Projet de Programmation Avancée

Instructions

- Projets en groupes de 3 à 4 personnes.
- Envoyez votre notebook à : chakib.fettal@u-paris.fr
- Le fichier doit être nommé "Nom1-Prenom1_Nom2-Prenom2_....ipynb".
- Le rendu étant un notebook, la présentation et le fait d'avoir des commentaires (dans le markdown et/ou code) ainsi que la lisibilité de votre code seront pris en compte dans la notation.
- Deadline: 21/12/2025 à 23h59.

Partie "Problem Solving"

Dans cette partie, on utilisera principalement les fonctionnalités de base de Python pour résoudre des problèmes algorithmiques.

Vacances et bagage

Vous allez en vacances et vous avez un nombre d'objets à prendre avec vous, cependant, il n'y pas assez d'éspace pour tout prendre dans votre voiture.

Vous essayez donc d'assigner un nombre à chaque objet représentant son importance tout en lui assignant un nombre representant l'espace qu'il va prendre dans la voiture. Déterminez les objets à inclure afin que vous ne dépassiez pas la limite d'espace dans la voiture et que l'importance totale des objets inclues soit la plus élevée possible.

Créer une fonction `take_objects(obj_importance, obj_sizes, car_space)` où :

- `obj_sizes` de taille n_{objets} représente les poids de chaque objet.
- `obj_importance` de taille n_{objets} représente l'importance de chaque objet.
- `car_space` est l'espace totale disponible dans la voiture.

Essayer de trouver une solution avec la meilleure complexité possible.

Fusionner plusieurs listes triées

Pendant le cours, on vous a montré comment fusionner deux listes triées en une seule liste triée. Ici, on vous demande de faire la même chose, mais avec un nombre quelconque de listes triées.

Étant donné un tableau "lists" contenant k listes, chaque liste est triée par ordre croissant. Fusionnez toutes les listes chaînées en une seule liste chaînée triée en ordre croissant et renvoyez-la.

Attention, il y a un moyen de faire ça en $O(n \log(k))$ où n est la taille totale des listes et k est le nombre de listes. Donner la solution la plus optimale possible.

Permutations

Étant donné un tableau nums d'entiers distincts, retourner toutes les permutations possibles. Vous pouvez retourner la réponse dans n'importe quel ordre. Par exemple :

```
entrée : nums = [1,2,3]

sortie : [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Sudoku

Écrivez un programme pour résoudre un puzzle de Sudoku en remplissant les cellules vides.

La solution d'un Sudoku doit satisfaire à toutes les règles suivantes :

- Chacun des chiffres 1 à 9 doit apparaître exactement une fois dans chaque ligne.
- Chacun des chiffres 1 à 9 doit apparaître exactement une fois dans chaque colonne.
- Chacun des chiffres 1 à 9 doit apparaître exactement une fois dans chacune des 9 sous-cases 3x3 de la grille.

Créer une fonction `sudoku(puzzle)` et afficher la solution du puzzle suivant avec matplotlib (mettre les cases remplies par votre algorithm dans une autre couleur).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Représenter le puzzle en entrée avec une table 2D de taille 9×9 avec les cases vides représentées par des 0.

Points dans un carré

On vous donne un entier `side` représentant la longueur du côté d'un carré dont les coins sont aux positions `(0, 0)`, `(0, side)`, `(side, 0)` et `(side, side)` sur un plan cartésien. On vous donne également un entier positif k et un tableau 2D d'entiers points, où points[i] = [xi, yi] représente la coordonnée d'un point situé sur le bord du carré.

Vous devez sélectionner k éléments parmi points de sorte que la distance de Manhattan minimale entre n'importe quelle paire de points soit maximisée.

Retournez la valeur maximale possible pour la distance de Manhattan minimale entre les k points choisis.

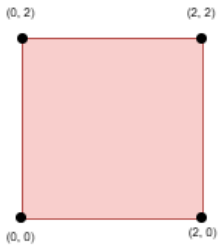
La distance de Manhattan entre deux cellules `(xi, yi)` et `(xj, yj)` est définie comme : `|xi - xj| + |yi - yj|`.

Exemple 1 :

Entrées: side = 2, points = [[0,2],[2,0],[2,2],[0,0]], k = 4

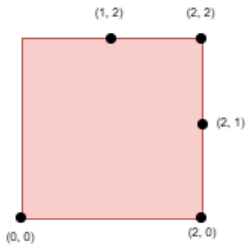
Sortie: 2

Explication :



Prendre tous les points

Exemple 2 :



Entrée: side = 2, points = [[0,0],[1,2],[2,0],[2,2],[2,1]], k = 4

Sortie: 1

Explication :

Prendre les points (0, 0), (2, 0), (2, 2), and (2, 1).

Consignes

- Vous ne pouvez utiliser ni faire d'import de librairies (meme celles de base) pour les deux premiers exercices.
- Vous ne pouvez utiliser que Matplotlib pour la visualization du Sudoku.
- Vous pouvez utiliser tout ce que vous voulez pour le reste.

Partie Calcul Scientifique

Dans cette deuxième partie, on va utiliser des librairies de calcul scientifique pour résoudre des problèmes de classification et de régression. On va essayer de créer des modèles de machine learning en utilisant des librairies comme NumPy, Pandas et CVXPY.

Pour les exercices restants, nous considérons les datasets suivants :

```
datasets = [
    make_moons(noise=0.3, random_state=0),
    make_circles(noise=0.2, factor=0.5, random_state=1),
    make_blobs(n_samples=100, centers=2, n_features=2, center_box=(0, 20), random_state=0)
]
```

Chaque élement de cette liste représente un couple (X, y) tel que $X \in \mathbb{R}^{n \times 2}$ est une matrice de données de taille 2D et $y \in \{0, 1\}^n$ est un vecteur de labels de taille n.

Régression Logistique en NumPy

On va construire sur ce qu'on avait fait en cours avec la régeréssion linéaire et essayer de créer un modèle linéaire pour la classification. Etant donné un ensemble de variables explicatives $\mathbf{X} \in \mathbb{R}^{n \times d}$ et une variable expliqué $\mathbf{x} \in \{0, 1\}^d$ qui represente la classe d'appartenance de chaque point. Notre objectif est de trouver β qui minimise l'erreur de prédiction :

$$L(\beta) = \sum_i y_i \log(\sigma(\mathbf{x}_i \beta)) + (1 - y_i) \log(1 - \sigma(\mathbf{x}_i \beta))$$

Tel que σ represente la foncion sigmoide ou la fonction logitstique :

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

On appelle le modèle qui découle de cette formulation la régréssion logistique. C'est une technique de modélisation pour prédire des résultats binaires (Oui/Non) en utilisant des coefficients pour expliquer la relation entre les variables explicatives et le résultat.

Créez une classe `LogisticRegresion` qui continent (entre autres) les méthodes :

- `fit(X, y)` : calcule les paramètres du modèle selon les données
- `predict(X)` : crée les prédictions pour X.

Optimisation par descente de gradient

Comme pour la régression linéaire, il faut rajouter une colonne de 1 à X. L'algorithme de descente de gradient est le suivant :

- λ = valeur singulière la plus grande de X
- $\alpha = \frac{4}{\lambda^2}$
- t = 0
- Tant que $t < n_{\text{iterations}}$:
  - $\nabla L = X^T(\sigma(X\beta) - y)$
  - $\beta = \beta - \alpha \nabla L$
  - $t = t + 1$

Optimisation par la méthode Newton

Une méthode qui a une vitesse de convergence plus grande que celle de la descente de gradient est celle de Newton. Son pseudo code est le suivant :

- $t = 0$
- Tant que $t < n_{\text{iterations}}$:
  - $H = \nabla^2 L = X^T \text{diag}\left(\frac{\sigma(X\beta)}{1-\sigma(X\beta)}\right) X$
  - $\beta = \beta - H^{-1}.\nabla L$
  - $t = t + 1$

Ajouter un paramètre `optimizer` au constructeur qui permet de choisir la méthode d'optimisation. La méthode par défaut est supposé être la descente de gradient.

Comparaison de la vitesse de convergence

Sur les trois datasets, afficher l'évolution de la fonction objectif des deux méthodes en fonction de l'itération (sur le même plot pour chaque dataset).

Consignes

- Afficher les frontières de décision sur les trois jeux de données pour les deux méthodes et colorer les points par rapport aux prédictions du modèle.
- Vous ne pouvez utiliser que NumPy et Matplotlib.

Arbres et forêts en Pandas

Ici, on va coder deux variantes d'algorithmes d'arbres.

Arbres de Décision

L'algorithme des arbres de décision est une méthode d'apprentissage automatique largement utilisée pour la classification et la régression. Le principe de l'algorithme des arbres de décision repose sur la création d'une structure arborescente qui permet de prendre des décisions basées sur des caractéristiques (variables) des données. Voici comment fonctionne cet algorithme pour la classification :

1. **Sélection de la caractéristique de division :** L'algorithme commence par sélectionner la caractéristique (variable) qui, lorsqu'elle est utilisée pour diviser les données, maximise la séparation entre les différentes classes de sortie. L'objectif est de trouver la caractéristique qui rend les sous-ensembles de données résultants aussi homogènes que possible en termes de classe.

2. **Division des données :** Une fois la caractéristique de division sélectionnée, les données d'entraînement sont divisées en sous-ensembles en fonction des valeurs possibles de cette caractéristique. Par exemple, si la caractéristique est la "couleur", les données peuvent être divisées en sous-ensembles pour chaque couleur (rouge, vert, bleu, etc.).

3. **Répétition :** Les étapes 1 et 2 sont répétées pour chaque sous-ensemble résultant. L'algorithme cherche à diviser chaque sous-ensemble en utilisant la caractéristique qui maximise la séparation des classes.

4. **Arrêt :** L'algorithme s'arrête lorsqu'une condition d'arrêt est satisfaite. Cela peut être lorsque tous les sous-ensembles sont suffisamment homogènes (par exemple, tous les exemples dans un sous-ensemble appartiennent à la même classe).

5. **Attribution de classe :** Une fois que l'arbre de décision est construit, il peut être utilisé pour attribuer une classe ou une étiquette à de nouvelles données en suivant les règles de classification définies par l'arbre. Les données sont acheminées le long des branches de l'arbre en fonction de leurs caractéristiques jusqu'à ce qu'une feuille de l'arbre soit atteinte, ce qui correspond à la classe attribuée.

Créer une classe DecisionTree qui contient entre autres :

- Un constructeur `DecisionTree(criterion)` où `criterion` est le critère d'impureté utilisé dans l'arbre. Il peut soit être l'entropy ou l'impurité gini.
- la méthode `fit(X, y)` qui crée l'arbre de decision.
- la méthode `predict(X)` qui retourne la classe de chaque observation dans X.

Algorithm 11: `Classification And Regression Trees (CART)`

Input : D: dataset of pairs $\{(x_i, y_i)\}_{i=1}^{n}$,
H: impurity function.
Output: T: binary decision tree.
Procedure `TreeGrowing` (T: tree, m: current node, Q: current dataset)
  if $|Q| = 1$ **then**
    Make m a leaf node by setting $T[m] \leftarrow$ `NULL`;
    return;
  end
  foreach *candidate split* $\theta \leftarrow (j, t)$ **do**
    Partition Q into left and right sets:

$$Q_\theta^- \leftarrow \{(x, y) \in Q \mid x_j \leq t\} \quad \text{and} \quad Q_\theta^+ \leftarrow Q \setminus Q_\theta^-;$$

    Compute the impurity incurred from the split:

$$G_\theta(Q) \leftarrow \frac{\left|Q_\theta^-\right|}{|Q|} H(Q_\theta^-) + \frac{\left|Q_\theta^+\right|}{|Q|} H(Q_\theta^+);$$

  end
  Save the split $T[m] \leftarrow \theta^*$ with the lowest incurred impurity where $\theta^* \leftarrow \arg\min_\theta G_\theta(Q)$;
  Grow the left subtree of m, `TreeGrowing` (T, m^-, $Q_{\theta^*}^-$);
  Grow the right subtree of m, `TreeGrowing` (T, m^+, $Q_{\theta^*}^+$);
begin
  Initialize the binary decision tree T;
  Grow the tree starting from the root `TreeGrowing` (T, 0, D);
end

Ici pour `criterion="gini"` nous avons :

$$H(Q) = 1 - \sum_{i=0}^{k-1} \left(\frac{n_i}{n}\right)^2$$

où n_i est le nombre de points dans la classe i qui se trouve dans Q et k est le nombre de classes. Et pour `criterion="entropy"` nous avons :

$$H(Q) = -\sum_{i=0}^{k-1} \frac{n_i}{n} \log_2 \frac{n_i}{n}$$

Forêt Aléatoire

Le principe du Random Forest (forêt aléatoire en français) est une technique d'apprentissage automatique largement utilisée pour la classification, la régression et d'autres tâches liées à la prédiction. Il appartient à la catégorie des méthodes d'ensemble, qui consistent à combiner les prédictions de plusieurs modèles pour obtenir une prédiction plus robuste et précise. Voici comment fonctionne le Random Forest :

- Création de multiples arbres de décision : Le Random Forest crée un certain nombre d'arbres de décision indépendants. Chaque arbre est construit en utilisant un sous-ensemble aléatoire des données d'apprentissage et un sous-ensemble aléatoire des caractéristiques. Cela signifie que chaque arbre est entraîné sur un échantillon différent des données et des caractéristiques.

- Entraînement des arbres de décision : Chaque arbre de décision est construit en suivant les règles d'un arbre de décision classique. Il divise les données d'apprentissage en fonction des caractéristiques pour minimiser l'erreur de prédiction. Cependant, étant donné que les arbres sont construits avec des sous-ensembles aléatoires de données et de caractéristiques, ils sont différents les uns des autres.

- Agrégation des prédictions : Une fois que tous les arbres sont construits, le Random Forest agrège leurs prédictions pour obtenir une prédiction finale. Pour la classification, il s'agit généralement d'un vote majoritaire parmi les arbres. Pour la régression, il s'agit souvent de la moyenne des prédictions.

L'un des avantages clés du Random Forest est sa capacité à réduire le sur-ajustement (overfitting). En utilisant des sous-ensembles aléatoires de données et de caractéristiques, les arbres individuels sont plus simples et plus sujets au surajustement. L'agrégation des prédictions à partir de plusieurs arbres contribue également à améliorer la généralisation du modèle.

Créer une classe RandomForest qui contient entre autres :

- Le constructeur `RandomForest(n_estimators, criterion)` où `n_estimators` est le nombre d'arbres dans la forêt. `criterion` selection les criteria des arbres constituant la forêt.
- la méthode `fit(X, y)` qui crée la forêt.
- la méthode `predict(X)` qui retourne la classe de chaque observation dans X.

Algorithm 13: `Bootstrap Aggregating Classifier`

Input : D: dataset of pairs $\{(x_i, y_i)\}_{i=1}^{n}$,
L: loss function,
$h_1, \ldots, h_M$: classifiers.
Output: F: bagged classifier.

for $m \in \{1, \ldots, M\}$ **do**
  Bootstrap a sample D_m from D;
  Fit classifier h_m on D_m;
end
Aggregate classifiers via a voting $F(x) \leftarrow \arg\max_y \mathbb{1}_{h_m(x)=y}$

Dans ce pseudocode M correspond à `n_estimators` . Chaque h est un arbre different et boostrap signifie choisir n observations aléatoirement avec un tirage avec remise depuis le dataset.

Consignes

- Vous ne pouvez utiliser que Pandas et Matplotlib (Vous pouvez néanmoins utiliser NumPy dans la partie visualisation et pour les fonctions mathématiques (comme np.log2, par exemple).
- Afficher les frontières de décision sur les trois jeux de données pour chaque méthode et colorer les points par rapport aux prédictions du modèle.

Machine à Vecteurs de Support en NumPy et CVXPY

Le but ici est d'apprendre à se documenter sur des librairies qui n'ont pas étés vu pendant le cours. Nous allons donc utiliser la libraire CVXPY qui permet de faire l'optimisation convexe. Plus spécifiquement, nous allons utiliser CVXPY pour résoudre le problème des Machines à Vecteurs de Support.

Le principe des machines à vecteurs de support, également connu sous le nom de support vector machines (SVM) en anglais, est une technique de machine learning utilisée pour la classification et la régression. L'objectif des SVM est de trouver un hyperplan (une frontière de décision) qui sépare de manière optimale les données en deux classes (dans le cas de la classification binaire). L'hyperplan est défini de manière à maximiser la marge entre les deux classes.

Voici les principes fondamentaux des SVM :

- **Séparation linéaire** : Les SVM cherchent un hyperplan qui peut séparer linéairement les données en deux classes. Cela signifie que l'hyperplan doit être une ligne droite (dans le cas de la classification en deux dimensions), un plan (dans le cas de la classification en trois dimensions), ou un hyperplan de dimension supérieure (dans le cas de données de dimension plus élevée).

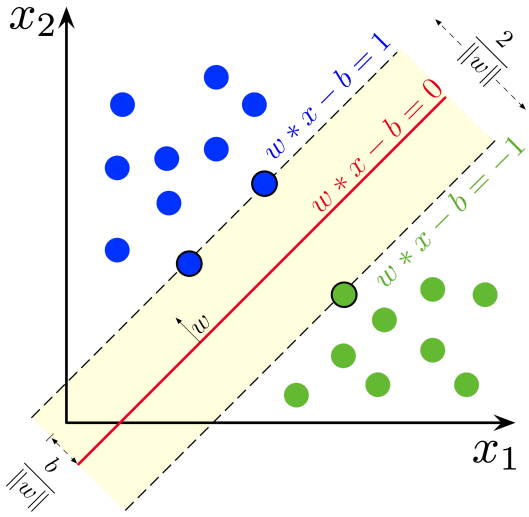
- **Marge maximale** : L'hyperplan trouvé par les SVM est celui qui maximise la marge entre les points de données les plus proches de chaque classe. Ces points de données les plus proches sont appelés vecteurs de support. La marge est la distance entre l'hyperplan et ces vecteurs de support. Maximiser la marge permet d'obtenir un modèle de classification robuste.

- **Fonction de coût** : Les SVM utilisent une fonction de coût pour pénaliser les erreurs de classification. Une pénalité est appliquée en fonction de la distance entre les points mal classés et l'hyperplan. L'objectif est de minimiser cette fonction de coût tout en maximisant la marge.

- **Transformation non linéaire** : Lorsque les données ne peuvent pas être séparées de manière linéaire, les SVM utilisent des transformations non linéaires pour projeter les données dans un espace de dimension supérieure où elles peuvent être séparées linéairement. Cette technique est appelée le "noyau" (kernel trick) et permet d'appliquer des SVM à des problèmes de classification non linéaire.

SVM linéaire

Etant donné un ensemble de données $X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n \times d}$ et un ensemble de labels $y \in \{-1, 1\}^n$. Le problème d'opimisation associé au SVM est le suivant :



$$\max_{\alpha} \quad \sum_{i=1}^{N} \alpha_i - \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j, \tag{1}$$

$$s.t. \quad 0 \leq \alpha_i \leq C, \qquad \forall i \in \{1, \dots, N\} \tag{2}$$

Pour prédire la classe d'une nouvelle observation x_{test}, la règle est la suivante :

$$\begin{aligned} y^{\text{test}} &= \text{sign}\left(\mathbf{w}^{\mathbf{T}} \mathbf{x}_{\text{test}} + b\right) \\ &= \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i x_i^T x_{\text{test}} + b\right) \end{aligned}$$

tel que

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

SVM non-linéaire (kernel RBF)

On peut étendre le cas précedent en utilisant l'astuce du kernel. Dans la formulation précedente remplace tous les produits $\mathbf{x}_i^T \mathbf{x}_j$ par le scalaire $K(x_i, x_j)$ tel que la formule de ce scalaire est définie par rapport au kernel choisi. Ici nous allons utiliser le kernel RBF qui est donné par :

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2) \quad \text{tel que} \quad \gamma = \frac{1}{2 \text{ var}(X)}$$

Consignes

- Créer une classe SVM qui contient entre autres :
  - Le constructeur `SVM(kernel, C)` où `kernel` prend ses valeurs dans 'linear' ou 'rbf' et qui sert à choisir laquelle des deux méthodes utiliser. `C` est le paramètre utilisé dans le problème d'optimisation RBF.
  - La méthode `fit(X, y)` qui résout le problème d'optimisation
  - La méthode `predict(X)` qui retourne la classe de chaque observation dans `X`.
- Afficher les frontières de décision sur les trois jeux de données pour les deux méthodes et colorer les points par rapport aux prédictions du modèle. Utiliser `C=0.025` pour le kernel linéaire et $C = +\infty$ pour le RBF.
- Vous ne pouvez utiliser que NumPy, CVXPY et Matplotlib.