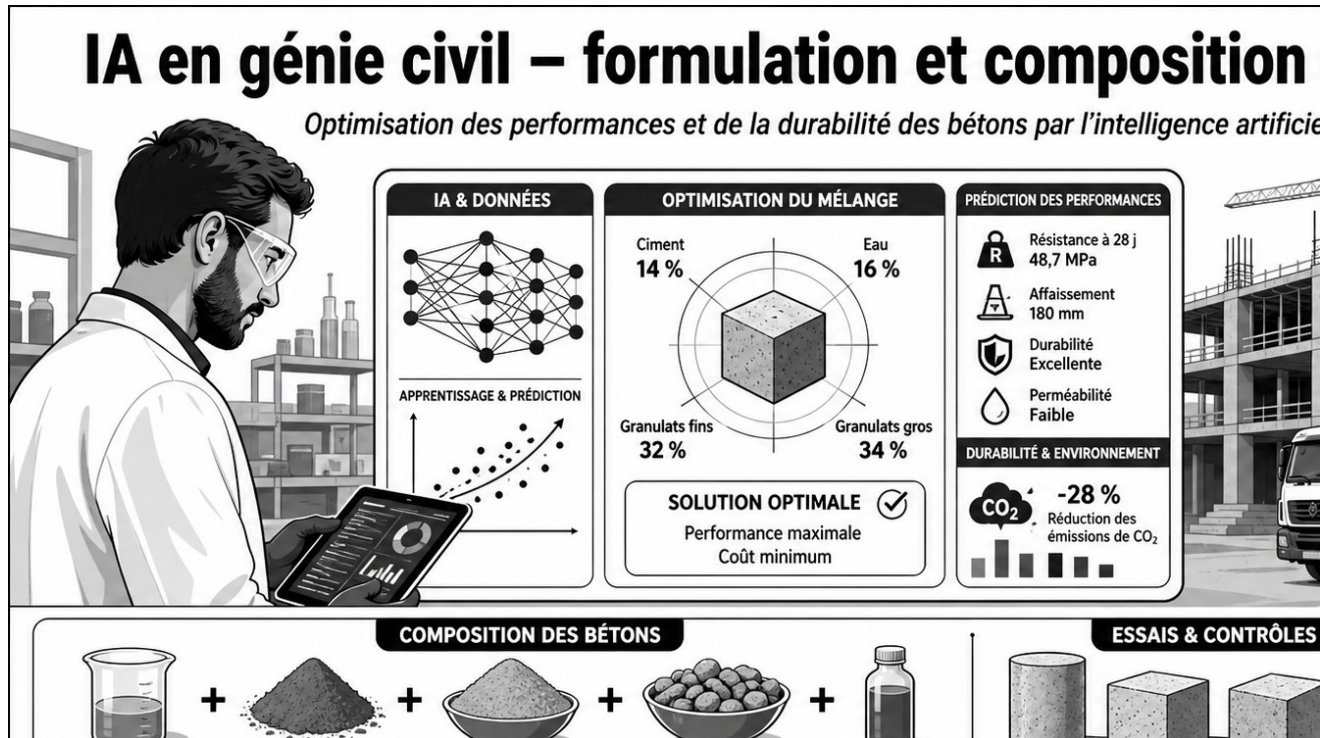


Durée : 2 heures | Aucun document autorisé | Calculatrice non autorisée

Les candidats sont invités à rédiger avec clarté, à soigner la présentation des réponses.



L'intelligence artificielle occupe aujourd'hui une place de plus en plus importante dans presque tous les domaines, tels que la santé, l'industrie, l'agriculture, l'éducation, les transports, l'énergie et la gestion des données. Grâce à sa capacité à analyser de grandes quantités d'informations, à prévoir des résultats et à aider à la prise de décision, l'IA permet d'améliorer la performance, la précision et l'efficacité des systèmes.

Dans le domaine du génie civil, elle offre de nouvelles perspectives pour la conception, le suivi et l'optimisation des ouvrages, notamment à travers la modélisation des matériaux, la prévision du comportement des structures, la maintenance des infrastructures et la réduction de l'impact environnemental.

Dans ce contexte, la présente épreuve s'intéresse plus particulièrement à l'application de l'intelligence artificielle dans la formulation et la composition des bétons, en mettant l'accent sur l'optimisation des proportions des constituants, l'amélioration des performances mécaniques, la durabilité du matériau et la maîtrise des coûts et de l'empreinte carbone.

Partie I : Base de Données Relationnelle

Dans le cadre d'un projet collaboratif de préparation scientifique, des étudiants de classes préparatoires travaillent en équipe pour proposer des solutions à un problème de génie civil lié à la formulation et à la composition des bétons. Leur objectif est de comparer plusieurs formulations destinées à différents ouvrages, d'analyser les matériaux utilisés, d'enregistrer les essais réalisés en laboratoire, puis d'exploiter ces informations pour retenir les formulations les plus adaptées aux contraintes de résistance imposées.

Afin de structurer ce travail collaboratif, on souhaite mettre en place une base de données relationnelle permettant de gérer les matériaux disponibles, les ouvrages visés, les formulations proposées par les étudiants, la composition de chaque formulation et les essais effectués sur ces formulations.

Le schéma de la base de données relationnelle décrit ci-dessous a été élaboré pour les besoins de l'énoncé.

■ Matériau (nom_materiau, unite, cout_unitaire)

La table Matériau enregistre les informations relatives aux matériaux utilisés dans les formulations de béton.

— **nom_materiau** : nom du matériau, de type chaîne de caractères, clé primaire.

Exemples : "ciment", "eau", "sable", "gravier".

— **unite** : unité de mesure du matériau, de type chaîne de caractères.

Exemples : "kg", "L".

— **cout_unitaire** : coût unitaire du matériau, de type réel.

■ Ouvrage (nom_ouvrage, resistance_minimale)

La table Ouvrage enregistre les informations relatives aux ouvrages de génie civil pour lesquels les étudiants doivent proposer des formulations.

— **nom_ouvrage** : nom de l'ouvrage, de type chaîne de caractères, clé primaire.

Exemples : "dalle", "poteau", "poutre", "fondation".

— **resistance_minimale** : résistance minimale exigée pour l'ouvrage, de type réel.

Exemples : 25.0, 30.0, 35.0.

■ Formulation (id_formulation, date_creation, #nom_ouvrage)

La table Formulation enregistre les formulations de béton proposées par les étudiants pour différents ouvrages.

— **id_formulation** : identifiant d'une formulation, de type entier, clé primaire.

— **date_creation** : date de création de la formulation, de type date selon le format "AAAA-MM-JJ".

— **nom_ouvrage** : nom de l'ouvrage concerné par la formulation, de type chaîne de caractères, clé étrangère faisant référence à la table Ouvrage.

■ CompositionFormulation (#id_formulation, #nom_materiau, quantite)

La table CompositionFormulation enregistre la composition détaillée de chaque formulation en matériaux.

— **id_formulation** : identifiant de la formulation, de type entier, clé étrangère faisant référence à la table Formulation.

— **nom_materiau** : nom du matériau utilisé dans la formulation, de type chaîne de caractères, clé étrangère faisant référence à la table Matériau.

— **quantite** : quantité du matériau utilisée dans la formulation, de type réel.

Exemples : 350.0 pour le ciment, 180.0 pour l'eau, 700.0 pour le sable.

La combinaison des clés étrangères **id_formulation** et **nom_materiau** forme la clé primaire de la table CompositionFormulation, garantissant l'unicité de chaque matériau dans une formulation donnée.

- Essai (id_essai, #id_formulation, date_essai, nb_jours, resistance)

La table Essai enregistre les résultats des essais effectués sur les formulations réalisées.

- **id_essai** : identifiant d'un essai, de type entier, clé primaire.
- **id_formulation** : identifiant de la formulation testée, de type entier, clé étrangère faisant référence à la table Formulation.
- **date_essai** : date de l'essai, de type date selon le format "AAAA-MM-JJ".
- **nb_jours** : âge du béton au moment de l'essai, de type entier.

Exemples : 7, 14, 28.

- **resistance** : résistance mesurée lors de l'essai, de type réel.

Exemples : 22.5, 31.8, 38.2.

Remarques :

- Une formulation est destinée à un seul ouvrage.
- Un ouvrage peut être associé à plusieurs formulations.
- Une formulation peut contenir plusieurs matériaux.
- Un même matériau peut intervenir dans plusieurs formulations.
- Une formulation peut faire l'objet de plusieurs essais à différents âges.

Algèbre Relationnel :

Q1. Lister les noms des ouvrages dont toutes les formulations ont une quantité de ciment supérieure à 100.

SQL :

Q2. Lister les noms des matériaux qui interviennent dans toutes les formulations.

Q3. Lister les noms des matériaux ainsi que leurs quantités utilisées dans la formulation destinée à l'ouvrage "fondation" ayant le coût total le plus élevé.

Q4. Calculer, pour chaque mois de l'année 2026, le nombre d'essais dont la résistance mesurée dépasse la résistance minimale requise par l'ouvrage correspondant.

Indication : on pourra utiliser la fonction strftime('%m', date_essai) pour extraire le mois.

SQLITE

Dans la suite, les fonctions demandées doivent être écrites en Python en désignant par cur le curseur de la connexion à la base de données.

Q5. Écrire une fonction `materiau_Formulation` qui prend en entrée `cur` : le curseur de la connexion à la base de données ; et `id_formulation` : un entier représentant l'identifiant d'une formulation.

La fonction doit retourner un dictionnaire où :

- chaque **clé** est le nom d'un matériau ;
- chaque **valeur** est la quantité correspondante dans la table `CompositionFormulation`.

Q6. Écrire une fonction `list_formulations` qui prend en entrée `cur` et `nom_ouvrage`. La fonction doit retourner une liste de dictionnaires de la forme `[ {nom_materiau : quantité} , {nom_materiau : quantité} , ... ]` correspondant aux formulations admissibles de l'ouvrage `nom_ouvrage`. Une formulation est dite admissible si elle possède au moins un essai vérifiant `resistance > resistance_minimale`

Q7. Écrire une fonction `creerFichiersOuvrages(cur)` qui crée, pour chaque ouvrage de la base, un fichier texte portant le nom de cet ouvrage. Dans chaque fichier, la fonction doit écrire une ligne par formulation admissible de l'ouvrage. Chaque ligne contient les couples (`nom_materiau : quantite`) de la formulation sous le format :

`nom_materiau : qte, nom_materiau : qte, nom_materiau : qte , etc`

Exemple

Une ligne du fichier `fondation.txt` peut être : `ciment : 350.0, eau : 180.0, sable : 700.0, gravier : 900.0`

Partie II : Programmation Orientée Objet

Après avoir conçu une base de données relationnelle permettant de stocker les informations relatives aux ouvrages, aux formulations, aux matériaux et aux essais de résistance, on souhaite désormais exploiter ces données dans une application Python orientée objet. En effet, au-delà du stockage, il devient nécessaire d'organiser les données, de les manipuler et de les préparer pour une utilisation en **intelligence artificielle**.

Dans le domaine du génie civil, les techniques de **Machine Learning** permettent d'aider à la prise de décision en estimant, à partir de la composition d'un béton et des conditions d'essai, une grandeur importante comme sa **résistance**. Une telle approche peut assister l'ingénieur dans le choix d'une formulation adaptée à un ouvrage donné, tout en réduisant le nombre d'essais à réaliser.

Dans cette optique, on se propose d'implémenter, selon une approche orientée objet, les classes suivantes :

- **Serie** : représente une colonne de données numériques, correspondant à une variable du jeu de données, comme la quantité de ciment, la quantité d'eau ou la résistance mesurée.
- **DataFrame** : représente un tableau de données composé de plusieurs objets Serie.
- **RegressionLineaire** : représente un modèle de régression permettant d'estimer la résistance du béton.
- **Pipeline** permet d'orchestrer les différentes étapes d'un processus de prédiction de la résistance.

Interface de la classe Serie

Rôle et responsabilités

La classe Serie représente une colonne de données numériques d'un jeu de données. Elle permet de stocker les valeurs d'une variable, comme la quantité de ciment, la quantité d'eau ou la résistance mesurée, puis d'effectuer quelques traitements statistiques utiles avant l'apprentissage d'un modèle.

Attributs

- ♣ label : nom de la série, de type chaîne de caractères.
- ♣ valeurs : vecteur numpy de valeurs numériques de la série.

Méthodes à implémenter

Q8. `def __init__(self, d)` : initialise les attributs de la série à partir d'un dictionnaire d contenant une unique paire {nom de la serie : liste des valeurs de la serie}

Q9. `def std(self)` : retourne l'écart-type des valeurs de la série :

$$\sigma_x = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2} \quad \text{Avec : } \bar{x} \text{ designe la moyenne des } x_i$$

Q10. `def standardiser(self)` : retourne une nouvelle série portant le même nom et des valeurs standardisées selon la formule :

$$z_i = \frac{x_i - \bar{x}}{\sigma_x}$$

Q11. `def correlation(self, other)` : retourne le coefficient de corrélation entre les valeurs de la série courante et les valeurs de la série other (sachant que les deux series ont le même nombre de valeurs)

$$r = \frac{\sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{n-1} (x_i - \bar{x})^2} \sqrt{\sum_{i=0}^{n-1} (y_i - \bar{y})^2}}$$

Q12. `def __eq__(self, val)` : retourne un vecteur booléen mask tel que `mask[i] = True` si la valeur de la série à l'indice i est égale à val, et False sinon.

Interface de la classe DataFrame

Rôle et responsabilités

La classe DataFrame représente un tableau de données composé de plusieurs objets Serie.

On suppose que **toutes les séries d'un même DataFrame ont le même nombre de valeurs**.

Attributs

- ♠ series : liste d'objets Serie.
- ♠ columns : vecteur numpy.ndarray contenant les noms des séries, dans le même ordre que dans series.
- ♠ shape : tuple de la forme (nombre_de_lignes, nombre_de_colonnes).

Si le DataFrame contient p séries et que chaque série contient n valeurs, alors : $shape=(n,p)$

Méthodes à implémenter

Q13. def __init__(self, dict_Serie) : initialise les attributs à partir d'un dictionnaire de la forme :

{nomSerie1 : listeValeurs , nomSerie2 : liste_Valeurs, ...}.

Q14. def projection(self, nomSerie) : retourne le vecteur des valeurs de la série de nom nomSerie. Si aucune série de ce nom n'existe, la méthode doit traiter cette exception et retourne None.

Q15. def to_array(self) : retourne une matrice X de type numpy.ndarray contenant les valeurs des séries du DataFrame. La colonne d'indice j de cette matrice correspond à la série dont le nom est columns[j].

Q16. def __getitem__(self, key) : permet d'accéder à une valeur précise du DataFrame à partir du tuple key de la forme (i, nomSerie). La méthode retourne donc la valeur située à la position i dans la liste des valeurs de la série de nom **nomSerie**.

Q17. def selection(self, mask) : permet de filtrer les valeurs des series du DataFrame à l'aide d'un vecteur booléen mask. Le booléen mask[i] précise si la valeur d'indice i dans chaque liste de valeurs de serie doit être gardée. La méthode retourne un nouveau DataFrame ne contenant que les series dont les valeurs sont gardé (pour lesquelles mask[i] = True) .

Q18. def matrice_correlation(self) : retourne une matrice carrée numpy.ndarray C contenant les coefficients de corrélation entre toutes les séries du DataFrame. L'élément $C[i, j]$ correspond donc à la corrélation entre les séries de noms columns[i] et columns[j]

Interface de la classe RegressionLineaire

Rôle et responsabilités

La classe RegressionLineaire représente un modèle de régression linéaire permettant de prédire la **résistance** du béton à partir des variables explicatives x_i (matériaux de composition) contenues dans un objet DataFrame, comme les quantités de ciment, de gravier, de sable, d'eau et le nombre de jours.

Le modèle est de la forme :

$$\widehat{resistance} = a_0 + a_1x_1 + a_2x_2 + \dots + a_px_p$$

où :

- a_0 est la constante du modèle ;
- $a_1, a_2, \dots, a_p$ sont les coefficients associés aux variables explicatives.

Exemple de données :

Considérons 4 formulations :

id_formulation	ciment	eau	sable	gravier	nb_jours
1	350	180	700	900	28
2	400	160	650	950	28
3	300	190	720	870	7
4	380	170	680	920	14

Alors la matrice des variables explicatives est :

$$X = \begin{pmatrix} 350 & 180 & 700 & 900 & 28 \\ 400 & 160 & 650 & 950 & 28 \\ 300 & 190 & 720 & 870 & 7 \\ 380 & 170 & 680 & 920 & 14 \end{pmatrix}$$

Attributs

- ♣ **V_coef** : vecteur numpy.ndarray contenant les coefficients du modèle. Sa valeur initiale est None.
- ♣ **constante** : réel représentant la constante du modèle. Sa valeur initiale est 0.

Méthodes à implémenter

Q19. `def __init__(self)` : initialise les attributs avec leurs valeurs initiales.

Q20. `def matrice_augmentee(self, objDataFrame)` : retourne la matrice augmentée X_1 , obtenue à partir de la matrice X des valeurs des séries contenues dans `objDataFrame`, en lui ajoutant une première colonne composée uniquement de 1.

Q21. `def fit(self, objDataFrame, Serie_resistance)` : calcule le vecteur des coefficients A à partir de la matrice augmentée X_1 et du vecteur cible y à partir des valeurs de `Serie_resistance` selon la formule suivante :

$$A = (X_1^T X_1)^{-1} X_1^T y$$

$$A = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_p \end{pmatrix}$$

$$y = \begin{pmatrix} resistance_0 \\ resistance_1 \\ \vdots \\ resistance_{n-1} \end{pmatrix}$$

Avec :

La méthode met ensuite à jour les attributs : **constante** avec a_0 et **V_coef** avec $(a_1, \dots, a_p)$.

Interface de la classe Pipeline

Rôle et responsabilités

La classe Pipeline permet d'orchestrer les différentes étapes d'un processus de prédiction de la résistance du béton à partir des données enregistrées dans la base de données.

Elle permet notamment de :

- charger les données d'apprentissage à partir d'une liste d'identifiants de formulations ;
- construire les objets DataFrame et Serie correspondants ;
- créer un modèle de régression et l'entraîner ;
- charger des données de test à partir d'une autre liste d'identifiants de formulations
- prédire les résistances associées ;
- évaluer la qualité des prédictions par comparaison avec les résistances mesurées dans la base de données

Attributs

- **cur** : curseur de connexion à la base de données.
- **composants** : liste des noms des variables explicatives utilisées par le modèle. Exemple : ["ciment", "gravier", "sable", "eau", "nb_jours"].
- **modele** : une instance de la classe RegressionLineaire. Sa valeur initiale est None

Méthodes à implémenter

Q22. def __init__(self, nom_base) : initialise l'attribut cur : le curseur de connexion à partir de la base de données nommée nom_base, initialise l'attribut composants à une liste vide et l'attribut modele à None

Q23. def charger_donnees(self, liste_id_formulation) : charge, à partir d'une liste d'identifiants de formulations, les variables explicatives (composants de formulations) et la résistance associée depuis la base de données.

La méthode initialise l'attribut composants avec les noms des matériaux utilisés dans les formulations et retourne un tuple contenant (objDataFrame, Serie_resistance).

Q24. def fit(self, liste_id_formulation_train) : charge les données d'apprentissage à partir de la liste liste_id_formulation_train, crée une instance de RegressionLineaire et l'affecte à l'attribut **modele** puis appelle la méthode fit de ce modèle afin d'initialiser ses attributs constante et V_coef.

Q25. def predict(self, liste_id_formulation_test) : Écrire la méthode predict en s'inspirant du pseudo-algorithme suivant :

1. Charger les données de test correspondant aux identifiants contenus dans liste_id_formulation_test en appelant la méthode charger_donnees.
2. Récupérer :
 - objDataFrameTest : le DataFrame des variables explicatives ;
 - Serie_resistance_test : la série des résistances réelles.
3. Transformer objDataFrameTest en matrice de données en utilisant la méthode to_array.
4. Pour chaque ligne de cette matrice :
 - calculer la résistance prédite à l'aide du modèle linéaire déjà entraîné, selon la formule :

$$\widehat{resistance} = constante + \sum_{j=0}^{p-1} V_coef[j] \times x_j$$

- constante est l'attribut de l'objet modele ;
- V_coef[j] est le coefficient associé à la variable explicative d'indice j ;
- x_j est la valeur de la variable explicative d'indice j pour la formulation considérée.

5. Regrouper toutes les résistances prédites dans une nouvelle série de nom "resistance_predite"

6. Ajouter cette série à objDataFrameTest et retourne ce nouvel objet DataFrame.

Q26. def evaluer(self, liste_id_formulation_test) : charge les données de test, calcule les résistances prédites, puis compare ces valeurs aux résistances réelles de la base de données. La méthode retourne l'erreur quadratique moyenne

$$MSE = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2$$

où :

- y_i désigne la résistance réelle de la formulation d'indice i ;
- $\hat{y}_i$ désigne la résistance prédite pour cette même formulation ;