce à `librosa`. Puis on crée les datasets correspondant aux données d'entraînement et de test avant de définir notre modèle. Nous avons d'abord essayé l'algorithme de classification SVM grâce à `scikit learn`, mais les résultats étaient mauvais (0.16217 sur Kaggle). Nous sommes alors passés à Catboost. Ensuite, on fait un `model.fit()` pour entraîner le modèle. Puis on fait un `predict` sur notre `xtest`, et on convertit le résultat en dataframe. Plus tôt, nous avons déjà créé un dataframe contenant les id des morceaux de test, il nous suffit d'y ajouter le nouveau pour obtenir un dataframe à deux colonnes, une pour les id et l'autre pour les genres correspondant. Puis nous exportons simplement le dataframe obtenu en fichier csv grâce à la fonction `to_csv()`. Dans un premier temps nous avons obtenu un score de 0.49900 avec comme paramètres 100 itérations, 0.003 de learning rate, 10 de profondeur et la loss function `MultiClass`.

Nous avons ensuite testé différentes valeurs pour les paramètres d'itérations, de profondeur et de learning rate, ainsi que d'autres losses, dont des customs. Au final notre meilleur score est : 0.61926 avec comme paramètre :

```
model = CatBoostClassifier(iterations=900,
                           learning_rate=0.2,
                           depth=5,
                           loss_function='MultiClass')
```

## 4 Classification à l'aide de melspectros

*Lien colab*

On commence par charger les melspectrogrammes dans `x_train`, `y_train`, `x_test` et `track_id` (qui contient les id des fichiers de test) grâce à `pickle`. On normalise et reshape `x_train` et `x_test`. Après avoir créé notre modèle (Sequential) on ajoute des couches de convolution et de max pooling, puis un flatten et des dense. On compile le modèle et on l'entraîne grâce à `x_train` et `y_train`. On effectue ensuite un predict sur `x_test` dont on stocke les résultats dans une liste `predictions`. Puis pour chaque track de `predictions` on regarde quelle valeur parmi les 8 est la plus haute (chaque valeur correspondant à la probabilité que le morceau soit du genre correspondant). On récupère ensuite l'index de cette valeur (entre 1 et 8), qui correspond au genre le plus probable, et on l'ajoute à la liste des genres `l`. Puis on ouvre le fichier `test.csv` et on en fait un dataframe comme précédemment. On a eu un problème, il n'y avait que 4002 melspectrogrammes alors que nous devons évaluer 4008 morceaux. Nous allons donc attribuer le genre 1 aux 6 morceaux dont nous n'avons pas les données. On crée une liste de résultats contenant autant d'éléments qu'il y a d'ids de morceaux de test. Puis on parcourt le dataframe des ids de test grâce à deux compteurs : un global et un que l'on incrémentera uniquement lorsque les id correspondront. Si l'id est égal à celui d'un des genres de la liste `l` alors on ajoute ce genre à l'index courant dans la liste résultats. Sinon on lui attribue arbitrairement la valeur 1. On crée ensuite un dataframe contenant les résultats et on l'exporte en csv.

Avec comme fonction `loss categorical_crossentropy`, l'optimizer Adam de keras et comme metrics `accuracy`, 7 epochs et trois couches de `convolution2D` de 64 à 16 (noyau 3,3), on obtient 0.27495 et c'est le meilleur que nous ayons pu obtenir. Nos résultats variaient toujours entre 0.2 et 0.27.

## 5 Classification à l'aide de VGG

Sources dans l'archive.

Dans `VGGish_process.ipynb` : On parcourt les fichiers des répertoires de test et de train et on leur applique `embedding_from_fn()`. Pour éviter d'avoir à les recalculer à chaque fois (quelques secondes par fichier sur plus de 8000 fichiers) on enregistre les datasets avec `pickle`.

Dans `rendu4.ipynb` : Nous n'avons pas réussi à faire fonctionner le programme, les données étant organisées différemment des pickles précédents. Nous avons cependant mis en commentaire notre idée du programme.