Faculté des Sciences
Aix*Marseille Université

Compte rendu du mini projet
Prédiction de l'année d'une chanson

Saïda GUEZOUI
M2 MAS DS

Année universitaire
2021 / 2022

Encadrant : Guilhem BONAFOS

Mail universitaire : saida.guezoui@etu.univ-amu.fr

Remerciements

Je voudrais remercier Mr Guilhem pour son cours de Python très instructif et organisé, pour ce projet intéressant qui nous a permis de mieux comprendre et pratiquer les différents algorithmes d'apprentissage en Data Science, chose qu'on a pas eu l'occasion de faire précédemment.
Je remercie également Mr Pierre Pudlo pour ses efforts et son temps consacré pour nous encadrer.

Contents

1 Introduction et présentation générale

Il est aujourd'hui extrêmement simple d'accéder à une grande quantité de musiques. On peut alors appliquer des algorithmes basés sur l'analyse statistique des données pour extraire ou prédire des caractéristiques. Par exemple, est-il possible à partir des données d'une chanson de dater cette dernière ? Si oui, avec quelle précision ?

Dans ce projet, on s'intéresse justement à l'estimation de la date d'une chanson avec le maximum de précision. Pour ce faire, on utilisera des méthodes d'apprentissage automatique (sous-domaine de l'intelligence artificielle).

Nous allons donc étudier plusieurs algorithmes de régression et comparer leur performance tout au long de ce travail grâce à différentes expériences. De plus, nous développerons un programme permettant de réutiliser l'ensemble des outils implémentés à cette occasion.

Notre travail se compose de deux grandes parties : la première est une étude mathématique des différentes méthodes pour arriver aux meilleurs résultats possibles ; et la seconde partie est le développement d'une bibliothèque Python3 pour pouvoir reproduire les résultats de la plus simple des façons.

2 Lecture et manipulation des données

Une chanson est généralement enregistrée sous la forme d'un fichier Mp3 contenant les données compressées d'une musique et des informations relatives à cette dernière (meta data). Cependant, pour ce projet, nous utiliserons des données fournies dans le fichier "YearPredictionMSD_100.npz" sous le format "npz" contenant deux matrices et un vecteur comme suit :

Donnée	Nature	Taille	Signification
X_labeled	Matrice	4578 x 90	Les chansons ayant des dates de sorties
X_unlabeled	Matrice	4578 x 90	Les chansons dont nous n'avons pas des dates de sorties
y_labeled	Vecteur	4578	Les années des chansons

Pour pouvoir utiliser ces données, on commence par importer ce fichier "YearPredictionMSD_100.npz" qui contient X_labeled, X_unlabeled et y_labeled à l'aide de la fonction *data_load()* qui se trouve dans le fichier script "data_utils.py". On effectue l'importation des données dans le fichier "exp_load_data.py".
Puis, dans un autre fichier "exp_visualisation.py", on visualise les données d'apprentissage, nous obtenons un histogramme qui correspond au nombre de musiques en fonction des années, voir figure 1.

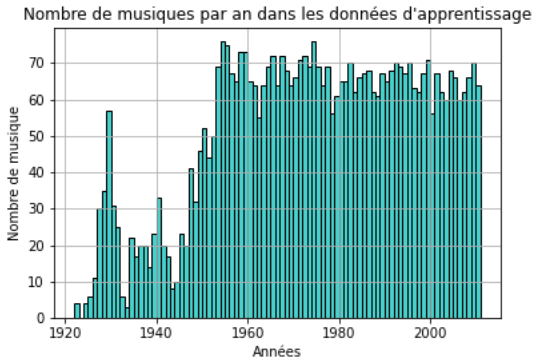


Figure 1 : Nombre de chansons par année

Nous rappelons que les dates de sortie des chansons sont réparties entre 1922 et 2011. D'après la figure 1, le nombre de chansons avant 1950 est relativement faible (inférieur à 58 chansons par an), contrairement aux années suivantes où on arrive jusqu'à 75 chansons par année.

L'étape suivante consiste à mélanger les données d'apprentissage, autrement dit, permuter les données de X_labeled et y_labeled en conservant les correspondances entre les deux variables. Cette étape est l'une des parties primordiales en Machine Learning tout comme - nous verrons plus tard - la validation croisée.

Pour ce faire, on utilise la fonction *randomize_data()* écrite dans le fichier "data_utils.py". Cette fonction prend en paramètre la matrice X_labeled, le vecteur y_labeled et les renvoie avec leurs lignes permutées. En d'autres termes, on redéfinie l'ordre des observations de manière aléatoire.

Afin de tester notre fonction randomize_data(), nous l'utilisons sur des données simples pour X et y dans le fichier "exp_randomize_data.py". La fonction renvoie bien une version permutée de notre matrice X et de notre vecteur y.

3 Validation croisée

Avant de commencer l'étude des méthodes de régression, nous appliquons une dernière transformation pour tester la fiabilité de nos méthodes : la validation croisée. Le principe est de diviser aléatoirement les données étiquetées de notre ensemble en deux sous ensembles distincts :

- S_train : ensemble de données d'apprentissage contenant 2/3 des données de l'échantillon initial.
- S_valid : ensemble de données de validation contenant 1/3 des données de l'échantillon initial.

Pour ce faire, nous implémentons la fonction *data_split()* qui prend en argument les données permutées renvoyées par la fonction précédente et un ratio entre 0 et 1 qui correspondant à la proportion d'exemples pour le train par rapport au test. On se retrouve donc avec deux couples de deux variables : ($X1$, $y1$) constituant l'ensemble S_train et ($X2$, $y2$) constituant l'ensemble S_valid.

On teste nos différentes fonctions dans le fichier "test_data_utils.py".

4 Algorithmes d'apprentissage élémentaire

Dans cette partie, nous allons étudier plusieurs algorithmes de régression pour prédire l'année d'une chanson. Quelque soit la méthode de régression, l'objectif est toujours le même : estimer la fonction objective f définie par :

$$\begin{array}{rccc} f_{w,b} & : & \chi & \rightarrow & Y \\ & & x & \mapsto & x^T w + b \end{array}$$

Avec $w \in \chi$ et $b \in \mathbb{R}$

Par la suite, l'ensemble des algorithmes de régression seront implémentés dans le fichier "algorithms.py".

4.1 Régression linéaire aux moindres carrés

On commence avec la méthode de régression la plus connue : régression linéaire aux moindres carrés. Le pseudo-code suivant décrit le fonctionnement de l'algorithme qu'on a remplit dans la classe "LinearRegressionLeastSquares" du fichier linear_regression.py :

Algorithm 1 - Régression linéaire aux moindres carrés

Require: $X \in \mathbb{N} \times \mathbb{M}$ et $y \in \mathbb{R}^N$
$\tilde{X} \leftarrow [X; 1_N]$
$\tilde{w} \leftarrow \tilde{X}^+ y$
$w \leftarrow (\tilde{w}(0), ..., \tilde{w}(M-1))$
$b \leftarrow \tilde{w}(M)$
return w, b

4.2 Algorithmes simplicites : des estimateurs constants

Au delà de la régression linéaire aux moindres carrées, il est possible d'implémenter des algorithmes simplistes pour prédire l'année d'une musique. Le point commun entre ces différentes méthodes est qu'elles renvoie systématiquement la même valeur indépendamment de la données en entrée. Ils permettent de trouver une fonction de prédiction constante.

a. Algorithme de l'apprentissage de l'estimateur constant moyen

Le premier algorithme renvoie la moyenne des années des musiques de l'ensemble d'apprentissage. Il a été programmé dans la classe "LinearRegressionMean" du même fichier.

Algorithm 2 - Estimateur constant moyen

Require: $X \in \mathbb{N} \times \mathbb{M}$ et $y \in \mathbb{R}^N$
$w \leftarrow 0_M$
$b \leftarrow \frac{1}{N} \sum_{n=0}^{N-1} y(n)$
return w, b

b. Apprentissage de l'estimateur constant médian

Le second algorithme renvoie l'année médiane de l'ensemble d'apprentissage. La classe associée est "LinearRegressionMedian".

Algorithm 3 - Estimateur constant médian

Require: $X \in \mathbb{N} \times \mathbb{M}$ et $y \in \mathbb{R}^N$
$w \leftarrow 0_M$
$b \leftarrow median(y)$
return w, b

Le calcul de la médiane est implémenté avec la fonction numpy.median().

c. Apprentissage de l'estimateur constant majoritaire

Enfin, le dernier algorithme de cette famille renvoie l'année majoritaire de l'ensemble d'entraînement. Cette fois-ci on remplit l'algorithme dans la classe "LinearRegressionMajority".

Algorithm 4 Apprentissage de l'estimateur constant majoritaire

Require: $\mathbf{X} \in \mathbb{R}^{N \times M}$ et $\mathbf{y} \in \mathbb{R}^N$
  $\mathbf{w} \leftarrow \mathbf{0}_M$ (vecteur nul de dimension M)
  $(\mathbf{b}, \mathbf{a}) \leftarrow \text{histogram}(\mathbf{y})$
  $b \leftarrow \mathbf{a}(\text{argmax}(\mathbf{b}))$
  return $\mathbf{w}, b$

Le calcul de l'année majoritaire est implémenté avec la fonction numpy.unique() et numpy.argmax(). On récupère les années des musiques avec la première fonction et on détermine l'année qui a la plus grande occurence avec la seconde fonction.

On retrouve dans le fichier "test_linear_regression.py", les tests vérifiant le bon fonctionnement des différentes implémentations.

5 Expérience avec ressources limitées

Maintenant que nous avons différentes méthodes implémentées, nous pouvons les étudier. Commençons par chercher à déterminer leurs performances selon la taille de l'ensemble d'apprentissage afin de déterminer qui est le meilleur parmi les quatre. Ces expériences sont faites dans le fichier "exp_training_size.py" où on apprend nos différents modèles sur différents ensembles de taille croissante, nous affichons dans le graphe suivant la valeur de l'erreur lorsqu'on applique ces modèles pour prédire sur les données d'entraînement (tracé en pointillé) et sur les données de validation (tracé continue).

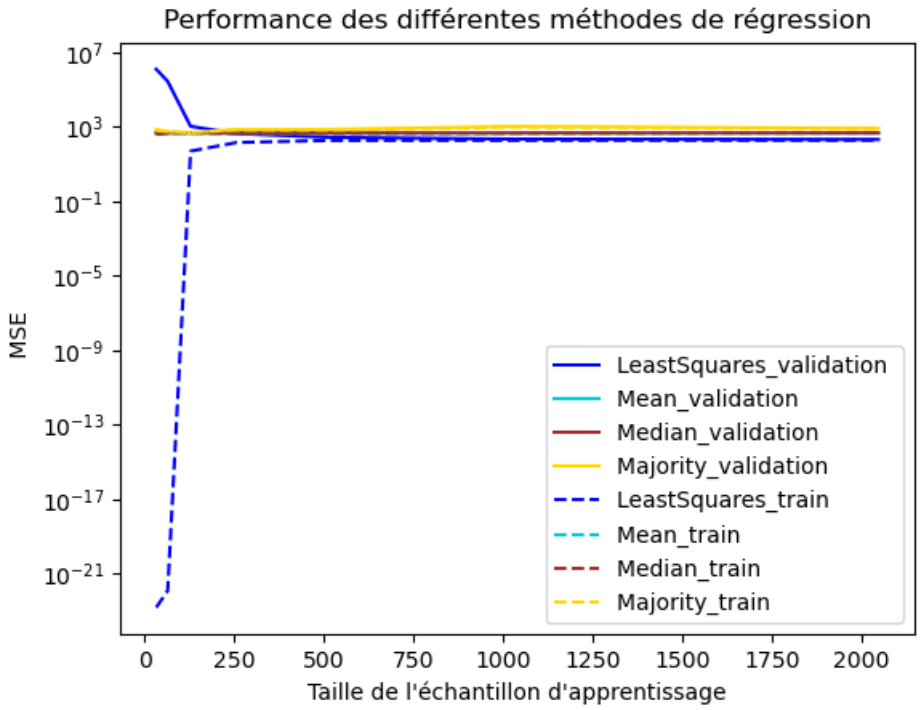


Figure 2: Performance des algorithmes d'apprentissage avec la pseudo inverse

D'après les résultats, on observe que le MSE des algorithmes simplistes (estimateur constant majoritaire, médian et moyen) se concentre autour de 10^3 pour l'ensemble d'entraînement et de validation. En revanche, pour l'algorithme de régression aux moindres carrés, il y a une différence entre les deux ensembles. En effet, le MSE train augmente au début lorsque la taille de l'ensemble d'apprentissage est faible et se stabilise autour de 10^2 lorsque la taille de l'ensemble d'apprentissage devient important. Concernant l'erreur de validation, elle diminue avec l'augmentation de la taille de l'échantillon d'apprentissage.

Dans ce cas, on a un problème de sur apprentissage avec la régression aux moindres carrées : le MSE sur l'ensemble d'entraînement augmente à partir d'une certaine taille de ce dernier.

On peut aussi comparer les différentes méthodes par rapport à leur temps d'apprentissage : on cherche donc maintenant à évaluer le temps de calcul des différents algorithmes.

La figure ci-dessous montre l'évolution du temps de calcul des quatre algorithmes selon la taille de l'ensemble d'entraînement. On observe que les algorithmes simplistes ont un temps de calcul rapide et constant, contrairement à l'algorithme de régression aux moindres carrés qui a un temps de calcul croissant et plus important que les autres mais dont la courbe peut être majorée par une droite linéaire (complexité linéaire à vue d'oeil). On constate que les tendances obtenues correspondent bien aux coûts computationnels annoncés dans le sujet.

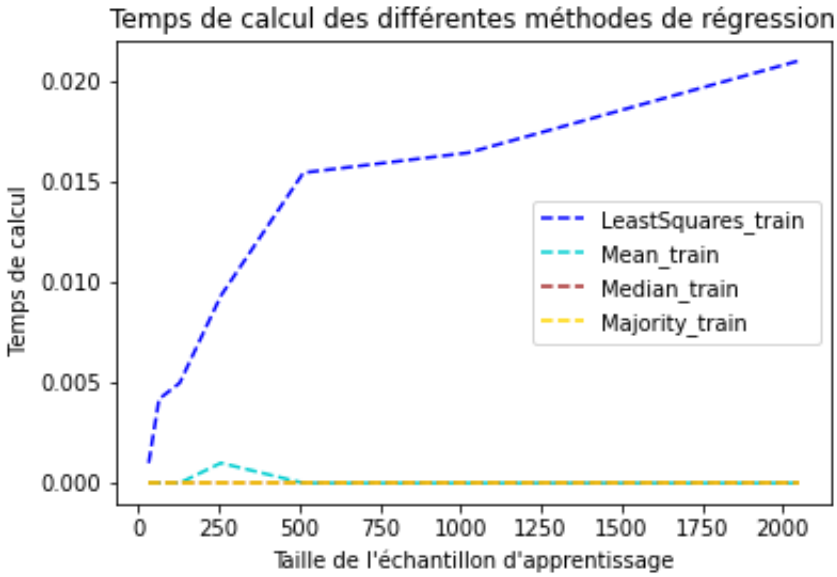


Figure 3: Temps de calcul des différents algorithmes

6 Algorithmes avec régularisation

Dans cette partie, nous allons étudier d'autres méthodes de régressions pour réduire le sur-apprentissage sur nos données d'entraînement. En effet, nous avons observé dans la partie précédente que l'algorithme de régression aux moindres carrés voyait ses résultats se dégrader lorsque l'ensemble d'entraînement atteignait une certaine taille. Pour contrecarrer ce phénomène, on peut utiliser une nouvelle famille : les algorithmes avec régularisation.

La régularisation consiste à pénaliser la complexité des modèles afin de limiter le sur-apprentissage. Par la suite, nous étudierons deux types de ces algorithmes : la régularisation Ridge et et la régularisation parci-

monieuse avec les modèle MP et OMP. Contrairement à la régressions aux moindres carrés, ces algorithmes possèdent un hyper-paramètre qui permet de gérer la régularisation.

Dans le cas de la régularisation parcimonieuse, on cherche des solutions linéaires parcimonieuses $f_{w,b}(x) = x^T w + b$ avec des vecteurs creux w (dont la plupart des composantes sont nulles) mais en limitant le nombre de valeurs de w non-nulles, ce qui va limiter la complexité du modèle.

6.1 Régression Ridge

La régression Ridge est un algorithme qui permet de pénaliser le terme $||w||_2^2$. On peut ajuster le poids de cette pénalité par rapport au terme d'erreur des moindres carrés par une constante que l'on note $\lambda > 0$.

Le principe est similaire à celui de la régression linéaire avec une contrainte quadratique sur les coefficients.Cette technique est utile lorsque les variables explicatives sont fortement corrélées. Elle permet d'optimiser le problème suivant :

$$argmin_w \frac{1}{2}||Xw - y||_2^2 + \frac{\lambda}{2}||w||_2^2$$

La solution est obtenue en annulant le gradient par rapport à w, elle est définie par :

$$w* = (X^t X + \lambda I)^{-1} X^T Y$$

On choisit λ de telle manière à ce que la matrice $X^t X + \lambda I$ soit inversible. L'objectif de ce modèle est de biaiser un peu la prédiction, afin de diminuer l'erreur standard.

La fonction ridge_regression(X, y, lambda) qu'on a complété dans le fichier "linear_regression" prend en argument le coefficient de pénalisation lambda, un ensemble d'apprentissage donné par la matrice X et le vecteur y, et renvoie la solution w.

6.2 Régression Matching Pursuit (MP)

L'algorithme Matching Pursuit fournit une solution w qui contient au plus k_{max} composantes non-nulles. On implémente ce modèle avec la fonction mp(X, y, n_{iter}) dans le fichier "algorithms.py" qui prend en argument un entier n_{iter}, un ensemble de données étiquetées (X,y) et renvoie le couple (w, error_norm) : le vecteur parcimonieux w et un vecteur error_norm de taille n_iter+1 dont la k-ième composante sera égale à l'énergie du résiduel au début de l'itération k (donc le dernier coefficient est l'énergie du résiduel à la fin de l'algorithme).

Algorithm 4 - MP

Require: $X = [a_0, ..., a_{M-1}] \in \mathbb{R}^{N \times M}$ t.q. $\forall m, ||a_m||_2 = 1, y \in \mathbb{R}^N, k_{max} \in \mathbb{N}^*$
  $r \leftarrow y$ (Initialisation du résiduel)
  $w \leftarrow 0_M$ (Initialisation de w)
  for $k = 1$ à k_{max} **do**
    $\forall m, c_m \leftarrow < a_m, r >$ (Calcul des corrélations du résiduel et des composantes)
    $\hat{m} \leftarrow argmax|c_m|$ (Sélection de la composante la plus corrélée)
    $w(\hat{m}) \leftarrow x(\hat{m}) + c_{\hat{m}}$ (Mise à jour de la décomposition)
    $r \leftarrow r - c_{\hat{m}} \cdot a_{\hat{m}}$ (Mise à jour du résiduel)
  end for
  return w

6.3 Régression Orthogonal Matching Pursuit (OMP)

L'algorithme OMP est une version améliorée de l'algorithme MP. Contrairement à la régression MP, après chaque étape de l'itération, tous les coefficients extraits sont mis à jours : c'est-à-dire que l'on calcule la projection orthogonale des coefficients du sous espace Ω (voir le pseudo-code ci-dessous). On implémente cet algorithme dans la fonction omp(X, y, n_iter) qui prend les mêmes arguments que pour le Mp. Les étapes de la méthode sont décris ci-dessous :

Algorithm 5 - OMP

Require: $X = [a_0, ..., a_{M-1}] \in \mathbb{R}^{N \times M}$ t.q. $\forall m, ||a_m||_2 = 1, y \in \mathbb{R}^N, k_{max} \in \mathbb{N}^*$
  $r \leftarrow y$ (Initialisation du résiduel)
  $w \leftarrow 0_M$ (Initialisation de w)
  $\Omega \leftarrow \emptyset$ (Initialisation du support de décomposition)
  for $k = 1$ à k_{max} **do**
    $\forall m, c_m \leftarrow < a_m, r >$ (Calcul des corrélations du résiduel et des composantes)
    $\hat{m} \leftarrow argmax|c_m|$ (Sélection de la composante la plus corrélée)
    $\Omega \leftarrow \Omega \cup \{\hat{m}\}$ (Mise à jour du support)
    $w(\Omega) \leftarrow X_\Omega^+ y$ (Mise à jour de la décomposition)
    $r \leftarrow y - Xw$ (Mise à jour du résiduel)
  end for
  return w

7 Intégration des algorithmes pour la régression linéaire

Afin d'adapter les algorithmes précédents à notre problème de régression, une étape de pré-traitement et de post-traitement ont été rajoutées. En effet, lors du pré-traitement, on cherche à centrer y (soustraire b = $\overline{y}$ de y) et ensuite normaliser toutes les colonnes a_m de la matrice X. Cette normalisation est faite à l'aide de la fonction "normalize_dictionnary(X)" qu'on a complétée dans le fichier "algorithms.py", elle prend en paramètres X et renvoie sa version normalisée et le vecteur contenant les coefficients de normalisation.

Concernant l'étape du post-traitement, l'algorithme appliqué à $\tilde{X}$ et $\tilde{y}$ qui sont des versions normalisées de X et y, renvoie un vecteur $\tilde{w}$ tel que $\tilde{X}\tilde{w} \approx \tilde{y}$.

Par la suite, on complète les classes "LinearRegressionRidge","LinearRegressionMp" et "LinearRegressionOmp" dans le fichier "linear_regression.py" qui nous permettent d'avoir le paramètre b et w liés respectivement aux algorithmes Ridge, Mp et OMP.

Pour pouvoir utiliser ces classes, on a complété le fichier "exp_algorithms" où on va fixer les hyper-paramètres k_{max} et λ arbitrairement et faire varier le nombre d'exemples des données de l'ensemble d'apprentissage. Ici, on fixe k_{max} à 20 itérations et λ à 0.3. Nous obtenons donc les résultats suivants :

7.1 Régression Ridge

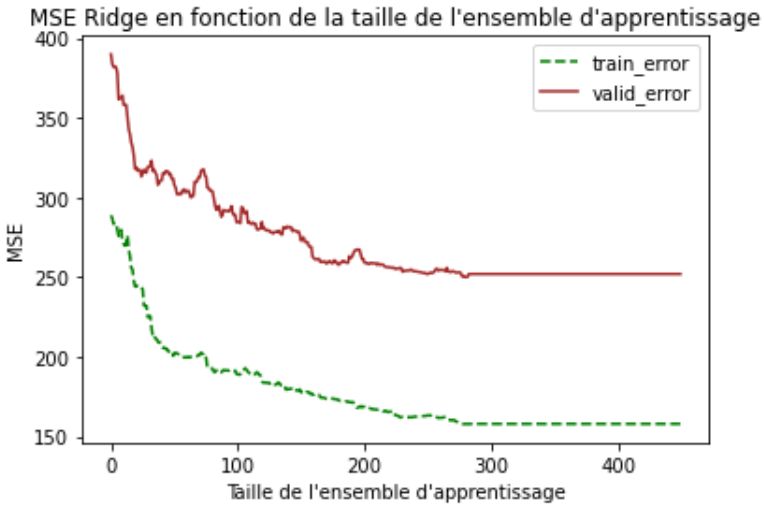


Figure 4 : Performance de la régression Ridge pour λ fixé

On constate qu'avec l'algorithme de la régression Ridge, il y a un écart important entre l'erreur de validation et l'erreur d'entraînement. En effet, le MSE de validation est plus grand et on remarque aussi que plus le nombre de données d'entrainement est élevé, plus on a moins d'erreur pour la prédiction sur les deux ensembles d'entraînement et de validation, à partir de 300 exemples, les deux MSE se stabilisent.

7.2 Régression Mp

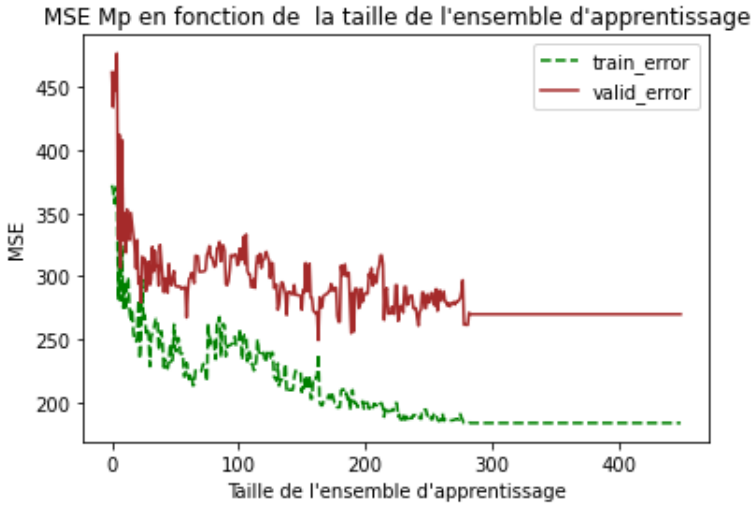


Figure 5 : Performance de la régression Mp pour k_{max} fixé

On remarque un résultat similaire avec l'algorithme avec un écart moins important, néanmoins, avec Mp l'erreur de validation décroît plus rapidement pour un nombre de données d'apprentissage relativement faible (inférieur à 80 exemples).

7.3 Régression Omp

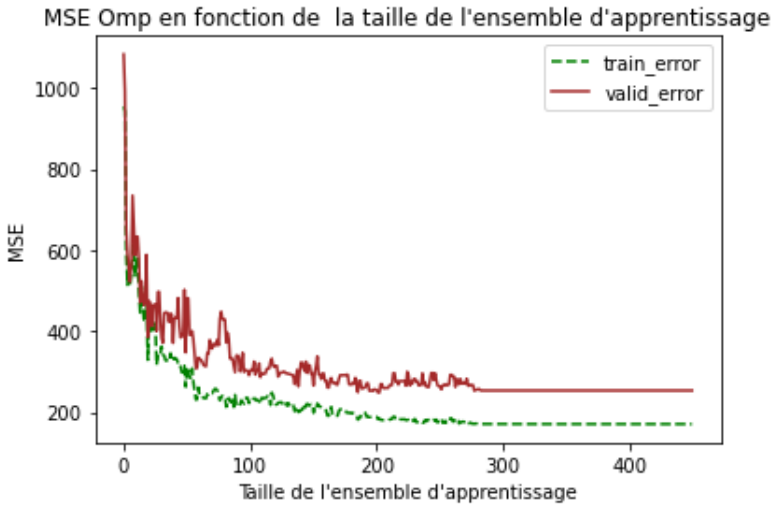


Figure 6 : Performance de la régression Omp pour k_{max} fixé

L'algorithme Omp fournit un résultat plus satisfaisant dans ce cas, il n'y a pas une grande différence entre les deux MSE et l'algorithme généralise bien par rapport aux deux précédents.

8 Expérience pour déterminer les hyper-paramètres

Maintenant, on cherche à déterminer le meilleur hyper-paramètre pour les algorithmes Mp, Omp (k_{max}) et Ridge (λ). Nous complétons cette fois-ci le fichier "exp_hyperparam" pour évaluer la performance des trois algorithmes en fonction de la valeur de l'hyper-paramètre associé.

Commençons par observer l'évolution des résultats sur l'ensemble d'entraînement et de validation selon différentes valeurs pour l' hyper-paramètre de chaque modèle. Dans les parties suivantes, nous allons étudier les courbes de l'évolution du mse pour l'entraînement et la validation pour dégager une tendance générale pour chaque algorithme.

Un des moyens les plus simples pour trouver l'hyper-paramètre automatiquement est d'entraîner un même modèle avec une suite de valeurs pour l'hyper-paramètre différentes. Ensuite, on récupère la valeur de l'hyper-paramètre qui minimise l'erreur de validation et on ré-entraîne notre modèle avec cette valeur.

8.1 Régression Ridge

On commence avec l'algorithme de Ridge. Les deux courbes suivantes montrent les performances de la régression Ridge selon la valeur de λ.

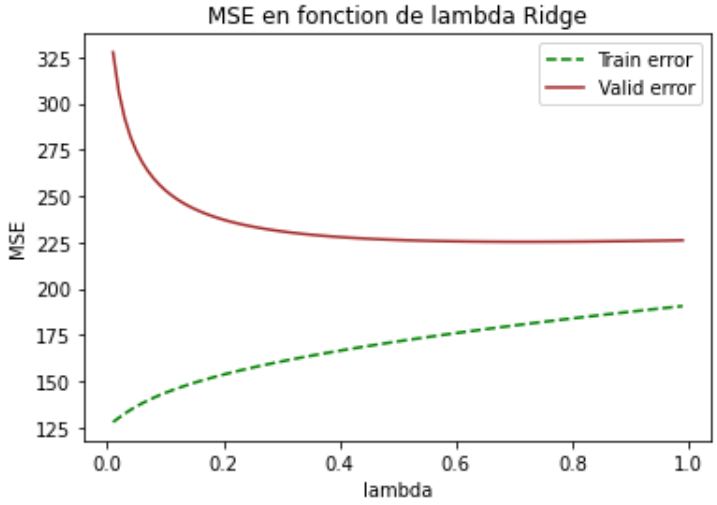


Figure 7 : Performance de la régression Ridge en fonction de λ

L'algorithme Ridge est connu pour sa capacité à mieux généraliser à partir d'un jeu de données. Ceci est dû au fait qu'on pénalise le terme $||w||_2^2$; autrement dit, on pénalise la complexité de la courbe de régression. Dans notre cas, nous avons bien que l'erreur sur l'ensemble d'entraînement est plus élevée et croit lorsque λ augmente. De plus, l'erreur de validation décroît jusqu'à 225 où se stabilise pour $\lambda = 0.4$.

8.2 Régression Mp

Pour le cas de la régression MP, la tendance n'est pas pareil.

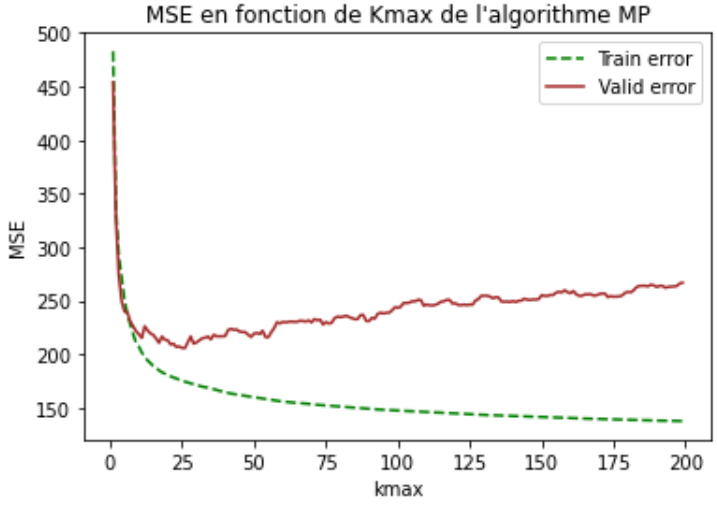


Figure 8 : Performance de la régression Mp en fonction de k_{max}

En effet, l'erreur de validation croit de manière significative à partir d'un certain rang de k_{max} (≈ 15 pour cet exemple) tandis que l'erreur d'entraînement décroît avec l'augmentation de k_{max}. Ici, nous avons donc un exemple flagrant de sur-apprentissage.

8.3 Régression Omp

On finit avec la régression OMP. Dans l'exemple ci-dessous, une nouvelle tendance dans l'évolution du mse nous frappe

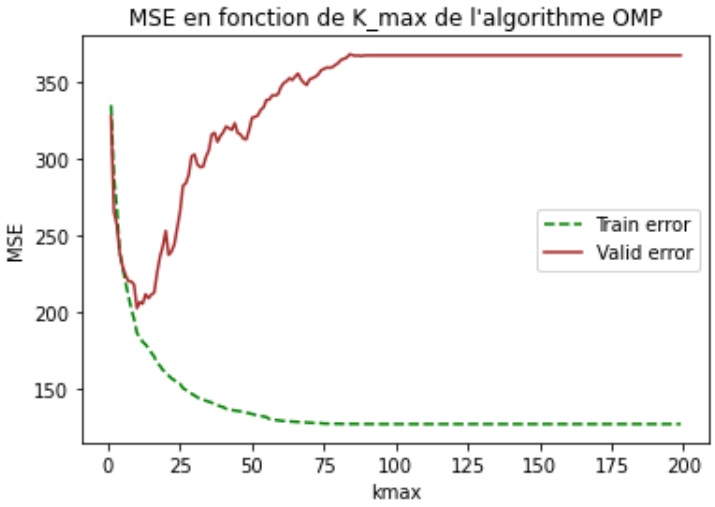


Figure 9 : Performance de la régression Omp en fonction de k_{max}

On remarque immédiatement que le problème du sur-apprentissage est bien plus important que pour l'algorithme MP. A partir du 15ème rang de k_max eur de validation croit à partir de $k_{max} \approx 15$ où l'algorithme rencontre un problème de sur-apprentissage.

9 Estimation des hyperparamètres par validatiton croisée

Nous rappelons que dans la partie précédente, nous avons fixé les deux hyper-paramètres d'une façon non automatique et celà n'est pas pratique. C'est pour celà qu'on va utiliser la fonction "learn_all_with_algo(X, y)" où algo désigne quel algorithme parmi les trois (Ridge, Mp et Omp).
Cette fonction va estimer l'hyperparamètre par validation croisée, elle ne prend en argument qu'un ensemble étiqueté X, y, sans hyperparamètre. De cette manière, l'hyperparamètre est sélectionné en fonction de la taille de l'ensemble d'apprentissage (X, y).

On crée donc les fonctions pour les trois algorithmes dans le fichier "learn_all_with_ALGO.py".

Le tableau ci-dessous donne les résultats des meilleurs hyper-paramètres pour les trois méthodes de régression sur 40 entraînements et validations différentes.

Itérations	Résultat pour Ridge	Résultats pour MP	Résultats pour OMP
1	0.48	61	7
2	0.44	80	9
3	0.32	43	15
4	0.55	11	10
5	0.34	63	21
6	0.37	13	11
7	0.37	69	28
8	0.25	13	11
9	0.99	16	10
10	0.39	46	24
11	0.26	45	14
12	0.51	93	11
13	0.21	19	14
14	0.76	54	18
15	0.3	42	10
16	0.4	72	15
17	0.34	13	11
18	0.62	54	13
19	0.54	34	6
20	0.44	45	22
21	0.43	82	37
22	0.29	58	10
23	0.28	13	11
24	0.82	50	37
25	0.2	45	9
26	0.89	13	12
27	0.28	34	8
28	0.59	97	32
29	0.28	42	11
30	0.13	43	23
31	0.54	32	16
32	0.51	99	25
33	0.24	95	29
34	0.5	78	45
35	0.25	38	10
36	0.3	91	22
37	0.29	73	27
38	0.32	63	30
39	0.23	42	42
40	0.11	26	8

Ainsi, d'après les résultats, on peut dire qu'en moyenne, le meilleur hyper-paramètre pour l'algorithme de régression Ridge est 0.41, 50 pour la régression MP et 18 pour la régression OMP.

De plus, on retrouve les grandes particuliarité des méthodes dans les résultats du tableau : la régression OMP converge plus rapidement que le MP (son k_{max} optimal est inférieur à celui du MP) et que la régression de Ridge généralise au mieux autour d'un λ égal à 0.41.

10 Utilisation de nouvelles méthodes

Maintenant, on utilise les fonctions crées dans la partie précédente dans le fichier "plot_exp_training_size.py" pour afficher sur un même graphique les performance de prédiction des trois algorithmes en fonction de la taille de l'ensemble d'apprentissage.

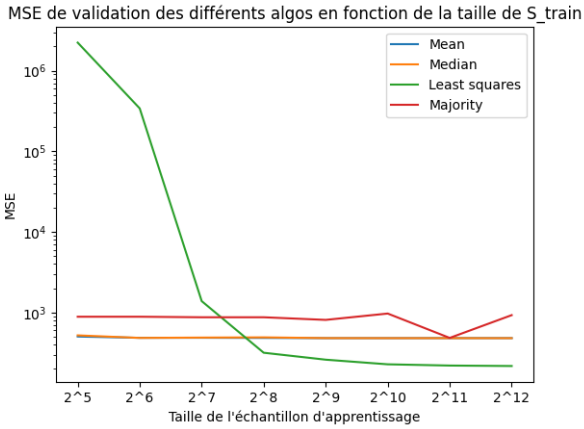


Figure 10 : Performance des premiers algorithmes sur l'ensemble de validation

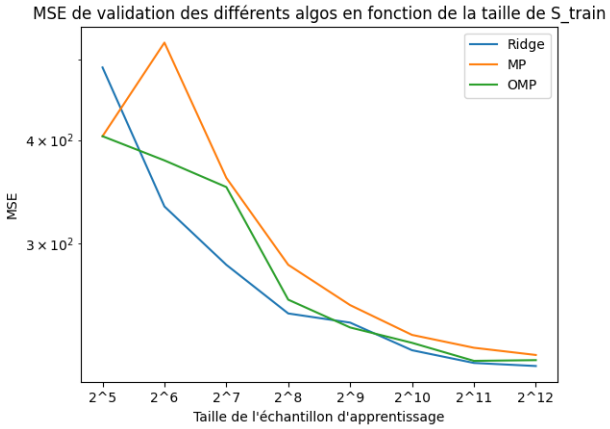


Figure 11 : Performance des derniers algorithmes sur l'ensemble de validation

Nous rappelons que l'échelle de l'axe des ordonnées est différente entre la figure 10 et 11, c'est pour cela que nous avons affiché les résultats sur 2 figures séparées.

En affichant l'erreur de validation pour les différents algorithmes, on remarque que la régression linéaire aux moindres carrés arrive à mieux prédire lorsqu'on a une taille de l'échantillon d'entraînement plus élevée. Dans la figure 11, les deux algorithmes de MP et OMP ont un MSE élevé (suppérieur à 600) et qui décroit lentement en fonction de la variation de la taille de l'ensemble d'apprentissage. En revanche, la régression Ridge a un MSE relativement faible (inférieur à 380 et qui devient très faible rapidement quand on augmente la taille de notre échantillon. Nous pouvons donc considérer la régression Ridge comme l'algorithme le plus performant dans notre cas.

En utilisant la fonction "learn_best_predictor_and_predict_test_data()", qui permet de prédire les dates manquantes des chansons du fichier initial "LearPredictionMSD100.npz", nous avons pu obtenir les résultats qu'on a stocké dans le fichier "test_prediction_results.npy".

11 Refactoring

L'objectif du refactoring est le fait de retravailler le code de notre programme pour le rendre plus lisible et donc plus maintenable. Concrètement, le refactoring consiste à retravailler la présentation du code, la modification des algorithmes par une réorganisation des unités de traitements par exemples (factorisation des fonctions, ajout de variables intermédiaires pour une meilleurs compréhension, etc) ou encore une modification de la conception.

Dans notre cas, la refactorisation est assez simple car le code est peu important (moins de 1000 lignes). Mais certains points ont été retravaillé :

1. Le premier est l'organisation des fichiers de codes ;

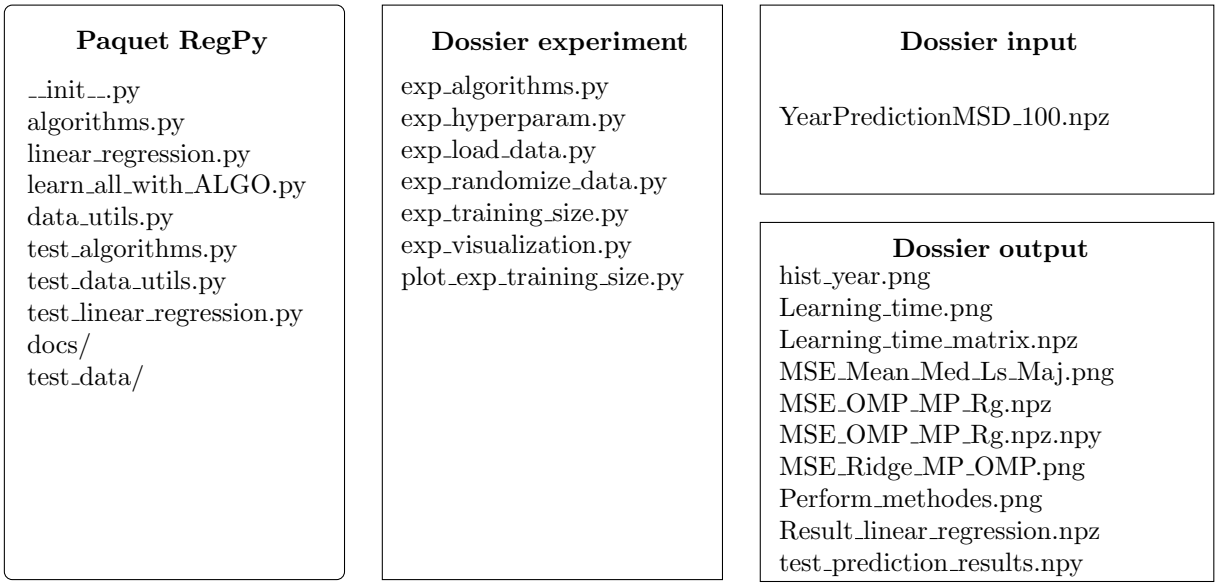
2. Le second est de regrouper le code des algorithmes, des classes, des tests et des autres outils développé pour charger et manipuler les données dans un paquet. Pour ce faire, quelques modifications dans le code et l'ajout de fichiers ont du être fait ;

3. Le troisième point est l'ajout de commentaires et le renommage de certaines variables pour rendre le code plus compréhensible.

11.1 Organisation des fichiers de code

Dans cette partie, nous allons décrire l'organisation de nos fichiers de code dans son ensemble ; c'est-à-dire l'ensemble du code que l'on retrouve dans notre projet : l'implémentation des fonctions, les tests, les expériences et les affichages des résultats.

Le résultat final est le suivant :



On a donc regrouper l'ensemble de nos méthodes de régressions et nos fonctions dans un paquet que l'on nomme RegPy. Le dossier experiment contient l'ensemble des scripts d'expérimentation des performances de nos modèles de régression et l'affichage des résultats. Le dossier input stocke les données initiales. Enfin, le dossier output accueille les productions des différents scripts du dossier experiment.

11.2 Paquet Python

Nous avons décidé de créer un paquet RegPy. Pour ce faire nous avons regrouper les fichiers de code dans un même dossier. Nous avons créer le fichier __init__.py qui indique à l'interpréteur Python que notre dossier est un paquet. De plus, le fichier __init__.py importe l'ensemble des sous modules (algorithms.py, linear_regression.py, learn_all_with_ALGO.py, etc) lorsqu'on import le paquet RegPy.

Il y a aussi des fichiers de tests des différents sous-modules que l'on différencie des autres sous-modules en ajoutant le préfixe test_.

Pour utiliser un paquet, il suffit d'ajouter son emplacement dans le path system de Python. L'exemple suivant illustre la procédure :

```
import sys
sys.path.append(<chemin de l emplacement du paquet regpy>)
import regpy as rp
```

De plus, lorsqu'un sous-module de regpy est utilisé, on ajoute au path system, le chemin de ce dernier avec la commande suivante :

```
import os, sys
DIR_PATH = os.sep.join(__file__.split(os.sep)[:-1])
sys.path.append(DIR_PATH)
```

Cette technique permet de garder une cohérence dans les appelles des sous-modules. Une autre technique est l'utilisation du paquet distutils pour installer notre module.

11.3 Ajout de commentaires et création d'une documentation

La dernière partie de notre refactoring est l'enrichissement de commentaires dans notre code. Ce travail permet certaines choses : la première est de rendre le code plus compréhensible pour une future mise à jour et la seconde est de créer une documentation à l'aide du logiciel Sphinx.

Sphinx est un logiciel qui génère une documentation à partir d'un code source. Il s'appuie sur les docstrings (commentaire délimité par les triples guillemets """) qui se situe dans les fonctions et classes permettant de fournir une description de ces dernières. Pour chaque fonction, on ajoute si besoin une description son action, de ses arguments et de sa sortie.

11.4 Problèmes rencontrés

Lors de ce projet, on a rencontré un problème avec la programmation de l'algorithme MP et OMP. En effet, dans le fichier "exp_hyperparam.py", il fallait entraîner les modèles de régressions (LinearRegressionRidge, LinearRegressionMp et LinearRegressionOmp) afin de déterminer les performances de des trois algorithmes. Néanmoins, au début, il y avait un problème de limite du nombre d'itération qui a été confondue avec le nombre de colonnes de la matrice "X_labeled", on avait donc une erreur dès qu'on dépassait 90 itérations. Cette erreur a aussi causé problème pour l'algorithme MP qui avait un MSE train et valid qui croîtraient au lieu de décroître.