Mini-Projet Java :
Conception d'un mini `scikit-learn` Orienté Objet

Programmation Orientée Objet & Machine Learning

Table des matières

1 Objectifs pédagogiques

Ce mini-projet a pour but de vous faire concevoir, en Java, un **mini-framework de Machine Learning** inspiré de `scikit-learn`, en mettant l'accent à la fois sur :

1.1 Objectifs en Programmation Orientée Objet

- Comprendre et utiliser :
  - les **classes abstraites** et les **interfaces**,
  - l'**héritage** (`extends`),
  - le **polymorphisme dynamique**,
  - l'**encapsulation** (`private`, `protected`, `public`).
- Organiser un projet en **packages** et sous-packages cohérents.
- Définir une API uniforme pour des modèles différents : `train`, `predict`, `score`.
- Écrire du code lisible, commenté, testé, structuré en plusieurs fichiers.

1.2 Objectifs en Machine Learning

- Implémenter une **régression linéaire** (en 1D) entraînée par **descente de gradient**.
- Implémenter l'algorithme des **k plus proches voisins (KNN)** :
  - pour la **régression**,
  - pour la **classification**.
- Utiliser la **distance euclidienne** en dimension quelconque.
- Implémenter des **métriques** :
  - Coefficient de détermination R^2 pour la régression.
  - **Accuracy** pour la classification.
- Implémenter une séparation **train / test** (`trainTestSplit`).
- Implémenter une **normalisation** (Min-Max) et une **standardisation** (Z-score), inspirées de `scikit-learn`.
- Construire un **pipeline complet** :

$$\text{Données brutes} \rightarrow \text{Prétraitement} \rightarrow \text{Entraînement} \rightarrow \text{Prédiction} \rightarrow \text{Score}$$

2 Contexte général

Vous allez concevoir un ensemble de classes Java, organisées en packages, qui permettent de :

— Entraîner différents modèles (`train`).
— Prédire sur de nouvelles données (`predict`).
— Évaluer la performance (`score`) sur un jeu de test.
— Prétraiter les données : `MinMaxScaler`, `StandardScaler`.
— Séparer les données en train/test (`DataUtils.trainTestSplit`).

Les modèles requis :

— **Régression linéaire 1D** (`LinearRegression`).
— **KNN Régression** (`KNNRegression`).
— **KNN Classification** (`KNNClassification`).

Tous ces modèles devront partager une interface commune en héritant de la classe abstraite `MLModel`.

3 Rappels théoriques détaillés

Cette section présente les bases mathématiques et algorithmiques nécessaires à la compréhension et à l'implémentation du mini-framework.

3.1 Structure des données en Java : tableaux 1D et 2D

Tableaux 1D

Un tableau 1D en Java est une structure linéaire :

```
double[] vecteur = new double[5];
vecteur[0] = 1.2;
```

Dans ce projet, un tableau 1D est utilisé pour :
- Représenter un **vecteur de caractéristiques** (features) $\vec{x} = (x_1, \ldots, x_p)$.
- Représenter un vecteur de valeurs cibles ou prédites : `double[] yTrue`, `double[] yPred`.

Tableaux 2D

Un tableau 2D est un tableau de tableaux :

```
double[][] dataset = new double[n][p];
```

Convention OBLIGATOIRE dans tout le projet :
- Chaque **ligne** `dataset[i]` représente une **observation** (un exemple).
- Les **colonnes 0 à p-2** sont les **features**.
- La **dernière colonne (p-1)** est la **cible** y.

Exemple régression 1D :

```
double[][] dataset = {
    {1.0, 2.0},    // x=1, y=2
    {2.0, 4.1},    // x=2, y=4.1
    {3.0, 5.9}     // x=3, y=5.9
};
```

Exemple classification 2D (deux features, label en dernière colonne) :

```
double[][] dataset = {
    {1.2, 0.5, 0},
    {2.1, 1.8, 1},
    {0.8, 0.3, 0}
};
```

3.2 Régression linéaire

La régression linéaire 1D cherche à modéliser la relation entre une variable x et une variable cible y par une droite :

$$\hat{y} = mx + b$$

où :
- m = pente (`slope`),
- b = ordonnée à l'origine (`intercept`),

— $\hat{y}$ = valeur prédite par le modèle.

L'objectif est de trouver les valeurs (m, b) qui minimisent l'erreur entre les valeurs prédites $\hat{y}_i$ et les valeurs réelles y_i.

3.3 Fonction de coût (Erreur Quadratique Moyenne)

On utilise la fonction de coût :

$$J(m, b) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

— Plus J est petit, meilleur est l'ajustement.
— J est toujours positive.

3.4 Descente de gradient

La descente de gradient est une méthode itérative pour trouver les paramètres qui minimisent la fonction de coût. À chaque itération, on met à jour m et b dans la direction opposée au gradient de J.

Dérivées partielles

$$\frac{\partial J}{\partial m} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_i$$

$$\frac{\partial J}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)$$

Mise à jour des paramètres

Étant donné un **taux d'apprentissage** α (learning rate) :

$$m \leftarrow m - \alpha \frac{\partial J}{\partial m} \quad , \quad b \leftarrow b - \alpha \frac{\partial J}{\partial b}$$

Ces mises à jour sont répétées `numEpochs` fois.

Rôle du learning rate

— Si α est trop grand : la descente de gradient risque de diverger.
— Si α est trop petit : la convergence est très lente.
— Il faut choisir un compromis (par exemple 0.01 ou 0.001).

3.5 Algorithme KNN

K-Nearest Neighbors (KNN) est un algorithme **non paramétrique** : il ne calcule pas de paramètres de modèle, mais **mémorise les données d'entraînement**.

Distance euclidienne

Pour deux vecteurs de caractéristiques $\vec{x} = (x_1, \ldots, x_p)$ et $\vec{z} = (z_1, \ldots, z_p)$, la distance euclidienne est :

$$d(\vec{x}, \vec{z}) = \sqrt{\sum_{j=1}^{p} (x_j - z_j)^2}$$

En Java, on utilisera typiquement :

```java
double sum = 0.0;
for (int j = 0; j < a.length; j++) {
    double diff = a[j] - b[j];
    sum += diff * diff;
}
double distance = Math.sqrt(sum);
```

KNN pour la régression

Pour prédire la valeur cible d'un exemple $\vec{x}$:

1. Calculer la distance entre $\vec{x}$ et chaque exemple $\vec{x}_i$ du train.
2. Trier les exemples par distance croissante.
3. Sélectionner les k plus proches voisins.
4. Retourner la **moyenne** des valeurs y_i de ces voisins.

KNN pour la classification

Pour prédire la classe d'un exemple $\vec{x}$:

1. Calcul des distances aux exemples d'entraînement.
2. Tri par distance.
3. Sélection des k plus proches voisins.
4. **Vote majoritaire** sur les labels (classes) de ces voisins.
5. En cas d'égalité, choisir une règle simple et fixe (ex : plus petite classe).

3.6 Métriques d'évaluation

Coefficient de détermination R^2

Le score R^2 mesure la qualité dun modèle de régression :

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

où $\bar{y}$ est la moyenne des y_i.
— $R^2 = 1$: prédiction parfaite.
— $R^2 = 0$: modèle aussi bon que la moyenne constante.
— $R^2 < 0$: modèle pire quune prédiction constante.

Accuracy

Pour un problème de classification, laccuracy est définie par :

$$Accuracy = \frac{\text{nombre de prédictions correctes}}{\text{nombre total de prédictions}}$$

3.7 Normalisation Min–Max

La normalisation Min–Max transforme chaque feature x en :

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

où $x_{\min}$ et $x_{\max}$ sont calculés sur le **train set**.
— Résultat : toutes les valeurs sont ramenées dans $[0, 1]$.
— Important pour KNN : évite quune feature à grande échelle domine la distance.

3.8 Standardisation Z-Score

La standardisation (Z-score) transforme chaque valeur en :

$$x' = \frac{x - \mu}{\sigma}$$

où μ est la moyenne et σ lécart-type sur le **train set**.
— Données centrées sur 0, variance 1.
— Souvent utilisée pour les modèles paramétriques et les réseaux de neurones.

3.9 Train–Test Split

On sépare le dataset en deux parties disjointes :

$$dataset = train \cup test, \quad train \cap test = \varnothing$$

Objectif :
— Entraîner le modèle sur **train**.
— Évaluer le modèle sur **test** (données jamais vues auparavant).
Proportions typiques : 80% train, 20% test.

Fonctions aléatoires à utiliser en Java

Pour mélanger les données avec reproductibilité, on utilisera :

```
import java.util.Random;

Random rand = new Random(seed);
int index = rand.nextInt(n); // entier entre 0 et n-1
```

Pour mélanger les lignes d'un tableau 2D, on peut :
— soit convertir le tableau en `List<double[]>` puis utiliser `java.util.Collections.shuffle`,
— soit implémenter un mélange de Fisher-Yates en permutant les lignes du tableau avec `rand.nextInt`.

3.10 Pipeline complet de Machine Learning

Le pipeline typique est :

1. Charger ou générer les données.
2. Appliquer `trainTestSplit`.
3. Appliquer le prétraitement (scaler) :
   — `fit` sur le train,
   — `transform` sur train et sur test.
4. Entraîner le modèle (`train`).
5. Prédire sur le test (`predict`).
6. Calculer le `score` (R^2 ou accuracy).

3.11 Correspondance avec scikit-learn

scikit-learn	**Projet Java**
`fit(X, y)`	`train(double[][] dataset)`
`predict(X)`	`predict(double[] / double[][])`
`score(X, y)`	`score(double[][] testSet)`
`MinMaxScaler`	`MinMaxScaler`
`StandardScaler`	`StandardScaler`
`train_test_split`	`DataUtils.trainTestSplit`

4 Structure du projet : packages et classes

Le projet doit être organisé en packages comme suit :

- `ml.core`
  - `MLModel` (classe abstraite)
- `ml.linear`
  - `LinearRegression`
- `ml.knn`
  - `KNNRegression`
  - `KNNClassification`
- `ml.preprocessing`
  - `Preprocessor` (interface)
  - `MinMaxScaler`
  - `StandardScaler`
- `ml.model_selection`
  - `DataUtils` (fonctions utilitaires, `trainTestSplit`)
- `ml.metrics`
  - `Metrics` (r^2, accuracy, éventuellement MSE)
- `ml.app`
  - `Main` (point dentrée, démonstration)

Les sections suivantes détaillent toutes les classes à implémenter.

5 Package `ml.core` : classe `MLModel`

Rôle

`MLModel` est une classe abstraite représentant un modèle générique de Machine Learning. Toutes les classes de modèles (`LinearRegression`, `KNNRegression`, `KNNClassification`) doivent **hériter** de cette classe.

Attributs

`name`	String	protected	Nom lisible du modèle (`"Linear Regression"`, `"KNN Regression (k=3)"`, etc.)

Constructeur

```
public MLModel(String name)
```

— Doit simplement initialiser l'attribut `this.name = name;`

Méthodes publiques

`public void printStatus()`

— Affiche un message du type :

```
System.out.println("Modèle : " + name + " (prêt)");
```

`public abstract void train(double[][] dataset);`

— Méthode abstraite à implémenter dans chaque sous-classe.
— Rôle : entraîner le modèle sur un dataset complet (features + cible en dernière colonne).

`public abstract double predict(double[] input);`

— Méthode abstraite.
— Rôle : prédire une valeur (réelle ou classe) pour une observation.
— `input` contient uniquement les features.

`public double[] predict(double[][] inputs)`

— Méthode concrète pouvant être définie dans `MLModel`.
— Applique simplement `predict` à chaque ligne de `inputs`.

`public abstract double score(double[][] testSet);`

— Méthode abstraite.
— Rôle : calculer un **score** sur un dataset de test.
— Convention :
  — `LinearRegression`, `KNNRegression` : R^2.
  — `KNNClassification` : accuracy.

`public String getName()`

— Retourne le nom du modèle.

6 Package `ml.linear` : classe `LinearRegression`

Rôle

Implémenter une régression linéaire en 1D entraînée par descente de gradient.

$$\hat{y} = mx + b$$

Attributs

`slope`	double	private	Pente m de la droite
`intercept`	double	private	Ordonnée à l'origine b
`learningRate`	double	private	Taux d'apprentissage α
`numEpochs`	int	private	Nombre d'itérations de la descente de gradient

Constructeurs

`public LinearRegression()`

- Doit appeler le constructeur parent avec `"Linear Regression"`.
- Initialise :
  - `slope = 0.0;`
  - `intercept = 0.0;`
  - `learningRate` à une valeur par défaut (ex : 0.01).
  - `numEpochs` à une valeur par défaut (ex : 1000).

`public LinearRegression(double learningRate, int numEpochs)`

- Permet de fixer les hyperparamètres.

Méthodes publiques (implémentation de `MLModel`)

`public void train(double[][] dataset)`

- Vérifie que `dataset` est valide (non null, longueur > 0).
- Initialise les paramètres via `initializeParameters()`.
- Lance la boucle de descente de gradient via `gradientDescentLoop(dataset)`.

`public double predict(double[] input)`

- Suppose une feature unique : `double x = input[0];`
- Retourne : `slope * x + intercept;`

`public double score(double[][] testSet)`

- Extrait le vecteur `yTrue` (dernière colonne) et les features (colonne 0).
- Calcule `yPred` en appelant `predict` sur chaque observation.
- Retourne `Metrics.r2Score(yTrue, yPred)`.

Méthodes privées (obligatoires)

`private void initializeParameters()`

- Fixe `slope = 0.0;` et `intercept = 0.0;`

`private boolean isDatasetValid(double[][] dataset)`
— Vérifie que `dataset != null` et `dataset.length > 0`.

`private void gradientDescentLoop(double[][] dataset)`
— Pour `epoch` de 0 à `numEpochs-1` :
  — calculer les gradients via `computeGradients(dataset)`,
  — mettre à jour les paramètres via `updateParameters(...)`.
  — (optionnel) calculer et afficher le coût.

`private double[] computeGradients(double[][] dataset)`
— Calcule $\partial J/\partial m$ et $\partial J/\partial b$ comme dans la partie théorique.
— Retourne un tableau `new double[]gradSlope, gradIntercept;`

`private void updateParameters(double gradSlope, double gradIntercept)`
— Met à jour `slope` et `intercept` avec le learningRate.

`private double computeCost(double[][] dataset)`
— Calcule le coût MSE sur `dataset`.

7 Package `ml.knn` : KNN Régression et Classification

7.1 Classe `KNNRegression`

Rôle

Implémenter KNN pour la régression.

Attributs

`k`	int	private	Nombre de voisins
`trainingData`	double[][]	private	Données d'entraînement mémorisées

Constructeur

```
public KNNRegression(int k)
```

— Vérifie que `k > 0`.
— Initialise `name`, par ex : `"KNN Regression (k=" + k + ")"`.

Méthodes publiques

`public void train(double[][] dataset)`
— Stocke `trainingData = dataset;`.

`public double predict(double[] input)`
— Pour chaque ligne `row` de `trainingData` :
  — Extraire les features de `row` (toutes les colonnes sauf la dernière).
  — Calculer la distance euclidienne entre `input` et ces features.
— Trier les lignes par distance croissante.
— Prendre les `k` lignes les plus proches.
— Retourner la **moyenne** des labels (dernière colonne) de ces k voisins.

`public double score(double[][] testSet)`
— Extraire `yTrue` (labels réels) du testSet.
— Calculer `yPred` avec `predict`.
— Retourner `Metrics.r2Score(yTrue, yPred)`.

Méthode privée de distance

`private double euclideanDistance(double[] a, double[] b)`
— Implémente la formule :

$$d(\vec{a}, \vec{b}) = \sqrt{\sum (a_j - b_j)^2}$$

7.2 Classe `KNNClassification`

Rôle

Implémenter KNN pour la classification (labels discrets).

Attributs

Identiques à `KNNRegression` : `k` et `trainingData`.

Constructeur

```
1 public KNNClassification(int k)
```

— Initialise `name`, par ex. `"KNN Classification (k=" + k + ")"`.

Méthodes publiques

`public void train(double[][] dataset)`

— Mémorise les données dentraînement dans `trainingData`.

`public double predict(double[] input)`

— Même logique que KNNRegression pour la partie tri par distance.
— Sur les `k` voisins, récupérer les labels (dernière colonne).
— Déterminer la classe la plus fréquente (vote majoritaire).
— Retourner cette classe (sous forme de `double`, par ex. 0.0, 1.0).

`public double score(double[][] testSet)`

— Extraire `yTrue`.
— Calculer `yPred`.
— Retourner `Metrics.accuracy(yTrue, yPred)`.

Méthode de distance

— Identique à `KNNRegression`.

8 Package `ml.preprocessing`

8.1 Interface `Preprocessor`

Rôle

Définir un préprocesseur de données générique, inspiré de lAPI de `scikit-learn`.

Méthodes

```
void fit(double[][] dataset);
double[][] transform(double[][] dataset);
double[][] fitTransform(double[][] dataset);
```

- `fit` : calcule les paramètres internes du préprocesseur (min/max ou moyennes/écarts-types).
- `transform` : retourne une version transformée du dataset.
- `fitTransform` : combine les deux (utile sur le train set).

8.2 Classe `MinMaxScaler`

Rôle

Normaliser chaque colonne entre 0 et 1.

Attributs

`min`	double[]	private	Valeur minimale par colonne
`max`	double[]	private	Valeur maximale par colonne
`fitted`	boolean	private	Indique si `fit` a été appelé

Méthodes

`public void fit(double[][] dataset)`

- Pour chaque colonne `j`, calcule `min[j]` et `max[j]`.

`public double[][] transform(double[][] dataset)`

- Pour chaque valeur `x = dataset[i][j]` :

$$x' = \frac{x - min[j]}{max[j] - min[j]}$$

- Si `max[j] == min[j]`, alors la colonne est constante : renvoyer 0.
- Retourner un **nouveau tableau**, ne pas modifier le dataset original.

`public double[][] fitTransform(double[][] dataset)`

- Appelle `fit(dataset)` puis `transform(dataset)`.

8.3 Classe `StandardScaler`

Rôle

Standardiser chaque colonne en Z-score.

Attributs

`mean`	double[]	private	Moyenne par colonne
`std`	double[]	private	Écart-type par colonne
`fitted`	boolean	private	Indique si `fit` a été appelé

Méthodes

`public void fit(double[][] dataset)`
- Calcule `mean[j]` et `std[j]` pour chaque colonne.

`public double[][] transform(double[][] dataset)`
- Applique : $(x - mean[j])/std[j]$.
- Si `std[j] == 0`, renvoyer 0.

`public double[][] fitTransform(double[][] dataset)` Même principe que pour `MinMaxScaler`.

Important :
- **fit** doit être appelé uniquement sur le train set.
- **transform** doit être appelé sur le train set et le test set avec les paramètres calculés sur le train set.

9 Package `ml.model_selection` : `DataUtils`

Classe `DataUtils`

Classe interne `SplitResult`

```
public static class SplitResult {
    public double[][] trainSet;
    public double[][] testSet;
}
```

Méthode `trainTestSplit`

```
public static SplitResult trainTestSplit(double[][] dataset,
                                          double testRatio,
                                          long seed)
```

- `dataset` : données complètes.
- `testRatio` : fraction des données réservées au test (ex : 0.2).
- `seed` : graine pour le générateur aléatoire.

Étapes à implémenter

1. Vérifier que `dataset` n'est pas `null` et contient au moins une ligne.
2. Créer un tableau d'indices `int[] indices = {0,1,...,n-1}`.
3. Utiliser `java.util.Random rand = new Random(seed);` pour mélanger ces indices (par ex. via un algorithme de Fisher-Yates).
4. Calculer les tailles :
   - `int testSize = (int) (testRatio * n);`
   - `int trainSize = n - testSize;`
5. Créer les tableaux `trainSet` et `testSet`, et recopier les lignes du dataset en suivant les indices mélangés.
6. Remplir et retourner un `SplitResult`.

10 Package `ml.metrics` : classe `Metrics`

Rôle

Centraliser les fonctions de calcul des métriques.

Méthodes statiques

`public static double r2Score(double[] yTrue, double[] yPred)`

- Calcule le R^2 décrit dans la partie théorique.
- Utiliser les boucles `for` pour calculer :
  - la moyenne de `yTrue`,
  - la somme des carrés des résidus,
  - la somme des carrés totale.

`public static double accuracy(double[] yTrue, double[] yPred)`

- Compare chaque `yTrue[i]` et `yPred[i]`.
- Compte le nombre de prédictions correctes.
- Retourne le ratio entre ce nombre et le nombre total.
- Pour la classification, vous pouvez utiliser `Math.round` sur `yPred[i]`.

(Optionnel) `public static double mse(double[] yTrue, double[] yPred)`

- Calcule la MSE si nécessaire.

11 Package `ml.app` : classe `Main`

Rôle

Démontrer l'utilisation du framework sur des données de régression et de classification.

Tâches à réaliser

1. Créer un dataset de **régression** (par exemple $y \approx 2x + bruit$).
2. Créer un dataset de **classification** binaire (par ex. deux nuages de points).
3. Appliquer `DataUtils.trainTestSplit` sur chaque dataset.
4. Appliquer un prétraitement (MinMax ou StandardScaler) sur les features :
   — `fit` sur le train,
   — `transform` sur le train et le test.
5. Créer des instances de :
   — `LinearRegression`,
   — `KNNRegression` (k=3, par exemple),
   — `KNNClassification` (k=3, par exemple).
6. Pour chaque modèle :
   — appeler `printStatus()`,
   — appeler `train(trainSet)`,
   — calculer `double s = model.score(testSet);`,
   — afficher `s` avec un message explicite :
      — "Score (R2) du modèle ... = ..." pour la régression.
      — "Score (accuracy) du modèle ... = ..." pour la classification.

12 Expérimentations demandées

Les étudiants doivent :
— Tester plusieurs valeurs de `learningRate` (par ex : 0.1, 0.01, 0.001) et `numEpochs` pour `LinearRegression`.
— Tester plusieurs valeurs de `k` (1, 3, 5, 7) pour `KNNRegression` et `KNNClassification`.
— Comparer les résultats avec et sans prétraitement (MinMax / StandardScaler).
— Observer limpact du choix du `testRatio` dans `trainTestSplit`.
— Discuter les performances (R^2, accuracy) dans un rapport.

13 Livrables et barème

Livrables

— Code Java complet, organisé en packages comme décrit.
— Rapport PDF (3 à 5 pages) avec :
  — un rappel rapide de la démarche,
  — les choix de paramètres,
  — les résultats (scores),
  — une analyse critique des résultats.

Barème indicatif (sur 20)

Conception `MLModel` + API `score` + organisation du projet	3 pts
Implémentation `LinearRegression` (descente de gradient + R^2)	6 pts
Implémentation `KNNRegression` (distance euclidienne + R^2)	4 pts
Implémentation `KNNClassification` (vote majoritaire + accuracy)	4 pts
Prétraitement (MinMaxScaler, StandardScaler) + `trainTestSplit`	2 pts
Rapport et clarté de l'analyse	1 pt
Total	20 pts

14 Remarques finales

— Toutes les classes et méthodes décrites sont **obligatoires**, sauf mention explicite.
— Le format des données (features + cible en dernière colonne) doit être respecté.
— Le code doit être compilable, correctement indenté, commenté et structuré.
— Vous pouvez ajouter des fonctionnalités supplémentaires (bonus) si vous le souhaitez, à condition de respecter d'abord toutes les exigences de base.