GTI770-TP04-Equipe_02_rapport

August 7, 2019

1 Laboratoire 4 : Développement d'un système intelligent

Étudiants	Alexandre Laroche - LARA12078907Marc-Antoine Charland - CHAM16059609Jonathan Croteau-Dicaire - CROJ10109402
Cours	GTI770 - Systèmes intelligents et apprentissage machine
Session	Été 2019
Groupe	02
Numéro du laboratoire	TP-04
Professeur	Prof. Alessandro L. Koarich
Chargé de laboratoire	Pierre-Luc Delisle
Date	7 août 2019 (23h55)

Département du génie logiciel et des technologies de l'information

2 1. Introduction

Le développement sans cesse des technologies du web ouvre constamment des portes sur le partage et l'accès à l'information. Parmi les marchés qui se développent rapidement, celui de la distribution et de l'accès aux pièces musicales en ligne se taille une place importante dans l'univers de la musique. La demande en gestion des collections de pièces musicales ne fait que grandir. Pour ces raisons, un sérieux intérêt s'est construit autour de la mise en place d'un système qui automatise la création de listes de lecture dans le but de répondre à des besoins utilisateurs personnalisés. C'est à ce moment qu'entre en jeu l'apprentissage machine.

Dans le secteur de l'apprentissage machine, lorsqu'un problème de classification survient, effectuer une analyse préliminaire est primordiale non seulement pour obtenir une meilleure vue d'ensemble du problème actuel, mais surtout pour éviter de refaire les erreurs que certains ont déjà expérimentées et pour donner une ligne directrice aux méthodes à employer quant à la résolution du problème. Dans le cas présent, il est question de concevoir et implémenter un système intelligent qui a la capacité de résoudre un problème de classification afin de reconnaître le style musical d'un ensemble de données représentant une série de pièces musicales. Dans le cadre du quatrième laboratoire, plusieurs documents de littérature ont été fournis pour orienter les recherches. Une lecture approfondie des documents a permis de soulever plusieurs similarités et points importants. En effet, le sujet qui concerne les techniques de combinaison d'algorithmes, la réduction de dimensions des ensembles de données ainsi que l'impact de la sélection des primitives sur la

précision de classification est au coeur du sujet qui concerne le problème.

Le document littéraire Facilitating comprehensive benchmarking experiments on the million song dataset écrit par Alexander Schindler, Rudolf Mayer, and Andreas Rauber souligne plusieurs informations importantes quant à la ligne directrice à opter. Le document indique que les algorithmes de classification tels que SVM et KNN pointent vers une accuracy pouvant dépasser les 27% avec l'utilisation de la primitive SpectrumDescriptor. Aussi, par ordre de performance descendante, les primitives SSD, Derivatives of the jMIR spectral features et jMIR MFFC implémentation rendent intéressant l'utilisation de l'algorithme de classification RandomForest. Enfin, le document montre l'utilisation d'une stratification au-delà que simplement sur la proportion de chaque classe. La raison est simple, le numéro d'une pièce musicale d'un album peut se retrouver à la fois dans l'ensemble de données utilisé pour l'entraînement que celui des tests. Dans un autre ordre d'idées, le document littéraire trouvé sur internet portant le titre Genre Classification for million Song Dataset Using Confidence-Based Classifiers Combination écrit par Yajie Hu et Mitsunori Ogihara présente une méthode de combinaison de modèles qui pointe vers une accuracy dépassant les 80%. Parmi les modèles, le réseau de neurones en faisait partie. Les primitives qu'ils ont utilisées sont des échantillons audios, des social tags, des paroles ainsi que des termes (noms) d'artistes.

Face au problème de classification actuel, une stratégie a été mise en place pour ne rien manquer. Tout d'abord, il faut effectuer les tests nécessaires pour évaluer l'ensemble de primitives qui offre la meilleure performance sur les différents modèles à investiguer. D'ailleurs, les premiers modèles à investiguer seront KNN, SVM, MLP et RandomForest. Ensuite, effectuer un prétraitement qui met l'accent sur la normalisation ainsi que la stratification des données. Par la suite, identifier la meilleure technique de réduction de dimensionnalité pour les différents modèles en cours d'investigation. Aussi, il faut évaluer et mettre en place les méthodes de validation pour confirmer la bonne orientation du projet. Ensuite, il sera question de mettre au clair la sélection des modèles de bases ainsi qu'évaluer et optimiser leurs paramètres en fonction des ensembles de primitives. Enfin, c'est à ce moment qu'on y verra la conception du système final, les hyperparamètres du métamodèle, les résultats finaux du système ainsi que des pistes d'améliorations.

3 2. Configuration et traitement des données

3.1 2.1. Configuration

tensorflow-gpu 1.13.1
- Nvidia GTX 960
- 2 GB GDDR5
- 1.1 GHz
- 1024 CUDA Cores
- Intel Core i7-4790K Devil's Canyon
- 4 Cores
- 8 Threads
- 4.4 GHz
- 16 GB DDR3

3.2 2.2. Choix des ensembles de primitives

Afin de choisir les ensembles de primitives utilisées, une expérimentation a été conduite. Pour ce faire, un MLP très peu optimisé a été entraîné pour chacun des ensembles de primitives. La figure

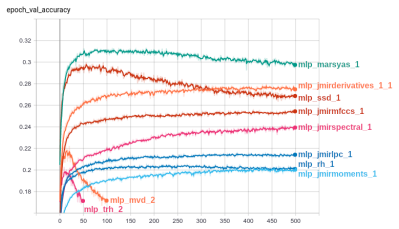
2.2.1 présente l'évolution de l'accuracy de validation de chacun des MLPs. Ainsi, les trois ensembles de primitives différents ont été choisie en ordre décroissant de leur accuracy de validation lors de cette expérimentation.

Tableau 2.2.1 Résultats des MLP sur chaque ensemble de primitives

[7]:
```
feature_set_experiments_results_path = os.path.join(constants.LOGS_PATH,␣
 ↪'rapport', 'feature_set_experiment_run_results.json')
feature_set_experiments_results =␣
 ↪read_json_ordered_dict(feature_set_experiments_results_path)

display(pd.DataFrame(feature_set_experiments_results).reindex(['best_epoch',␣
 ↪'loss', 'val_loss', 'accuracy', 'val_accuracy']).transpose())
```

	best_epoch	loss	val_loss	accuracy	val_accuracy
mlp_marsyas_1	97.0	2.088358	2.243324	0.352197	0.311799
mlp_ssd_1	46.0	2.128508	2.291429	0.341734	0.297040
mlp_jmirderivatives_1	331.0	2.211810	2.355624	0.313602	0.277519
mlp_jmirmfccs_1	475.0	2.383962	2.449880	0.270453	0.254156
mlp_jmirspectral_1	494.0	2.453926	2.488728	0.247703	0.239481
mlp_mvd_2	13.0	2.418119	2.585535	0.269047	0.223859
mlp_jmirlpc_1	420.0	2.545402	2.601319	0.229031	0.214363
mlp_rh_1	116.0	2.568651	2.649563	0.223337	0.203364
mlp_trh_2	10.0	2.513337	2.644916	0.240142	0.202974
mlp_jmirmoments_1	495.0	2.604361	2.639236	0.208641	0.200050



feature_set_experiments_val_acc

Figure 2.2.1 Évolution de l'accuracy de la validation d'un MLP sur chaque ensemble de primitives

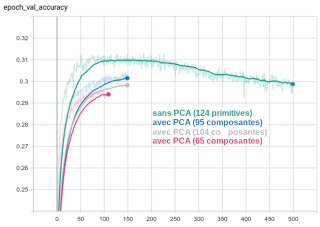
3.3 2.3. Normalisation et mise à l'échelle des données

Un premier prétraitement appliqué est une normalisation des données d'entrées à l'aide de la classe `sklearn.preprocessing.StandardScaler`. Cette méthode de normalisation est choisie, car elle présentait de meilleurs résultats lors des expérimentations conduites au tp3 que d'autres méthodes de normalisation et de mise à l'échelle.

3.4 2.4. Réduction de la dimensionnalité

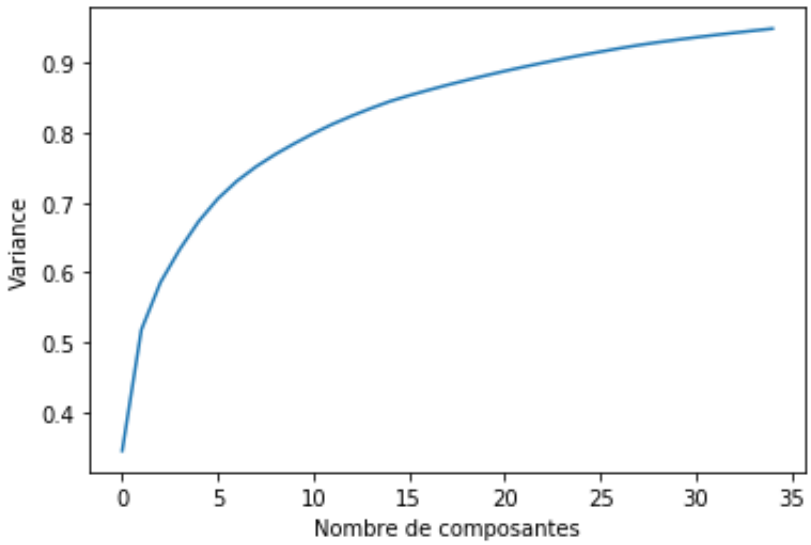
Une multitude d'expérimentations ont été conduites pour déterminer si l'utilisation d'une technique de réduction de dimensionnalité est pertinente avec les ensembles de primitives choisies. Les techniques évaluées lors de ces expérimentations sont l'analyse des composantes principales (PCA), l'analyse discriminante linéaire (LDA) et l'utilisation d'un autoencoder en prétraitement. La suite de cette section décrit des expérimentations qui ont été effectuées à ce sujet. Pour la concision de ce rapport et par manque de contexte sur le système final, seulement les expérimentations les plus pertinentes pour la prise des décisions ont été décrites. La section ń 5. Conception du système final ż indique l'utilisation de ces techniques dans le système finale. Notez que pour ces expérimentations la technique de validation utilisée est un split train/test 80/20, sauf pour indication contraire.

Pour ce qui est de l'utilisation de PCA, cette technique ne semble pas pertinente pour l'ensemble de primitives Marsyas. La figure 2.4.1. montre que l'accuracy de validation d'un MLP non optimisé est réduit lorsque que PCA est utilisé en prétraitement. En détail, cette figure compare l'évolution de l'accuracy de validation de quatre MLP entraînés avec ou sans PCA en prétraitement. Les trois utilisations de PCA ont respectivement utilisé 65, 95 et 104 composantes principales. Préalablement, une visualisation du la variance cumulative des composantes a été effectuée. La figure 2.4.2. présente cette visualisation. De plus, cette analyse a déterminé que seulement 35 composantes principales sont nécessaires pour expliquer 99.5% de la variance. Cependant, réduire les 124 primitives de l'ensemble Marsyas à seulement 35 composantes semble extrême. Cela explique le choix du nombre de composantes testées. Enfin, face à ces résultats, PCA n'a pas été utilisé sur l'ensemble Marsyas.



mlp_marsyas_pca_val_acc

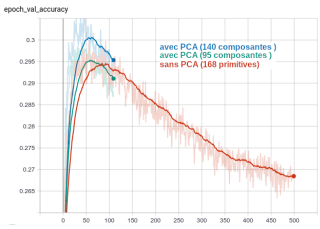
Figure 2.4.1. Évolution de l'accuracy de la validation d'un MLP avec PCA sur Marsyas



cumulative_variance_pca_marsyas

Figure 2.4.2. Variance cumulative des composantes principales de PCA sur Marsyas Notez qu'une visualisation préliminaire de la variance cumulative a aussi été effectuée pour les expérimentations avec PCA et LDA décrits dans la suite de cette section du rapport. Par souci de concision, les figures présentant ces visualisations sont omises.

À première vue, PCA semble légèrement approprié sur l'ensemble de primitives SSD (Statistical Spectrum Descriptor) utilisé par un MLP non optimisé. La figure 2.4.3. montre que l'utilisation de PCA avec 140 composantes augmente l'accuracy de validation d'environ 0,5 %. Ici, 104 composantes principales expliquent 99.5 % de la variance. En revanche, plus tard dans le laboratoire, lors de l'ajout d'un nouveau modèle de base, soit un MLP partiellement optimisé, une expérimentation a révélé que des résultats quasi égaux peuvent êtres obtenues avec ou sans PCA. Par simplicité, la décision de ne pas utiliser PCA sur ce modèle a été prise.

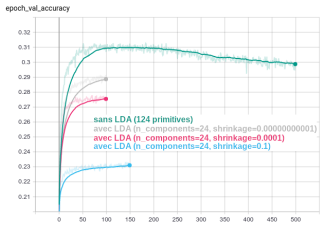


mlp_ssd_pca_val_acc

Figure 2.4.3. Évolution de l'accuracy de la validation d'un MLP avec PCA sur SSD Additionnellement, la technique LDA a aussi été testée sur l'ensemble Marsyas à l'aide d'un MLP. Tout comme PCA, l'utilisation de LDA n'est pas favorable pour cet ensemble. La figure 2.4.4. présente l'évolution de l'accuracy de validation au cours des epochs d'entraînement. Malheureusement, le nombre de composantes maximales permises par LDA est définit par :

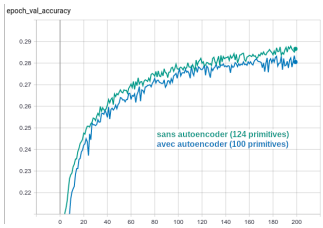
```
n_component <= min(n_classes - 1, n_features)
```

De ce fait, seulement 24 composantes soit `n_classes - 1` peuvent êtres utilisés avec LDA sur l'ensemble Marsyas. Ainsi, cette expérimentation a permis de comprendre que LDA est très peu approprié aux ensembles qui contiennent largement plus de 24 primitives tel que Marsyas, SSD et JMIR Derivatives. Pour cette raison, LDA n'est pas utilisé.



mlp_marsyas_lda_val_acc

Figure 2.4.4. Évolution de l'accuracy de la validation d'un MLP avec LDA sur Marsyas Enfin, la dernière technique de réduction de dimensionnalité évaluée est l'autoencoder. Le choix de considérer cette technique revient au fait que c'est bien connu qu'un autoencoder linéaire à une seule couche caché peut approximer PCA et qu'un autoencoder non-linéaire peut surpasser PCA. De ce fait, lors de ce travail, deux autoencoders non-linéaires ont été produits respectivement pour Marsyas et SSD. Malheureusement, l'utilisation des autoencoders n'a pas donné des résultats concluent. Pourtant, la conception ainsi que l'utilisation semble correcte. Pour information, suite à l'entraînement de ces autoencoders, seulement la partie ń encoder ż est utilisée pour traiter les données. Le code clé de l'autoencoder pour Marsyas est présenté ci-dessous. Aussi, la figure 2.4.5. présente l'accuracy de validation de cet autoencoder.



mlp_marsyas_autoencoder_val_acc

Figure 2.4.5. Évolution de l'accuracy de la validation d'un MLP avec l'autoencoder sur Marsyas

3.5 2.5. Méthodes de validation

Telle que décrit plus tôt, les expérimentations préliminaires portant sur le choix des ensembles de primitives, de même que sur l'utilisation de la réduction de dimensionnalité, ont été validés en utilisant un simple split train/test 80/20. Puisque la précision de ces expérimentations est peu importante, la rapidité d'exécution de ce type de validation a été favorisée.

Suite à ces expérimentations, l'ensemble de données complet a été divisé en deux sous-ensembles, soit *base* et *meta*. Ces ensembles contiennent respectivement 80 % et 20 % des données. De plus, ils ont respectivement été utilisés pour l'entraînement des modèles de base et du métamodèle.

En ce qui concerne l'entraînement des modèles de base, celui-ci a débuté par une recherche des meilleurs hyperparamètres. Cette recherche a été conduite sous une validation croisée de type K-folds où `k=5`. Par la suite, chaque modèle de base final a été entraîné avec ses meilleurs hyperparamètres sur la totalité de l'ensemble *base* sans validation. Les modèles de base finaux ont ensuite été combinés tels que décrits dans la section 5.

Par la suite, le métamodèle a été entraîné sur l'ensemble *meta*. Puisque le métamodèle agrège les résultats des modèles de base, c'est important que sont entraînement ne soit pas effectué à l'aide de données qui ont déjà été vues par les modèles de base. Cela explique pourquoi l'ensemble de données original a été séparé en deux. Ainsi, la recherche des meilleurs hyperparamètres de ce métamodèle a été effectuée sous une validation croisée de type K-folds où `k=5`. Enfin, suite à la découverte de ces hyperparamètres, le métamodèle a été entraîné sur l'ensemble *meta* complet.

4 3. Choix des modèles de bases

Les trois modèles de bases choisis sont un MLP, un Random Forest et un SVM. Ces trois modèles ont été choisis dus à leur réputation comme apprenant fort. De plus, face à la complexité du problème à l'étude, prioriser des modèles capables de définir des frontières non-linéaire semble être un choix approprié. Aussi, ce sont les modèles avec lesquels les membres de l'équipe sont le plus à l'aise. Parallèlement, les MLP sont reconnues pour avoir de bonnes performances sur de grand ensembles de données. Dans les faits, la structure d'un MLP n'est pas dépendante de la taille de l'ensemble de données. Au contraire, la taille des SVM dépend de la taille de l'ensemble de données. Cela s'explique par le fait qu'un ensemble de données de plus grande taille peut contenir une plus grande quantité de vecteurs de support. Par conséquent, le temps d'entraînement d'un SVM grandit rapidement avec la taille de l'ensemble de données. Ainsi, le choix de SVM a été fait en partie à contrecoeur dû au temps nécessaire à son entraînement.

Ce dernier point explique pourquoi KNN n'a pas été considéré; ce modèle est particulièrement lent à entraîner sur un grand nombre de données. Aussi il est faible au fléau de la dimension. Cela à été pris en compte puisque, certains ensembles de primitives ont un grand nombre de dimensions, par exemple l'ensemble Temporal Rhythm Histograms à 420 dimensions.

Pour ce qui est des modèles de Bayes, ceux-ci n'ont pas été considérés, car ces modèles n'ont pas eu une bonne performance dans les laboratoires précédents.

Enfin, l'énoncé de ce travail est quelque peu ambigu. Celui-ci demande pour la question présente de décrire la structure des modèles telle que les réseaux de neurones. En même temps, l'énoncé demande de décrire les hyperparamètres des modèles de bases à la question 4. Puisque nous considérons la structure d'un réseau de neurones comme étant un hyperparamètre, cette exigence est répondue à la question 4.

5 4. Hyperparamètres et résultats des modèles de base

Tel que demandé par l'énoncé, cette section présente les hyperparamètres et les résultats des trois modèles de base entraînés sur les trois ensembles de données différents. Cependant, pour concision, les autres modèles de base ne sont pas présentés dans cette section.

5.1 4.1. Hyperparamètres et résultats du MLP sur l'ensemble Marsyas

Pour le MLP de base, ce modèle a été entraîné sur l'ensemble de primitives Marsyas. Le code suivant présente la structure du réseau :

```
inputs = Input(shape=124)

layer = Dense(90, activation='relu')(inputs)
layer = Dense(85, activation='relu')(layer)

layer = Dropout(0.10)(layer)

layer = Dense(80, activation='relu')(layer)
layer = Dense(75, activation='relu')(layer)
layer = Dense(70, activation='relu')(layer)

layer = BatchNormalization()(layer)

layer = Dense(65, activation='relu')(layer)
layer = Dense(60, activation='relu')(layer)

layer = BatchNormalization()(layer)

layer = Dense(55, activation='relu')(layer)
layer = Dense(50, activation='relu')(layer)

layer = Dropout(0.10)(layer)

layer = Dense(45, activation='relu')(layer)
layer = Dense(40, activation='relu')(layer)

layer = Dropout(0.10)(layer)

layer = Dense(30, activation='relu')(layer)

outputs = Dense(25, activation='softmax')(layer)

model = Model(inputs=inputs, outputs=outputs)
```

Voici une liste des autres hyperparamètres de ce modèles :

Hyperparamètre	Valeur
Optimizer	Adam
Taux d'apprentissage	par défaut (0.001)
Taux d'apprentissage	par défaut (0.001)
Fonction de loss	categorical crossentropy
Batch size	2000
Epochs	110

Tableau 4.1.1. Résultats de l'entrainement du MLP de base sur Marsyas

[4]:
```
base_mlp_marsyas_results_path = os.path.join(constants.LOGS_PATH, 'rapport',␣
 ↪'new_base_mlp_marsyas_cv_1_run_results.json')
base_mlp_marsyas_results = read_json_ordered_dict(base_mlp_marsyas_results_path)
base_mlp_marsyas_df = pd.DataFrame([base_mlp_marsyas_results]).transpose().
 ↪reindex(['avg_accuracy', 'avg_val_accuracy', 'avg_batch_val_accuracy',␣
 ↪'avg_loss', 'avg_val_loss', 'avg_val_f1_macro', 'avg_val_f1_micro',␣
 ↪'train_time'])

display(base_mlp_marsyas_df)
```

	0
avg_accuracy	0.325711
avg_val_accuracy	0.297646
avg_batch_val_accuracy	0.297646
avg_loss	2.188799
avg_val_loss	2.298860
avg_val_f1_macro	0.274868
avg_val_f1_micro	0.297646
train_time	329.797179

C'est important de noter que les valeurs de l'accuracy et de loss sont des moyennes sur ces 5 folds. Curieusement, l'accuracy de validation de ce modèle de 29.8 % est plus basse que celle du MLP entraîné sur le même ensemble lors de l'expérimentation décrite à la section 2.2. Cela s'explique probablement par le fait que le modèle entraîné à la section 2.2 a seulement été validé avec un split train/test 80/20.

De plus, le score F1 de type macro, c'est-à-dire que cette mesure ne prend pas en compte l'imbalance des données, est de 27.5 %. Pour sa part, le score F1 micro, qui prend en compte l'imbalance des données, est équivalent à l'accuracy de validation. Cela dit, l'imbalance des données réduit le score F1 de 2.3 %.

Le temps d'exécution de la validation croisée est de 5.5 minutes à l'aide de la configuration décrite dans la section 2.1.

5.2 4.2. Hyperparamètres et résultats du Random Forest sur l'ensemble SSD

Dans un premier temps, il est évident que ce n'est pas tous les hyperparamètres d'un modèle comme celui du RandomForest qui ont nécessairement un réel impact sur la précision de classification. En fait, tout dépend du problème ainsi que de la quantité de données à traiter. Puisque le modèle en question se trouve à être très efficace avec de ensembles de données de grandes tailles, l'ensemble SSD est dans ce cas plus intéressant.

Avant d'effectuer une série de tests et ajustements sur les paramètres du modèle en question, une analyse doit d'abord être effectuée pour mieux comprendre leurs impacts. Comme point de départ, c'est au niveau du code qu'on vient afficher les paramètres par défaut du modèle pour ensuite effectuer une recherche sur la marge des valeurs à tester. Une fois l'analyse terminée, elle a permis à l'équipe de dresser une liste des paramètres qui ont non seulement un impact sur l'apprentissage du modèle, mais qui répond surtout au problème actuel. Ci-dessous, vous trouverez la liste des paramètres qui optimisent au mieux le modèle ainsi qu'une courte description

et justification sur la raison d'une telle prise de décision. Il faut indiquer que les chiffres (%) utilisés dans la description des paramètres ci-dessous sont des estimations de haut niveau tirés de plusieurs expérimentations effectuées sur le modèle RandomForest.

- **n_estimator**
  Représente le nombre d'arbres du modèle et de façon générale, plus le nombre d'arbres est élevé, plus l'apprentissage est meilleur. Cependant, utiliser une valeur trop élevée peut grandement ralentir le processus d'entraînement. Certains de nos tests préliminaires montrent qu'une valeur trop élevée pourrait diminuer la performance des tests de 5% et une valeur trop basse pourrait baisser la performance des tests de 10%.

  - Valeur par défaut: 10
  - Valeurs à tester: [10, 30, 60, 80, 100]

- **max_depth**
  Représente la profondeur de chaque arbre de la forêt. Plus l'arbre est profond, plus il se divise et plus il capture de l'information sur les données. Dans le cas présent, le paramètre servira à mieux comprendre son impact sur le taux d'erreurs autant au niveau des tests qu'au niveau de l'apprentissage. Certains de nos tests préliminaires semblent pointer vers un surapprentissage lorsque la valeur est trop élevée. Aussi, il semble que les arbres prédisent correctement les données d'entraînement, mais qu'il ne parvient pas à généraliser les résultats pour les nouvelles données. Ces théories seront confirmées un peu plus loin dans le rapport.

  - Valeur par défaut: None
  - Valeurs à tester: [None, 10, 30, 50, 70, 80]

- **min_samples_split**
  Représente le nombre minimal d'échantillons requis pour fractionner un noeud interne de l'arbre. Augmenter la valeur du paramètre fait en sorte que chaque arbre de la forêt devient plus contraint, car il doit prendre en compte un plus grand nombre d'échantillons sur chaque noeud. Certains de nos tests préliminaires ont montré que trop augmenter la valeur a comme conséquence une baisse significative de la performance des tests et de l'apprentissage du modèle pouvant aller jusqu'à 20%. En d'autres mots, l'apprentissage du modèle baisse en performance ce qui résulte en un sous-apprentissage des données.

  - Valeur de départ: 2
  - Valeurs à tester: [2, 4, 6, 8, 10]

- **min_sample_leaf**
  Représente le nombre d'échantillons minimum requis pour être un noeud de type feuille. Ce paramètre reste très similaire à celui du min_samples_split puisque si la valeur est trop élevée, on se retrouve avec du sous-apprentissage. Certains de nos tests préliminaires montrent qu'augmenter ou diminuer la valeur de 1 peut résulter en une baisse de 25% sur la performance de tests ainsi que sur la performance d'entraînement. Il faut donc absolument investiguer le paramètre en question.

  - Valeur par défaut: 1
  - Valeurs à tester: [1, 2, 3, 4, 5]

- **max_features**
  Représente le nombre de fonctionnalités à prendre en compte lors de la recherche du meilleur ''split". La documentation de sklearn indique que la recherche d'un ''split" ne s'arrête pas tant qu'au moins une partition valide des échantillons de noeud soit trouvé même si cela nécessite de dépasser le nombre de fonctionnalités indiqué en paramètre. En d'autres mots, le modèle peut tomber en sur-apprentissage. Puisque la performance de test semble être plus difficile à contrôler, aucun test préliminaire ne sera effectué pour apprendre à mieux connaître ses limites. Cependant, on inclut tout de même ce paramètre dans la liste pour en apprendre davantage.

  - Valeur par défaut: 'auto'
  - Valeurs à tester: ['auto' , 'sqrt']

- **bootstrap**
  Représente le cas où des échantillons bootstrap sont utilisés durant la construction des arbres. Le paramètre en question ne semble pas jouer énormément sur la performance de test ou d'entraînement du modèle. Cependant, il serait intéressant de voir si l'utilisation entière de l'ensemble de données pour la construction de chaque arbre vient optimiser le modèle.

  - Valeur par défaut: True
  - Valeurs à tester: [True, False]

Maintenant que la compréhension des hyperparamètres du modèle RandomForest se concrétise, il est désormais temps d'effectuer le nécessaire pour trouver la combinaison idéale d'hyperparamètres pour optimiser au mieux le modèle. Pour y arriver, plusieurs méthodes existent telles que GridSearchCV et RandomizedSearchCV. Les deux méthodes explorent exactement le même espace de paramètres et les résultats dans la recherche des meilleurs paramètres sont très similaires. Cependant, RandomizedSearchCV offre un temps d'exécution nettement plus intéressant que celui du GridSearchCV. Pour cette simple raison, l'utilisation de RandomizedSearchCV est mis de l'avant. Plusieurs tutoriaux indiquent que l'utilisation de 100 itérations avec un fold de 5 est assez bien balancé pour couvrir un assez grand intervalle de paramètres tout en minimisant le sur-apprentissage. Cependant, l'utilisation de 100 itérations pour trouver la meilleure combinaison de paramètres requiert une puissance de calcul qui dépasse largement celle auquelle les membres de l'équipe ont accès. En fait, l'ensemble de données du problème actuel est trop grand pour que nous puissions utiliser 100 itérations. Le paramètre du RandomizedSearchCV portant sur le nombre d'itérations a donc dû être réduit à 25. L'exécution de la recherche sur la meilleure combinaison de paramètres discutés plus haut a duré un total de deux heures sur un ordinateur ayant la configuration énoncée à la section 2.1. Ci-dessous, vous trouverez les paramètres par défaut du modèle et ceux trouvés avec RandomizedSearchCV pour venir optimiser le modèle.

Comparaison entre les hyperparamètres du Random Forest

Hyperparamètre	Par défaut	Optimisé	Conclusion haut niveau
n_estimator	10	80	Gain sur la performance des tests
max_depth	None	80	Réduction d'erreurs sur les données d'apprentissage
min_samples_split	2	4	Réduction du sur-apprentissag
min_sample_leaf	1	1	n/a
max_features	'auto'	'auto'	n/a
bootstrap	True	False	Utilisation complète du ''dataset"

La première exécution du modèle RandomForest sans hyperparamètres a donné une accuracy de 20% (0.2035) avec un résultat F1 de 19% (0.1916) sur l'ensemble de validation. Avec l'utilisation des hyperparamètres trouvés, le résultat de précision a augmenté à 29% (0.2915), soit un gain de 9% et a augmenté le résultat F1 à 27% (0.2688), soit un gain de 8%. À la lumière des résultats obtenus et face au problème actuel de classification, plusieurs conclusions haut niveau peuvent être faites. L'utilisation des nouveaux hyperparamètres a permis au modèle d'améliorer sa performance au niveau des tests, d'améliorer le contrôle du taux d'erreurs autant au niveau des tests qu'au niveau de l'apprentissage et d'avoir un meilleur équilibre entre sur-apprentissage et sous-apprentissage.

5.3 4.3. Hyperparamètres et résultats du SVM sur l'ensemble JMIR Derivatives

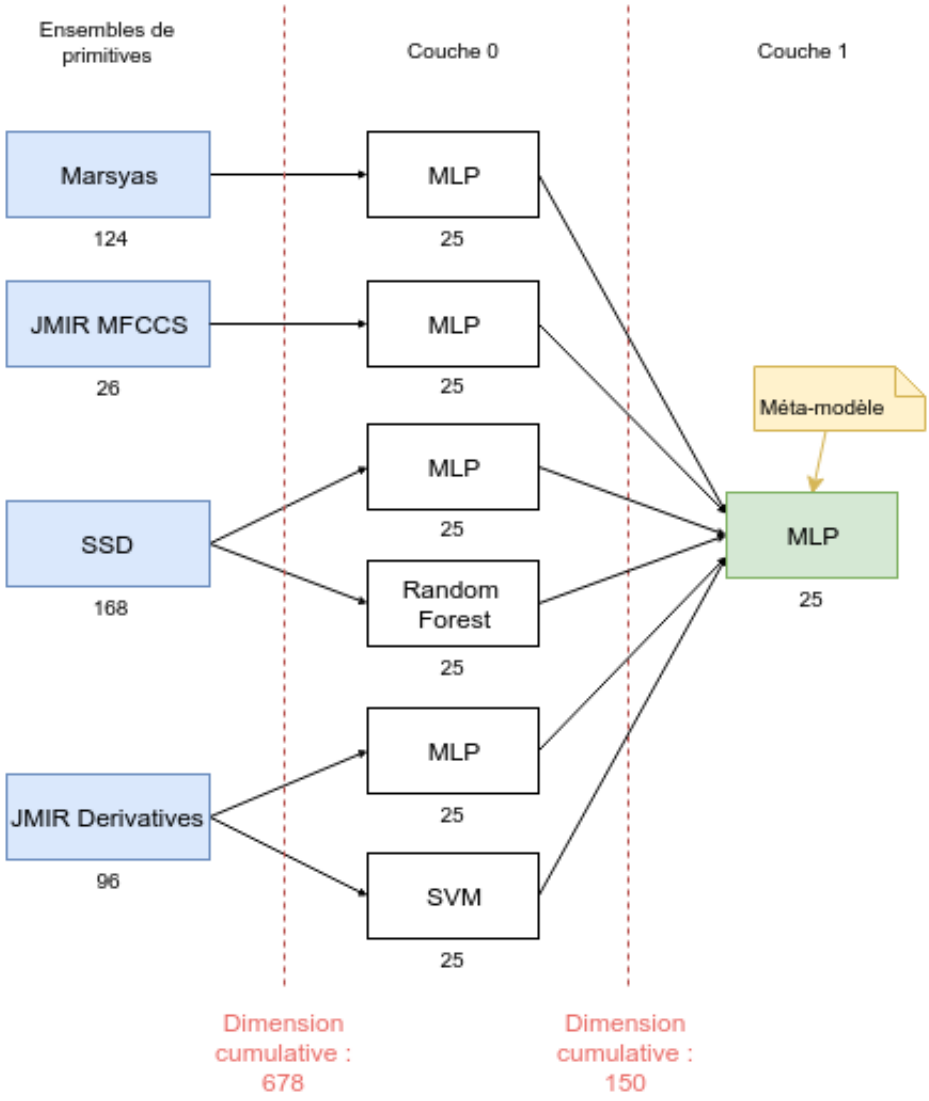
Afin d'identifier les meilleurs hyperparamètres pour le SVM sur l'ensemble JMIR Derivatives, une recherche en grille a été conduite sur une période de 3 jours. Les paramètres optimaux trouvés durant cette recherche sont les suivants :

Hyperparamètre	Valeur
kernel	rbf
C	10
gamma	0.001

Le SVM optimal a obtenu une accuracy de validation de 26.87 %. Ce résultat est inférieur à celui du MLP lors de l'expérimentation décrite dans la section 2.2. Cela s'explique peut être par la divergence entre les méthodes de validations. Cela serait surprenant que la cause soit l'utilisation d'un SVM au lieu d'un réseaux de neurones.

6 5. Conception du système final

La stratégie de combinaison utilisée par le système final est un *Stacking ensemble*. Cette stratégie consiste à agréger les sorties des modèles de base à l'aide d'un métamodèle tel que présenté dans la figure 5.1.



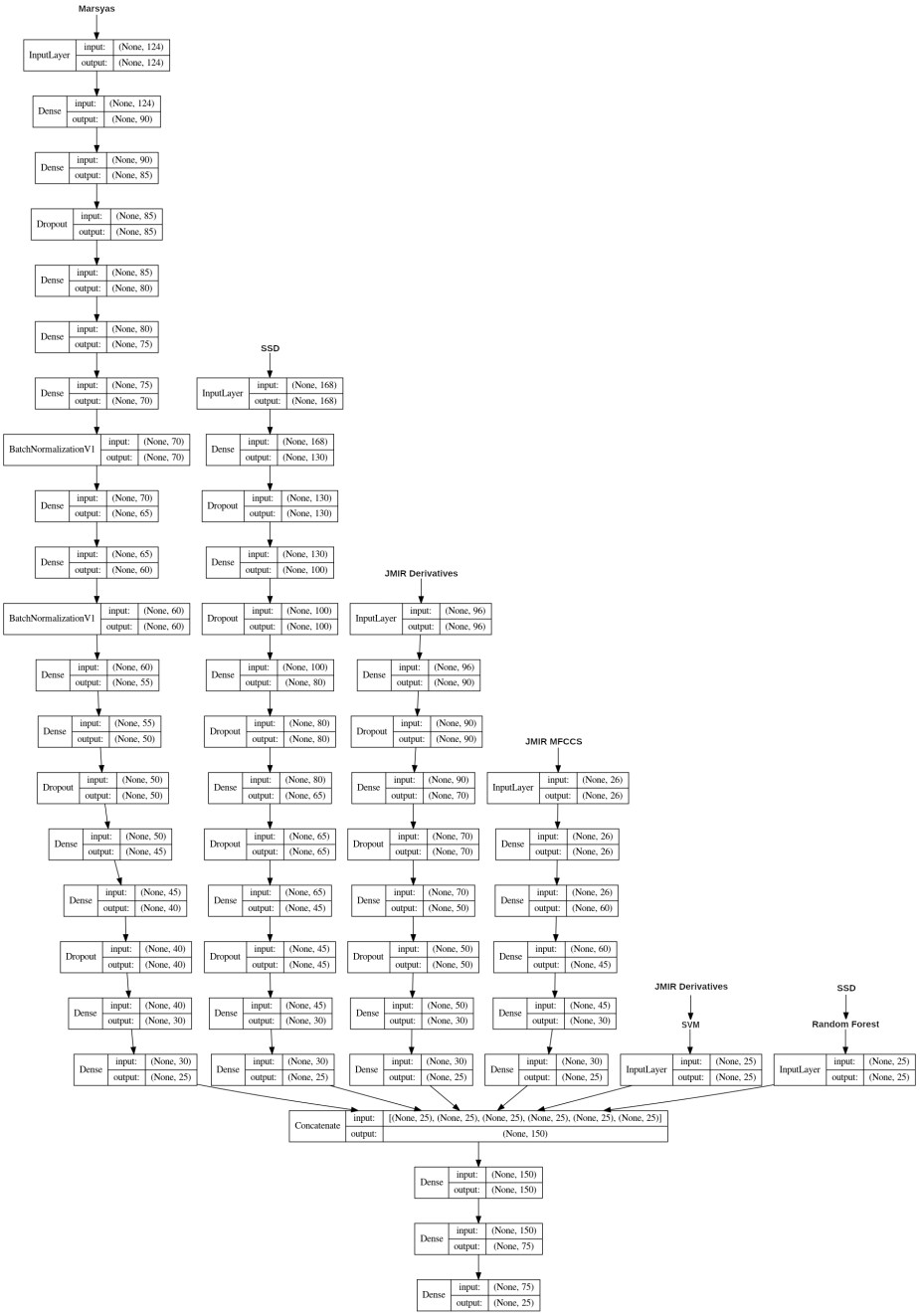
strategie_combinaison

Figure 5.1. Structure de l'ensemble de modèles Cette stratégie de combinaison ressemble beaucoup au vote pondéré. À vrai dire, le méta-modèle est responsable de définir une fonction de vote et d'assigner les poids aux modèles de base. De plus, la stratégie *Stacking ensemble* peut utiliser une frontière de décision non-linéaire. C'est la raison pourquoi la stratégie de vote tel qu'implémenté par la librairie scikit-learn n'a pas été choisie. Parallèlement, contrairement à la stratégie *Bagging* cette stratégie permet de combiner des modèles de base de différents algorithmes. De ce fait, cette stratégie supporte une plus grande diversité. Enfin, les stratégies de *Boosting* semblent plus utilisées pour la combinaison d'apprenants faibles. Cela va à l'encontre des algorithmes d'apprentissage choisis pour ce travail.

Ainsi, le système final repose sur l'utilisation de quatre ensembles de primitives, soit les trois présenté à la section 2.2 en plus de l'ensemble JMIR MFCCS. Selon l'expérimentation décrite à la section 2.2, cette ensemble est le quatrième plus performant. De plus, suite à la conception des trois modèles de base, trois autres modèles ont été ajoutés au système final. Ces modèles supplémentaires comportent un MLP sur l'ensemble SSD, un MLP sur JMIR Derivatives et un

autre MLP sur JMIR MFCCS. Ces modèles ont été ajoutés pour accroître la diversité. Plus de détails sur ces trois modèles additionnels seront présentés à la section 6. D'ailleurs, le méta-modèle de ce système est aussi un MLP.

Le choix d'utiliser un MLP pour méta-modèle revient en partie au fait que cet algorithme est non-linéaire, mais surtout parce qu'il est convivial. À vrai dire, les MLPs sont des apprenants fort, relativement facile à optimiser, et rapide à entraîner. SVM et Random Forest ont aussi été considérés comme algorithme pour le méta-modèle. Seulement, les SVMs sont véritablement pénibles à entraîner sur de très grands ensembles de données dus au temps d'entraînement. De plus, les membres de l'équipe ont plus d'expérience dans l'entraînement de réseaux de neurones que de SVMs ou de Random Forests. Enfin, utiliser un MLP comme méta-modèle comporte aussi des avantages au niveau de l'implémentation. Entre autres, la librairie utilisée, soit `tensorflow.keras`, permet de combiner des modèles. Ainsi, la couche de sortie des MLPs de base sont utilisées comme couches d'entrées du MLP méta-modèle. Par le fait même, cette librairie permet le partage des poids entre les MLPs des couches inférieures vers ceux des couches supérieures de l'ensemble. La figure 5.2 présente en détaille la structure des couches de l'ensemble.



ensemble

Figure 5.2. Structure détaillée de l'ensemble de modèles Ci-dessous est présentée une version simplifiée du code clé pour combiner les modèles de base au méta-modèle de même que pour effectuer une prédiction.

```
# Chargement des MLPs de base.
mlp_marsyas_model = load_model(mlp_marsyas_file_path)
mlp_ssd_model = load_model(mlp_ssd_file_path)
mlp_derivs_model = load_model(mlp_derivs_file_path)
mlp_mfccs_model = load_model(mlp_mfccs_file_path)
```

```
mlp_marsyas_model.trainable = False
mlp_ssd_model.trainable = False
mlp_derivs_model.trainable = False
mlp_mfccs_model.trainable = False

# Chargement des modèles sklearn
svm_derivs_model = pickle.load(open(svm_derivs_file_path, 'rb'))
rf_ssd_model = pickle.load(open(rf_ssd_file_path, 'rb'))

# Couches d'entrées reçevant les prédictions des modèles sklearn.
aux_input_svm = Input(shape=(25,), name='aux_input_0')
aux_input_rf = Input(shape=(25,), name='aux_input_1')

# Liste des couches d'entrées de l'ensemble de modèles.
ensemble_inputs = [
    mlp_marsyas_model.input,
    mlp_ssd_model.input,
    mlp_derivs_model.input,
    mlp_mfccs_model.input,
    aux_input_svm,
    aux_input_rf
]

# Concatène la sortie des MLPs de base et des couches d'entrées des modèles sklearn.
meta_inputs = concatenate([
    mlp_marsyas_model.output,
    mlp_ssd_model.output,
    mlp_derivs_model.output,
    mlp_mfccs_model.output,
    aux_input_svm,
    aux_input_rf
])

# Couche d'entrée du méta-modèle
layer = Dense(150, activation='relu', name='meta_input')(meta_inputs)

layer = Dense(75, activation='relu', name='meta_1')(layer)

# Couche de sortie du méta-modèle
meta_outputs = Dense(25, activation='softmax', name='meta_output')(layer)

ensemble_model = Model(inputs=ensemble_inputs, outputs=meta_outputs)

ensemble_model.compile(
    optimizer=Adam(),
    loss='categorical_crossentropy',
)
```

```
# Pour effectuer une prédiction
prediction = ensemble_model.predict([
    marsyas_features,
    ssd_features,
    derivs_features,
    mfccs_features,
    svm_derivs_model.predict_proba(derivs_features),
    rf_ssd_model.predict_proba(ssd_features)
])
```

C'est important de noter que les modèles de la librairie `sklearn` ne peuvent pas être combinés implicitement aux modèles de `tensorflow.keras`. À vrai dire, les prédictions effectuées par les modèles `sklearn` doivent être données en paramètre au modèle de l'ensemble aux côtés des primitives destinées aux MLPs de base.

7 6. Hyperparamètres finaux des modèles et résultats finaux

Le tableau 6.1 présente les hyperparamètres des modèles finaux. Notez que les hyperparamètres des modèles de base sont les même que ceux présentés dans la section 4. Pour la structure des couches des MLPs, voir la figure 5.2.

Modèle	Ensemble de primitives	Hyperparamètre	Valeur
Random Forest	SSD	n_estimator	80
"	"	max_depth	80
"	"	min_samples_split	4
"	"	min_sample_leaf	1
"	"	max_features	auto
"	"	bootstrap	False
SVM	JMIR Derivatives	kernel	rbf
"	"	C	10
"	"	gamma	0.001
MLP	Tous	optimizer	Adam
"	"	taux d'apprentissage	par défaut (0.001)
"	"	fonction loss	categorical crossentropy
"	"	fonction d'activation (couche de sortie)	softmax
"	"	fonction d'activation (autres couches)	relu
"	"	Batch size	2000
MLP	Marsyas	taux de dropout	0.10
"	"	epochs	110
MLP	SSD	taux de dropout	0.025
"	"	epochs	50
MLP	JMIR Derivatives	taux de dropout	0.025
"	"	epochs	150
MLP	JMIR MFCCS	epochs	150

Modèle	Ensemble de primitives	Hyperparamètre	Valeur
MLP	Méta-modèle	epochs	100

Tableau 6.1 Hyperparamètres des modèles finaux Le score du système final sur le site de kaggle au moment de la rédaction est de 0.36167.

Le tableau 6.2 présente les résultats finaux de l'ensemble lors de la validation croisée.

Tableau 6.2 Résultats finaux de l'ensemble

[4]:
```
ensemble_results_path = os.path.join(constants.LOGS_PATH, 'rapport',␣
 ↪'ensemble_final_run_results.json')
ensemble_results = read_json_ordered_dict(ensemble_results_path)
ensemble_df = pd.DataFrame([ensemble_results]).transpose().
 ↪reindex(['avg_accuracy', 'avg_val_accuracy', 'avg_batch_val_accuracy',␣
 ↪'avg_loss', 'avg_val_loss', 'avg_val_f1_macro', 'avg_val_f1_micro',␣
 ↪'train_time'])

display(ensemble_df)
```

```
                              0
avg_accuracy           0.377837
avg_val_accuracy       0.360558
avg_batch_val_accuracy 0.364986
avg_loss               2.014422
avg_val_loss           2.084618
avg_val_f1_macro       0.360648
avg_val_f1_micro       0.360558
train_time          2743.034329
```

Les résultats du système final sont plutôt bons. L'accuracy de validation de 36.06% est supérieur à celle de chacun de modèles de base. De plus, les scores F1 macro et micro sont respectivement de 36.06% et 36.006%. De façon générale, lorsque le score F1 macro est inférieur au score F1 micro, cela démontre l'effet négatif du débalancement des données sur la performance du modèle. Cette situation est le cas pour les modèles de base. Or, pour le système final les deux scores F1 sont presque qu'égaux. Cela peut être interprété comme quoi le système final est plus robuste au débalancement de donnée qu'un seul des modèles de base. Cependant, cela ne serait pas prudent d'affirmer que ce débalancement n'a pas d'effet négatif sur la performance du système final. Cette robustesse au débalancement est sûrement une conséquence de la diversité des algorithmes d'apprentissage de même que des ensembles de données.

Une première difficulté a été rencontrée lors de l'ajout du Random Forest à l'ensemble de modèles. Initialement, l'ensemble de modèles contenait seulement des réseaux de neurones. À ce moment, le fait d'entraîner le méta-modèle sur des données déjà vues par les réseaux de neurones causait seulement un léger surapprentissage. Cependant, lorsque le modèle de base du Random Forest a été ajouté à la combinaison de modèles, un très grand surapprentissage a été observé. Cela est dû au fait que le Random Forest semble mémoriser les exemples d'apprentissage, même s'il n'est pas en état de surapprentissage. Bref, le méta-modèle a dû être entraîné sur des données

qui n'ont jamais été vues par les modèles de base. Pour ce faire, l'ensemble de données original a été divisé en les sous-ensembles *base* et *meta* tel qu'expliqué à la section 2.5. Par la suite, chacun des modèles de base a été entraîné à nouveau sur cette nouvelle division des données. Suite au réentraînement, la combinaison de modèles comprenant les MLPs et le Random Forest avait une performance inférieure à celle de la combinaison de modèles comprenant seulement les MLP. Cela est dû au fait que les modèles de base de même que le méta-modèle disposaient de moins de données. Une possible solution à ce problème est énoncée dans la section 7.1. La performance du système a été accrue avec l'ajout du SVM et du MLP sur l'ensemble JMIR MFCCS.

Par la suite, une seconde difficulté fut le temps d'apprentissage démesuré du SVM. Une amélioration est proposée dans la section 7.4 pour réduire le temps d'apprentissage du SVM.

8 7. Pistes d'améliorations

8.1 7.1. Méthode de validation croisée imbriquée

Les membres de l'équipe auraient préféré utiliser une autre méthode pour la validation du méta-modèle. Tel qu'expliqué à la section 6, l'ensemble de données a dû être séparé en deux sous-ensemble; respectivement pour l'entraînement des modèles de base et pour l'entraînement du méta-modèle. Par conséquent, les modèles de base et le méta-modèle ne sont pas entraînées sur la totalité des données disponibles. Une solutions à ce problème est d'utiliser une méthode de validation croisée imbriquée.

8.2 7.2. Ajouter plus de modèles de base

Suite aux résultats obtenus avec la combinaison de modèles, ajouter davantage de modèles de base sur des ensembles de primitives différents permettrait d'améliorer la performance du système final.

8.3 7.3. Balancer les données

Balancer les données pourrait aussi améliorer la performance du système. Dans les faits, l'ensemble de données utilisé pour ce travail est débalancé. Ainsi, le tableau 7.3.1 montre que certaine classe ont une proportion plus forte alors d'autres plus faible. La librairie imbalanced-learn a été testée pour faire de l'oversampling à l'aide de la méthode SMOTE. Cependant, cette expérimentation a seulement permis d'accroitre l'accuracy de validation de 0.5 %. Face à cette amélioration négligeable, l'utilisation de cette librairie a été mise de côté. Or, avec du recul ne pas utiliser cette librairie était une erreur. 0.5 % d'accuracy de plus est une raison valable d'entraîner à nouveau les modèles.

Classe	nb d'observation	Proportion
POP_INDIE	11858	6.60%
ROCK_COLLEGE	10856	6.05%
ROCK_CONTEMPORARY	10829	6.03%
HIP_HOP_RAP	10581	5.89%
DANCE	9885	5.51%
METAL_ALTERNATIVE	9195	5.12%

Classe	nb d'observation	Proportion
POP_CONTEMPORARY	8959	4.99%
ROCK_HARD	8720	4.86%
ROCK_ALTERNATIVE	8333	4.64%
EXPERIMENTAL	7932	4.42%
COUNTRY_TRADITIONAL	7316	4.07%
ROCK_NEO_PSYCHEDELIA	7266	4.05%
ELECTRONICA	7148	3.98%
METAL_HEAVY	7031	3.92%
JAZZ_CLASSIC	6568	3.66%
METAL_DEATH	6485	3.61%
FOLK_INTERNATIONAL	6465	3.60%
PUNK	6306	3.51%
POP_LATIN	5048	2.81%
GOSPEL	4580	2.55%
BLUES_CONTEMPORARY	4511	2.51%
RNB_SOUL	4107	2.29%
GRUNGE_EMO	4096	2.28%
REGGAE	3433	1.91%
BIG_BAND	2047	1.14%

Tableau 7.3.1. Répartition des données en classes

8.4 7.4. Ensemble de SVMs

Le temps d'apprentissage du SVM a été d'une longueur pénible. Les causes de ce temps d'apprentissage autres que l'algorithme sont la taille de l'ensemble de données et le fait que la librairie LibSVM n'utilise pas le multithreading. Ainsi, le temps d'entraînement du SVM de même que ça performance de classification peut être améliorée en constituant un ensemble de SVMs. Chaque SVM doit être entraîné sur un sous-ensemble de l'ensemble de primitives originalement assigné au SVM. Ainsi, 8 SVM pourraient être entrainés en parallèle sur des thread différents.

9 8. Conclusion

La conception d'un système intelligent qui a la capacité d'automatiser la classification de façon optimale ne va pas sans plan stratégique. Dans le cas présent, le problème de classification des pièces musicales en fonction du genre fut adressé en évaluent les ensembles de primitives, en investiguant les forces et faiblesse des algorithmes face au problème, en analysant le type de prétraitement et technique de réduction de dimensionnalité à employer, en établissant des méthodes de validation, en optimisant les hyperparamètres des modèles et en combinant les classificateurs dans le but de maximiser la performance du système. Le travail effectué sur la conception du système intelligent a été réalisé entre autres grâce aux outils tels que Jupyter Notebook, Anaconda, Tensor Flow et Tensor Board sous le langage de programmation Python.

À la suite de ce travail, des résultats très intéressants ont été obtenus. Entre autres, le système final a atteint une accuracy de validation de 36.06 %, soit environ 7 % de plus que le meilleur modèle de base. Par ailleurs, le score F1 macro de ce système a atteint 36.06 %. Par-dessus tout,

au moment de la rédaction, soit 4h30 avant la date de remise, ce système intelligent se classe en deuxième position dans la compétition kaggle avec un score de 0.36167. Tout compte fait, ces résultats démontrent l'importance des techniques de combinaison de modèles, plus particulièrement la technique de *Stacked ensemble*.

Enfin, il faut mentionner que le problème de classification de pièces musicales n'est pas nécessairement un nouveau concept dans le monde de l'apprentissage machine. En effet, des entreprises comme Spotify qui offre à ses clients des listes de lectures personnalisées ou comme l'entreprise Shazam qui fait de la reconnaissance audio en temps réel grâce à des techniques comme celle de la transformée de fourier. Bien entendu, le problème auquel Shazam fait face n'est pas identique à celui qui a été abordé dans ce rapport, mais l'idée de déceler des variations pour créer une opportunité d'association en très présent. Cela dit, il devient encore plus intéressant d'aborder le sujet sachant que l'évolution des marchés de distribution de musiques en ligne est en constante évolution. Seulement, certains problèmes persistent tels que l'accès aux enregistrements même pour des raisons de droit. Par conséquent, le problème de classification automatique ne sera pas résolu de si tôt.

10 Fonctions utilitaires

[2]:
```
from collections import OrderedDict
from IPython.display import display, Markdown
import json
import os

import numpy as np
import pandas as pd

import src.constants as constants
```

[3]:
```
def read_json_ordered_dict(path):

    with open(path, 'r') as file:
        content = json.load(file, object_pairs_hook=OrderedDict)

    return content
```