SDATA PROJECT : Supervised classification, regression and data analysis

By Louis Bahrman and Solal Bizeul

Exercice 1

Dans ce premier exercice, nous travaillons avec un dataset provenant de l'API de Spotify, contenant 31278 chansons. Ces chansons peuvent être classifiées en 15 genres de musique différents : [Dark Trap, Underground Rap, Trap Metal, Emo, Rap, RnB, Pop, Hiphop, techhouse, techno, trance, psytrance, trap, dnb, hardstyle]

1. Data analysis

Tout d'abord, nous commençons par importer toutes les librairies de classification et de visualisation de data dont nous pourrions avoir besoin. Puis, nous importons les datasets en provenance de github et nous procédons au prétraitement des données. En observant le head du train_dataset, nous remarquons immédiatement trois choses : la présence de valeurs nulles (NaN) dans la colonne "song_name", le fait que les colonnes "id", "uri", "track_href", et analysis_url" ne semblent pas donner d'information utilisables pour prédire le genre d'une chanson, et le fait que la colonne "type" ne semble contenir que la valeur "audio_features".

Valeurs nulles

En effectuant une somme sur les colonnes du nombre de valeurs nulles, on peut remarquer que la colonne "song_name" contient 15625 valeurs nulles, ce qui correspond environ à la moitié du dataset. Normalement, pour remplir les valeurs manquantes, on peut essayer de les remplacer par la moyenne ou le mode sur la colonne. Ici, puisqu'il s'agit de noms de chansons, c'est impossible de faire ce genre de procédé. De plus, il serait également malvenu d'effacer les lignes qui contiennent une valeur nulle, puisque le nombre de ces lignes s'élèvent à 15000. La seule solution restante est d'effacer la colonne "song_name". Nous remarquons également qu'aucune autre colonne ne contient des valeurs nulles, ce qui conclut donc notre pré-traitement des valeurs nulles.

Valeurs n’apportant pas d’information

En regardant le nombre de valeurs uniques que prend la colonne “type”, nous concluons que cette colonne contient pour toutes les lignes la même valeur : “audio_features”. Cette colonne ne donne pas d’informations utiles : nous avons donc choisi de la supprimer du dataset. De même, les colonnes “id”, “uri”, “track_href” et “analysis_url” ne contiennent que des références à un morceau particulier sous la forme d’un code alphanumérique choisi au hasard. Ils sont tous équivalents, sachant que “uri” = spotify:track:“id”, “track_href” = https://api.spotify.com/v1/tracks/“id” et “analysis_url” = “https://api.spotify.com/v1/audio-analysis/“id". Puisque ces colonnes sont équivalentes et les codes choisis au hasard, ces données ne nous apportent rien : nous avons donc choisi de supprimer ces quatre colonnes.

Valeurs aberrantes

Nous avons comparé l’amplitude des valeurs données dans la page 1 du sujet par rapport aux statistiques obtenus par le train.describe(), et en avons conclu qu’aucune des valeurs ne nous semblaient aberrantes. Cette étape de pré-traitement peut donc être ignorée.

Data Visualisation

Pour visualiser les données avec lesquelles nous allions travailler, nous avons choisi tout d’abord d’utiliser des histogrammes afin de représenter la répartition de chacune des features. Nous avons remarqué que la variable time_signature avait pour valeur 4 en très grande majorité : il fallait donc vérifier si les chansons ayant un time_signature différent appartenaient toutes au même genre. Ceci n’était pas le cas. A partir des histogrammes, nous avons également constaté que speechiness, acousticness et instrumentalness semblaient avoir une grande majorité de leurs valeurs entre 0 et 0.1. Il sera donc peut-être difficile de tirer des informations utiles de ces variables. Nous avons ensuite fait un un seaborn pairplot coloré selon les genres afin voir les corrélations entre les variables. Le nombre de genres étant assez grand (15) et le nombre de points étant très grand (30000), le pairplot est assez difficile à interpréter pour la plupart des variables. Malgré cela, quelques caractéristiques peuvent en être décelées, comme le bas tempo des chansons de rap, par exemple. Pour quantifier plus précisément ces corrélations, nous avons choisi d’utiliser la fonction df.corr() pour avoir les chiffres exactes, et la fonction heatmap pour les visualiser. La corrélation la plus grande est celle entre “instrumentalness” et “duration_ms” qui est de 0.6. Les corrélations sont également difficiles à interpréter car nous ne savons pas ce que signifie une “speechiness” haute ou basse par exemple: il est donc compliqué de savoir si sa corrélation négative de -0.3 avec la durée de la chanson semble logique ou non.

2. Classification

Prétraitement

Nous avons tout d'abord séparé le dataset train en X et y afin de pouvoir entraîner les classifiers. De plus, il fallait utiliser un Label Encoder sur la colonne genres pour transformer les noms en nombres de 0 à 14. Nous avons ensuite normalisé toutes les valeurs du dataset train et du dataset test grâce à StandardScaler.

Nous avons enfin mis en place un Stratified KFold qui sera utilisé dans toutes les fonctions de classification. Le but du KFold est de d'assurer que le modèle fitté n'est pas undertrain ou overtrain : si les valeurs obtenus par le KFold sont stables, cela signifie que l'algorithme est satisfaisant et que les résultats que l'on obtiendra en testant ce classifier sur le dataset de test seront similaires aux valeurs de KFold. Si, au contraire, les valeurs obtenues par le KFold sont extrêmes, c'est-à-dire que l'amplitude des valeurs obtenues par le KFold est très grande, cela ne présage rien de bon pour l'utilisation de l'algorithme sur le dataset de test. Les classifiers du KFold obtenant un petit score sont undertrain, tandis que ceux obtenant un grand score sont overtrain : cet algorithme n'est donc pas fiable pour prédire les genres des chansons.

Nous avons choisi d'utiliser un Stratified KFold plutôt qu'un KFold classique. En effet, nous avons remarqué que certains genres étaient très peu représentés comparé à d'autres dans le dataset train. Par exemple, il n'y a que 336 chansons de pop tandis qu'il y a 4378 chansons de Underground Rap. En utilisant un KFold classique, nous aurions pu être confrontés à un cas où toutes les chansons de pop étaient dans le test du KFold, et il était donc impossible pour l'algorithme de correctement les classer puisqu'il n'a pas pu train avec ce genre. Le Stratified KFold résout ce problème en s'assurant que la distribution des genres dans le dataset est préservée dans les KFolds. Par exemple, la pop représente environ 1% du dataset : avec un Stratified KFold, ce genre représente également 1% du train et 1% du test. Ainsi, les folds obtenus sont représentatifs du dataset dans son ensemble. Nous avons choisi d'utiliser random_state pour pouvoir reproduire nos résultats, et shuffle pour mélanger les données prises par le KFold. Nous avons décidé d'utiliser 10 Folds car cela nous donne suffisamment de folds pour vérifier la stabilité de l'algorithme, mais suffisamment peu pour éviter des folds overtrainés et d'autres undertrainés. En effet, nous avons essayé d'utiliser 100 folds avec le Decision Tree Regressor et nous obtenions des F1 scores qui variaient entre 0.3 et 0.9, exemplifiant le phénomène d'overfitting et d'underfitting.

Fonction générale de classification

Cette fonction sera utilisée pour chaque algorithme de classification testé, afin de faciliter leur mise en place. Elle prend en paramètre un classifier, et renvoie 4 variables : le Score F1 Micro

pour chacun des 10 folds, la moyenne de ces scores, le score le F1 Score du dataset de test et, si nous voulons l'utiliser pour des analyses plus poussées, l'array contenant les prédictions pour les données de test. Cette fonction nous permet d'avoir une vue d'ensemble sur l'efficacité de prédiction d'un algorithme en quelques lignes.

Modèles de classification

Pour cet exercice, nous avons choisi de tester un grand nombre d'algorithmes différents afin de pouvoir prédire les genres de la meilleure manière. Cela nous permettait également de voir le comportement et l'efficacité avec le type de données utilisées, c'est-à-dire un set de données plutôt petit, avec peu de corrélations marquées entre les variables. Sachant que nous avons utilisé 15 algorithmes différents pour estimer les genres, nous n'allons pas décrire avec précision les résultats et tous les paramètres utilisés pour chaque méthode. Cependant, il semble important de souligner certains aspects généraux de notre étude. Tout d'abord, nous avons utilisé à chaque fois random_state = 0 afin de s'assurer que nos résultats soient reproductibles. Pour certains algorithmes, comme le LinearSupportVectorMachine, le nombre d'itérations n'était pas assez grand pour que le modèle converge correctement : nous avons donc augmenté max_iter à 10000, ce qui était suffisant.

Linear Models

Nous avons commencé par tester des modèles linéaires : la Logistic Regression et la Stochastic Gradient Descent.

Logistic Regression est un classifier simple essayant de maximiser l'entropie du modèle. Pour cet algorithme, le paramètre le plus intéressant était le 'solver' qui change l'algorithme appliqué pour l'optimisation. En testant tous les algorithmes disponibles, nous n'avons pas remarqué de grande différence de résultat entre les algorithmes. Tous les résultats étaient stables et ne variaient peu, voire étaient égaux pour les algorithmes "newton-cg", "sag" et "saga. Par contre, le temps d'exécution, lui, changeait considérablement, d'un maximum de 2 minutes à un minimum de 20 secondes. Le solver "sag" semblait alors optimal pour notre problème avec un f1 de 0.584 et un temps d'exécution de 21s.

Le Stochastic Gradient Descent est un algorithme qui met à jour le modèle en fonction du gradient du loss. A nouveau, le paramètre le plus important est "loss", qui donne la fonction de loss utilisée. En testant toutes les fonctions adaptées à la classification, nous pouvons remarquer que les performances fluctuent : la fonction 'log' est stable et a un score satisfaisant de 0.54 en moyenne, tandis que 'perceptron' est instable et n'a qu'un score de 0.33. La fonction 'log' semble optimale ici avec un temps d'exécution acceptable de 12 secondes.

Discriminant Analysis

Nous avons ensuite testé des modèles utilisant des surfaces linéaires et quadratiques : le Linear Discriminant Analysis et le Quadratic Discriminant Analysis.

Pour le Linear, le paramètre décisif est le “solver” : nous avons donc expérimenté avec les trois solvers proposés. A nouveau, les trois solvers étaient stables et donnaient tous un f1 score d’environ 0.53. Ce qui était impressionnant avec cette méthode était le temps d'exécution qui dépassait à peine 1 seconde.

Pour le Quadratic, il n’y a aucun paramètre qui semble primordial. Avec les paramètres de base, nous obtenons un bon score de 0.61 de moyenne avec des KFolds stables et un temps d'exécution de 1.3 secondes. C’est donc un algorithme efficace en termes de score-temps.

Naive Bayes

Les méthodes de Naive Bayes sont des modèles probabilistes basés sur le théorème de Bayes. Cet algorithme est naïf car il suppose que les features sont deux-à-deux indépendantes.

Nous avons testé le GaussianNB et le BernoulliNB : en effet, les autres algorithmes comme le ComplementNB ne fonctionnent pas avec des valeurs négatives.
Le GNB n’a pas de paramètres à changer, mais les paramètres automatiques nous donnent un score stable de 0.59 en un temps record de 300ms.

Le paramètre important pour le BNB semblait être la valeur alpha, mais pour des valeurs entre 0.1 et 100, elle a peu d’influence. Au-delà de 100, le score diminue progressivement. La valeur optimale semble être autour de 50. Avec ces paramètres, cela donne un score de 0.474 plutôt stable, avec un temps d'exécution de 700ms.

Decision Tree

Le DecisionTreeClassifier est un classifier classique qui utilise un arbre de décision. Le critère le plus important est le criterion : “gini” ou “entropy”. Les résultats avec ces deux options sont similaires : on obtient un score stable d’environ 0.56 en une dizaine de secondes.

Support Vector Machine

Les SVMs sont des méthodes de classification supervisée qui divisent l’espace des données en une série d’hyperplans pour maximiser la distance entre les séparatrices et les points. Nous

avons utilisé le Support Vector Classification classique, et la version linéaire LinearSVC. La classification utilisant le support Nu NuSVC ne fonctionnait pas car il était impossible de trouver une valeur de Nu plausible (les données ne se prêtaient pas à cette méthode).
Le paramètre fondamental pour le SVC est le kernel, qui correspond à la fonction utilisée par la méthode pour calculer les hyperplans. Nous avons pu tester tous les kernels, sauf le “precomputed” car ce kernel a besoin que la matrice de train soit carré, ce qui n’est pas notre cas. Peu importe le kernel, ces algorithmes sont stables. Le score des kernels “poly”, “linear” et “rbf” est environ de 0.62, tandis que celui de “sigmoid” est de 0.47. Le temps d'exécution, quant à lui, est plutôt grand par rapport aux autres méthodes testées pour l'instant (4 minutes).

Pour le LinearSVC, il semble intéressant au premier abord de changer le “loss” à “hinge” mais cela ne fait pas converger l’algorithme, même en utilisant un nombre d’itération max à 100k. Par contre, le paramètre dual est très utile. En effet, ce paramètre transforme automatiquement le problème de classification en problème dual, et travaille donc avec la transposée de la matrice de train. L’algorithme essaie d'entraîner 15 valeurs avec 31000 features, ce qui n’est absolument pas optimal. Ceci se voit dans les résultats : on obtient les mêmes résultats avec les deux méthodes (0.54), mais dual = True prend 14 minutes à s'exécuter tandis que dual = False prend 10 secondes. Il ne faut donc pas oublier de changer ce paramètre.

Nearest Neighbors

Le KNN Classifier est un algorithme classique de Nearest Neighbors Search. Pour cet algorithme, nous pouvons changer deux paramètres importants : le nombre de voisins considérés lors de l'exécution de l’algorithme (n_neighbors) et l’importance de chacun des voisins dans le calcul des prédictions. En faisant des tests, nous avons trouvé que la valeur optimale de voisins à considérer est de 30 environ. Le fait que les poids sont calculés de manière uniforme ou de façon à donner plus d’importance aux voisins les plus proches ne semblent pas changer énormément le score final. Nous obtenons un score final stable de 0.57 en 30 secondes avec 30 voisins.

Neural network

Nous avons ensuite essayé de classifier les musiques à l’aide d’un réseau neuronal. Le défi était d’obtenir la prédiction la plus exacte possible en évitant l’overfitting, ce qui a été notre plus grande contrainte dans le choix du modèle. En effet, un réseau neuronal comportant trop de couches, ou des couches comportant trop de neurones, comprendrait plus d’hyperparamètres que de données. Nous avons ainsi éliminé les ResNet, qui étaient trop propices à l’overfitting. De même, la largeur des hidden-layers est fixée à 64, meilleur compromis évitant l’overfitting. Nous avons sélectionné la fonction d’activation ‘relu’ qui offre des meilleures performances à réseau égal que les fonctions non linéaires ‘sigmoid’ ou ‘tanh’.

Nous n'avons pas utilisé les k-folds, car chaque entraînement risquait de converger vers un maximum local différent. Afin d'éviter l'overfitting, nous avons enfin ajouté des layers de Dropout entre chacune des 3 hidden layers, avec une rate égale à 0.1. Enfin, un softmax est appliqué afin d'obtenir la catégorie.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 64)                896
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_7 (Dense)              (None, 15)                975
=================================================================
Total params: 10,191
Trainable params: 10,191
Non-trainable params: 0
```

Nous choisissons un batch_size égal à 128 et 150 epochs.

Le temps nécessaire pour effectuer la compilation et le fit du modèle est de 71.2 secondes, sans k-folds et en utilisant le GPU disponible sur Google Collab.

Ensemble Methods

Ces méthodes d'ensemble sont les dernières que nous avons traitées, car elles combinent plusieurs algorithmes pour renforcer l'apprentissage. Nous avons tenté d'utiliser deux méthodes d'averaging (Bagging et RandomForest) et deux méthodes de boosting (AdaBoost et Gradient Tree Boosting). Les méthodes d'averaging prennent en compte plusieurs estimateurs entraînés indépendamment et en font la moyenne, tandis que les méthodes de boosting construisent les estimateurs l'un après l'autre afin de réduire le biais du modèle final. Ces méthodes d'ensemble prennent beaucoup de temps à exécuter, en particulier lorsqu'on augmente le nombre d'itérations maximum.

Le RandomForestClassifier entraîne un grand nombre d'arbres de décision tout en introduisant du hasard dans la façon de choisir les meilleures feuilles afin de réduire la haute variance qui est un problème dans le TreeClassifier classique. Nous avons testé les deux possibilités pour "criterion", ce qui n'a pas changé le résultat de manière significative. Nous avons également tenté d'augmenter le nombre d'estimators à 500, ce qui n'accroît que très peu le score, mais

multiplie par 5 le temps d'exécution. Le résultat final avec “gini” et 100 estimators donne un score stable de 0.65 environ en 2 minutes.

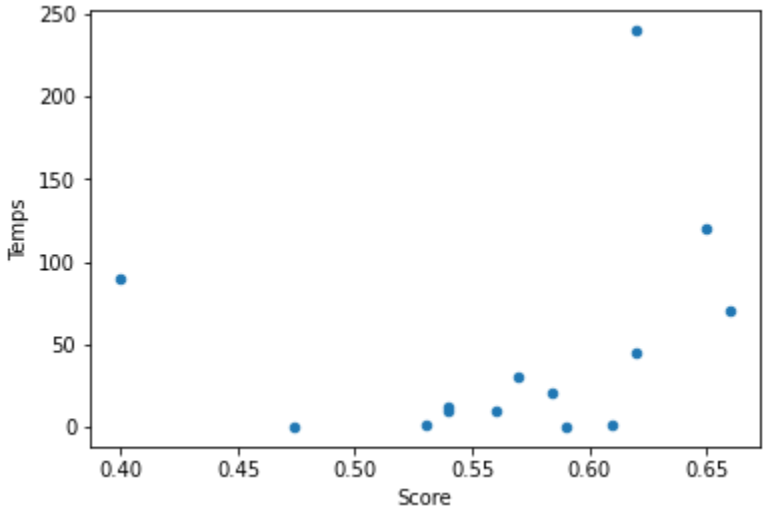
Le Bagging Classifier utilise l’algorithme du base_classifier donné en argument et l’applique à des sélections du dataset entier, puis agrége les résultats pour former un rendu final, dans le but de limiter la variance du base_classifier. Nous avons tenté d’utiliser l’algorithme qui a donné les meilleurs résultats jusqu’ici (hors du random_forest qui prend énormément de temps si utilisé comme paramètre pour le bagging), c’est-à-dire le SVC. Nous avons obtenu un score stable de 0.64 pour un temps d'exécution de 36 minutes, ce qui est très long. En utilisant le paramètre par défaut (Decision Tree), nous obtenons un score de 0.62 en un temps de 45 secondes, ce qui est beaucoup plus raisonnable.

Le AdaBoost nous permet de fit des modèles peu efficaces un grand nombre de fois sur des versions modifiées du dataset original en appliquant des poids différents à chaque point, dans le but que l'algorithme se concentre sur les points difficiles à prédire. Nous avons choisi de garder le base estimator d’après notre expérience avec le bagging. En jouant sur les paramètres learning_rate et n_estimators, nous n’arrivons pas à obtenir un score au-dessus de 0.4 (convergence en 90 secondes), ce qui montre que cette technique n’est pas adaptée à nos données.

Le Gradient Boosting optimise des fonctions de “loss” afin de fit des arbres de décision sur le gradient de ces fonctions. Cet algorithme nous donne le meilleur résultat pour l’instant, avec un score stable de 0.68 en 22 minutes, en utilisant les paramètres par défaut. Lorsqu’on augmente le learning rate ou le nombre d’estimators, l’algorithme prend plus longtemps à converger et donne en général des résultats plus mauvais.

Conclusion

En conclusion, nous avons produit un graphique qui résume toutes les méthodes utilisées.



Nous pouvons voir que deux algorithmes semblent optimaux : le neural network, représenté par le point le plus à droite, ainsi que le QDA qui a un score de 0.61 tout en ayant un temps d'exécution de 1.3 secondes. Nous n'avons pas ploté le Gradient Boosting car son temps d'exécution de 1320 secondes rendait tout le graphe illisible. En tant que data scientist, il faut savoir équilibrer la performance de l'algorithme et le temps d'exécution. Dans le cadre d'un véritable projet de data pour une entreprise, nous n'aurions pas forcément le temps ni la puissance de calcul pour fitter un Gradient Boosting sur des millions de données. Il faut donc apprendre à estimer à quel point nous pouvons réduire le temps de calcul, tout en gardant un score acceptable (notion à définir selon le scope du projet). Pour le challenge de classification, nous avons voulu envoyer la prédiction du réseau de neurones, malgré le fait que le Gradient Boosting ait un meilleur score. En effet, Mr.Itier nous avait prévenu qu'il serait difficile d'utiliser un réseau neuronal sur ce projet à cause de la petite taille des données. Pourtant, cette technique nous a renvoyé un score tout à fait acceptable, et nous souhaitions voir si le réseau que nous avions conçu pouvait prédire correctement les genres, plutôt que d'utiliser un Gradient Boosting par défaut.

Exercice 2

1. Prédiction de la popularité

Pré-traitement

Les données ne comportent aucune valeur nulle ou incohérente.
Seule la colonne genre semble inexploitable. Le nombre de genres distincts est en effet égal au nombre de lignes de données, ce qui signifie que le genre est différent pour chaque chanson. Pour l'instant, nous nous contenterons de supprimer cette colonne.

Visualisation basique

Selon les histogrammes, les features se séparent majoritairement en deux groupes: celles suivant une loi gaussienne et celles ayant un pic clair vers le haut ou le bas de l'axe. Plus précisément, la matrice de corrélation nous permet de savoir que la popularité est corrélée négativement avec l'acousticness, et positivement avec l'energy et la loudness.

Régression

Nous commençons par séparer la colonne popularity, puis nous normalisons les données avec Standard Scaler. Nous ne pouvons pas utiliser de Stratified KFold car c'est un problème de régression : nous choisissons donc d'utiliser un KFold classique, avec 5 folds afin d'éviter l'overtrain qui se fait naturellement sur ce problème. Nous allons utiliser le r2 afin de comparer les scores de chaque méthode. Finalement, nous définissons une fonction régression équivalente à la fonction classification définie dans l'exercice 1.

Sachant que les algorithmes suivants correspondent presque exactement à ceux exploités dans l'exercice 1, il n'est pas nécessaire d'expliquer à nouveau leur fonctionnement.

Régression Linéaire

Nous avons essayé de prédire la popularité des chansons grâce aux algorithmes classiques de régression linéaire : LinearRegression, Ridge, Lasso, Bayesian Ridge, Lasso Lars et Stochastic Gradient Descent. Toutes ces méthodes ont des caractéristiques similaires : scores assez bas d'environ 0.27 sur le KFold et de 0.33 sur le test, assez peu stables par KFold (variation de 0.24 à 0.3) et temps d'exécution très rapide d'environ 50ms. De plus, le paramètre important (alpha qui correspond au paramètre de régularisation) n'a pas beaucoup d'influence sur le résultat. Ces méthodes linéaires ne semblent pas adaptées à ces données.

Support Vector Machine

Les SVMs étaient relativement efficaces lors de l'exercice de classification, mais le résultat dépendait entre autres du kernel utilisé. Pour la régression, nous retrouvons la même chose. Selon le kernel, le score varie de 0.39 pour "rbf", à -1.05 pour "sigmoid" en passant par 0.23 pour les autres kernels ainsi que pour LinearSVR. Comme les modèles linéaires, ces algorithmes ne sont pas très stables par le KFold (les résultats varient de 0.16 à 0.3) mais ils s'exécutent rapidement (1s environ).

Nearest Neighbors

Pour le Nearest Neighbors, nous optimisons le nombre de voisins considérés lors de la prédiction grâce à n_neighbors, et nous prenons l'optimisation des poids des points en fonction de la distance. Avec ces paramètres optimaux, nous obtenons un résultat plutôt stable avec une moyenne de 0.5 en 100ms environ.

Classifier Tree

Les arbres de décision peuvent être utilisés également pour faire de la régression, même si ce n'est pas leur raison d'être principal. Cet algorithme n'est pas adapté à notre problème : nous obtenons des scores très bas, voire négatifs. De plus, le KFold n'est absolument pas stable, peu importe le criterion utilisé. Cette méthode ne sera pas incluse dans le récapitulatif pour ces raisons.

Ensemble Methods

Nous avons utilisé les mêmes méthodes que dans la classification : RandomForest, Bagging, AdaBoost, Gradient Boosting. Comparé au Tree, le RandomForest est plutôt efficace, avec une stabilité convenable et un r2 de 0.55 avec n_estimators = 1000 (temps d'exécution de 90 secondes). De même, le Bagging nous donne des bons résultats stables d'environ 0.55 avec 100 estimators. Les techniques de boosting donnent des résultats un peu moins bons, avec l'AdaBoost à 0.37 et le Gradient à 0.51. Ces méthodes étaient tout du moins relativement stables et rapides à exécuter.

NN

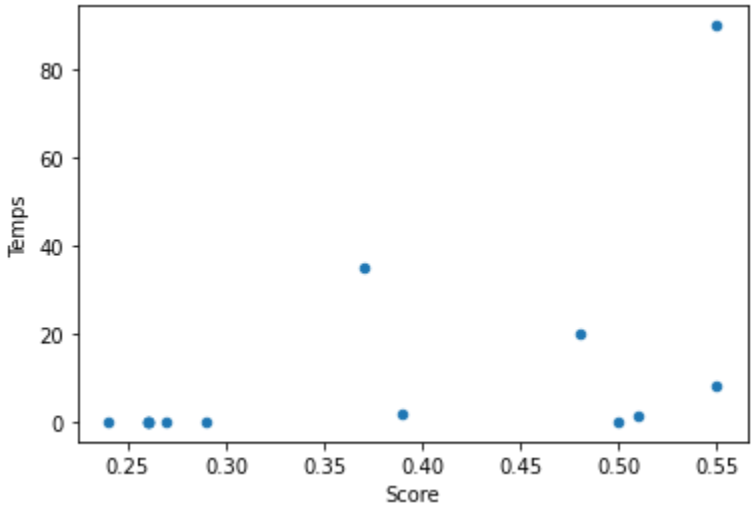
Nous utilisons un réseau neuronal afin d'effectuer la régression. Le nombre de lignes très faible de ce second dataset rend le modèle d'autant plus susceptible d'overfitting si il est trop complexe, c'est pourquoi il nous faut drastiquement limiter le nombre de paramètres entraînables. Nous choisissons la structure suivante :

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 32)                416
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_7 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,505
Trainable params: 1,505
Non-trainable params: 0
```

Il n'y a que 2 hidden layers de 32 neurones chacun. Le dropout rate est ici encore fixé à 0.1.

Le R2 est ainsi de 0.48, pour un temps de fitting de 20 secondes.

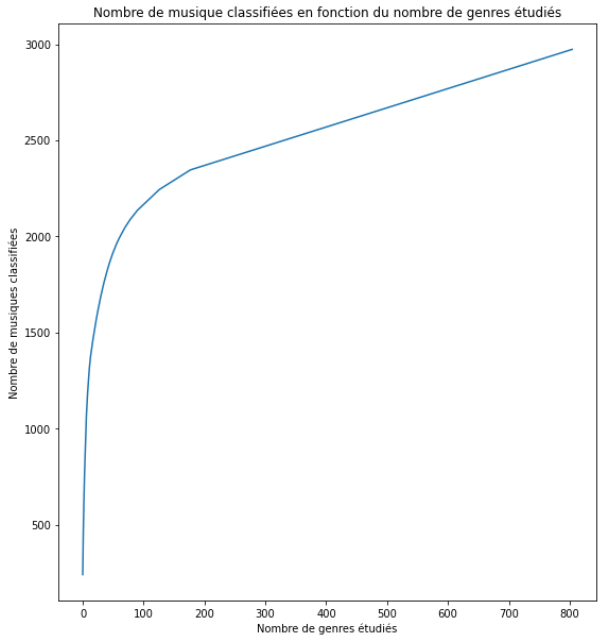
Récapitulatif

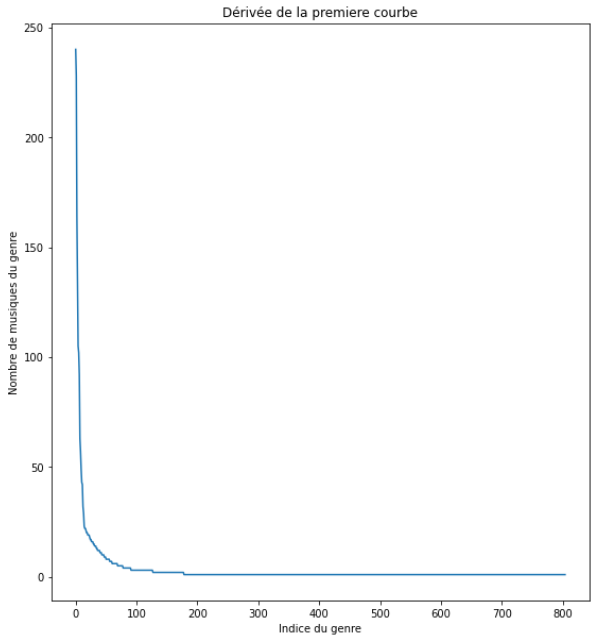


Nous avons produit un graphique ci-dessus représentant toutes les techniques utilisées. Nous pouvons remarquer que celles qui semblent le plus efficaces sont le Bagging, qui atteint un score de 0.55 en 8.2 secondes, et, pour une bonne estimation très rapide, le Gradient Boosting qui atteint 0.51 en 1.3 secondes. A nouveau, comme pour la classification, il faut trouver l'équilibre entre le score et le temps d'exécution, concept qui devient primordial lorsque le travail de data scientist se fait en entreprise en temps et puissance de calcul limités.

2. Traitement du Genre

La classe genre, que nous avions ignorée au début de l'exercice, peut néanmoins être exploitée. Nous avions remarqué qu'il n'y avait qu'un unique morceau par genre, et nous avons donc considéré que la classification induite par cette colonne n'apportait aucune information. Néanmoins, l'étude sémantique des genres peut nous être utile. En effet, la plupart des genres sont composés de plusieurs mots, un adjectif et un genre plus large. Ainsi le genre 'spanish pop' est un sous-genre de la 'pop' venant d'Espagne. On peut ainsi réduire le nombre de genres en catégorisant une musique dans le genre le plus large possible, c'est-à-dire celui comprenant le plus de musiques. On parvient ainsi à réduire le nombre de genres distincts de 2973 à 805.
En affichant le nombre de chansons classifiées en fonction du nombre de genres étudiés, on obtient une courbe de la forme suivante, affine par morceaux.





La dernière partie est de pente égale à 1, correspondant à l'ensemble des genres n'apparaissant que dans une seule musique. Nous choisissons donc de ne considérer que les genres représentant plus d'une musique, et de classer les autres dans un même genre.

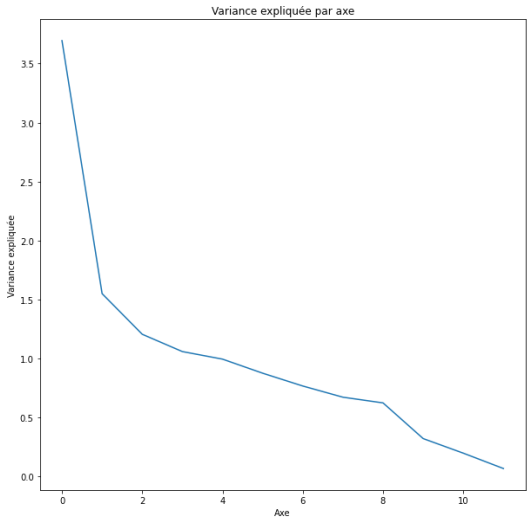
Néanmoins, cette méthode a des désavantages.
Tout d'abord, les genres représentés par un unique morceau restent très nombreux, et représentent plus d'⅓ des musiques.
Ensuite, elle favorise les genres les plus présents, au détriment de l'équilibre entre les genres. Par exemple, 40 genres comprennent le mot 'japanese'. Pour autant, le genre 'japanese pop' n'est pas classé dans 'japanese' mais dans 'pop', ce qui réduit la taille du genre 'japanese'. Le genre 'japanese' contiendra alors tous les sous-genres moins connus, et qui n'ont en commun

que l'appellation 'japanese' et non le véritable genre de la musique, qui pourra varier du 'rockabilly' au 'jazztronica'.

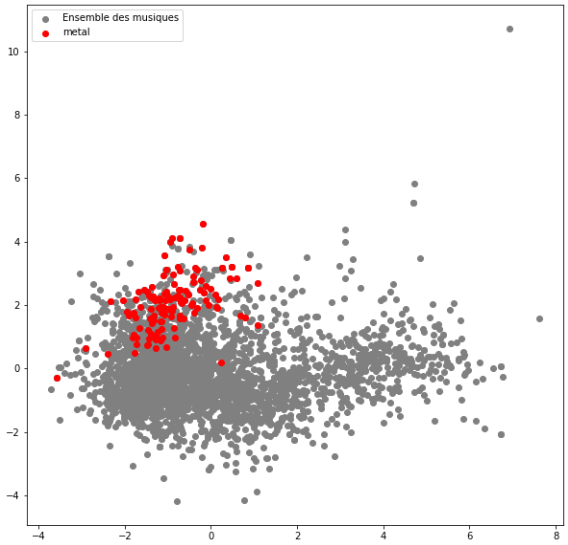
Nous avons appliqué cette nouvelle colonne genre à la prédiction de la popularité. En ajoutant une colonne "new_genre" au DataFrame original, nous allons pouvoir vérifier que notre travail a été profitable en comparant les scores obtenus sans colonne genre, et ceux obtenus avec la colonne "new_genre". Les résultats sont les suivants : nous obtenons un score de 0.53 en moyenne pour le Bagging, ce qui est plus petit que le score original de 0.55. Néanmoins, pour le réseau de neurone, grâce à la nouvelle colonne genre, le r2 augmente de 0.47 à 0.53.Nous pouvons donc en conclure que cette étape de changement de la colonne genre peut apporter des bénéfices en termes de performance selon les algorithmes. En affinant encore ce traitement, nous pourrions surement arriver à augmenter de manière significative le score.

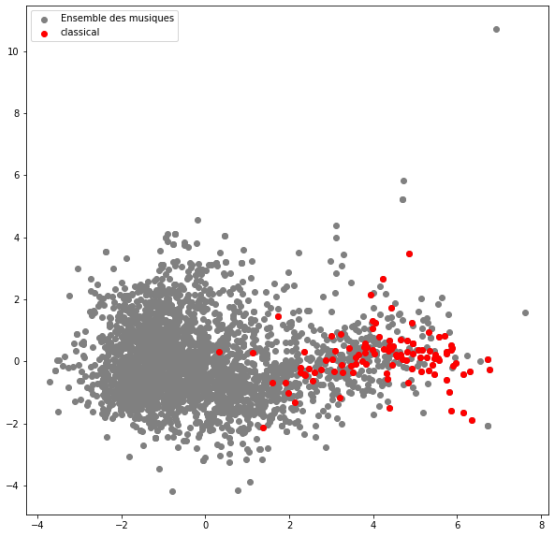
Visualisation

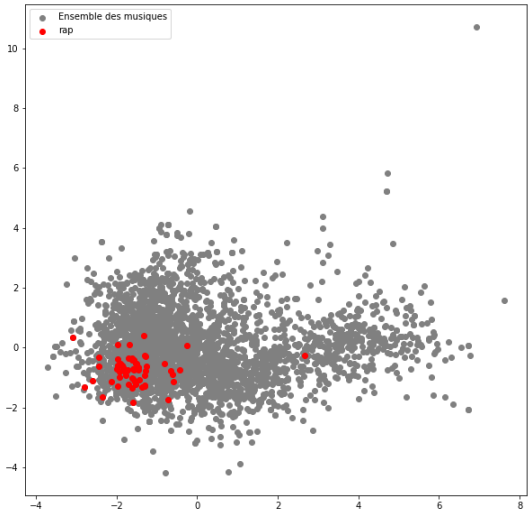
Afin de mieux visualiser les données, nous devons réduire leur dimension. Le graphe de la variance expliquée par axe nous indique que deux axes suffisent à conserver une grande inertie.

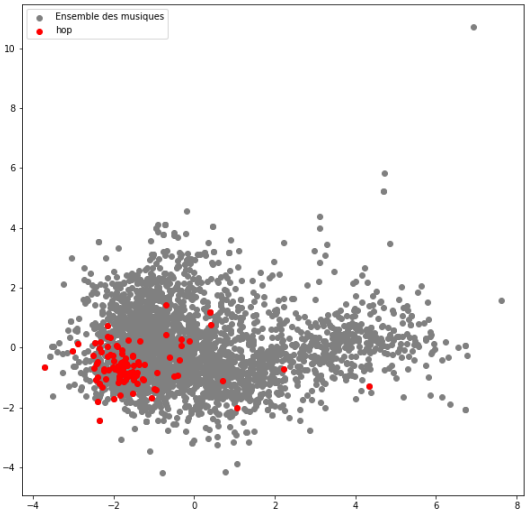


Cette projection permet de montrer la cohérence de notre algorithme de reclassement des genres, comme le montre le graphe suivant

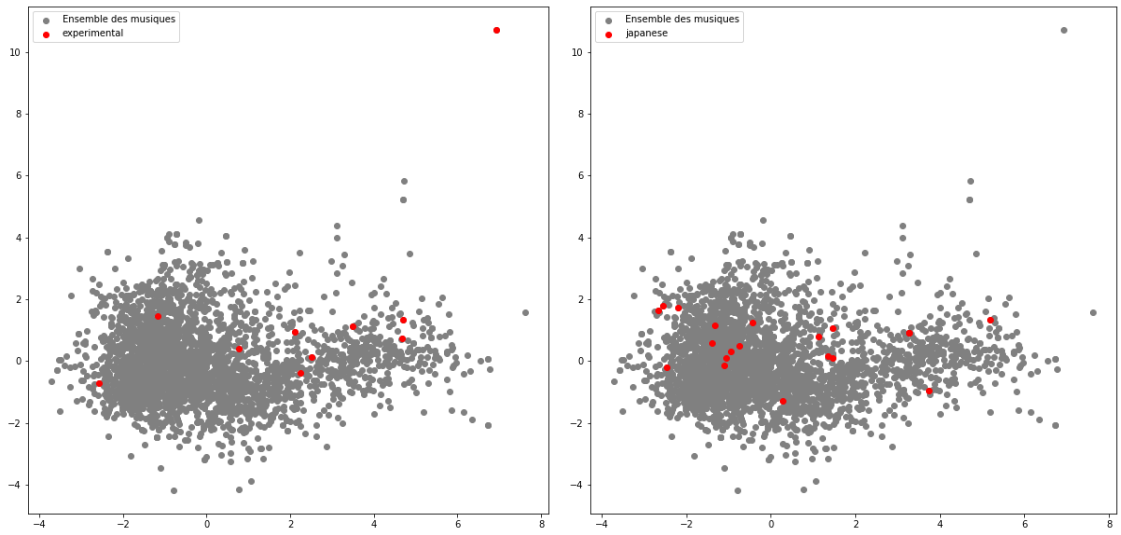






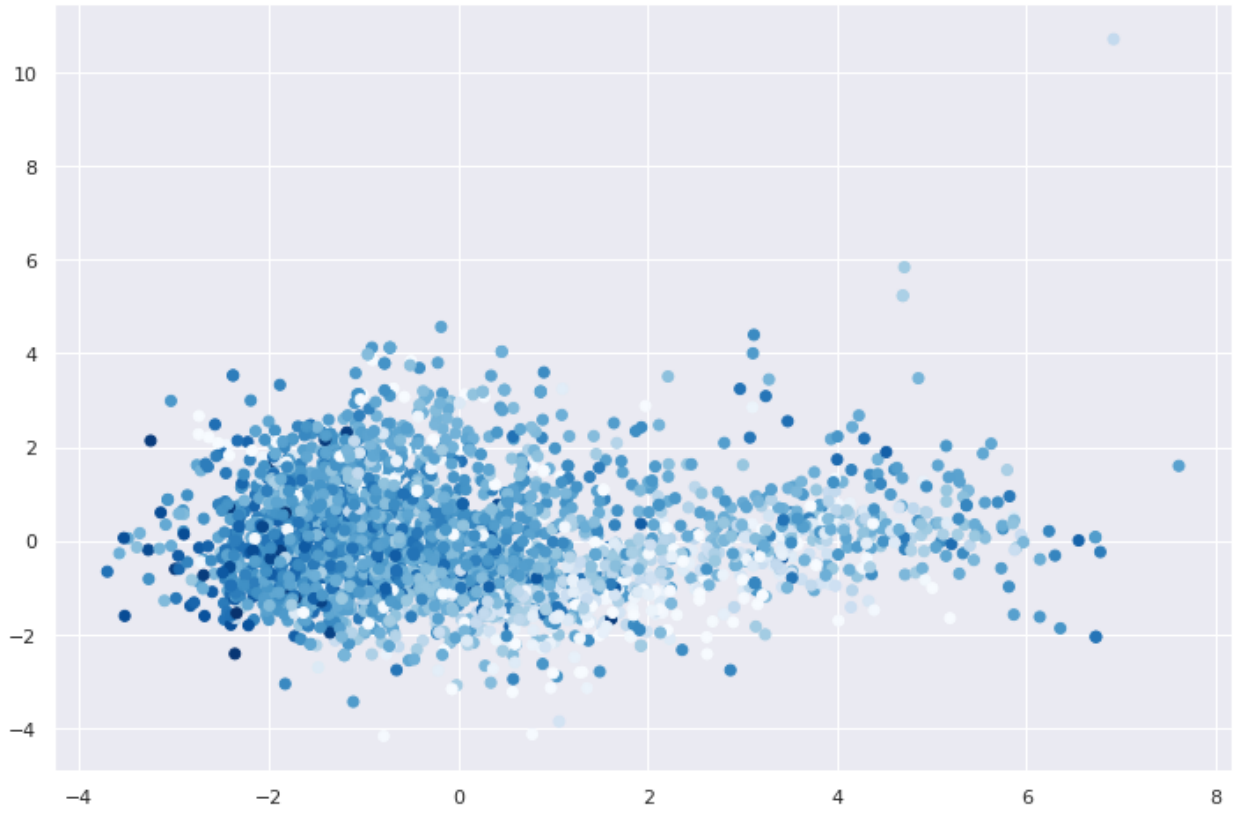


Certains genres créés artificiellement ne sont néanmoins pas assez représentatifs, comme ‘experimental’, ou ’Japanese’, ce qui montre les défauts énoncés plus haut.



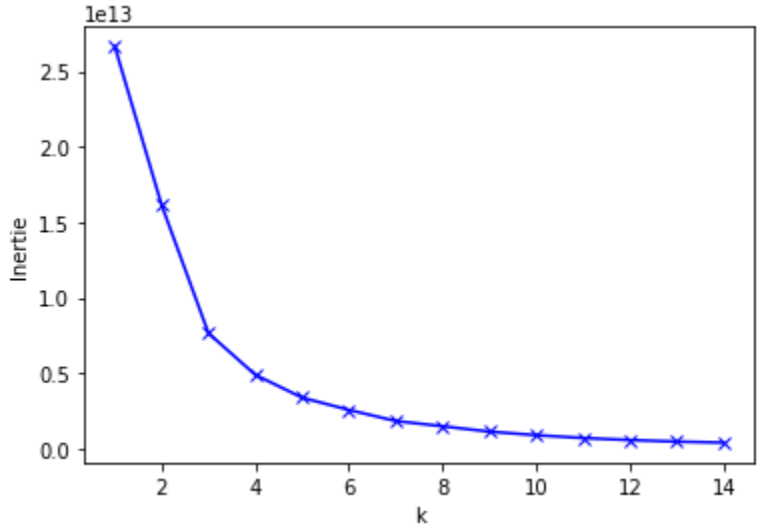
Visualisation de la popularité

Nous avons ensuite essayé de déterminer visuellement les musiques les plus populaires. Voici une projection des morceaux sur les axes obtenus par l'ACP. Plus la couleur des points est foncée, plus le morceau est populaire. On remarque que les chansons les plus populaires sont en bas à gauche, ce qui correspond à une forte dansabilité, une forte énergie et une faible instrumentalness.



Clustering

Un des traitements possibles sur ces données est le clustering. Il serait intéressant de voir si nous pouvons arriver à effectuer un clustering avec l'algorithme K-Means afin de représenter correctement les données. Nous avons commencé par calculer le K optimal à utiliser grâce à la métrique de l'inertie. Nous en avons conclu que le K optimal était 5. Nous avons ensuite fitté un algorithme de K-Means sur les données (sans la colonne genre), puis l'avons tracé en utilisant les axes trouvés par l'ACP.



Le résultat ci-dessus est peu concluant. Nous pouvons voir que la classe vert foncé qui est composée de quelques points est bien définie, et se sépare clairement des autres classes. Les autres classes sont moins bien séparés : la classe verte est centrée vers le bas et la jaune vers le haut. Il faudrait possiblement améliorer l'ACP pour obtenir des axes plus représentatifs des clusters obtenus.

Conclusion

Nous avons trouvé quelques idées qui auraient été intéressantes à mettre en place afin d'utiliser ce dataset rigoureusement. Tout d'abord, nous pourrions comparer les 5 clusters obtenus par le K-Means aux 5 genres nouvellement définis par notre étude des genres, afin de voir s' ils correspondent partiellement ou aucunement. Une autre idée serait de construire un système de recommandation de musique en utilisant un 1-NN Search. A partir d'une chanson

choisie par l'utilisateur, l'algorithme irait chercher son plus proche voisin. L'utilisateur pourrait également choisir un niveau d'aléatoire entre 1 et 10 qui augmenterait le "cercle" autour de la chanson originale, et l'algorithme choisirait une chanson au hasard dans ce cercle. Le cercle serait petit pour alea=1, mais il comprendrait toutes les chansons dans le dataset pour alea = 10. Nous pourrions aussi ajouter un paramètre de popularité pour que l'utilisateur puisse être recommandé une chanson connue ou obscure selon son envie. En conclusion, cet exercice était passionnant car il nous a permis d'exercer nos connaissances et nos intuitions de futurs data scientists.