Programmation Python - M2 Data Science

Énoncé du projet à rendre

Table des matières

1 Présentation générale

1.1 Le challenge

Nous avons accès à une énorme quantité de musique numérique sur nos ordinateurs, nots téléphones, en streaming, etc. Imaginez que l'on vous fasse écouter une chanson que vous ne connaissez pas : seriez-vous capable de la dater ? Avec quelle précision : un an, dix ans, cinquante ans ? Si on demandait à un musicologue qui a une grande expérience d'écoute de musiques, quelle pourrait être la précision de sa réponse ?
En passant de l'homme à la machine, nous aimerions qu'un ordinateur puisse estimer l'année d'une chanson à partir du contenu sonore de façon automatique, en lui donnant au préalable un nombre limité d'*exemples d'apprentissage*.

1.2 Les données

En général, une chanson est stockée dans un fichier, par exemple au format `.mp3` ou `.wav`. Ce fichier contient les données brutes de l'enregistrement et vous pouvez l'écouter avec un lecteur. Pour simplifier la tâche, vous n'allez pas travailler directement sur ces fichiers mais sur une transformation de ces fichiers : chaque fichier a été transformé en un vecteur de 90 coefficients représentatif de son contenu sonore. Effectuer cette transformation ne fait pas partie du projet, vous travaillerez directement à partir des coefficients transformés, qu'on appelle descripteurs (*features* en anglais).
Vous allez accéder à 4578 exemples de chansons dont à la fois le vecteur de descripteurs ainsi extrait et l'année sont fournis. En pratique, ces données sont disponibles dans le fichier `YearPredictionMSD_100.npz` sur la page Ametice de l'UE. Il contient un dictionnaire dont la clé `'X_labeled'` permet d'accéder à une matrice de format 4578×90 : chaque ligne correspond au contenu d'une chanson donné sous la forme de 90 descripteurs (*features*) qui ont été extraits de l'enregistrement. La clé `'y_labeled'` permet d'accéder à un vecteur de taille 4578 correspondant aux années des chansons. Une dernière clé `'X_unlabeled'` vous donnera également accès à une autre matrice de 2289 autres chansons pour lesquelles l'année ne vous est pas fournie. **Votre but est de développer un programme qui estime l'année de ces chansons avec la plus grande précision. À la fin du projet, nous comparerons vos résultats aux vraies dates.**
Pour en savoir plus. Dans les challenges internationaux pour des tâches de prédiction à partir de données, on procède exactement de la même façon : les participants ont accès à un ensemble d'exemples d'apprentissage où l'information à prédire est fournie (on parle d'exemples étiquetés) et à un ensemble de données non-étiquetées sur lequel la performance des différents systèmes est évaluée pour départager les participants. Ceux-ci envoient leurs prédictions obtenues sur les données non-étiquetées à l'organisateur du challenge, qui est le seul à posséder les vraies étiquettes, et peut ainsi évaluer la précision des prédictions. Il est très fréquent que l'étape d'extraction des descripteurs à partir des données brutes (enregistrements, images, etc.) soit réalisée en amont, de sorte que les candidats ne travaillent que sur les descripteurs, comme dans ce projet. Dans notre cas, les données fournies proviennent d'un jeu de données appelé « Million Song Dataset » [1] qui est utilisé pour différentes tâches de prédiction (année, genre musical, etc.) et d'apprentissage en général. Les descripteurs qui ont été extraits ici sont des descripteurs de « timbre musical » dont la moyenne sur tout un enregistrement donne un vecteur de taille 12 et la covariance donne un vecteur de taille 78, soit un vecteur de taille 90 qui a ensuite subi une normalisation.
Autres fichiers de données. Sur Ametice, vous avez également accès à deux autres fichiers `YearPredictionMSD_2_100.npz` et `YearPredictionMSD_2_12_100.npz` qui contiennent les données des mêmes chansons mais avec un nombre de descripteurs plus élevé, respectivement 180 et 4185. L'utilisation de ces fichiers est optionnelle : vous pourrez éventuellement voir si vos algorithmes sont plus performants avec des données en plus grande dimension et vous rendre compte des différences de temps de calcul.

1.3 Comment apprendre à un ordinateur à dater une chanson ?

Vous serez guidés dans ce projet pour concevoir un tel programme. Vous allez en particulier apprendre des techniques et des notions d'*apprentissage supervisé* :

- on dispose d'un certain nombre de chansons pour lesquelles l'année est connue ;
- ces exemples étiquetés sont utilisés pour entrainer un modèle à estimer l'année à partir du contenu sonore ;
- le but est qu'il puisse utiliser cet apprentissage pour prédire l'année d'une nouvelle chanson inconnue – c'est-à-dire une chanson qui n'est pas dans l'ensemble d'apprentissage ;
- une question importante est d'estimer la performance de notre système à *généraliser* ce qu'il apprend, c'est-à-dire sa précision quand il prédit l'année d'une chanson n'appartenant pas à l'ensemble d'apprentissage.

1. http ://labrosa.ee.columbia.edu/millionsong/

1.4 Objectifs et contenu du projet

Ce mini-projet peut-être divisé en plusieurs parties, dont les objectifs principaux sont multiples. Il ne s'agit pas de devenir des experts de l'analyse d'enregistrements musicaux, mais plutôt :

- de monter en compétence en développement logiciel et en programmation, en Python, en étant capable de mettre en œuvre un projet de programmation incluant des algorithmes dans le cadre d'applications sur des données réelles ; ceci vous sera utile pour le bon déroulement des projets de M2 et pour votre stage ; ceci comprend notamment :
  - l'expérience de programmation et l'acquisition de bonnes pratiques de programmation : maîtriser le langage, diviser le code en plusieurs fichiers et fonctions, tester les fonctions codées, les intégrer ensemble, etc.
  - la manipulation de données
  - la capacité à formaliser un algorithme (en pseudocode) et à le mettre en œuvre avec un langage de programmation
- d'adopter une méthodologie typique de résolution de problème en sciences des données : exploration et manipulation d'un jeu de données, traitements, apprentissage, réglage des paramètres, évaluation des performances par validation croisée ;
- de vous familiariser avec une structuration du code pour les sciences des données, avec une architecture « à la `scikit-learn` »
- de voir le lien entre le problème posé et les techniques de sélection de variables pour construire des modèles dits parcimonieux.

1.5 Méthodologie de programmation

Ce projet sera l'occasion de développer vos compétences par l'expérience en programmation, et notamment d'assimiler de bonnes pratiques de programmation pour gagner en efficacité. La plus grande partie de ces notions ont déjà été introduites, nous en verrons peut-être d'autres. Le style de programmation adopté pour ce projet repose sur les fonctions et les classes : pour un certain nombre de fonctionnalités, vous devrez coder des fonctions (par exemple, l'algorithme Matching Pursuit) ; pour d'autres tâches, vous devrez créer et manipuler des classes (par exemple, pour l'apprentissage et la prédiction des années). Ce type de programmation s'oppose à la programmation uniquement par scripts [2] ou par notebooks (qui ne sont pas appropriées pour un projet de cette taille). L'utilisation de scripts sera restreinte à des cas particulier : les expériences et autres programmes utilisant les fonctions et classes.
L'architecture du code a été pensée pour vous aider et vous donner un modèle : l'en-tête des fonctions et méthodes est donné (nom, type des arguments en entrée et des données en sortie), certains tests seront aussi fournis. Il vous appartient de comprendre la logique de cette architecture, de la respecter (ne pas changer les noms, les entrées et sorties), de demander de l'aide si besoin, de signaler les possibles erreurs dans l'énoncé.
Les tests sont fournis. Comprenez-les, utilisez-les et inspirez-vous en pour écrire du code pour débugguer vos fonctions. Il est conseillé d'installer le paquet `spyder-unittest` qui permet de sélectionner facilement les résultats des tests qui vous intéressent. L'installation peut se faire de plusieurs façons (liste non-exhaustive) :

- avec Anaconda en ligne de commande, `conda install -c spyder-ide spyder-unittest` ou `conda install -c conda-forge spyder-unittest`
- avec le navigateur Anaconda, ajoutez le channel `conda-forge` et installez le paquet `spyder-unittest`
- avec pip3, `pip3 install spyder-unittest`

1.6 Évaluation de votre travail

L'évaluation se fera à partir d'un rapport/compte-rendu ainsi que du code produit (date à préciser ensemble). Les conseils donnés dans les séances précédentes et dans les autres UE sont à appliquer à la fois pour le rapport (clarté, rigueur, etc.) et pour le code (indentation, commentaires, tests, etc.). Le plagiat ne sera pas toléré [3].

2 Lecture et manipulation des données

Vous complèterez le code fourni en étudiant sa structure et sa documentation.

2. On appelle script un fichier exécutable, qui contient des lignes de code en dehors des fonctions/classes.
3. `https://sciences.univ-amu.fr/sites/sciences.univ-amu.fr/files/charte_ant-plagiat_amu.pdf`

2.1 Lecture du fichier de données d'apprentissage

Dans le fichier `data_utils.py`, complétez la fonction `load_data(filename)` qui prend en argument une chaîne de caractères désignant un nom de fichier de données au format `npz` et renvoie 3 `nd-arrays` : une matrice `X_labeled`, un vecteur `y_labeled` et une matrice `X_unlabeled` correspondant respectivement aux exemples étiquetés, à leurs étiquettes et aux exemples non-étiquetés. Fonction utile : `numpy.load`.
Pour débugguer et vérifier cette fonction, complétez le script `exp_load_data.py` (et lancez-le progressivement à chaque fois) :

- appelez la fonction `load_data` avec le nom du fichier qui contient les données d'apprentissage pour les charger dans des variables `X_labeled`, `y_labeled` et `X_unlabeled` ;
- pour chacune des trois variables, affichez (avec `print`) son type, son nombre de dimensions, son format ;
- affichez les 5 premières valeurs de `y_labeled` et les 5 dernières ;
- affichez les 2 premiers coefficients de la première ligne et de la dernière ligne de chaque matrice, et affichez de même le dernier coefficient de la première et la dernière ligne.

Fonctions utiles : `type, numpy.ndarray.ndim, numpy.ndarray.shape, print`.
On fait couramment des petites expériences comme précédemment pour débuguer son code. On peut ensuite consolider la démarche en élaborant des tests. Une façon pratique de procéder pour les novices se fait en deux temps : tout d'abord, faites un script comme le précédent, comme vous en avez peut-être déjà l'habitude, pour faire tourner votre code sur quelques exemples ; puis transformez tout ou partie de ce code en tests, ce qui permet de garder ensuite ces tests et de les exécuter régulièrement pour vérifier que le code est à jour. Complétez le fichier `test_data_utils`, pour qu'il y ait une classe `TestLoadData` pour tester la fonction `load_data`, avec une méthode de test correspondant à chaque étape de l'expérience précédente, comme cela est prévu dans le code fourni. Certains tests sont fournis pour que vous puissiez les comprendre et prendre exemple dessus. Fonctions utiles : `assert_array_equal, assert_array_almost_equal` (dans `numpy.testing`).
Principales compétences acquises : cette partie vous a en principe permis de maitriser la lecture élémentaire de fichiers de données numpy (format `.npz`) ; de voir une façon de tester d'une fonction (quand on écrit une fonction, il est utile de la tester simplement avant de l'utiliser dans des contextes plus complexes) ; de revoir la notion de fonction (définition et appel).

2.2 Manipulation des données d'apprentissage

Vous vous demandez peut-être à quoi ressemblent les données d'apprentissage que l'on vous a fournies. Tracez l'histogramme des années y présentes dans les données d'apprentissage afin de voir combien la base d'apprentissage comporte de chansons pour chaque année entre 1922 et 2011. Le code permettant de tracer cet histogramme est à écrire dans le script `exp_visualization.py` et vous sauvegarderez l'histogramme dans un fichier `hist_year.png` pour pouvoir l'inclure dans votre rapport. Fonctions utiles : `hist` et `savefig` (dans `matplotlib.pyplot`).
Principales compétences acquises : visualisation de données via un histogramme et sauvegarde d'une figure dans un fichier image.

2.3 Mélanger les données d'apprentissage

Une pratique courante lorsque l'on manipule des données d'apprentissage que l'on suppose *i.i.d.* est de les mélanger soi-même. Dans le cas de nos exemples d'apprentissage, il faudra mélanger les lignes de `X_labeled` et de `y_labeled` pour obtenir un ordre aléatoire des exemples, tout en gardant la correspondance entre `X_labeled` et de `y_labeled` (pour chaque entier i, la i-ième année dans `y_labeled` doit correspondre à la chanson de la i-ième ligne de `X_labeled`). Soit $N, M \in \mathbb{N}$.

- Définissez mathématiquement une permutation de l'ensemble $\{1, ..., N\}$.
- Soit $\mathbf{X} \in \mathbb{R}^{N \times M}$ et $\mathbf{y} \in \mathbb{R}^N$. Définissez mathématiquement $\tilde{\mathbf{X}} \in \mathbb{R}^{N \times M}$ et $\tilde{\mathbf{y}} \in \mathbb{R}^N$ qui sont des versions « mélangées » de $\mathbf{X}$ et $\mathbf{y}$, respectivement, en utilisant une unique permutation pour mélanger aléatoirement et de façon identique les lignes de $\mathbf{X}$ et les coefficients de $\mathbf{y}$.
- Écrivez la fonction `randomize_data(X, y)` (dans le fichier `data_utils.py`) qui prend en argument une matrice `X` et un vecteur `y` et renvoie leurs versions mélangées aléatoirement `Xr`, `yr`. Fonction utile : `numpy.random.permutation`.
- utilisez `test_randomize_data` pour débugguer cette fonction en générant les données simples $X = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & \cdots & \cdots & \vdots \\ \vdots & & \vdots & \\ 16 & 17 & 18 & 19 \end{pmatrix}$ et $y = \begin{pmatrix} 0 \\ -4 \\ -8 \\ -12 \\ -16 \end{pmatrix}$ en appelant `randomize_data` et en affichant les données de départ et le résultat. Les tests de la fonctoin `randomize_data` vous sont par ailleurs fournis : vous les utiliserez non seulement pour valider votre code mais également, en étudiant le code du

test, pour comprendre ce qui est testé. Vous pourrez en particulier voir comment il est possible de tester qu'un fonction renvoie bien une erreur quand cela est attendu.

3 Validation croisée

Mesure de performance. En pratique, pour mesurer la performance d'une fonction de prédiction $f : \mathcal{X} \mapsto \mathcal{Y}$ à partir d'un ensemble $\mathcal{S} \subset \mathcal{X} \times \mathcal{Y}$, on calculera l'erreur quadratique moyenne

$$E_{\mathcal{S}}(f) = \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x},y)\in\mathcal{S}} (y - f(x))^2 \tag{1}$$

où $|\mathcal{S}|$ désigne le cardinal de $\mathcal{S}$. La racine carrée de $E_{\mathcal{S}}(f)$ donne un ordre de grandeur de la précision de l'estimation sur une année.

Naïvement, vous pourriez utiliser toutes les données étiquetées $\mathcal{S}_l$ pour apprendre la fonction de prédiction et laisser l'évaluateur calculer la performance de votre fonction sur les données non-étiquetées. Avec cette approche, vous ignorez complètement la performance de votre fonction avant les résultats de l'évaluateur. Ceci pose problème. Par exemple, si vous trouvez plusieurs fonctions possibles, vous n'êtes pas en mesure de sélectionner vous-même la plus performante ; de même, vous n'êtes pas en mesure de savoir si une fonction coûteuse en temps de calcul donne de meilleurs résultats qu'une fonction plus rapide. Le principe de validation croisée permet d'estimer soi-même les performances de prédiction d'une fonction à partir des données disponibles. Le protocole le plus simple consiste à :

1. diviser les données étiquetées $\mathcal{S}_l$ en deux ensembles formant une partition : un ensemble d'apprentissage $\mathcal{S}_{train}$ et un ensemble de validation $\mathcal{S}_{valid}$ (avec donc $\mathcal{S}_{train} \cup \mathcal{S}_{valid} = \mathcal{S}_l$ et $\mathcal{S}_{train} \cap \mathcal{S}_{valid} = \emptyset$;
2. apprendre la fonction f sur l'ensemble d'apprentissage S_{train}, i.e. calculer $f = \mathcal{L}(\mathcal{S}_{train})$;
3. estimer la performance sur l'ensemble de validation $E_{\mathcal{S}_{valid}}(f)$.

De cette façon, les données utilisées pour estimer la performance sont des exemples nouveaux qui n'ont pas été utilisés lors de l'apprentissage, à la manière des exemples nouveaux que l'évaluateur utilisera.

En utilisant le code fourni et en le plaçant dans `data_utils.py`, écrivez une fonction `split_data(X, y, ratio)` qui prend en arguments une matrice X d'exemples (les lignes de X) et un vecteur y d'étiquettes associées, et en extrait une partition en deux ensembles d'exemples étiquettés (X1, y1) et (X2, y2) formant une partition aléatoire de (X, y). Le dernier argument ratio est un nombre entre 0 et 1 correspondant à la proportion d'exemples à mettre dans (X1, y1). Dans cette fonction, vous utiliserez la fonction `randomize_data` vue précédemment pour mélanger les exemples avant de construire la partition. Vous testerez cette fonction avec les tests unitaires fournis, à mettre dans `test_data_utils.py`. Ils permettent de vérifier que la partition obtenue est correcte.

4 Algorithmes d'apprentissage élémentaires

Dans cette partie, vous allez étudier et coder plusieurs algorithmes d'apprentissage simples pour la prédiction de l'année d'une chanson. Chaque algorithme d'apprentissage sera une façon spécifique de trouver une fonction de prédiction $f : \mathcal{X} \mapsto \mathcal{Y}$ à partir d'un ensemble d'apprentissage $\mathcal{S}_{train}$. Un algorithme d'apprentissage $\mathcal{L}$ sera codé en Python par une méthode `fit(X,y)` prenant en argument un ensemble d'apprentissage et calculant les paramètres qui caractérisent la fonction de prédiction obtenue. La fonction de prédiction sera codée par une méthode `predict(X)` qui prend en argument un ensemble d'exemples (en général différents des exemples d'apprentissage) et renvoie le vecteur des étiquettes prédites. Pour chaque nouvelle approche, une nouvelle classe sera créée avec les méthodes `fit` et `predict` spécifiques à l'approche. Cette architecture avec une classe et ces deux méthodes `fit` et `predict` est inspirée de la librairie `scikit-learn`.
Dans cette séance, vous allez aborder deux stratégies pour apprendre une fonction de prédiction f linéaire (cf. 4.1) :

— En 4.2, la régression linéaire aux moindres carrés ;
— En **??**, un ensemble de stratégies très simples, qui donnent des fonctions f peu performantes mais néanmoins intéressantes pour interpréter les résultats et comparer les différentes stratégies.

Les deux stratégies sont indépendantes : n'hésitez pas à passer de l'une à l'autre selon votre avancement, vos envies, etc. D'autres stratégies, plus élaborées, seront proposées en section 6.

4.1 Régression linéaire

Pour rappel, la régression linéaire consiste à chercher une fonction linéaire, i.e. de la forme

$$f_{\mathbf{w},b} : \mathcal{X} \mapsto \mathcal{Y} \tag{2}$$

$$\mathbf{x} \mapsto \mathbf{x}^T\mathbf{w} + b \tag{3}$$

où $\mathbf{w} \in \mathcal{X}$ et $b \in \mathbb{R}$. Nous utiliserons une classe générique `LinearRegression` qui a deux attributs `w` et `b` et une méthode `predict` qui implémente l'équation 3 pour l'étape de prédiction. Cette classe est donnée intégralement dans le fichier `linear_regression.py`.
Nous allons étudier plusieurs algorithmes d'apprentissages : chaque algorithme estime d'une façon différente les paramètres `w` et `b` à partir d'un ensemble d'apprentissage. Pour chaque approche, vous utiliserez une nouvelle classe qui hérite de `LinearRegression` et qui aura une méthode spécifique `fit`.

4.2 Régression linéaire aux moindres carrés

On parle de régression linéaire aux moindres carrés lorsque l'on minimise l'erreur quadratique moyenne pour estimer $\mathbf{w} \in \mathcal{X}$ et $b \in \mathbb{R}$. En considérant un ensemble d'apprentissage $\mathcal{S}_{train}$, le problème s'écrit donc

$$(\hat{\mathbf{w}, b}) = \arg\min_{(\mathbf{w},b)} E_{\mathcal{S}_{train}}(f_{\mathbf{w},b}) \tag{4}$$

Résoudre ce problème permet donc d'estimer un jeu de paramètres $(\hat{\mathbf{w}}, \hat{b})$qui caractérise une fonction de prédiction $f_{\hat{\mathbf{w}},\hat{b}}$: on voit qu'on apprend bien une fonction de prédiction à partir d'un ensemble d'apprentissage. La résolution fait intervenir la pseudo-inverse $X^+ = (X^TX)^{-1}X^T$ de X. L'algorithme 1 synthétise la procédure d'apprentissage, qui consiste à calculer la pseudo-inverse après avoir ajouté à X un vecteur de taille N composé de 1 afin d'apprendre simultanément tous les paramètres.

Algorithm 1 Apprentissage de la fonction de régression linéaire aux moindres carrés

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

Dans la classe `LinearRegressionLeastSquares` du fichier `linear_regression.py` fourni, complétez la méthode `fit(X, y)` qui prend en argument un ensemble d'apprentissage (matrice `X` et vecteur `y`) et calcule les paramètres de la fonction de régression linéaire aux moindres carrés en appliquant l'algorithme 1. Fonction utile : la pseudo-inverse est donnée par `numpy.linalg.pinv`.

4.3 Algorithmes simplistes : des estimateurs constants

Attention ! Les approches données ici sont tellement simplistes qu'elles peuvent vous déstabiliser, vous paraitre stupides et inutiles. Avec un peu de recul, vous verrez que ce n'est pas le cas.
On peut adopter une des trois stratégies suivantes pour construire une fonction de prédiction simpliste :

- renvoyer toujours l'année moyenne calculée sur l'ensemble d'apprentissage ;
- renvoyer toujours l'année médiane calculée sur l'ensemble d'apprentissage ;
- renvoyer toujours l'année la plus fréquente (ou majoritaire) dans l'ensemble d'apprentissage.

Ce sont des stratégies très simples et rapides, dont la logique est loin d'être absurde. On peut se demander quel est le taux d'erreur pour chaque stratégie, et lequel de ces choix est le meilleur.

On souhaite trouver une fonction de prédiction constante, de la forme

$$f_b : \mathcal{X} \mapsto \mathcal{Y} \tag{5}$$

$$\mathbf{x} \mapsto b$$

où $b \in \mathbb{R}$. Il n'y a donc qu'un paramètre $b \in \mathbb{R}$ à apprendre sur un ensemble d'apprentissage : cela revient à apprendre uniquement le coefficient constant d'une régression linéaire du type 3, en fixant $\mathbf{w} = 0$.
Les algorithmes 2, 3 et 4 décrivent les procédures d'apprentissage pour ces trois stratégies. La fonction *median* de l'algorithme 3 calcule la médiane des coefficients d'un vecteur et la fonction *histogram* de l'algorithme 4 renvoie l'histogramme $\mathbf{h} \in \mathbb{N}^p$ des valeurs d'un vecteur $\mathbf{y}$ ainsi que les valeurs $a \in \mathbb{R}^p$ correspondantes, p désignant le nombre de valeurs différentes dans y.

Remarque : l'apprentissage est ici très peu coûteux (calcul de la moyenne, la médiane ou l'étiquette majoritaire) : la complexité algorithmique est en $\mathcal{O}(N)$ pour un ensemble d'apprentissage de taille N.

Dans `linear_regression.py` écrivez la méthode `fit(X, y)` des classes `LinearRegressionMean`, `LinearRegressionMedian` et `LinearRegressionMajority` correspondant aux approches données par les algorithmes 2, 3 et 4 respectivement. Remarquez que par souci d'homogénéité entre tous les algorithmes

Algorithm 2 Apprentissage de l'estimateur constant moyen

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)
  $b \leftarrow \frac{1}{N} \sum_{n=0}^{N-1} \mathbf{y}(n)$
  return $\mathbf{w}, b$

Algorithm 3 Apprentissage de l'estimateur constant médian

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)
  $b \leftarrow \text{median}(\mathbf{y})$
  return $\mathbf{w}, b$

d'apprentissage, nous avons conservé l'argument X même s'il n'est pas utilisé ensuite.

Vous verrez dans le fichier `test_linear_regression.py` que l'on peut créer des tests simples, sur de petites données élémentaires, afin de tester et corriger le code.

5 Expérience avec ressources limitées

Vous avez un ou plusieurs algorithmes d'apprentissage : lequel est le meilleur et dans quelles situations ? Il s'agit maintenant de les mettre en application et d'évaluer leurs performances à travers une expérience. Il est courant que les ressources disponibles soient limitées pour réaliser la phase d'apprentissage :

- les données disponibles peuvent être en nombre limité : par exemple, si peu de données sont fournies ; si leur étiquetage est coûteux ; si la mémoire disponible pour les stocker est limitée ;
- le temps de calcul peut être limité : par exemple, pour ne pas décharger la batterie sur un portable.

Il convient donc de bien maîtriser les conséquences que de telles limitations peuvent avoir sur la qualité de l'apprentissage. Ces dimensions seront un axe d'étude dans la suite de ce mini-projet. Dans cette partie, vous allez mesurer les performances des méthodes déjà codées lorsque la taille de l'ensemble d'apprentissage varie afin de simuler des ressources plus ou moins limitées.

Choisissez une stratégie d'estimation constante que vous avez codée.
Dans un nouveau fichier `exp_training_size.py`, codez l'expérience suivante :

- chargez les données étiquetées depuis le fichier `YearPredictionMSD_100.npz` ;
- séparez-les aléatoirement pour constituer un ensemble d'apprentissage maximal $\mathcal{S}^O_{train}$ (2/3 des données) et un ensemble de validation $\mathcal{S}_{valid}$ (1/3 des données) ;
- construisez l'ensemble $\mathcal{N} = \{2^5, 2^6, ..., 2^11\}$ correspondant à la taille des ensembles d'apprentissage à tester ;
- créez deux vecteurs `train_error` et `valid_error` de la taille de $\mathcal{N}$ afin de stocker les erreurs de prédiction sur les ensembles d'apprentissage et de validation ;
- pour chaque $N \in \mathcal{N}$,
  - construisez $\mathcal{S}_{train}$ en sélectionnant les N premiers exemples de $\mathcal{S}^O_{train}$;
  - appliquez l'algorithme d'apprentissage à l'ensemble d'apprentissage courant $\mathcal{S}_{train}$;
  - en utilisant la fonction de prédiction obtenue, estimez les étiquettes des exemples de $\mathcal{S}_{valid}$;
  - calculez l'erreur quadratique moyenne sur l'ensemble de validation $\mathcal{S}_{valid}$ et stockez-la dans `valid_error` ;
  - estimez également l'erreur de prédiction sur $\mathcal{S}_{train}$ et stockez-la dans `train_error`.
- sauvegardez $\mathcal{N}$, `valid_error` et `train_error` dans un fichier `.npz`.

Fonctions utiles : `numpy.savez`.

Complétez le code de `exp_training_size.py` pour estimer également les performances des autres stratégies d'estimation constante en utilisant exactement les mêmes ensembles d'apprentissage et de validation. Les vecteurs `valid_error` et `train_error` deviendront des matrices dont chaque colonne

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 $\mathbf{w}, b$

contiendra les performances d'un estimateur constant.

Complétez le code de `exp_training_size.py` pour estimer également les performances de la fonction de régression linéaire aux moindres carrés en utilisant exactement les mêmes ensembles d'apprentissage et de validation, et en ajoutant une nouvelle colonne aux matrices `valid_error` et `train_error`.

Dans un fichier `plot_exp_training_size.py`, chargez les résultats depuis votre fichier `.npz` et affichez sur une même figure :

— la performance des méthodes de prédiction sur l'ensemble de validation en fonction de la taille de l'ensemble d'apprentissage, en traits pleins, avec une couleur par méthode ;
— la performance des méthodes sur l'ensemble d'apprentissage en fonction de la taille de l'ensemble d'apprentissage, en pointillés, avec les mêmes couleurs que précédemment.

Vous pouvez choisir des axes logarithmiques grâce aux fonctions `semilogx`, `semilogy` ou `loglog`. Vous ajouterez un titre, des noms aux axes, une légende et sauvegarderez la figure au format `.png`.
Expliquez les tendances obtenues. Comparez notamment l'allure des deux courbes relatives à la régression linéaire aux moindres carrés. Quel intérêt voyez-vous à étudier la fonction de prédiction constante ?

On souhaite mesurer le temps de calcul des fonctions d'apprentissage. Pour cela, on enregistre le temps donné par une horloge juste avant et juste après l'appel à la fonction et on calcule la différence.
Étudiez la documentation du module time (`https://docs.python.org/3.8/library/time.html`) pour sélectionner les fonctions d'horloge appropriées.
Modifiez ensuite votre fichier `exp_training_size.py` pour mesurer les temps de calcul de chaque méthode en fonction de la taille de l'ensemble d'apprentissage. Vous stockerez les temps de calcul obtenus dans une matrice `learning_time` de dimension similaire aux matrices d'erreur et la sauvegarderez dans votre fichier `.npz`.
Modifiez ensuite votre fichier `plot_exp_training_size.py` pour afficher dans une nouvelle figure le temps de calcul de chaque méthode en fonction de la taille de l'ensemble d'apprentissage et sauvergardez la figure au format `.png`.
Les tendances obtenues correspondent-elles aux coûts computationnels annoncés ?

6 Algorithmes avec régularisation

Problème de sur-apprentissage. Vous avez vu que lorsque le nombre d'exemples d'apprentissage est faible, la régression linéaire au moindres carrés $\mathbf{w}_{LS} \leftarrow (\mathbf{X}_{train}^T \mathbf{X}_{train})^{-1} \mathbf{X}_{train}^T \tilde{\mathbf{y}}$ ne donne pas une solution satisfaisante : l'erreur en généralisation calculée sur l'ensemble de validation est très élevée. L'écart important entre l'erreur d'apprentissage et l'erreur de validation fait ressortir un phénomène de sur-apprentissage des 91 paramètres du modèle.

Régularisation. La régularisation est un mécanisme permettant de limiter le sur-apprentissage en favorisant certaines solutions et en pénalisant les autres. Dans cette séance, deux types de régularisation seront vues : régularisation ridge et régularisation parcimonieuse. Chaque régularisation s'accompagne d'un hyper-paramètre à ajuster, permettant de contrôler l'importance de la régularisation. L'ajustement optimal de cet hyper-paramètre fera également partie des objectifs du projet.

Solutions parcimonieuses. Notre principale stratégie contre le sur-apprentissage consistera à chercher des solutions linéaires $f_{\mathbf{w},b} : \mathbf{x} \mapsto \mathbf{x}^T\mathbf{w} + b$ dites parcimonieuses, c'est-à-dire avec des vecteurs $\mathbf{w}$ contenant beaucoup de zéros. En limitant le nombre de valeurs de $\mathbf{w}$ non-nulles, on limite la complexité du modèle (cette notion de complexité peut s'apparenter au nombre de paramètres, au nombre de degrés de liberté, à la richesse, à l'expressivité du modèle, etc.). Une solution parcimonieuse a d'autres avantages qui, selon les contextes, portent des noms différents : réduction de dimension, sélection de variables, compression, échantillonnage compressif, etc.
La quête d'algorithmes permettant de trouver des solutions parcimonieuses a suscité de nombreux efforts car le problème est difficile d'un point de vue calculatoire (le problème optimal est de la famille des problèmes NP-difficiles). Vous allez vous concentrer dans cette séance sur une famille d'algorithmes qui font référence : les algorithmes gloutons, parmi lesquels Matching Pursuit. Ces algorithmes partent d'une solution $\mathbf{w}$ nulle et ajoutent les coefficients non-nuls un par un par une suite d'optimisations locales, selon le principe de sélection progressive que vous aurez peut-être vu en statistiques. À noter qu'il existe d'autres approches pour trouver des solutions parcimonieuses telles que les algorithmes d'optimisation convexes qui optimisent un critère global en affinant $\mathbf{w}$ par itérations successives.
Le travail demandé consiste à mettre en œuvre au moins un algorithme d'estimation parcimonieuse parmi ceux proposés, à le tester et, sur les données du problème, à évaluer l'erreur en validation et l'erreur en apprentissage pour $|\mathcal{S}_{train}| = 500$.

Nous nous concentrons ici sur les algorithmes seulement, en dehors de la problématique applicative de régression linéaire. Les algorithmes seront codés dans le fichier `algorithms.py` et les tests de base sont fournis dans `test_algorithms.py`.

6.1 Solution avec régularisation de Tikhonov (*ridge regression*)

La régularisation de Tikhonov consiste à ajouter un terme de pénalité en $\|\mathbf{w}\|_2^2$ dans la fonctionnelle à minimiser. Le poids de cette pénalité par rapport au terme d'erreur des moindres carrés est ajusté par une constante $\lambda_{rigde} > 0$ à fixer au préalable. Lorsque l'on régularise ainsi un problème de régression linéaire aux moindres carrés, on parle aussi de régression linéaire d'arête (*linear ridge regression*). Cette régularisation ne fournit pas une solution parcimonieuse mais nous l'utilisons car elle est très utilisée.

Reformulation du problème. Pour $\lambda_{ridge} > 0$ fixé, le problème d'optimisation pour effectuer l'apprentissage s'écrit

$$\arg \min_{\mathbf{w} \in \mathbb{R}^M} \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \frac{\lambda_{ridge}}{2} \|\mathbf{w}\|_2^2 \tag{6}$$

La solution de ce problème donne une erreur d'apprentissage moins bonne que celle des moindres carrés mais une erreur de généralisation généralement meilleure. Le sur-apprentissage est ainsi limité.

Solution. Le problème a une solution analytique qui s'obtient en annulant le gradient par rapport à $\mathbf{w}$, comme dans le cas de la régression linéaire aux moindres carrés. L'algorithme 5 décrit la procédure.

Algorithm 5 Ridge regression (régularisation de Tikhonov)

Require: $\mathbf{X} \in \mathbb{R}^{N \times M}$, $\mathbf{y} \in \mathbb{R}^N$, $\lambda_{ridge} > 0$
  $\mathbf{w} \leftarrow (\mathbf{X}^T\mathbf{X} + \lambda_{ridge}\mathbf{I}_M)^{-1}\mathbf{X}^T\mathbf{y}$
  return $\mathbf{w}$

Écrivez une fonction `ridge_regression(X, y, lambda)` qui prend en argument le coefficient de pénalisation `lambda` et un ensemble d'apprentissage donné par la matrice X et le vecteur y, et renvoie la solution w obtenue en appliquant l'algorithme 5.

6.2 Les algorithmes gloutons

Les algorithmes gloutons fournissent une solution sous-optimale à un problème global généralement difficile. Dans le cas de la recherche d'une solution parcimonieuse, choisir les valeurs non-nulles dans $\mathbf{w}$ est un problème combinatoire difficile et les méthodes gloutonnes sélectionnent un par un ces coefficients. Chaque itération comporte deux étapes : sélection du meilleur coefficient et mise à jour de $\mathbf{w}$. Les algorithmes ont donc la forme suivante :

- initialiser $\mathbf{w}$ au vecteur nul $\mathbf{O}_M$ de dimension M ;
- initialiser le vecteur d'erreur résiduelle $\mathbf{r}$ à $\mathbf{y}$;
- répéter
- sélectionner la composante $\hat{m} \in \{0, ..., M-1\}$ pour réduire au mieux l'erreur résiduelle ;
- mettre à jour $\mathbf{w}$ et $\mathbf{r}$.

Les algorithmes présentés ici se distinguent par leur complexité algorithmique et la qualité de la solution, c'est-à-dire l'erreur d'apprentissage obtenue pour un nombre fixé de coefficients non-nuls de $\mathbf{w}$:

- l'algorithme Matching Pursuit (MP, section 6.2.1) est le plus rapide et fournit la moins bonne solution ;
- l'algorithme Orthogonal Matching Pursuit (OMP, section 6.2.2) est moins rapide que MP et fournit une meilleure solution grâce à un meilleur critère de mise à jour que MP.

Dans un contexte de ressources limitées, les deux algorithmes sont donc intéressants à étudier afin de trouver un bon compromis entre temps de calcul et erreur.

6.2.1 L'algorithme Matching Pursuit (MP)

Pour un entier k_{max} fixé à l'avance, l'algorithme Matching Pursuit fournit une solution $\mathbf{w}$ qui contient au plus k_{max} composantes non-nulles. Il est donné par l'algorithme 6 et est expliqué ci-dessous.

Résiduel. L'erreur résiduelle $\mathbf{r}$ est la différence entre le vecteur à modéliser $\mathbf{y}$ et sa modélisation $\mathbf{X}\mathbf{w}$ au cours de l'algorithme. Cette notion fournit un invariant de l'algorithme : $\mathbf{r} = \mathbf{y} - \mathbf{X}\mathbf{w}$. Il est notamment vérifié à l'initialisation lorsque $\mathbf{w} = \mathbf{0}_M$ et $\mathbf{r} = \mathbf{y}$ (ligne 1 et 3) et à la fin de chaque itération.

Algorithm 6 Matching Pursuit

Require: $\mathbf{X} = [\mathbf{a}_0, ..., \mathbf{a}_{M-1}] \in \mathbb{R}^{N \times M}$ t.q. $\forall m, \|\mathbf{a}_m\|_2 = 1, \mathbf{y} \in \mathbb{R}^N, k_{max} \in \mathbb{N}^*$
1. $\mathbf{r} \leftarrow \mathbf{y}$ (Initialisation du résiduel)
2. $\mathbf{w} \leftarrow \mathbf{0}_M$ (Initialisation de $\mathbf{w}$)
3. **for** $k = 1$ à k_{max} **do**
4. $\quad \forall m, c_m \leftarrow \langle \mathbf{a}_m, \mathbf{r} \rangle$ (Calcul des corrélations du résiduel et des composantes)
5. $\quad \hat{m} \leftarrow \arg\max_m |c_m|$ (Sélection de la composante la plus corrélée)
6. $\quad \mathbf{w}(\hat{m}) \leftarrow \mathbf{x}(\hat{m}) + c_{\hat{m}}$ (Mise à jour de la décomposition)
7. $\quad \mathbf{r} \leftarrow \mathbf{r} - c_{\hat{m}} \mathbf{a}_{\hat{m}}$ (Mise à jour du résiduel)
8. **end for**
9. **return** $\mathbf{w}$

Étape de sélection (lignes 4 et 5). On souhaite sélectionner la compsante $\mathbf{a}_m$ qui permet de réduire au mieux l'énergie du résiduel. En optimisant par rapport à l'indice m et au poids c_m, on résout donc

$$\arg\min_m \min_{c_m \in \mathbb{R}} \|\mathbf{r} - c_m \mathbf{a}_m\|_2^2 \tag{7}$$

Pour m fixé, l'optimisation intérieure par rapport à c_m se résout en annulant la dérivée de la fonctionnelle :

$$0 = \frac{d}{dc_m} \|\mathbf{r} - c_m \mathbf{a}_m\|_2^2 = -2(\mathbf{a}_m^T \mathbf{r} - c_m) \text{ car } \|\mathbf{a}_m\|_2 = 1 \tag{8}$$

donc $c_m = \mathbf{a}_m^T \mathbf{r} = \langle \mathbf{a}_m, \mathbf{r} \rangle$. Le problème 7 devient

$$\begin{aligned} \arg\min_m \|\mathbf{r} - \langle \mathbf{a}_m, \mathbf{r} \rangle \mathbf{a}_m\|_2^2 &= \arg\min_m \|\mathbf{r}\|_2^2 + \|\mathbf{a}_m\|_2^2 \langle \mathbf{a}_m, \mathbf{r} \rangle^2 - 2\langle \mathbf{a}_m, \mathbf{r} \rangle^2 \\ &= \arg\min_m \|\mathbf{r}\|^2 - \langle \mathbf{a}_m, \mathbf{r} \rangle^2 \\ &= \arg\max_m \langle \mathbf{a}_m, \mathbf{r} \rangle^2 \\ &= \arg\max_m |c_m| \end{aligned} \tag{9}$$

On retrouve bien les étapes de sélection du vecteur le plus corrélé données aux lignes 4 et 5.

Étape de mise à jour (lignes 6 à 7). L'énergie du résiduel est réduite de façon optimale au sens de l'équetion 7 en posant $\mathbf{r} \leftarrow \mathbf{r} - c_{\hat{m}} \mathbf{a}_{\hat{m}}$ (ligne 7). La mise à jour du vecteur $\mathbf{w}$ s'obtient via l'invariant

$$\mathbf{y} = \mathbf{r} + \mathbf{X}\mathbf{W} = (\mathbf{r} - c_{\hat{m}} \mathbf{a}_{\hat{m}}) + (c_{\hat{m}} \mathbf{a}_{\hat{m}} + \mathbf{X}\mathbf{w}) = (\mathbf{r} - c_{\hat{m}} \mathbf{a}_{\hat{m}}) + \mathbf{X}(\mathbf{w} + c_{\hat{m}} \mathbf{e}_m) \tag{10}$$

où $\mathbf{e}_m$ est le m-ième vecteur de la base canonique de $\mathbb{R}^M$. La dernière partie de cette équation correspond bien à la somme du nouveau résiduel $\mathbf{r} - c_{\hat{m}} \mathbf{a}_{\hat{m}}$ et de la nouvelle décomposition $\mathbf{X}(\mathbf{w} + c_{\hat{m}} \mathbf{e}_m)$ (ligne 6).

Propriété de MP. Vous pouvez vérifier qu'à chaque étape, le résiduel $\mathbf{r}$ est orthgonal au dernier vecteur sélectionné $\mathbf{a}_{\hat{m}}$: l'algorithme ne sélectionne jamais deux fois de suie la même composante. En revanche, il peut sélectionner deux fois la même composante à plusieurs itérations d'intervalle.

Écrivez une fonction `mp(X, y, n_iter)` qui prend en argument un entier `n_iter` et un ensemble d'apprentissage donné par la matrice X et le vecteur y, et applique l'algorithme 6 pour renvoyer un couple d'objets `(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 (le dernier coefficient est l'énergie du résiduel à la fin de l'algorithme). Ce dernier vecteur est utile pour tester la fonction en vérifiant que cette énergie décroit.

6.2.2 L'algorithme Orthogonal Matching Pursuit (OMP)

OMP est une variante de MP où l'étape de mise à jour est améliorée. Il est décrit par l'algorithme 7. On y construit explicitement l'ensemble Ω des indices des coefficients non-nuls de $\mathbf{w}$, appelé support de $\mathbf{w}$ et l'étape de mise à jour utilise la sous-matrice $\mathbf{X}_\Omega$ correspondant aux colonnes de $\mathbf{X}$ indexées par Ω.

Résiduel. Comme pour MP, l'erreur résiduelle $\mathbf{r}$ caractérise l'invariant de l'algorithme $\mathbf{r} = \mathbf{y} - \mathbf{X}\mathbf{w} = \mathbf{y} - \mathbf{X}_\Omega \mathbf{w}(\Omega)$, au moment de l'initialisation ($\mathbf{w} = \mathbf{0}_M$, $\mathbf{r} = \mathbf{y}$ et $\Omega = \emptyset$) et à la fin de chaque itération.

Étape de sélection (lignes 5 et 6). Cette étape est la même que pour MP et sélectionne la composante qui permet de réduire l'énergie du résiduel, c'est-à-dire la solution du problème 7.

Algorithm 7 Orthogonal Matching Pursuit

Require: $\mathbf{X} = [\mathbf{a}_0, ..., \mathbf{a}_{M-1}] \in \mathbb{R}^{N \times M}$ t.q. $\forall m, \|\mathbf{a}_m\|_2 = 1, \mathbf{y} \in \mathbb{R}^N, k_{max} \in \mathbb{N}^*$
  $\mathbf{r} \leftarrow \mathbf{y}$ (Initialisation du résiduel)
  $\mathbf{w} \leftarrow \mathbf{0}_M$ (Initialisation de la décomposition)
  $\Omega \leftarrow \emptyset$ (Initialisation du support de décomposition)
  for $k = 1$ à k_{max} **do**
    $\forall m, c_m \leftarrow \langle \mathbf{a}_m, \mathbf{r} \rangle$ (Calcul des corrélations du résiduel et des composantes)
    $\hat{m} \leftarrow \arg\max_m |c_m|$ (Sélection de la composante la plus corrélée)
    $\Omega \leftarrow \Omega \cup \{\hat{m}\}$ (Mise à jour du support)
    $\mathbf{w}(\Omega) \leftarrow \mathbf{X}_\Omega^+ \mathbf{y}$ (Mise à jour de la décomposition)
    $\mathbf{r} \leftarrow \mathbf{y} - \mathbf{X}\mathbf{w}$ (Mise à jour du résiduel)
  end for
  return $\mathbf{w}$

Étape de mise à jour (lignes 7, 8 et 9). Les règles de mise à jour de MP réduisent l'énergie du résiduel de façon optimale en ne jouant que sur la contribution de la composante sélectionnée à l'itération courante. Pour une meilleure réduction de l'énergie du résiduel, OMP effectue une mise à jour de l'ensemble des coefficients sélectionnés, dont les indices sont stockés dans Ω (ligne 7). Plutôt que s'appuyer sur le problème 7, on résout donc le problème :

$$\arg \min_{\mathbf{u} \in \mathbb{R}^{|\Omega|}} \|\mathbf{y} - \mathbf{X}_\Omega \mathbf{u}\|_2^2 \tag{11}$$

dont la solution (ligne 8 de l'algorithme) est bien connue puisqu'il s'agit d'un problème de régression linéaire aux moindres carrés restreint à la sous-matrice $\mathbf{X}_\Omega$.

Propriété d'OMP. Vous pouvez vérifier qu'à chaque étape, le résiduel $\mathbf{r}$ est orthogonal à l'ensemble des vecteurs déjà sélectionnés : on a $\forall m \in \Omega, \langle \mathbf{a}_m, \mathbf{r} \rangle = 0$. Cette propriété vient de la projection orthogonale effectuée à la ligne 8 et a pour conséquence que le nombre de coefficients non-nuls de $\mathbf{w}$ est exactement égal au nombre d'itérations effectuées.

Écrivez une fonction `omp(X, y, n_iter)` qui prend en argument un entier `n_iter` et un ensemble d'apprentissage donné par la matrice X et le vecteur y, et applique l'algorithme 7 pour renvoyer un couple d'objets `(w, error_norm)` : le vecteur parcimonieux $\mathbf{w}$ 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 (le dernier coefficient est l'énergie du résiduel à la fin de l'algorithme). Ce dernier vecteur est utile pour tester la fonction en vérifiant que cette énergie décroit.

7 Intégration des algorithmes pour la régression linéaire

Vous pouvez remarquer que les algorithmes introduits dans la section précédente ne permettent pas de résoudre directement notre problème de régression pour deux raisons :

— ils n'estiment pas le paramètre constant b (ils estiment un modèle linéaire plutôt qu'affine) ;

— ils nécessitent une matrice $\mathbf{X}$ dont les colonnes $\mathbf{a}_m$ sont normalisées ($\|\mathbf{a}_m\|_2 = 1$).

L'adaptation à notre problème consiste à ajouter une étape de pré-traitement et une étape de post-traitement autour de l'appel à l'algorithme.

Pré-traitement des données. Tout d'abord, le coefficient b est appris dans la phase de pré-traitement :

$$b = \frac{1}{N} \sum_{n=0}^{N-1} \mathbf{y}(n) \tag{12}$$

puis on effectue un centrage des données $\mathbf{y}$ en posant

$$\tilde{\mathbf{y}} = \mathbf{y} - b \tag{13}$$

Ensuite, il faut normaliser les colonnes $\mathbf{a}_m$ de la matrice $\mathbf{X} = [\mathbf{a}_0, ..., \mathbf{a}_{M-1}]$ pour que leur norme euclidienne soit unitaire. On définira donc

$$\forall m \in \{0, ..., M-1\}, \tilde{\mathbf{a}}_m = \frac{\mathbf{a}_m}{\alpha(m)} = \|\mathbf{a}_m\|_2 \tag{14}$$

et

$$\tilde{\mathbf{X}} = [\tilde{\mathbf{a}}_0, ..., \tilde{\mathbf{a}}_{M-1}] \tag{15}$$

Cette normalisation peut être effectuée matriciellement, sans boucle (à faire selon votre niveau).

Dans le fichier `algorithms.py`, complétez la fonction `normalize_dictionnary(X)` qui prend en argument un dictionnaire $\mathbf{X}$ et renvoie sa version normalisée $\tilde{\mathbf{X}}$ ainsi que le vecteur α contenant les coefficients de normalisation.

Post-traitement du résultat. L'algorithme appliqué à $\tilde{\mathbf{X}}$ et $\tilde{\mathbf{y}}$ renvoie un vecteur $\tilde{\mathbf{w}}$ tel que $\tilde{\mathbf{y}} \approx \tilde{\mathbf{X}}\tilde{\mathbf{w}}$. Par un changement de variable dans $\tilde{\mathbf{X}}\tilde{\mathbf{w}} = \mathbf{X}\mathbf{w}$, on obtient alors la relation qui lie $\tilde{\mathbf{w}}$ et $\mathbf{w}$:

$$\forall m \in \{0, ..., M-1\}, \mathbf{w}(m) = \frac{\tilde{\mathbf{w}}(m)}{\alpha(m)} \tag{16}$$

Cette relation est utile afin de renvoyer $\mathbf{w}$ après l'application de l'algorithme qui fournit $\tilde{\mathbf{w}}$.

Régression linéaire avec un algorithme régularisé. L'algorithme 8 décrit la procédure pour résoudre notre problème de régression avec un des algorithmes présentés (régression ridge, MP, OMP).

Algorithm 8 Régression linéaire avec un algorithme régularisé

Require: $\mathbf{X} = [\mathbf{a}_0, ..., \mathbf{a}_{M-1}] \in \mathbb{R}^{N \times M}, \mathbf{y} \in \mathbb{R}^N$, hyperparamètre P (nom différent selon l'algorithme)
  Estimation de b (par équation 12)
  $\tilde{\mathbf{X}}, \alpha \leftarrow$ normalisation($\mathbf{X}$)
  $\tilde{\mathbf{w}} \leftarrow A(\tilde{\mathbf{X}}, \tilde{\mathbf{y}}, P)$ (algorithme MP, OMP ou ridge)
  $\mathbf{w} \leftarrow$ rescale($\tilde{\mathbf{w}}, \alpha$) (par équation 16)
  return $\mathbf{w}$

Dans le fichier `linear_regression.py`, selon les algorithmes que vous souhaitez utiliser, complétez les classes `LinearRegressionRidge, LinearRegressionMp` ou `LinearRegressionOmp`.

Au-delà des tests proposés pour vos fonctions et classes, utilisez un script dans un fichier séparé (`exp_xxxx.py`), où vous pouvez :
- fixer arbitrairement le paramètre k_{max} ou λ de votre algorithme ;
- charger les données étiquetées et en sélectionner un sous-ensemble (par exemple 500 exemples) ;
- apprenez sur ces données à l'aide de votre algorithme ;
- contrôler le comportement de votre algorithme en affichant et en commentant l'erreur du résidu en fonction des itérations (valeurs initiales et finales, décroissance, cohérence avec les coefficients de régression obtenus, nombre d'itérations).

En cas de problème, vous pouvez recourir au débogueur de Python via Spyder pour suspendre l'exécution pendant le déroulement de votre algorithme et contrôler pas-à-pas son déroulement (auto-formez-vous avec l'aide en ligne).

8 Expérience pour déterminer les hyper-paramètres

Dans chacun des algorithmes avec régularisation présentés, au moins un paramètre doit être fixé au préalable avant d'appliquer l'algorithme. On parle d'hyperparamètre. Dans les cas ci-dessus, il s'agit :
- pour la régression d'arrête, du coefficient de pénalisation λ ridge ;
- pour MP, OMP et OLS, du nombre d'itérations k_{max} ;

Fixer ces hyperparamètres est en général une étape difficile, qui peut demander un temps de calcul important. La tâche consiste en général à essayer plusieurs valeurs possibles et à comparer les résultats pour sélectionner la valeur donnant la meilleure performance. Ceci se fait généralement automatiquement par validation croisée (voir section 3). On se contentera ici de tracer les performances en fonction des valeurs testées pour visualiser le résultat.

En utilisant les données déjà fournies et en fixant le nombre d'exemples d'apprentissage à 500 (tirés aléatoirement), écrivez un script `exp_hyperparam.py` dans lequel, pour l'algorithme (ou les algorithmes) que vous avez mis en œuvre,
- séparez les données étiquetées en un ensemble d'apprentissage et un ensemble de validation ;
- définissez un ensemble de valeurs à tester pour votre hyperparamètre ;
- pour chaque valeur possible,
  - appliquez votre algorithme en utilisant cette valeur ;
  - calculez l'erreur sur l'ensemble d'apprentissage et sur l'ensemble de validation ;
- tracez sur une même figure les deux erreurs en fonction de la valeur de l'hyperparamètre.

Le choix de l'ensemble des valeurs à tester est à ajuster pour obtenir une courbe intéressante pour l'erreur en validation (vous pouvez prendre des valeurs espacées logarithmiquement pour λ). Commentez vos courbes :

— quelle est la valeur optimale de l'hyperparamètre ?
— comment varient les courbes avant et après cette valeur ?
— comment les deux courbes évoluent-elles comparativement ?

On souhaite maintenant utiliser les algorithmes étudiés sans avoir à ajuster à la main l'hyperparamètre, c'est-à-dire pouvoir avoir des fonctions associées de la forme `algo(X, y)` au lieu de `algo(X, y, hyperparam)`. Comment faire pour que le choix de l'hyperparamètre soit effectué automatiquement à l'intérieur de la fonction ?

9 Estimation des hyperparamètres par validatiton croisée

Vous avez développé une ou plusieurs méthodes qui apprennent/estiment les paramètres w et b d'une fonction de prédiction linéaire. Le vecteur w est estimé en utilisant une régularisation parcimonieuse ou de norme l_2. Ces méthodes dépendent d'un hyperparamètre k_{max} ou λ à fixer et vous avez mis en évidence l'influence de ce paramètre sur les performances, en termes d'erreur en apprentissage et d'erreur en validation.
Les méthodes proposées ne sont donc pas encore complètement automatiques puisqu'il faut fixer la valeur de l'hyperparamètre. C'est ce qu'il vous reste à faire. Pour chaque algorithme, par exemple MP, on souhaite avoir une fonction du type :

```
learn_all_with_mp(X, y)
```

qui ne prend en argument qu'un ensemble étiqueté X, y, sans hyperparamètre tel que le nombre d'itérations `n_iter` (ou λ_{ridge} pour la régression ridge). Une telle fonction va estimer l'hyperparamètre par validation croisée :

— générer des valeurs possibles de l'hyperparamètre `n_iter` ;
— créer des objets `LinearRegressionMp(n_iter)` pour chaque valeur de `n_iter` et les entrainer (méthode `fit`) sur un sous-ensemble d'apprentissage issu de X, y ;
— sélectionner une valeur optimale de l'hyperparamètre via un ensemble de validation issu de X, y et renvoyer l'objet `LinearRegressionMp` initialisé avec cet hyperparamètre et contenant les paramètres `(w, b)` appris.

De cette façon, l'hyperparamètre est sélectionné en fonction des données d'apprentissage (X, y) et s'adapte donc selon les contextes, par exemple si l'ensemble d'apprentissage est plus ou moins grand.

Bonne nouvelle ! Vous avez déjà tous les outils conceptuels, les algorithmes et une bonne partie du code pour répondre à la question « comment estimer mon hyperparamètre ? », il n'y a plus qu'à les intégrer. Le principe d'estimation des hyperparamètres par validation croisée est développé dans la partie 3. Vous pourrez utiliser votre ou vos méthodes d'apprentissage dans la partie 10 pour revenir au problème initial et obtenir des solutions meilleures qu'avec la régression linéaire aux moindres carrés. Vous pourrez alors proposer votre meilleure solution pour prédire les réponses sur l'ensemble non-étiqueté fourni. Vous aurez en outre de l'aide méthodologique pour finaliser proprement votre code (partie 11).

Considérons l'ensemble des données étiquetées disponibles dans le fichier `YearPredictionMSD_100.npz`. Vous allez faire une partition de ces données, non plus en deux mais en trois sous-ensembles :

— un ensemble d'apprentissage pour apprendre les paramètres (w, b) ;
— un ensemble de validation 1 pour apprendre l'hyperparamètre ;
— un ensemble de validation 2 pour estimer l'erreur de prédiction.

De cette manière, vous pouvez suivre la démarche de la séance 3 pour apprendre tous les paramètres et hyperparamètres avec l'ensemble d'apprentissage et l'ensemble de validation 1 ; puis il vous restera des données non utilisées (ensemble de validation 2) pour estimer la capacité de généralisation de votre fonction, c'est-à-dire son erreur de prédiction sur des exemples nouveaux, qui n'ont pas été utilisés dans la phase d'apprentissage des paramètres et hyperparamètres.

En remplaçant `ALGO` pour le nom d'un algorithme, écrivez une fonction `learn_all_with_ALGO(X, y)` qui prend un argument un ensemble d'exemples étiquetés donné par la matrice X et le vecteur y et renvoie l'objet `LinearRegressionALGO` appris de la façon suivante :

— partitionnez l'ensemble (X, y) en un ensemble d'apprentissage et un ensemble de validation 1, dans une proportion de 2/3 et 1/3 respectivement ;
— construisez un ensemble d'environ 20 valeurs possibles pour l'hyperparamètre (à ajuster vous-même intelligemment) ;

— pour chaque valeur possible de l'hyperparamètre, apprenez les paramètres (w, b) à partir de l'ensemble d'apprentissage en utilisant un objet `LinearRegressionALGO`, puis estimez l'erreur sur l'ensemble de validation ;
— sélectionnez la valeur de l'hyperparamètre qui minimise l'erreur de validation ;
— l'hypeparamètre étant fixé à la valeur optimale, vous pouvez ré-entrainer le modèle sur l'ensemble total (X, y) pour obtenir une estimation encore meilleure et renvoyer l'objet obtenu.

Remarques :
— l'ensemble (X,y) passé à cette fonction ne contient pas l'ensemble de validation 2 ;
— une bonne partie du code de la partie 8 peut être « recyclé » pour faire la fonction demandée. C'est une forme de refactoring.

10 Utilisation des nouvelles méthodes dans l'expérience de prédiction avec ressources limitées

Vous disposez désormais d'un ou plusieurs nouveaux algorithmes de régression linéaire.

En reprenant le fichier `plot_exp_training_size.py`, affichez sur un même graphique les performance de prédiction de tous les algorithmes que vous avez, en fonction de la taille de l'ensemble d'apprentissage. L'ensemble $\mathcal{S}_{valid}$ contenant un tiers des données étiquetées correspond à l'ensemble de validation 2. L'ensemble $\mathcal{S}_{train}$ dont la taille varie dans l'expérience correspond à l'ensemble (X, y) qui est divisé dans la fonction `learn_all_with_ALGO` pour obtenir l'ensemble d'apprentissage et l'ensemble de valisation 1. Quel est l'algorithme qui donne les meilleurs résultats lorsque $|\mathcal{S}_{train}| = 500$?

C'est confirmé, votre patron vous demande de lui fournir le prototype Python d'un futur programme pour téléphone portable, qui prédit l'année d'une chanson avec un ensemble d'apprentissage de taille 500 pour limiter la mémoire et la puissance utilisées. De cette façon, chaque utilisateur peut entrer ses 500 chansons préférées et obtenir une prédiction adaptée à la distribution de ses goûts musicaux. Vous devez maintenant convaincre votre patron que vous avez travaillé méthodiquement et que vos résultats sont fiables.
Écrivez une fonction `learn_best_predictor_and_predict_test_data()` qui
— partionne les données étiquetées en un ensemble de taille 500 et un ensemble de validation 2 contenant le reste des données ;
— en utilisant la fonction `learn_all_with_ALGO`, apprend les paramètres [w, b] de la fonction de prédiction à partir de l'ensemble de taille 500 ;
— calcule et affiche l'estimation de la performance en utilisant le reste des données étiquetées ;
— lit les données non-étiquetées dans le fichier `YearPredictionMSD_100.npz` ;
— prédit les étiquettes manquantes, les stocke dans un vecteur `y_test` et sauvegarde ce vecteur dans un fichier `test_prediction_results.npy` via la fonction `np.save` ; les années prédites peuvent être des nombres réels, inutile de les arrondir à des valeurs entières.

11 Refactoring

Le terme refactoring désigne le travail que l'on fait sur le code pour le rendre plus lisible, mieux structuré, plus simple, mieux « écrit », sans changer les fonctionnalités ni en ajouter. Ce travail est réalisé couramment pour de multiples raisons : la première version du code n'est pas toujours bien écrite et est issue d'un cheminement plus ou moins hasardeux lors du développement ; la structure initialement prévue n'est pas exactement la même que celle finalement obtenue ; le développeur a négligé la clarté de certaines parties ; le code doit être réutilisé dans un nouveau contexte et remanié en conséquence ; etc.
Parmi les opérations de refactoring, on peut citer :
— renommer les variables et les fonctions en choisissant des noms plus appropriés ;
— créer de nouvelles fonctions lorsqu'une fonction est trop longue ou pour « factoriser » certains morceaux du code et éviter ainsi que des morceaux identiques apparaissent à différents endroits ;
— simplifier le code par exemple en remplaçant certaines boucles par des opérations matricielles ;
— supprimer les parties du code inutiles (par exemple, les instructions mises en commentaire) ;
— retravailler la structure des fichiers, par exemple en créant des sous-répertoires pour les différents fichiers : dans notre cas, des sous-répertoires `algorithmes/` pour mettre tous vos algorithmes, `experiments/` pour les expériences, `tests/` pour les tests unitaires, `io/` (input-output) pour la lecture et l'écriture de fichiers, etc.
— faire un paquet Python avec notamment un fichier `setup.py`
— vérifier et corriger les commentaires dans le code : supprimer ce qui est redondant avec le code lui-même, ajouter l'information complémentaire nécessaire à la compréhension, etc.

Pour vous aider dans ces opérations, des conseils sont donnés dans le document *PEP 8 – Style Guide for Python Code* dont vous avez déjà entendu parler. Vous y trouverez notamment de bonnes conventions pour choisir les noms de variables et de fonctions, pour documenter le code et pour adopter d'autres bonnes pratiques de programmation. Utilisez ce document pour finaliser votre travail et gardez-le pour vos prochains projets en Python.
Attention : lorsque l'on effectue l'une de ces opérations, il y a toujours un risque d'introduire un nouveau bug. Il convient donc de prendre des précautions : sauvegarder fréquemment les nouvelles versions du code pour pouvoir revenir à une version fiable en cas de problème ; tester fréquemment les nouvelles versions du code (encore un intérêt d'avoir des tests unitaires !) ; effectuer les opérations de refactoring progressivement, sans tout changer à la fois ; etc.

Inspirez-vous des principes de refactoring évoqués pour finaliser le code que vous rendrez. Il sera d'autant plus apprécié que vous aurez pu le rendre clair.

Le travail à rendre sur Ametice comporte, dans un fichier `.zip` :
— un compte-rendu au format `pdf` ;
— l'ensemble de vos fichiers sources (`.py`) dans les répertoires appropriés, afin que l'on puisse directement les exécuter ;
— le fichier de résultats `test_prediction_results.npy` si vous avez pu l'obtenir.
La structure du compte-rendu peut suivre le déroulé du sujets, en apportant des réponses aux questions posées et une analyse critique de chaque résultat. Vous indiquerez également clairement, en faisant référence à votre code et en détaillant les éventuelles difficultés rencontrées : ce que vous avez fait et qui est terminé ; ce que vous avez fait et que vous n'avez pas pu terminer ; ce que vous n'avez pas pu faire du tout ; les parties que vous avez éventuellement faites à plusieurs ou qui vous ont été fournies. Le code doit être clair et vous devez indiquer comment le lancer pour reproduire tous vos résultats.
Le plagiat ne sera pas toléré, vous devez clairement mentionné la source de toute partie dont vous n'êtes pas l'auteur, dans le rapport et le code.