TSMA 2018
Music genre classification
Determine musical genre of audio tracks

17 décembre 2018

Teiki Pepin
Rapport
Master 2 Informatique
Université de Bordeaux

Table des matières

1 Introduction

Le but de ce projet est de mettre en place un système de classification de musiques par genres. On dispose d'une base de données de 8008 musiques au format mp3. Les 4000 premières sont labélisées par genres et servent à former un modèle d'entraînement tandis que les 4008 autres musiques servent à l'évaluation de notre modèle.

Les évaluation se font par soumission de manière automatique en ligne sur le site web Kaggle en fournissant un fichier au format csv des genres obtenus pour chaque musique de l'ensemble de test. Le plateforme web fournissant aussi un classement des participants, on peut ainsi observer sa performance par rapport aux autres étudiants dans le cadre de cet exercice.

Le premier objectif effectué au cours du projet est l'établissement d'un procédé permettant de traiter la lecture de nos fichiers musicaux, l'extraction de leur données importantes, la classification à partir de ces données et l'évaluation de notre classification. Une fois ceci atteint, le second objectif est d'améliorer le modèle de classification employé.

2 Pipeline de traitement et de classification des données

2.1 Architecture

Notre projet étant réparti sur de nombreux fichier, il est nécessaire de décrire le rôle de chaque fichier afin d'aider à la compréhension. On notera en premier le fait que diverses bibliothèques Python sont utilisées par ce projet et que celles-ci sont toutes incluse dans le code fourni. Afin de se servir de ces dernières et de la version de Python adaptée, il faut activer l'environnement virtuel qui a été configuré pour le projet via une commande (*source ./bin/activate*).

Notre répertoire *data* contient :

- 2 fichiers .txt représentant les IDs des fichiers corrompus de train et test
- 2 fichiers .pickle représentant les MFCC extraites de train et test
- 2 fichiers .pickle représentant un ensemble partiel de features extrait de train et test
- 2 fichiers .pickle représentant un ensemble global de features extrait de train et test
- 1 fichier .csv représentant les IDs et le genre des fichiers de train
- 1 fichier .csv représentant les IDs des fichiers de test
- 1 fichier .csv représentant les IDs et le genre des résultats sur test obtenus par le programme

Notre répertoire *src* contient :

- classify.py : chaîne de traitement des données allant de la lecture des pickles à l'exportation des résultats obtenus par classification

— input_output.py : fonctions de lecture et d'écriture de fichiers utilisées par classify.py
— models.py : modèles de classification utilisées par classify.py
— pickle_test_check.py : vérification des données dans le pickle de test
— pickle_train_check.py : vérification des données dans le pickle de train
— test_features.py : extraction des features de test
— train_features.py : extraction des features de train

2.2 Extraction des données

On utilise Librosa pour la lecture des fichiers audio. Les ensembles train et test sont traités séparément. On passe à Librosa le répertoire où sont situées nos fichiers avant de les lire un par un et d'en extraire les features qui nous intéresse. En début de projet, on commence par n'extraire que les Mel-frequency cepstral coefficients de nos fichiers. Ceci va évoluer par la suite d'abord en prenant une combinaison de features de chaque fichier puis la totalité des features proposés par Librosa à l'exception de *poly_features*. Pour ces features, on conserve la moyenne et la variance de leur valeurs prises au long de la musique. Pour chaque fichier ainsi traité, on ajoute dans un fichier pickle l'identifiant du fichier audio et ses valeurs extraites.

Il existe des fichiers audio invalides dans les données fournis : 3 dans train et 2 dans test. Ces fichiers ne peuvent pas être lus par Librosa et provoqueront un arrêt du programme lorsqu'on essaye de les traiter. Pour éviter cela, on vérifie la taille en octets de chaque fichier avant sa lecture. Si un fichier à une taille inférieure à un seuil prédéfini (50ko par défaut), on ajoute l'id du fichier corrompu à un fichier .txt avant de passer au fichier suivant sans traiter le fichier corrompu. Le fichier corrompu est ainsi absent des pickles de features et ne sera pas utilisé lors de la classification. Cependant, on n'oublie pas de remettre les id des fichiers corrompus de l'ensemble de test, avec un genre déterminé aléatoirement, lorsque l'on exporte le tout en .csv pour une évaluation en ligne.

2.3 Chaine de traitement

Dans notre algorithme de classification, on commence initialement par charger les données précédemment extraites : on lit les identifiants des fichiers corrompus de nos fichier .txt, on lit les features de nos fichiers .pickle et on lit le .csv de train pour avoir la vérité terrain des genres. Ce dernier fichier contenant toujours les informations des fichiers corrompus, on les supprime afin que nos informations correspondent à ce que nous avons stocké dans les pickle.

On divise ensuite notre ensemble train en deux parties : un ensemble d'entraînement et en ensemble de validation. Cet ensemble de validation nous servira à vérifier l'efficacité de nos modèles sans avoir à ce que l'on fasse une évaluation de l'ensemble de test sur Kaggle à chaque changement. On défini ensuite le modèle de classification qui sera utilisé et on l'applique à notre échantillon d'entraînement. On peut ensuite afficher le résultat obtenu sur notre ensemble

de validation et/ou exporter en .csv les genres prédits sur notre ensemble de test. On peut observer les étapes décrites jusqu'à présent dans la figure 1.

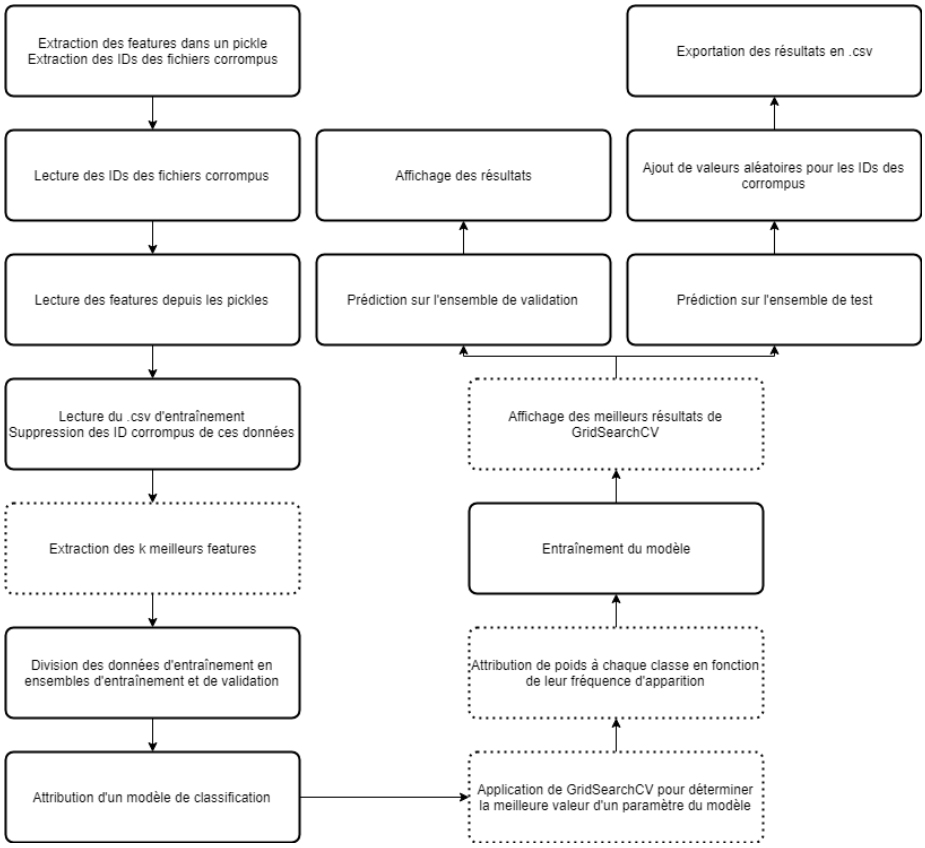

FIGURE 1 – *Affichage du pipeline de traitement des données pour la classification. Les procédés en pointillés ne sont appelés que lorsque l'on cherche à optimiser le modèle et ne sont pas utilisés pour des soumissions sur Kaggle.*

3 Amélioration des résultats

3.1 Utilisation de plusieurs algorithmes de Scikit-learn

On utilise la bibliothèque de machine learning Scikit-learn afin de classifier nos données. On essaye plusieurs algorithmes connus que la bibliothèque implémente et on observe les résultats dans le tableau ci-dessous. Les noms de méthodes sont donnés tels qu'ils sont présentés dans le fichier *models.py* où chaque fonction correspond à un modèle et ses paramètres particuliers.

La première méthode correspond à l'algorithme des plus proches voisins, la

suivante à un classification naïve bayésienne, la troisième à un arbre de décision et les cinq dernières à différents solveurs d'une régression logistique.

Méthode	Accuracy
KNN	0.3625
GaussNB	0.405
DecisionTree	0.365
LogisticSaga	0.2975
LogisticSag	0.3625
LogisticLbfgs	0.4375
LogisticLiblinear	**0.515**
LogisticNewton	0.5025

On atteint une classification à 51,5% correcte avec une régression logistique par solveur 'liblinear'.

3.2 Augmentation du nombre de features

Afin d'améliorer nos résultats, une première étape évidente consiste à extraire plus de données de nos fichiers audio. On utilise les fonction de Librosa afin de sauvegarder dans différents pickles un ensembles des MFCC, un ensemble de plusieurs features et un ensemble de tous les features proposés par Librosa sauf *poly_features*. Pour chaque feature, on ne sauvegarde que sa moyenne et sa variance. Cela nous fait un total de 406 données par musique dans le dernier cas.

Méthode	MFCC uniquement	Plusieurs features	Tous les features
KNN	0.3625	0.265	0.3025
GaussNB	0.405	0.385	0.305
DecisionTree	0.365	0.39	0.4375
LogisticSaga	0.2975	0.22	0.2375
LogisticSag	0.3625	0.225	0.275
LogisticLbfgs	0.4375	0.3375	0.315
LogisticLiblinear	**0.515**	0.53	0.3775
LogisticNewton	0.5025	**0.54**	0.46

On observe les résultats de la classification sur les méthodes présentées dans la section précédente. On remarque que certaines méthodes n'arrivent pas à gérer une quantité importante de features et proposent des résultats moins bien. On observe tout de même que, avec nos premiers algorithmes, on arrive à passer de 51,5% à 54% en rajoutant des features.

3.3 Utilisation de XGBoost

On décide d'utiliser la bibliothèque XGBoost permettant la classification des données ainsi que leur augmentation en appliquant des transformations à

nos données de bases. On fait appel à XGBoost par une fonction wrapper pour se conformer à notre modèle Scikit-learn actuel. Le format des données et les fonctions d'entrainement et de prédiction restent ainsi celles de Scikit-learn.

Plusieurs algorithmes de classification sont proposés par XGBoost, on utilisera l'algorithme *softmax* qui est le plus adapté à obtenir une classification multi-classes sans avoir besoin de conserver les probabilités d'appartenance.

On compare un appel à notre nouveau modèle non-paramétré (*model = XGBClassifier()*) aux résultats précédemment obtenus.

Méthode	MFCC uniquement	Plusieurs features	Tous les features
GaussNB	0.405	0.385	0.305
LogisticLiblinear	**0.515**	0.53	0.3775
LogisticNewton	0.5025	**0.54**	0.46
XGB1	0.5125	**0.54**	**0.5725**

On remarque que notre nouvelle méthode produit des résultats presque équivalents à nos meilleurs résultats avec peu de features et drastiquement supérieurs avec tous les features. En effet, cette nouvelle méthode s'améliore à chaque fois qu'on lui rajoute des features et ne s'en retrouve pas pénalisée. On atteint ainsi un taux de précision de 57,25%.

Il est aussi important de noter que dû à l'augmentation effectuée sur les données, cette méthode est beaucoup plus lente que les méthodes précédentes. Afin de réduire l'impact de ce facteur, il est important de passer à notre fonction une indication du nombre de threads qu'elle peut utiliser via le paramètre *n_jobs*.

3.4 Fonctions types

Méthode	Accuracy
XGB1	0.5725
XGBCustom1	0.6025
XGBCustom2	0.6025
XGBCustom3	0.5875
XGBCustom4	**0.61**

On compare notre fonction non-paramétrée à plusieurs fonctions types trouvées sur Internet qui sont appliquées à des cas similaires. On observe, sans trop de surprise, que on obtient de meilleurs résultats avec des fonctions paramétrées, notamment avec XGBCustom4 qui atteint les 61% de précision.

3.5 Paramétrage par GridSearchCV

On utilise la fonction GridSearchCV de Scikit-learn qui nous permet de définir un modèle initial et de définir des variables de ce modèle à tester. Pour chacune de ces variables, on définit les valeurs que l'on souhaite tester. Le programme va ensuite effectuer plusieurs tests avec différentes seed aléatoires pour

chacune de ces valeurs et calculer un résultat moyen pour chaque combinaison de paramètres proposées. C'est, bien évidemment, un procédé qui demande beaucoup de temps, il est donc important de limiter autant que possible les combinaisons à tester. On observe dans la figure 2 un exemple de résultat.

```
(Project) tepepin@jolicoeur:~/TSMA/Project/src$ python classify.py
5 corrupted files detected.
{'n_estimators': 542}
0.5576687578419071
```

FIGURE 2 – *Exemple de résultat pour une évaluation via GridSearchCV. On a fait varier le paramètre 'n_estimators' de 450 à 550 lors de l'exécution. La meilleur précision de 0.5577 a été obtenue avec 542 estimateurs.*

On propose d'améliorer progressivement le constructeur par défaut afin de déterminer des valeurs optimales pour chaque paramètre. Les constructeurs exactes sont visibles dans le fichier *models.py*. On présente le tableau suivant qui contient les évolutions à chaque étape permettant d'améliorer notre modèle :

Méthode	Feature(s) ajouté(s)	Accuracy
XGB1	n_jobs=-1, random_state=50	0.5725
XGB2	objective='multi :softmax', num_class=8	0.5725
XGB3	min_child_weight=6, max_depth=8	0.5875
XGB4	gamma=0.3	0.59
XGB5	subsample=0.8, colsample_bytree=0.8	0.59
XGB6	learning_rate=0.1, booster='gbtree'	0.59
XGB7	nb_estimators=300	**0.59**

On remarque qu'il n'y a pas de différence de résultat entre XGB1 et XGB2 car les paramètres ajoutés avec XGB2 sont ceux utilisés par défauts par XGB1. Le paramètre *objective* a tout de même été déterminé comme optimale par *GridSearchCV*.

De manière similaire, les résultats de XGB4 à XGB7 sont identiques. Cela est dû au fait que certaines de ces valeurs correspondent aux valeurs par défaut et que l'ensemble de validation sur lequel la précision est calculée ne contient que 400 fichiers (10% de l'ensemble train). Ce genre de différence ce remarquerait mieux sur l'échantillon de test de 4008 fichiers.

On remarque tout de même que nos résultats sont inférieurs à ceux de XGBCustom4. On décide donc de partir d'appliquer le même procédé à XGBCustom4. On prends grand soin d'appliquer la recherche de GridSearchCV au nombre d'estimateurs près (plutôt que de se contenter de la centaine la plus proche) ainsi que de chercher le meilleur *learning_rate* au millième près. Cela nous donne la fonction que l'on nommera XGBFinal.

Méthode	Accuracy
XGB7	0.59
XGBCustom4	0.61
XGBFinal	**0.6125**

3.6 Ajout de poids

On propose d'utiliser la fonction de Scikit-learn nommée *compute_sample_weight* afin d'attribuer un poids à chaque instance d'un genre spécifique. Ce poids fait office de coefficient qui représente la fréquence d'apparition d'un genre dans l'ensemble des données d'entraînement. En effet, tous les genres musicaux ne sont pas représentés à part égal. Cependant, la proportion de chaque genre dans l'ensemble d'entraînement n'est pas nécessairement la même que celle de l'ensemble de test.

Méthode	Accuracy sur validation	Accuracy sur Kaggle
XGBFinal sans poids	0.6125	**0.61220**
XGBFinal avec poids	**0.615**	0.60413

3.7 Extraction des meilleurs features

Nous avions précédemment observé que l'ajout d'une grande quantité de features n'est pas nécessairement favorable pour la classification. On propose donc d'utiliser la fonction *SelectKBest* de Scikit-learn afin d'évaluer la pertinence des features utilisés et de ne conserver que les meilleurs. On évalue nos résultat pour différentes quantités de features avec le modèle XGBFinal :

Nombre de features conservés	Accuracy
10	0.456
20	0.49
50	0.55
100	0.5625
150	0.585
200	0.615
250	0.61
300	0.615
350	0.61
400	**0.6175**
406	0.6125

On constate que notre classification est plus performante lorsque le nombre de features augmente, mais que celle-ci reste relativement stable lorsque l'on atteint les 200 features utilisés. La variance des résultats observée à partir de ce point est probablement due à des cas particuliers de classification présents dans les 400 musiques de validation. On suppose que l'effet de ces cas particuliers est rendu négligeable lorsque l'on passe au plus large échantillon de test. La sélection de features n'est donc pas nécessaire à utiliser avec notre algorithme,

mais cela peut cependant servir à réduire le temps de calcul dont le programme a besoin pour classifier en fonction des features.

3.8 Résultat final

Notre modèle final reste donc XGBFinal tel qu'on l'a précédemment défini sans modifications supplémentaires. On propose ci-dessous un tableau récapitulatif de ses paramètres :

Paramètres	Valeurs
n_jobs	-1
random_state	50
objective	'multi :softmax'
num_class	8
min_child_weight	1
max_depth	5
gamma	0
subsample	0.8
colsample_bytree	0.8
learning_rate	0.08
booster	'gbtree'
n_estimators	542
scale_pos_weight	1

On obtient le résultat tel que visible dans la figure 3 sur l'ensemble de validation et un résultat de 61,220% sur la partie publique du Kaggle. Il est important de noter que dans le cas d'une soumission Kaggle, on utilise la totalité des 4000 musiques de train pour l'entraînement, sans faire de sous-ensemble de validation. Ayant plus de données d'entraînement, notre performance de classification augmente par conséquent. On obtient aussi la matrice de confusion de la figure 4 et on peut voir la précision par genre dans la figure 5.

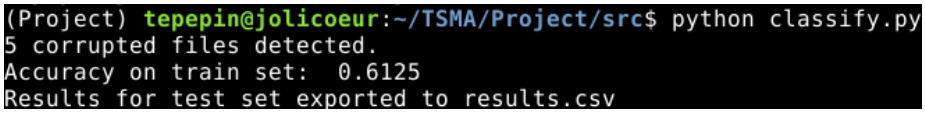

FIGURE 3 – *Affichage standard en console.*

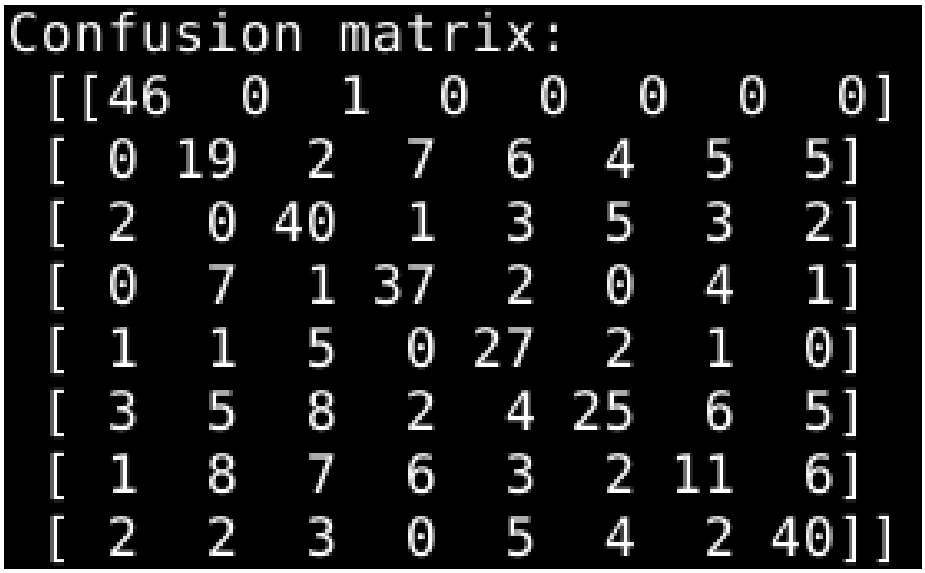

FIGURE 4 – *Matrice de confusion de notre modèle final. On a un total de 400 éléments dans notre ensemble de validation.*

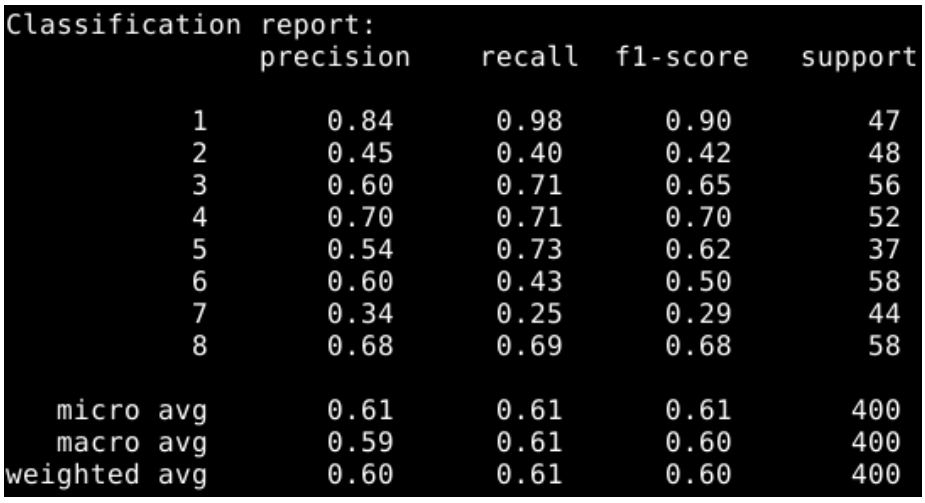

FIGURE 5 – *Rapport de classification de notre modèle final. On observe notamment la précision pour chaque classe.*

4 Conclusion

On a donc réussi à correctement attribuer un genre à 61,25% des musiques de l'ensemble de test. Ceci est comparable à des résultats obtenus dans ce domaine en 2001 [1] et 2002 [2] par Tzanetakis et Cook.

Ces derniers ont aussi fait effectuer des tests de classification par des étudiants universitaires. Les étudiants ont réussi à correctement classifier 53% des musiques en écoutant seulement un échantillon de 250ms. Ce taux de classification atteint 70% lorsque les étudiants écoutent au moins 3 secondes du fichier audio. Notre résultat obtenu par machine learning est ainsi très proche des résultats humains.

Cependant, les algorithmes de classification se sont nettement améliorés depuis ces publication. Les auteurs d'une publication de journal en 2016 [3] mette en avant un taux de classification par genres d'environ 84%. Ils le compare aussi à d'autres algorithmes récents qui obtiennent des résultats similaires ou meilleurs.

On pourrait améliorer notre algorithme actuel de plusieurs manières. Une première méthode serait d'extraire d'autres features que ceux proposés par Librosa. On pourrait aussi améliorer la qualité et la quantité du dataset d'entraînement utilisé.

Une autre méthode serait d'utiliser un algorithme de classification autre que *softmax* qui est proposé par XGBoost. Les auteurs de la dernière publication [3] utilise notamment un algorithme de Support Vector Machines pour leur classification.

Il serait également concevable d'utiliser d'autres modèles de classification appliqués uniquement à certaines classes avant d'appliquer la classification globale sur les autres classes. En effet, on remarque dans la figure 5 que les classes 7 et 2 sont fortement inférieures à la moyenne. Ce sont des classes dont le genre est difficile à catégoriser dû à leur ambiguïté.

5 Références

[1] George Tzanetakis, Georg Essl, and Perry Cook. Automatic musical genre classification of audio signals. In *Proc. of 2nd Annual International Symposium on Music Information Retrieval, Indiana University Bloomington, Indiana, USA*, 2001.

[2] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5) :293–302, 2002.

[3] Loris Nanni, Yandre MG Costa, Alessandra Lumini, Moo Young Kim, and Seung Ryul Baek. Combining visual and acoustic features for music genre classification. *Expert Systems with Applications*, 45 :108–117, 2016.