

UNIVERSITE DE SFAX Faculté des sciences de Sfax 	Année Universitaire : 2025-2026 Examen Informatique	
	Filière : MP2 – PC2 Nombre pages : 06 + Annexe 2 pages	Date :16/05/2026 Durée : 2 heures

PROBLÈME 1 (12 points)

Dans le contexte du changement climatique, l'analyse et la modélisation des données environnementales jouent un rôle essentiel dans la compréhension et la prévision des phénomènes extrêmes. Ce sujet propose une étude d'un système intelligent dédié au suivi et à la prédiction d'indicateurs climatiques. Dans cette partie, nous proposons une modélisation numérique orientée objet d'un système d'intelligence artificielle destiné à prédire l'évolution d'un indicateur climatique.

Dans la suite on suppose que le module **numpy** a été importé par : **import numpy as np**

Classe IndicateurClimatique

Cette classe représente une série temporelle d'un indicateur climatique mesuré sur une période donnée.

Attributs

- **temps** : vecteur représentant l'axe temporel, de type **numpy.ndarray** (float ou int), de la forme **array([t1, t2, ..., tn])**
- **valeurs** : vecteur des valeurs réelles des observations temporelles, de type **numpy.ndarray** (float), de la forme **array([v1, v2, ..., vn])**.

La méthode **init** initialise un objet à partir des paramètres **t** et **v** représentant respectivement son vecteur des **temps** et son vecteur des **valeurs** observées de l'indicateur climatique.

Structure de la classe

```
class IndicateurClimatique():
    def __init__(self, t, v):
        self.temps = t.copy()
        self.valeurs = v.copy()
```

Méthodes à implémenter

Q1. get_X(...) : Cette méthode retourne la matrice des variables explicatives **X** construite à partir de l'attribut **self.temps**.

$$X = \begin{bmatrix} 1 & t1 \\ \vdots & \vdots \\ 1 & tn \end{bmatrix}$$

X est une matrice de forme **(n, 2)** formée par deux colonnes :

- la première colonne est constituée de 1,
- la deuxième colonne est formée des valeurs de temps.

Q2. get_Y(...) : Cette méthode calcule et retourne une matrice colonne **Y** construite à partir de l'attribut **self.valeurs** formée par les valeurs réelles correspondants aux mesures de l'indicateur climatique :

$$Y = \begin{bmatrix} v1 \\ \vdots \\ vn \end{bmatrix}$$

Classe ModeleIA

Cette classe représente un modèle de prédiction appliqué à un indicateur climatique.

Attributs

- **indicateur** : instance de la classe **IndicateurClimatique** associée au modèle.
- **entraîne** : booléen indiquant si le modèle a été entraîné.
- **parametres** : vecteur de la forme **array([p1, p2])** de type **numpy.ndarray** contenant les paramètres estimés du modèle après entraînement.

Méthodes à implémenter

Q3. __init__ (...) : Constructeur qui :

- initialise l'attribut **indicateur** à partir du paramètre **ind**,
- initialise **entraîne** à **False**,
- initialise **parametres** à **None**.

La méthode **entraîner(...)** permet de calculer et stocker les paramètres du modèle dans l'attribut **parametres** à partir des données de l'attribut **indicateur**. Cette méthode sera redéfinie dans les classes filles.

def entraîner (self) :

#pour cette classe le script de cette méthode n'est pas demandé.

La méthode **predire(...)** permet de calculer et retourner les valeurs prédites sous forme de vecteur **array([vp1, vp2, ..., vpn])** à partir des paramètres estimés stockés dans l'attribut **parametres**. Cette méthode sera redéfinie dans les classes filles.

def predire(self) :

#pour cette classe le script de cette méthode n'est pas demandé.

Q4. calcul_precision(...) : calcule et retourne le **coefficient de détermination R²** du modèle après son entraînement, à partir des données de l'attribut **indicateur**.

$$R^2 = 1 - \frac{\sum (v_i - vp_i)^2}{\sum (v_i - \bar{v})^2}$$

où :

v_i : valeur réelle

vp_i : valeur prédite (retournée par la méthode **predire**)

$\bar{v}$: moyenne des valeurs réelles

La sous Classe **ModeleLineaire**

Cette classe hérite de la classe **ModeleIA** et permet d'estimer les paramètres **a** (*pente*) et **b** (*intercept*) pour un indicateur climatique en utilisant la régression linéaire simple ($\bar{v} = a.t + b$).

Q5. entraîner(...) : Calcule et stocke les paramètres **a** et **b** du modèle linéaire à partir des données réelles de **self.indicateur**, en suivant les étapes suivantes :

1. Récupérer les données :
 - **t** : vecteur des temps
 - **v** : vecteur des valeurs réelles
2. Calculer la pente **a** à l'aide de la formule analytique de la régression linéaire simple : $a = \frac{\sum_{i=1}^n (t_i - \bar{t})(v_i - \bar{v})}{\sum_{i=1}^n (t_i - \bar{t})^2}$
où :
 - $\bar{t}$: moyenne des valeurs du vecteur **temps**
 - $\bar{v}$: moyenne des valeurs du vecteur **valeurs**
3. Calculer l'intercept **b** : $b = \bar{v} - a.\bar{t}$
4. Stocker **a** et **b** dans l'attribut **parametres** sous forme de vecteur
5. Indiquer que le modèle est entraîné.

Q6. predire(...) : calcule et retourne les valeurs prédites **vp** à partir des paramètres **a** et **b**.

La méthode doit :

1. si le modèle est entraîné, effectuer les calculs et retourner le résultat, sinon afficher un message d'erreur.
2. Récupérer les paramètres **a** et **b**
3. Récupérer le vecteur des temps **t** de la forme **array([t1, t2, ..., tn])**
4. Appliquer le modèle linéaire: $vp_i = a.t_i + b$
5. Retourner les valeurs prédites sous forme de vecteur: **array([vp1, vp2, ..., vpn])**

La sous classe **ModeleLineaireMatriciel**

Cette classe hérite de la classe **ModeleIA** et implémente la régression linéaire via la formulation matricielle des moindres carrés. Contrairement à **ModeleLineaire** qui utilise la formule analytique, on utilise, ici, l'écriture générale :

$$\hat{Y} = X \beta$$

avec : $\beta = (X^T X)^{-1} X^T Y$

où :

- X^T désigne la transposée de la matrice **X**
- $(X^T X)^{-1}$ désigne l'inverse de la matrice $X^T X$

Q7. entraîner(...) : calcule et stocke les paramètres du modèle linéaire en utilisant la formulation matricielle des moindres carrés, en appliquant les étapes suivantes :

1. Récupérer les matrices **X** et **Y** retournées par les méthodes *get_X* et *get_Y* de l'indicateur climatique, où :
 - **X** : matrice des variables explicatives de forme **(n, 2)**
 - **Y** : matrice colonne des valeurs réelles de forme **(n, 1)**
2. Appliquer la formule matricielle :

$$\beta = (X^T X)^{-1} X^T Y$$

3. Stocker β dans l'attribut **parametres**
4. Indiquer que le modèle est entraîné

Q8. predire(...) : calcule et retourne les valeurs prédites à partir des paramètres estimés.

Cette méthode doit :

1. si le modèle est entraîné, effectuer les calculs et retourner le résultat, sinon afficher un message d'erreur.
2. Récupérer la matrice **X** à partir de la méthode *get_X*
3. Calculer les valeurs prédites $\hat{Y}$ selon la formule matricielle : $\hat{Y} = X \beta$
4. Convertir $\hat{Y}$ (matrice colonne de forme (n,1)) en un vecteur en utilisant la méthode **flatten** de **Numpy**
5. retourner le vecteur des valeurs prédites

PROBLÈME 2 (8 points)

Cette partie s'appuie sur la conception et l'exploitation d'une base de données relationnelle permettant de stocker les informations relatives aux stations météorologiques, aux capteurs, aux mesures collectées et aux événements climatiques.

Station (idStat, nomStat, paysStat, budget)

La table **Station** représente les stations météorologiques.

- **idStat** : identifiant unique de la station, clé primaire de type entier
- **nomStat** : nom de la station, de type chaîne de caractères
- **paysStat** : pays d'implantation de la station, de type chaîne de caractères
- **budget** : budget alloué à la station, de type réel.

Capteur (idCap, typeCap, statut, idStat#)

La table **Capteur** représente les capteurs installés dans chaque station.

- **idCap** : identifiant unique du capteur, clé primaire de type entier
- **typeCap** : type du capteur (exemple : température, CO₂, humidité...), de type chaîne de caractères
- **statut** : statut du capteur (actif / inactif), de type chaîne de caractères
- **idStat** : identifiant de la station responsable, clé étrangère qui fait référence à la table **Station**, de type entier

Mesure (idMes, dateMes, valeur, idCap#)

La table **Mesure** permet de stocker les mesures climatiques collectées par les capteurs.

- **idMes** : identifiant unique de la mesure, clé primaire de type entier
- **dateMes** : date de la mesure de type DATE ("AAAA-MM-JJ")
- **valeur** : valeur numérique mesurée par le capteur, de type réel
- **idCap** : identifiant du capteur, clé étrangère qui fait référence à la table **Capteur**, de type entier

Prediction (idPred, datePred, tempPred, niveauRisque, nomModele, idStat#)

La table **Prediction** stocke les prédictions générées par les modèles d'intelligence artificielle.

- **idPred** : identifiant unique de la prédiction, clé primaire, de type entier
- **datePred** : date de la prédiction, de type DATE ("AAAA-MM-JJ")
- **tempPred** : température prédite en degré Celsius, de type réel
- **niveauRisque** : niveau de risque estimé (Faible, Moyen, Élevé), de type chaîne de caractères
- **nomModele** : nom du modèle utilisé (exemple : Régression linéaire, Réseau de neurones, Forêt aléatoire...), de type chaîne de caractères
- **idStat** : identifiant de la station concernée, clé étrangère faisant référence à la table **Station**, de type entier

Evenement (idEvent, typeEvent, dateDebut, dateFin, gravite, idStat#)

La table **Evenement** représente les événements climatiques extrêmes enregistrés.

- **idEvent** : identifiant unique de l'événement, clé primaire, de type entier
- **typeEvent** : type d'événement (exemples : Canicule, Inondation, Sécheresse), de type chaîne de caractères
- **dateDebut** : date de début de l'événement, de type DATE ("AAAA-MM-JJ")
- **dateFin** : date de fin de l'événement, de type DATE ("AAAA-MM-JJ")
- **gravite** : niveau de gravité (échelle de 1 à 10), de type entier
- **idStat** : identifiant de la station concernée, clé étrangère faisant référence à la table **Station**, de type entier

Travail demandé

Exprimer en algèbre relationnelle la requête suivante :

Q1. Donner les identifiants des stations ayant enregistré des événements graves ($\text{gravite} \geq 7$) mais ne disposant pas d'un capteur dont le statut est actif.

Q2. Écrire la requête SQL qui permet de créer la table **Mesure** en respectant les contraintes d'intégrité (les autres tables sont supposées déjà créées).

Dans la suite, on suppose que les tables de la base de données sont remplies. Répondre aux questions par des requêtes SQL.

Q3. Augmenter de 5 % le budget des stations situées en Tunisie.

Q4. Donner les identifiants des stations ayant plus de 5 capteurs.

Q5. Donner les identifiants et les noms des stations dont la moyenne des températures mesurées est supérieure à 30°C après 2020.

Q6. Donner les identifiants et les noms des stations n'ayant enregistré aucun événement de type Canicule.

Dans la suite, les fonctions demandées doivent être écrites en Python, en désignant par **cur** le curseur d'exécution des requêtes.

Q7. Écrire la fonction **TempReelStat** qui prend deux paramètres (**cur** et **idS**, l'identifiant d'une station) et retourne la liste des températures réelles enregistrées pour cette station, triées par date croissante.

Q8. Écrire la fonction **TempPredStat** qui prend trois paramètres (**cur**, **idS** et **nomM**, respectivement identifiant d'une station et nom d'un modèle) et retourne la liste des températures prédites pour cette station par ce modèle, triées par date croissante.

Q9. Écrire la fonction **GenDict** qui prend deux paramètres (**cur** et **idS**) et retourne un dictionnaire où :

- Les clés sont les noms des modèles utilisés pour les prédictions de la station **idS**,
- Les valeurs sont l'erreur quadratique moyenne (**MSE**) associée à chaque modèle pour la station **idS**.

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

y_i : température réelle

$\hat{y}_i$: température prédite

n : nombre d'observations