Les Étapes Essentielles d'un Projet de Machine Learning

1. Définition du problème

- Identifier clairement ce que l'on cherche à prédire ou classifier.
- **Exemple** : Prédire quels clients risquent de quitter l'entreprise (churn), reconnaître des visages, détecter des fraudes.
- **Exemple** : Une entreprise de téléphonie mobile veut savoir **quels clients risquent de résilier leur abonnement**.
- Le modèle de machine learning va **prédire le churn client**, c'est-à-dire identifier les clients susceptibles de partir.

2. Collecte des données

- Rassembler les données nécessaires à partir de sources internes ou externes.
- Vérifier la qualité, la quantité et la pertinence des données.

3. Prétraitement des données

- Nettoyage : suppression des doublons, gestion des valeurs manquantes.
- Transformation : normalisation, encodage des variables catégorielles.
- Séparation : division en **training set**, **validation set** et **test set**.

4. Choix du modèle

- Sélectionner un algorithme adapté : régression, arbre de décision, SVM, réseau de neurones, etc.
- Tenir compte de la nature du problème (classification, régression, clustering…).

5. Entraînement du modèle

- Utiliser le **training set** pour ajuster les paramètres du modèle.
- Surveiller les performances sur le **validation set** pour éviter le surapprentissage (overfitting).

6. Évaluation du modèle

- Mesurer la performance avec des métriques comme :
  -Accuracy, **Precision**, **Recall**, **F1-score** (pour la classification)
  - RMSE, **MAE**, **R^2** (pour la régression)
- Utiliser le **test set** pour une évaluation finale et impartiale.

7. Mise en production

- Déployer le modèle dans un environnement réel (API, application, dashboard…).
- Assurer le suivi des performances et la maintenance du modèle.

8. Amélioration continue

- Collecter de nouvelles données.
- Réentraîner ou affiner le modèle selon les évolutions du contexte ou des performances.

Conclusion

Structurer un projet de machine learning de A à Z nécessite une approche méthodique et des compétences variées. En suivant les étapes décrites auparavant, vous serez en mesure de définir clairement votre problème, de collecter et préparer des données de qualité, d'explorer et visualiser ces données, de sélectionner et entraîner un modèle, d'évaluer ses performances, et enfin de le mettre en production.

Apprentissage supervisé

Rappel:

Les quatre indispensables étapes pour résoudre un problème d'apprentissage supervisé.

1) **Dataset**
   Y=target, X=feature
2) **Modèle**
   Développer le model avec les paramètres Xet Y (prédire Y en fonction de X)
3) **Fonction coût**
   Mesurer les erreurs entre Y et les prédictions
4) **Fonction de minimisation**
   Développer un algorithme de minimisation d'erreurs de la fonction coût.

Dataset vs Training Set

Dataset (jeu de données)

- C'est l'ensemble **complet** de données que l'on utilise dans un projet d'apprentissage automatique ou d'analyse.
- Il peut contenir **des milliers voire des millions d'exemples**.
- Il est généralement **divisé** en plusieurs sous-ensembles :
  - **Training set** (ensemble d'entraînement)
  - **Validation set** (ensemble de validation)
  - **Test set** (ensemble de test)

Training Set (ensemble d'entraînement)

- C'est une **partie** du dataset.
- Il est utilisé pour **entraîner le modèle** : c'est avec ces données que le modèle apprend à faire des prédictions.
- Le modèle ajuste ses **paramètres internes** (poids, biais, etc.) en fonction des erreurs qu'il fait sur ce jeu.

Exemple concret

Imagine un dataset de 10 000 images de chats et de chiens :

- **Training set** : 7 000 images → utilisé pour apprendre à distinguer chats et chiens.
- **Validation set** : 2 000 images → utilisé pour ajuster les hyperparamètres.
- **Test set** : 1 000 images → utilisé pour évaluer la performance finale du modèle.

Apprentissage supervisé

1) **DataSet**

En apprentissage supervisé, le dataset est composé de deux variables :

- Le target variable Y
- et le features variable $X(x_1, x_2, \dots, x_n)$

De tel sorte que $Y = f(X) = f(x_1, x_2, \dots, x_n)$

Représentation matricielle

Dataset(X,Y)

$Y = f(X)$	x_1	x_2	x_3	x_4	……….	x_n
y^1	x_1^1	x_2^1	x_3^1	x_4^1	……….	x_n^1
y^2	x_1^2	x_2^2	x_3^2	x_4^2	……….	x_n^2
y^3	x_1^3	x_2^3	x_3^3	x_4^3	……….	x_n^3
……..	….	….	….	….	……….	….
y^m	x_1^m	x_2^m	x_3^m	x_4^m	……….	x_n^m
target			***features***			

Avec
n : représente le nombre de feature**s (nombre de colonnes)**
m : le nombre d'exemple **(nombre de lignes)**

Exemple

Un Dataset sur des appartements

Datatset(X,y)			
Y=f(X)	x_1	x_2	x_3
Prix (DA)	Surface(m^2)	Qualité	Adresse postale
31300000	90	3	18003
72000000	110	5	18000
25000000	40	4	18004
29000000	60	3	18010
19000000	50	3	18009
……	…..	……	……
target	***features***		

n =3 ; m : 5

2) Le modèle

La regression linéaire

La **régression** est une méthode statistique qui permet de **modéliser la relation entre une variable dépendante (ou cible)** et une ou plusieurs **variables indépendantes (ou explicatives)**.

Définition simple

- On cherche à expliquer ou prédire une variable yy (par exemple le prix d'une maison) en fonction de variables XX (surface, nombre de chambres, localisation…).
- La régression construit une **fonction mathématique** qui relie XX à yy.
- Exemple en régression linéaire :

$$y=\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_n x_n+\varepsilon$$

Où :

- β_0 est l'intercept (constante),
- β_i sont les coefficients des variables,
- ε\varepsilon est l'erreur (partie non expliquée).

Objectifs de la régression

- **Prédire** une valeur future (ex. prédire le revenu en fonction du niveau d'étude).
- **Expliquer** l'influence des variables (ex. voir si la température influence la consommation d'énergie).
- **Quantifier** la force et la direction de la relation (positive, négative, forte, faible).

Types de régression

- **Régression linéaire simple** : une seule variable explicative.
- **Régression linéaire multiple** : plusieurs variables explicatives.
- **Régressions régularisées** : Lasso, Ridge, Elastic Net (ajout de pénalisation pour éviter le sur-apprentissage).
- **Régression logistique** : utilisée quand la variable cible est qualitative (ex. succès/échec).
- **Régressions non linéaires** : quand la relation n'est pas une droite mais une courbe.

En résumé : La régression est une **technique d'analyse prédictive et explicative** qui relie une variable cible à un ensemble de variables explicatives, en cherchant à minimiser l'erreur entre les valeurs observées et celles prédites par le modèle.

Exemple

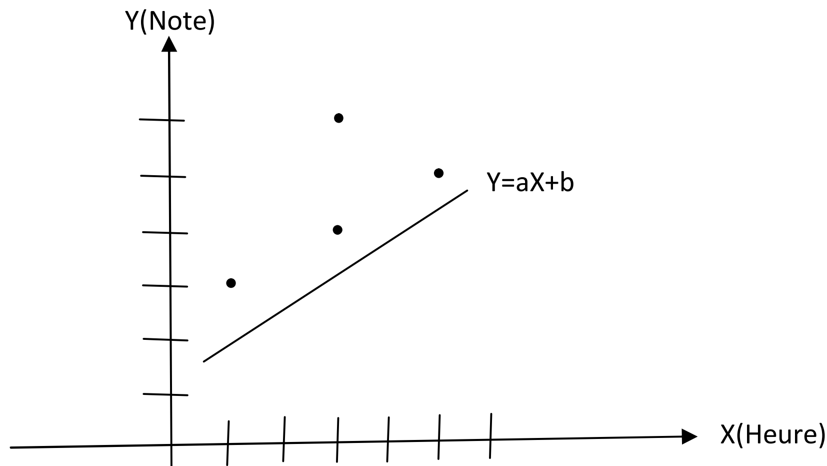
On reprend le nuage de points étudié précédemment :

Heure(X)	1	3	3	5
Note au test(Y)	3	4	6	5

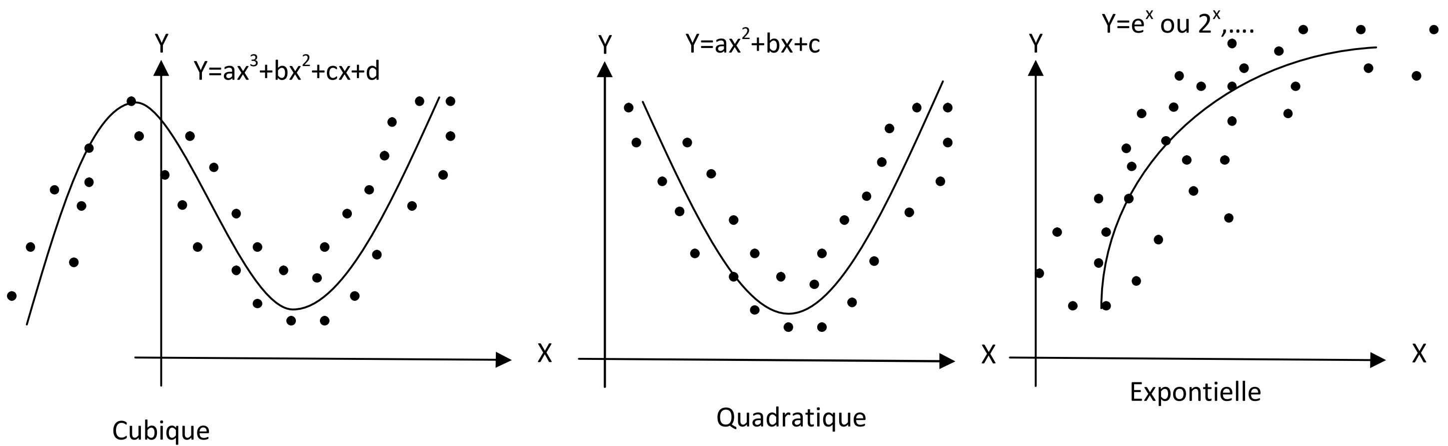
Dans ce contexte, **Y** désigne la variable cible (*target*), c'est-à-dire la valeur que l'on souhaite prédire : ici, la note obtenue au test. **X** représente une seule caractéristique (*feature*) : le nombre d'heures de révision.
L'objectif est d'établir une relation linéaire entre X et Y, exprimée par une droite affine de la forme :

$$Y=aX+b$$

Où **a** et **b** sont des paramètres inconnus à estimer.
Cette méthode correspond à une **régression linéaire simple**, utilisée pour prédire la note en fonction du temps de révision.
Dans le cadre de l'apprentissage supervisé, les paramètres **a** et **b** sont initialement attribués de manière aléatoire. Cela revient à tracer une droite arbitraire sur le graphe, qui ne reflète pas encore la tendance réelle des données. Au fil de l'entraînement, ces paramètres sont progressivement ajustés afin de **minimiser la fonction de coût**, permettant ainsi à la droite de mieux s'aligner avec les points observés.



NB : Un nuage de points peut parfois être mieux représenté par une courbe non linéaire, telle qu'une fonction quadratique, cubique ou exponentielle, selon la forme et la tendance des données. (Voir les graphes ci-dessous pour illustration.)

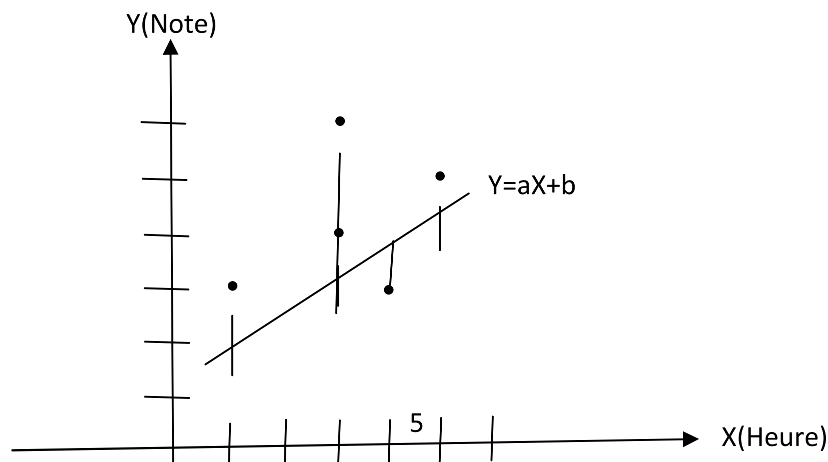


3) La fonction coût

La machine ne va mesurer les erreurs, elle va calculer son cout entre le modèle quelle est entrain de développer avecc les paramètres a et b qu'on a choisi aléatoire et les vrais valeurs y dans le dataset, elle va mesurer les erreurs entre le modèle et les différents points

4) Fonction de minimisation

Pour claculer les erreurs entre les points réels et les points sur la droite. On procède la projection des points sur la droite



Pour minimiser la fonction coût on peut utiliser la méthode du moindre carrées (voir l'exemple statistique page 16), maintenant il existe plusieurs méthodes, la plus populaire en machine learning, c'est la méthode de descente de gradient (gradient descent)

Étapes de la descente de gradient en régression linéaire

On cherche à ajuster une droite de la forme :

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

Où :

- x est la variable d'entrée (feature),
- $\hat{y}$ est la prédiction du modèle,
- a (pente) et b (l'intercept = ordonnée à l'origine) sont les **paramètres à apprendre**.

� Étapes de l'algorithme :

1. **Initialisation** : Choisir des valeurs aléatoires pour a et b
2. **Prédiction** : Calculer les prédictions $\hat{y}_i = ax_i + b$
3. **Calcul de la fonction coût** : Utiliser la **fonction de coût quadratique moyenne** :
$$J(a,b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$
4. **Calcul du gradient** : Calculer les dérivées partielles de J par rapport à a et b :
$$\frac{\partial J}{\partial a} = \frac{2}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i).x_i\,; \frac{\partial J}{\partial b} = \frac{2}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)\,;$$
5. **Mise à jour des paramètres** : On ajuste a et b dans la direction opposée au gradient :
$$a_i = a - \propto \frac{\partial J}{\partial a};\ b_i = b - \propto \frac{\partial J}{\partial b}$$
Où $\propto$ est le **taux d'apprentissage** (*learning rate*).
6. **Itération** : Répéter les étapes 2 à 5 jusqu'à ce que la fonction coût converge (devienne minimale).

Pourquoi utilise-t-on le carré des erreurs au lieu de la valeur absolue ?

On préfère utiliser :

$$(\hat{y}_i - y_i)^2 \text{ Plutôt que} |\hat{y}_i - y_i|$$

Pour plusieurs raisons :

1- Différentiabilité

- La fonction carrée est **lisse et dérivable partout et convexe**, ce qui est essentiel pour appliquer la descente de gradient.
Exemple

$f(x) = x^2$

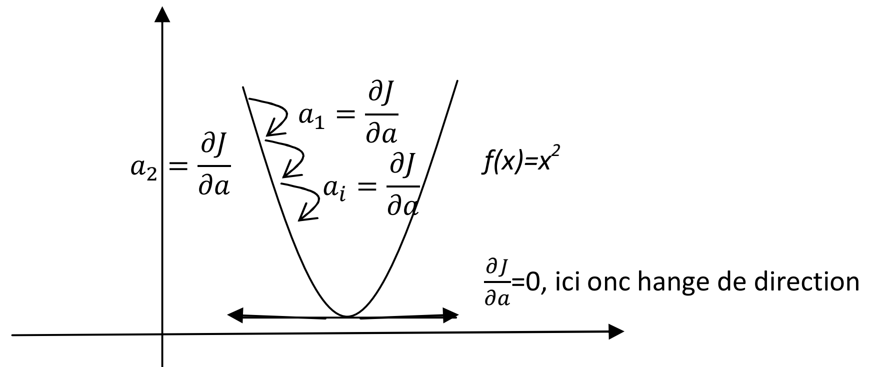
Sa courbe est **continue et lisse**

Sa dérivée est :$f'(x) = 2x$

Elle existe **pour tous les x**, y compris en zéro.

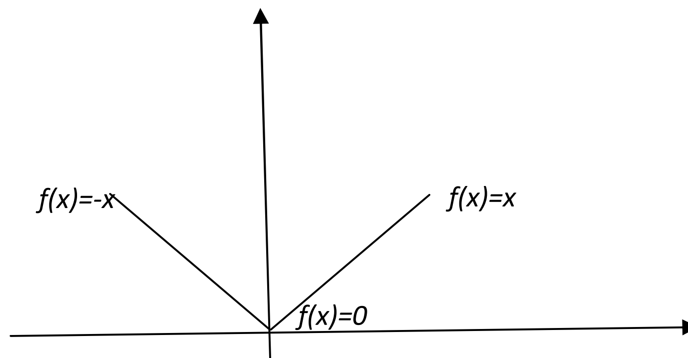
Cela permet à l'algorithme de **descente de gradient** de savoir dans quelle direction ajuster les paramètres à chaque étape.

- Sa courbe est **continue et lisse**



- La valeur absolue **n'est pas dérivable en zéro**, ce qui complique l'optimisation.
  Exemple
  $f(x) = |x|$
  $$f(x) = \begin{cases} x \; si \; x > 0 \\ -x \; si \; x < 0 \\ 0 \; si \; x = 0 \end{cases}$$

  La représentation graphique de la fonction $f(x) = |x|$



Sa dérivée est :
$$f(x) = \begin{cases} 1 \; si \; x > 0 \\ -1 \; si \; x < 0 \\ indéfini \; si \; x = 0 \end{cases}$$

2- Simplicité mathématique

- Les dérivées du carré sont simples à calculer et à manipuler.
- Cela permet une mise en œuvre plus efficace et plus stable de l'algorithme.

3- Pénalisation des grandes erreurs

- Le carré amplifie les grandes erreurs plus que les petites, ce qui pousse le modèle à les corriger en priorité.
- Cela peut être utile pour améliorer la précision globale du modèle.

Illustrons ça avec un exemple simple et visuel pour bien comprendre pourquoi on utilise le **carré des erreurs** $(\hat{y}_i - y_i)^2$ au lieu de la **valeur absolue**$|\hat{y}_i - y_i|$dans la descente de gradient.

Objectif : minimiser l'erreur entre prédiction et réalité

Exemple :

Imaginons qu'on essaie de prédire la note d'un élève en fonction de ses heures d'étude.

	Heures d'étude (X)	**Note réelle (Y)**	**Prédiction (ŷ)**	**Erreur**
X	2	5	4	-1
Y	4	7	9	+2

Comparaison des deux fonctions d'erreur

1. Valeur absolue :

Erreur totale=|4−5|+|9−7|=1+2=3

Mais la **dérivée** de |x| est :

$$f(x) = \begin{cases} 1 \; si \; x > 0 \\ -1 \; si \; x < 0 \\ indéfini \; si \; x = 0 \end{cases}$$

Pas de pente définie en zéro, donc la descente de gradient ne sait pas dans quelle direction aller.

2. Carré de l'erreur :

Erreur totale=(4−5)2+(9−7)2=12+22=1+4=5

La **dérivée** de $f(x) = x^2$ est :

$f'(x) = 2x$

Toujours définie, même en zéro. La descente de gradient peut calculer une pente claire et ajuster les paramètres.

En Visualisation les deux courbe associées aux deux fonctions, on vois que la courbe

- La fonction $|x|$ forme un **V** pointu au centre → pas de pente nette au sommet
- La fonction x^2 forme une **parabole lisse** → pente bien définie partout

Illustration avec des données réelles
Nous avons utilisé les données suivantes :

Heure d'étude (X)	Note obtenue (Y)
1	3
3	4
3	6
5	5

Programme Python

```
import numpy as np

# Données d'entraînement
X = np.array([1, 3, 3, 5])
Y = np.array([3, 4, 6, 5])

# Paramètres initiaux
a = 0.0
b = 0.0

# Hyperparamètres
learning_rate = 0.01
iterations = 100
m = len(X)

# Fonction coût
def cost(a, b, X, Y):
    y_pred = a * X + b
    return np.mean((y_pred - Y) ** 2)

# Descente de gradient
for i in range(iterations):
    y_pred = a * X + b
    da = (2/m) * np.sum((y_pred - Y) * X)
    db = (2/m) * np.sum(y_pred - Y)
    a -= learning_rate * da #Taux d'apprentissage
    b -= learning_rate * db
```

```
# Résultats finaux
final_cost = cost(a, b, X, Y)
print(f"Paramètres finaux : a = {a:.4f}, b = {b:.4f}")
print(f"Coût final : J(a,b) = {final_cost:.4f}")
```

Résultat
Nous avons appliqué la descente de gradient pendant 100 itérations. Voici les résultats :
Paramètres finaux : $a = 1.0752$, $b = 0.923$
Coût final : $J(a, b) = 1.5403$

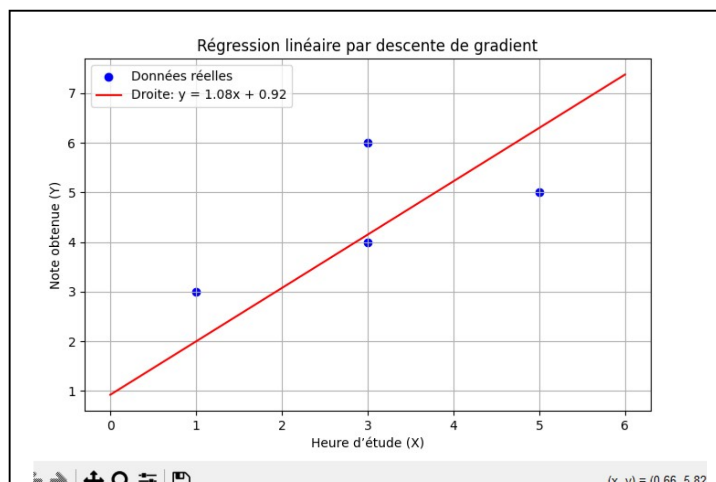
Exemple

Programme Python pour afficher les points de données et la droite de régression obtenue par descente de gradient :

Tracer :

- Les **points** : (1,3), (3,4), (3,6), (5,5)
- La **droite** : y=ax+by = ax + b avec a=1.0752a = 1.0752, b=0.9233

```
import matplotlib.pyplot as plt
import numpy as np

# Données
X = np.array([1, 3, 3, 5])
Y = np.array([3, 4, 6, 5])
# Paramètres de la droite obtenus par descente de gradient
a = 1.0752
b = 0.9233

# Droite de régression
x_line = np.linspace(min(X)-1, max(X)+1, 100)
y_line = a * x_line + b
# Tracé
plt.figure(figsize=(8, 5))
plt.scatter(X, Y, color='blue', label='Données réelles')
plt.plot(x_line, y_line, color='red', label=f'Droite: y = {a:.2f}x + {b:.2f}')
plt.xlabel('Heure d'étude (X)')
plt.ylabel('Note obtenue (Y)')
plt.title('Régression linéaire par descente de gradient')
plt.legend()
plt.grid(True)
plt.show()
```



Exercice

Ecrire le programme python pour trouver les coefficients de la droite de regression et afficher de cette droite avec les points associer pour l'exemple du Datatset(X,y) suivant :

Datatset(X,y)			
Y=f(X)	x_1	x_2	x_3
Prix (DA)	Surface(m^2)	Qualité	Adresse postale
31300000	90	3	18003
72000000	110	5	18000
25000000	40	4	18004
29000000	60	3	18010
19000000	50	3	18009
target	***features***		

Solution

Voici le programme Python complet qui utilise **scikit-learn** pour ajuster une **régression linéaire multivariée** à ton jeu de données, puis affiche les **points réels** et la **droite de régression** projetée sur une seule variable (par exemple, la surface).

Étapes du programme

1. Définir les variables explicatives $X=[x_1,x_2,x_3]$ et la cible y (le prix).
2. Ajuster un modèle de régression linéaire.
3. Afficher les **coefficients** de la droite : $y=a_1x_1+a_2x_2+a_3x_3+b$
4. Tracer les points et la droite projetée sur une seule variable (ex. : surface).

Avant d'utiliser python on va jeter un coup d'œil sur les biblithèque qu'on va être utilisé dans ce programme.

1)**numpy :** biblithèque importé pour crééer des tableaux linéaire et non linéaire

Déclaration d'un vecteur avec la bibliothèque numpy

y=(31300000, 72000000, 25000000, 29000000, 19000000)
y = np.array([31300000, 72000000, 25000000, 29000000, 19000000])

Déclaration d'une matrice avec la bibliothèque numpy

90	3	18003
110	5	18000
40	4	18004
60	3	18010
50	3	18009

```
X = np.array([
    [90, 3, 18003],
    [110, 5, 18000],
    [40, 4, 18004],
    [60, 3, 18010],
    [50, 3, 18009]
```

])
2) **matplolib :** bibliothèque pour dessiner les graphes 2D ou 3D.

La fonction `plt.scatter()` de la bibliothèque **Matplotlib** en Python est utilisée pour créer un **diagramme de dispersion** (scatter plot), qui permet de visualiser la relation entre deux ensembles de données numériques.

Définition

python
```
plt.scatter(x, y, ...)
```

- `x` : coordonnées horizontales (abscisses)
- `y` : coordonnées verticales (ordonnées)
- `...` : options facultatives (couleur, taille, transparence, etc.)

Utilité

- Visualiser la **corrélation** entre deux variables
- Identifier des **groupes**, **tendances**, ou **outliers**
- Très utilisé en **régression**, **classification**, **analyse exploratoire**

Exemple simple

python
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 1, 3, 7]

plt.scatter(x, y, color='red', marker='o')
plt.title("Exemple de scatter plot")
plt.xlabel("Variable X")
plt.ylabel("Variable Y")
plt.show()
```

Options utiles

Paramètre	Description	Exemple
`color`	Couleur des points	`'blue'`, `'green'`
`marker`	Forme des points	`'o'`, `'x'`, `'^'`
`s`	Taille des points	`s=100`
`alpha`	Transparence (0 à 1)	`alpha=0.5`
`c`	Couleur selon une autre variable	`c=z`

3) sklearn.linear_model : bibliothèque qui contient les différents models de regression

Dans sklearn on va définir ces variables et fonction utilisée dans le programme au dessous :

*model = LinearRegression() : c'est le moodèle qu'on va utiliser de sklearn (la regression linéaire)

*model.fit(X, y)

La méthode `model.fit(X, y)` est une **étape fondamentale dans l'apprentissage supervisé** en machine learning avec des bibliothèques comme `scikit-learn`.

Définition

python
```
model.fit(X, y)
```

- `model` : un modèle d'apprentissage (ex. : `LinearRegression()`, `RandomForestClassifier()`, etc.)
- `X` : les **données d'entrée** (features), généralement sous forme de tableau ou matrice (ex. : caractéristiques d'un matériau)
- `y` : les **valeurs cibles** (labels ou résultats attendus), comme la classe ou la valeur à prédire

Que fait `fit()` ?

- Il **entraîne le modèle** en ajustant ses paramètres internes (comme les coefficients dans une régression).
- Il **apprend la relation** entre les variables d'entrée `X` et la sortie `y`.
- Après cette étape, le modèle est prêt à **faire des prédictions** avec `model.predict(X_new)`.

Exemple concret

python
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
X = [[1], [2], [3], [4]]  # Entrées
y = [2, 4, 6, 8]          # Sorties

model.fit(X, y)           # Apprentissage
```

→ Le modèle apprend que la relation est : `y = 2 * x`

Résumé

Élément	Rôle
`X`	Données d'entrée (features)
`y`	Résultats attendus (labels ou valeurs)
`fit()`	Apprentissage du modèle

— Les coefficients (`model.coef_`)

- **Définition** : Ce sont les **poids attribués à chaque variable indépendante** (chaque colonne de `X`) dans le modèle.
- **Rôle** : Ils indiquent **l'impact de chaque variable** sur la prédiction.
- **Exemple** : Si `X` a 3 colonnes (densité, température, pression), alors `model.coef_ = [0.5, -1.2, 3.0]` signifie :
  - +0.5 unité de sortie par unité de densité
  - -1.2 unité par degré de température
  - +3.0 unité par unité de pression

— L'ordonnée à l'origine (`model.intercept_`)

- **Définition** : C'est la **valeur de sortie prédite quand toutes les entrées sont nulles**.
- **Rôle** : Il ajuste la droite ou le plan de régression pour qu'elle passe au bon endroit.
- **Exemple** : Si `intercept_ = 2.5`, alors la prédiction de base commence à 2.5 avant d'ajouter les effets des variables.

— La fonction de Prédictions du modèle (`model.predict(X)`)

- **Définition** : Calcule les **valeurs prédites** pour chaque ligne de `X` en appliquant la formule :

$$\hat{y} = \text{intercept} + \sum_{i} \text{coef}_i x_i$$

Exemple :
Si `X = [[1, 2, 3]]`, `coef_ = [0.5, -1.2, 3.0]`, `intercept = 2.5`, alors :

$$\hat{y} = 2.5 + (0.5 \cdot 1) + (-1.2 \cdot 2) + (3.0 \cdot 3) = 2.5 + 0.5 - 2.4 + 9 = 9.6$$

En résumé

Élément	Signification	Utilité principale
`model.coef_`	Poids des variables	Comprendre l'influence de chaque facteur
`model.intercept_`	Valeur de base	Ajuster la prédiction
`model.predict(X)`	Valeurs prédites pour les entrées `X`	Obtenir les résultats du mo

# Visualisation : projection sur la surface (x1)
x_surface = X[:, 0]
y_pred_surface = model.predict(X)

Ce petit extrait de code Python est utilisé pour **visualiser les prédictions d'un modèle de régression** en fonction d'une seule variable d'entrée. Voici une explication détaillée ligne par ligne :

Contexte

Ce code suppose que :

- `X` est une matrice de données d'entrée (features), avec plusieurs colonnes.
- `model` est un modèle de régression entraîné (par exemple avec `LinearRegression()`).
- L'objectif est de **projeter les prédictions sur une seule dimension** (la première colonne de `X`, notée `x1`).

Ligne 1 : `x_surface = X[:, 0]`

- **Signification** : On extrait la **première colonne** de la matrice `X`.
- **But** : Obtenir les valeurs de la **variable x1** (par exemple, la densité ou la température).
- **Notation** : `X[:, 0]` signifie "toutes les lignes, colonne 0".

Exemple :

Si `X = [[1, 2], [3, 4], [5, 6]]`, alors :

python

```
x_surface = [1, 3, 5]
```

Ligne 2 : `y_pred_surface = model.predict(X)`

- **Signification** : On utilise le modèle pour **prédire les valeurs de sortie** (y) pour chaque ligne de `X`.
- **But** : Obtenir les **valeurs estimées** par le modèle.
- **Résultat** : Un tableau de prédictions, de même longueur que `X`.

Exemple :

Si le modèle a appris `y = 2x1 + 3x2`, alors :

python

```
model.predict([[1, 2], [3, 4]]) → [2*1+3*2, 2*3+3*4] → [8, 18]
```

Visualisation typique

Ce code est souvent suivi par un `plt.scatter()` ou `plt.plot()` pour afficher :

python

```
import matplotlib.pyplot as plt

plt.scatter(x_surface, y_pred_surface, color='blue')
plt.xlabel("x1 (ex: densité)")
plt.ylabel("Prédiction du modèle")
plt.title("Projection des prédictions sur x1")
plt.show()
```

Résumé

Élément	Rôle
`X[:, 0]`	Extraire la première variable (x1)
`model.predict(X)`	Calculer les prédictions du modèle
Visualisation	Montrer comment les prédictions varient selon x1

Programme Pyhton

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Données X est une matrices (5,3) et y est un vecteur de (1,5)
X = np.array([
    [90, 3, 18003],
    [110, 5, 18000],
    [40, 4, 18004],
    [60, 3, 18010],
    [50, 3, 18009]
])
y = np.array([31300000, 72000000, 25000000, 29000000, 19000000])

# Modèle de régression
model = LinearRegression()
model.fit(X, y)

# Coefficients
coefficients = model.coef_
intercept = model.intercept_
print("Coefficients :", coefficients)
```

```
print("Intercept :", intercept)

# Visualisation : projection sur la surface (x1)
x_surface = X[:, 0]
y_pred_surface = model.predict(X)

plt.figure(figsize=(8, 5))
plt.scatter(x_surface, y, color='blue', label='Prix réel')
plt.plot(x_surface, y_pred_surface, color='red', label='Régression (projection sur surface)')
plt.xlabel('Surface (m²)')
plt.ylabel('Prix (DA)')
plt.title('Régression linéaire multivariée (projection sur surface)')
plt.legend()
plt.grid(True)
plt.show()
```

Résultat

- Tu obtiendras les **coefficients** associés à chaque variable : surface, qualité, adresse.
- Le graphique montre comment le **prix varie avec la surface**, en tenant compte des autres variables.

Liste des principales méthodes de régression **disponibles dans** scikit-learn **(sklearn), que tu peux importer comme tu le fais avec LinearRegression :**

Régressions dans sklearn.linear_model

from sklearn.linear_model import LinearRegression	# Régression linéaire classique
from sklearn.linear_model import Ridge	# Régression avec régularisation L2
from sklearn.linear_model import Lasso	# Régression avec régularisation L1
from sklearn.linear_model import ElasticNet	# Combinaison de Lasso et Ridge
from sklearn.linear_model import BayesianRidge	# Régression bayésienne
from sklearn.linear_model import SGDRegressor	# Régression par descente de gradient stochastique
from sklearn.linear_model import HuberRegressor	# Robuste aux valeurs aberrantes
from sklearn.linear_model import PassiveAggressiveRegressor	# Régression en ligne (streaming)
from sklearn.linear_model import RANSACRegressor	# Robuste aux outliers (échantillonnage aléatoire)
from sklearn.linear_model import TheilSenRegressor	# Robuste et non paramétrique

1) **Régression avec régularisation L2**

La **régularisation L2** dans le contexte de la régression en Python (souvent avec `scikit-learn`) est une **technique pour éviter le surapprentissage** (overfitting) en **pénalisant les grands coefficients** du modèle.

Qu'est-ce que la régularisation L2 ?

- Elle ajoute une **pénalité** à la fonction de coût du modèle.
- Cette pénalité est **proportionnelle au carré des coefficients**.
- Elle pousse le modèle à **réduire la complexité** en limitant les valeurs extrêmes des poids.

Formule de la régression linéaire avec L2 :

$$Coût = \sum (y_i - \hat{y}_i)^2 + \sum w_j^2$$

- y_i : valeur réelle
- $\hat{y}_i$: valeur prédite
- w_j: coefficient du modèle
- λ: paramètre de régularisation (plus il est grand, plus la pénalité est forte)

En Python avec `Ridge`

python

```
from sklearn.linear_model import Ridge

model = Ridge(alpha=1.0)  # alpha = λ
model.fit(X, y)
```

- `Ridge` est le nom du modèle de régression avec régularisation L2.
- `alpha` contrôle la force de la régularisation :
  - `alpha = 0` → équivalent à une régression linéaire classique
  - `alpha > 0` → pénalise les grands coefficients

Pourquoi utiliser L2 ?

- Réduit le **surapprentissage** sur des données bruitées.
- Améliore la **généralisation** du modèle sur de nouvelles données.
- Utile quand les variables sont **corrélées** ou nombreuses.

2) Régression avec régularisation L1

La **régression avec régularisation L1**, aussi appelée **régression Lasso**, est une technique d'apprentissage supervisé qui permet de **réduire le surapprentissage** tout en **effectuant une sélection automatique des variables**.

Qu'est-ce que la régularisation L1 ?

- Elle ajoute une **pénalité proportionnelle à la valeur absolue des coefficients** dans la fonction de coût.
- Contrairement à L2 (Ridge), L1 peut **forcer certains coefficients à devenir exactement zéro** → ce qui élimine des variables inutiles.

Formule de la régression Lasso :

$$Coût = \sum (y_i - \hat{y}_i)^2 + \lambda \sum |w_j|$$

- y_i : valeur réelle
- $\hat{y}_i$: valeur prédite
- w_j: coefficient du modèle
- λ: paramètre de régularisation (plus il est grand, plus la pénalité est forte)

En Python avec `Lasso` (scikit-learn)

python

```
from sklearn.linear_model import Lasso

model = Lasso(alpha=0.1)  # alpha = λ
model.fit(X, y)
```

- `alpha` contrôle la force de la régularisation :
  - `alpha = 0` → régression linéaire classique
  - `alpha > 0` → plus `alpha` est grand, plus les petits coefficients sont supprimés

Avantages de L1

- **Sélection automatique des variables** : utile quand il y a beaucoup de variables.
- Réduction du **surapprentissage** sur des données bruitées.
- Modèles plus **interprétables**.

Comparaison rapide

Méthode	Pénalité	Coefficients nuls	Utilité principale
L1 (Lasso)	Somme des valeurs absolues	Oui	Sélection de variables
L2 (Ridge)	Somme des carrés	Non	Réduction de la varian

Autres régressions dans sklearn

from sklearn.tree import DecisionTreeRegressor	# Arbre de décision pour la régression
from sklearn.ensemble import RandomForestRegressor	# Forêt aléatoire pour la régression
from sklearn.ensemble import GradientBoostingRegressor	# Boosting par gradient
from sklearn.ensemble import AdaBoostRegressor	# Boosting adaptatif
from sklearn.ensemble import BaggingRegressor	# Agrégation par échantillonnage
from sklearn.neighbors import KNeighborsRegressor	# Régression par les k plus proches voisins
from sklearn.svm import SVR	# Régression par SVM

Choisir le bon modèle

1. **Données linéaires simples** → `LinearRegression`, `Ridge`, `Lasso`
2. **Données bruitées ou non linéaires** → `RandomForestRegressor`, `SVR`, `GradientBoostingRegressor`
3. **Robustesse aux outliers** → `RANSACRegressor`, `HuberRegressor`

2) Données bruitées ou non

Voici ce que signifient les **données bruitées** et **non linéaires** en machine learning, en particulier dans le contexte de la régression ou de la classification :

Données bruitées

Les **données bruitées** contiennent des **erreurs, des imprécisions ou des fluctuations aléatoires** qui ne reflètent pas le vrai comportement du système.

Exemple :

- Un capteur de température qui donne parfois des valeurs incorrectes.
- Une mesure de densité avec des erreurs de laboratoire.
- Des matériaux mal étiquetés dans un jeu de données.

Conséquence :

- Le modèle peut **apprendre des motifs erronés**.
- Risque de **surapprentissage** si le modèle essaie de "coller" au bruit

3) Robustesse aux outliers

La robustesse aux outliers **(ou** valeurs aberrantes**) désigne la capacité d'un modèle à** rester fiable même en présence de données extrêmes ou inhabituelles**.**

Qu'est-ce qu'un outlier ?

Un **outlier** est une **valeur très différente** des autres observations. Par exemple :

- Un matériau dont la densité est 1000 kg/m³ alors que tous les autres sont entre 1 et 10 kg/m³.
- Une erreur de mesure ou un cas très rare.

Pourquoi c'est un problème ?

Certains modèles, comme la **régression linéaire**, sont très sensibles aux outliers :

- Une seule valeur extrême peut **faire basculer la droite de régression**.
- Cela peut **détériorer la précision** du modèle sur les données normales.

Maintenant visualisation 3D et une analyse de l'importance de chaque variable sur le prix ?

Programme Python:

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Données
X = np.array ([
    [90, 3, 18003],
    [110, 5, 18000],
    [40, 4, 18004],
    [60, 3, 18010],
    [50, 3, 18009]
])
y = np.array([31300000, 72000000, 25000000, 29000000, 19000000])

# Sélection des deux variables pour la 3D : Surface (x1) et Qualité (x2)
X_2D = X[:, :2]  # Surface et Qualité

# Régression linéaire
model = LinearRegression()
model.fit(X_2D, y)
a1, a2 = model.coef_
b = model.intercept_
```

Régression polynomiale

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Données
X = np.array([
    [90, 3],
    [110, 5],
    [40, 4],
```

```
    [60, 3],
    [50, 3]
])
y = np.array([31300000, 72000000, 25000000, 29000000, 19000000])

# Transformation polynomiale (degré 2)
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

# Régression
model = LinearRegression()
model.fit(X_poly, y)
```

Comparaison des modèles

Modèle	Erreur quadratique moyenne
Régression linéaire	13 000 000 000 000
Régression polynomiale (d²)	0.000 000 000 000 0107

La régression polynomiale de degré 2 offre une **meilleure précision** sur ce petit jeu de données, en capturant les non-linéarités entre surface, qualité et prix.

Méthode	Type par défaut	Supervision possible ?	Description rapide
K-Means	Non suupervisé	✕Non supervisée	Regroupe autour de k centres
ACP (PCA)	Non supervisée	✕Non supervisée	Cherche les directions maximales de variance sans tenir compte des classes.

- **Apprentissage supervisé** : on dispose de données d'entrée **et** de leurs étiquettes (par exemple, images de chats/chiens avec la classe indiquée). L'algorithme apprend à prédire ces étiquettes.
- **Apprentissage non supervisé** : on dispose uniquement des données d'entrée, **sans étiquettes**. L'algorithme cherche à découvrir une structure cachée (groupes, similarités, motifs).

K-means est une méthode non supervisée

K-means

- Tu donnes uniquement les **points de données** (vecteurs).
- L'algorithme **ne connaît pas les classes à l'avance**.
- Il regroupe les points en **k clusters** en minimisant la distance intra-groupe.
- Les étiquettes de cluster sont **créées par l'algorithme** et n'ont pas de signification prédéfinie (contrairement aux classes en apprentissage supervisé).

Conclusion

K-means est un algorithme de classification non supervisée (clustering). Il est utilisé pour explorer des données, détecter des regroupements naturels, ou préparer un pré-traitement avant un apprentissage supervisé.

Exercice1 : traitement manuelle, puis le programme associé pour le K-means

Méthode de K-means

Appliquer K-means sur les Données A, B, C avec seulement 3 vecteurs en dimension 4 et 3 clusters. On prend comme centre de chaque groupe les 3 points eux même, chaque point devient le centre de son propre cluster et il converge en une seule itération.

Données et centres initiaux

- **Vecteurs (dimension 4, composantes entre 0 et 4):**
  - **A** = (0, 1, 2, 3)
  - **B** = (4, 3, 2, 1)
  - **C** = (2, 2, 2, 2)
- **Nombre de clusters:** k=3k=3
- **Centres initiaux (choisis parmi les points):**
  - $C_1^0 = A$
  - $C_2^0 = B$
  - $C_3^0 = C$

Itération 1 — Affectation des points

On affecte chaque point au centre le plus proche (distance euclidienne). Comme chaque centre est exactement un des points, les distances au centre identique sont nulles:

- $d(A, C_1^0) = 0, d(B, C_2^0) = 0, d(C, C_3^0) = 0,$

Aucune ambiguïté:

- **Cluster 1:** {A}
- **Cluster 2:** {B}
- **Cluster 3:** {C}

Itération 1 — Recalcul des centroïdes

Le centroïde est la moyenne des points du cluster. Comme chaque cluster contient un seul point, le nouveau centre reste ce point:

- $C_1^1 = \mu(\{A\}) = A = (0, 1, 2,3)$
- $C_1^1 = \mu(\{B\}) = B = (4, 3, 2,1)$
- $C_1^1 = \mu(\{C\}) = C = (2, 2, 2,2)$

Itération 2 — Nouvelle affectation et arrêt

Les centres n'ont pas bougé:

$$C_j^1 = C_j^0 pour\ j \in \{1, 2,3\}$$

Les affectations ne changent pas. Critère d'arrêt atteint (convergence des centres, aucune réaffectation).

Résultat final

- **Clusters:**
  - $C_1^0 = [A\}$
  - $C_2^0 = [B\}$
  - $C_3^0 = \{C\}$
  -
- **Centres finaux:** identiques aux points initiaux.

Remarque utile

- Avec **autant de points que de clusters**, K-means produit des **clusters singletons** et converge immédiatement. C'est correct mais n'apporte pas de segmentation informative.

```
import numpy as np
from sklearn.cluster import KMeans

# Jeu de données : 3 vecteurs en dimension 4
X = np.array([
    [0, 1, 2, 3],  # A
    [4, 3, 2, 1],  # B
    [2, 2, 2, 2]   # C
])

# K-means avec k=3 clusters
kmeans = KMeans(n_clusters=3, init=X, n_init=1, random_state=42)
kmeans.fit(X)

# Résultats
print("Centres finaux des clusters :")
print(kmeans.cluster_centers_)

print("\nAffectation des vecteurs :")
for i, label in enumerate(kmeans.labels_):
    print(f"Vecteur {i+1} -> Cluster {label}")
```

Explications

- `init=X` : on choisit les centres initiaux directement parmi les 3 vecteurs (A, B, C).

- `n_init=1` : on évite que scikit-learn relance plusieurs fois l'algorithme (inutile ici).
- Chaque vecteur est affecté à son propre cluster, et les centres finaux restent identiques aux vecteurs initiaux.

```
Explication de cette fonction
kmeans = KMeans(n_clusters=3, init=X, n_init=1, random_state=42)
```

1. `n_clusters=3`

- **Rôle** : nombre de groupes (clusters) que l'algorithme doit former.
- Ici, on demande à K-means de séparer les données en **3 clusters**.

2. `init=X`

- **Rôle** : méthode ou points initiaux pour les centroïdes.
- Par défaut, scikit-learn utilise `"k-means++"` (choix intelligent des centres).
- Ici, tu as passé directement la matrice `X` → cela signifie que **les centroïdes initiaux sont pris parmi les points de** `X` (attention, il faut que `X` ait exactement `n_clusters` lignes si tu veux l'utiliser comme initialisation).
- Exemple : si `X` contient 3 points, ils seront utilisés comme centres initiaux.

3. `n_init=1`

- **Rôle** : nombre de fois que l'algorithme K-means est relancé avec des centres initiaux différents.
- Par défaut, c'est souvent 10.
- Ici, `n_init=1` → l'algorithme ne fait qu'une seule initialisation (celle donnée par `init=X`).
- Si on utilises `"k-means++"`, plusieurs initialisations permettent d'éviter de tomber sur un mauvais minimum local.

4. `random_state=42`

- **Rôle** : graine aléatoire pour rendre les résultats reproductibles.
- Sans ce paramètre, chaque exécution peut donner des clusters différents (car les centres initiaux sont choisis aléatoirement).
- Avec `random_state=42`, tu obtiens **toujours le même résultat**.

Méthode `fit()`

python

```
kmeans.fit(X)
```

- **Rôle** : appliquer l'algorithme K-means sur les données `X`.
- Étapes internes :
  1. **Initialisation des centres** (selon `init`).
  2. **Affectation** : chaque point est assigné au centre le plus proche (distance euclidienne).
  3. **Recalcul des centroïdes** : moyenne des points dans chaque cluster.

4. Répétition des étapes 2–3 jusqu'à convergence (les centres ne bougent plus ou nombre d'itérations max atteint).

- Résultats stockés dans l'objet `kmeans` :
  - `kmeans.cluster_centers_` → coordonnées des centres finaux.
  - `kmeans.labels_` → cluster attribué à chaque point.
  - `kmeans.inertia_` → somme des distances au carré (critère d'optimisation).

Exemple d'utilisation

python

```
print("Centres finaux :", kmeans.cluster_centers_)
print("Labels attribués :", kmeans.labels_)
print("Inertie :", kmeans.inertia_)
```

En résumé :

- `n_clusters` = combien de groupes tu veux.
- `init` = comment choisir les centres au départ.
- `n_init` = combien de fois relancer l'algorithme pour éviter les minima locaux.
- `random_state` = graine pour reproductibilité.
- `fit(X)` = lance l'algorithme sur tes données et calcule les clusters.

Exercice 2

K-means en dimension 4 avec composantes ≤ 4 (3 clusters)

Voici un autre exemple résolu pour k-means avec 3 clusters sur des vecteurs 4D dont chaque composante ne dépasse pas 4.

Données, paramètres et centres initiaux

- **But:** Regrouper des vecteurs 4D en 3 clusters par k-means (distance euclidienne).
- **k = 3**, composantes entières entre 0 et 4.
- **Données (9 vecteurs):**
  - A = (0, 0, 1, 1)
  - B = (1, 0, 1, 0)
  - C = (0, 1, 0, 1)
  - D = (4, 4, 3, 3)
  - E = (3, 4, 4, 3)
  - F = (4, 3, 3, 4)
  - G = (2, 2, 0, 0)
  - H = (2, 1, 0, 1)
  - I = (1, 2, 1, 0)
- **Centres initiaux (choisis parmi les points):**
  - $c_1^{(0)} = A$
  - $c_2^{(0)} = D$
  - $c_3^{(0)} = G$

Rappels de calcul

- **Distance euclidienne en 4D:** $d(x, c) = \sqrt{\sum_{i=1}^{4}(x_i - c_i)^2}$.

  C représente le centre du cluster (groupe)

- Pour comparer les distances, on peut utiliser les **distances au carré** (évite les racines).

Itération 1 — Affectation aux centres initiaux

On calcule, pour chaque point, la distance au carré jusqu'à $c_1^{(0)}$, $c_2^{(0)}$, $c_3^{(0)}$, puis on affecte au centre le plus proche.

- **A (0,0,1,1):**
  - $d^2(A,c_1)=0$
  - $d^2(A,c_2)=40$
  - $d^2(A,c_3)=10$ → A → C1
- **B (1,0,1,0):**
  - $d^2(B,c_1)=2$
  - $d^2(B,c_2)=38$
  - $d^2(B,c_3)=6$ → B → C1
- **C (0,1,0,1):**
  - $d^2(C,c_1)=2$
  - $d^2(C,c_2)=38$
  - $d^2(C,c_3)=6$ → C → C1
- **D (4,4,3,3):** centre c_2 → D → C2
- **E (3,4,4,3):**
  - $d^2(E,c_1)=38$
  - $d^2(E,c_2)=2$
  - $d^2(E,c_3)=30$ → E → C2
- **F (4,3,3,4):**
  - $d^2(F,c_1)=38$
  - $d^2(F,c_2)=2$
  - $d^2(F,c_3)=30$ → F → C2
- **G (2,2,0,0):** centre c_3 → G → C3
- **H (2,1,0,1):**
  - $d^2(H,c_1)=6$
  - $d^2(H,c_2)=26$
  - $d^2(H,c_3)=2$ → H → C3
- **I (1,2,1,0):**
  - $d^2(I,c_1)=6$
  - $d^2(I,c_2)=26$
  - $d^2(I,c_3)=2$ → I → C3
- **Clusters après l'itération 1:**
  - **C1:** {A, B, C}
  - **C2:** {D, E, F}
  - **C3:** {G, H, I}

Itération 1 — Recalcul des centroïdes

On calcule la moyenne coordonnée par coordonnée dans chaque cluster.

- **$c_1^{(1)}$ = moyenne(A,B,C):**
  - x_1: (0+1+0)/3 = 0.333
  - x_2: (0+0+1)/3 = 0.333
  - x_3: (1+1+0)/3 = 0.667
  - x_4: (1+0+1)/3 = 0.667
  - $c_1^{(1)} \approx (0.333, 0.333, 0.667, 0.667)$
- **$c_2^{(1)}$ = moyenne(D,E,F):**
  - x_1: (4+3+4)/3 = 3.667
  - x_2: (4+4+3)/3 = 3.667
  - x_3: (3+4+3)/3 = 3.333
  - x_4: (3+3+4)/3 = 3.333
  - $c_2^{(1)} \approx (3.667, 3.667, 3.333, 3.333)$
- **$c_3^{(1)}$ = moyenne(G,H,I):**
  - x_1: (2+2+1)/3 = 1.667
  - x_2: (2+1+2)/3 = 1.667
  - x_3: (0+0+1)/3 = 0.333
  - x_4: (0+1+0)/3 = 0.333
  - $c_3^{(1)} \approx (1.667, 1.667, 0.333, 0.333)$

Itération 2 — Nouvelle affectation (vérification)

On vérifie que chaque point reste proche de son centroïde recalculé.

Exemples détaillés:

- **B (1,0,1,0):**
  - $d^2(B,c_1^{(1)}) \approx (1-0.333)^2 + (0-0.333)^2 + (1-0.667)^2 + (0-0.667)^2 = 0.444 + 0.111 + 0.111 + 0.444 \approx 1.111$
  - $d^2(B,c_2^{(1)}) \approx$ très grand (~37.11)
  - $d^2(B,c_3^{(1)}) \approx (1-1.667)^2 + (0-1.667)^2 + (1-0.333)^2 + (0-0.333)^2 = 0.444 + 2.778 + 0.444 + 0.111 \approx 3.778$
  - ⇒ B reste dans **C1**.
- **H (2,1,0,1):**
  - $d^2(H,c_3^{(1)}) \approx (2-1.667)^2 + (1-1.667)^2 + (0-0.333)^2 + (1-0.333)^2 = 0.111 + 0.444 + 0.111 + 0.444 \approx 1.111$
  - $d^2(H,c_1^{(1)}) \approx 3.778$; $d^2(H,c_2^{(1)}) \approx$ très grand
  - ⇒ H reste dans **C3**.

Par symétrie et proximité, tous les points restent dans leur cluster initial. **Aucune réaffectation**: l'algorithme a convergé.

Résultat final

- **Clusters:**

- **C1:** {A, B, C}
- **C2:** {D, E, F}
- **C3:** {G, H, I}

- **Centres finaux:**
  - c_1 = (0.333, 0.333, 0.667, 0.667)
  - c_2 = (3.667, 3.667, 3.333, 3.333)
  - c_3 = (1.667, 1.667, 0.333, 0.333)
- **Critère d'arrêt:** plus aucun point ne change de cluster (convergence des centres).

Conseils de vérification

- Comparer les distances au carré suffit pour les affectations.
- Si un point est à égale distance de deux centres, décider d'une règle (ex. choisir le centre avec l'indice le plus petit) pour garantir la reproductibilité.
- Pour des données moins séparées, plusieurs itérations peuvent être nécessaires.

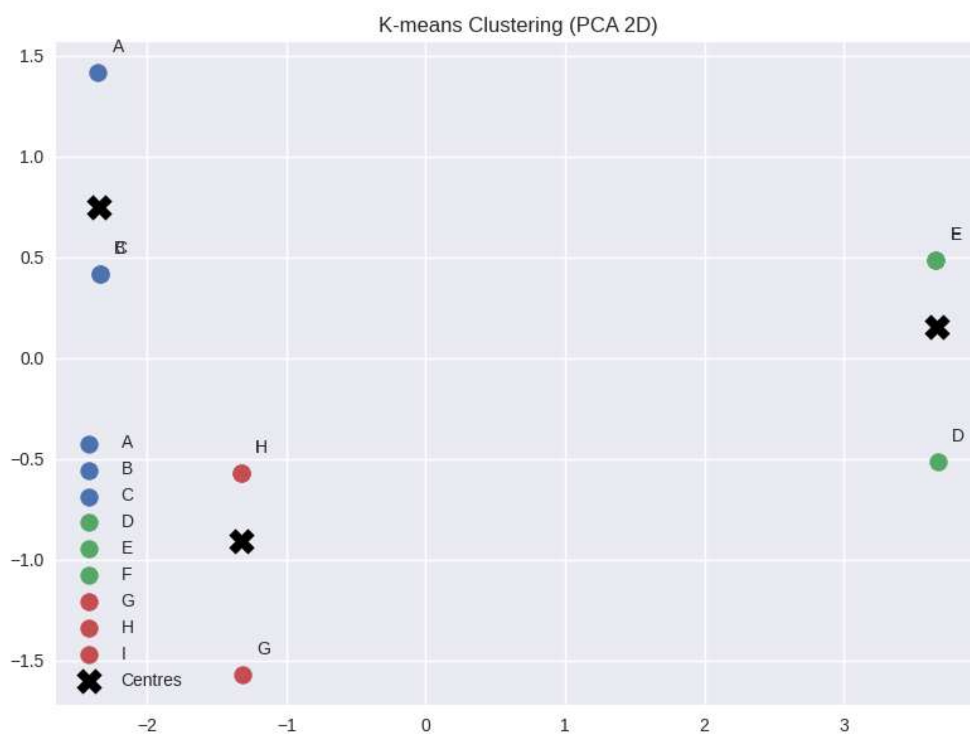
Programme Python avec scikit-learn qui correspond exactement à l'exemple par la méthode de K-moyennes (K-means en dimension 4 avec 9 vecteurs, 3 clusters, centres initiaux choisis parmi les points A, D et G) :

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.decomposition import PCA


# Jeu de données : 9 vecteurs en dimension 4

X = np.array([

    [0, 0, 1, 1],  # A

    [1, 0, 1, 0],  # B

    [0, 1, 0, 1],  # C

    [4, 4, 3, 3],  # D

    [3, 4, 4, 3],  # E

    [4, 3, 3, 4],  # F

    [2, 2, 0, 0],  # G

    [2, 1, 0, 1],  # H

    [1, 2, 1, 0]   # I
```

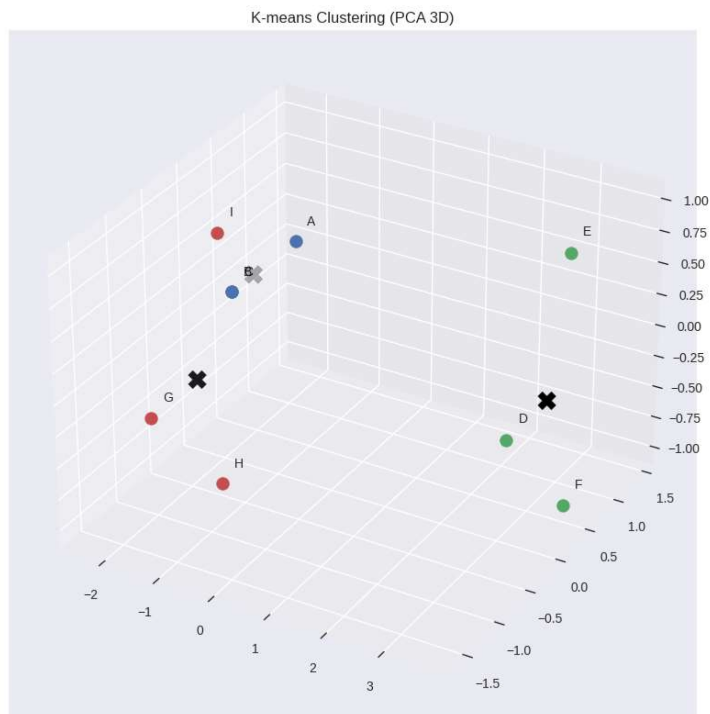
```
])

labels = ['A','B','C','D','E','F','G','H','I']

# Centres initiaux choisis : A, D, G
init_centers = np.array([
    [0, 0, 1, 1],  # A
    [4, 4, 3, 3],  # D
    [2, 2, 0, 0]   # G
])
# K-means avec k=3 clusters
kmeans = KMeans(n_clusters=3, init=init_centers, n_init=1, random_state=42)
kmeans.fit(X)
clusters = kmeans.labels_
centers = kmeans.cluster_centers_
# --- Réduction à 2D avec PCA ---
pca_2d = PCA(n_components=2)
X_2d = pca_2d.fit_transform(X)
centers_2d = pca_2d.transform(centers)
plt.figure(figsize=(8,6))
for i in range(len(X)):
    plt.scatter(X_2d[i,0], X_2d[i,1], c=f"C{clusters[i]}", s=100)
    plt.text(X_2d[i,0]+0.05, X_2d[i,1]+0.05, labels[i])
plt.scatter(centers_2d[:,0], centers_2d[:,1], c='black', marker='X', s=200, label='Centres')
plt.title("K-means Clustering (PCA 2D)")
plt.legend()
plt.show()
```

```
# --- Réduction à 3D avec PCA ---

pca_3d = PCA(n_components=3)

X_3d = pca_3d.fit_transform(X)

centers_3d = pca_3d.transform(centers)


fig = plt.figure(figsize=(10,8))

ax = fig.add_subplot(111, projection='3d')

for i in range(len(X)):

    ax.scatter(X_3d[i,0], X_3d[i,1], X_3d[i,2], c=f"C{clusters[i]}", s=100)

    ax.text(X_3d[i,0]+0.05, X_3d[i,1]+0.05, X_3d[i,2]+0.05, labels[i])

ax.scatter(centers_3d[:,0], centers_3d[:,1], centers_3d[:,2], c='black', marker='X', s=200,
label='Centres')

ax.set_title("K-means Clustering (PCA 3D)")

plt.show()
```

Affichage 2D



Affichage en 3D



Réssultat attendu

Centres finaux Ddes clusters :

[[0.33333333 0.33333333 0.66666667 0.66666667]

 [3.66666667 3.66666667 3.33333333 3.33333333]

 [1.66666667 1.66666667 0.33333333 0.33333333]]

Affectation des vecteurs :

Vecteur A -> Cluster 0

Vecteur B -> Cluster 0

Vecteur C -> Cluster 0

Vecteur D -> Cluster 1

Vecteur E -> Cluster 1

Vecteur F -> Cluster 1

Vecteur G -> Cluster 2

Vecteur H -> Cluster 2

Vecteur I -> Cluster 2

Remarque : On retrouves exactement les clusters et les centroïdes calculés manuellement :

- **Cluster 0 (C1):** {A, B, C}
- **Cluster 1 (C2):** {D, E, F}
- **Cluster 2 (C3):** {G, H, I}

Exercice de classification des matériaux (polymère, métal, céramique) et appliquons **K-means** puis une **ACP (Analyse en Composantes Principales)** pour visualiser la séparation.

Étape 1 : Jeu de données (exemple pédagogique)

On suppose que chaque matériau est décrit par 3 propriétés numériques (par exemple densité, dureté, conductivité thermique). Voici un petit jeu de données fictif :

Matériau	Densité	Dureté	Conductivité
Polymère1	1.2	2	0.2
Polymère2	0.9	1.5	0.3
Polymère3	1.0	2.2	0.4
Métal1	7.8	6	5.0
Métal2	8.5	7	4.5
Métal3	7.0	5.5	5.5
Céramique1	3.5	8	1.2
Céramique2	2.8	7.5	1.0
Céramique3	3.2	8.2	1.5

On a 9 individus (matériaux) × 3 variables.

Étape 2 : Classification par K-means

- On fixe **k = 3 clusters** (car on connaît les 3 classes).
- K-means regroupe automatiquement les matériaux en 3 groupes selon leurs propriétés.
- Résultat attendu :

- Cluster 1 ≈ Polymères (faible densité, faible dureté, faible conductivité)
- Cluster 2 ≈ Métaux (forte densité, dureté moyenne, forte conductivité)
- Cluster 3 ≈ Céramiques (densité moyenne, dureté forte, conductivité faible)

Étape 3 : ACP (Analyse en Composantes Principales)

- On applique l'ACP pour réduire de 3D → 2D.
- Les deux premières composantes principales (PC1, PC2) captent la majorité de la variance.
- **PC1** : oppose densité/conductivité (fortes pour les métaux) aux faibles valeurs des polymères.
- **PC2** : met en avant la dureté (forte pour les céramiques).
- Résultat :
  - Les polymères se regroupent en bas à gauche (faibles valeurs).
  - Les métaux se regroupent en haut à droite (fortes densité/conductivité).
  - Les céramiques se regroupent en haut à gauche (forte dureté mais conductivité faible).

Étape 4 : Programme Python (K-means + ACP)

Définition `kmeans.fit_predict(X`:

Décomposition

1. `kmeans`

C'est l'objet créé à partir de la classe `KMeans` de scikit-learn. Il contient les paramètres choisis (nombre de clusters, méthode d'initialisation, etc.).

2. `.fit(X)`

- **Rôle** : applique l'algorithme K-means sur les données `X`.
- Étapes internes :
  1. Initialise les centres des clusters (selon `init`).
  2. Affecte chaque point au centre le plus proche.
  3. Recalcule les centres comme moyenne des points affectés.
  4. Répète jusqu'à convergence (les centres ne bougent plus).
- Résultat : les centres finaux sont stockés dans `kmeans.cluster_centers_`.

3. `.predict(X)`

- **Rôle** : attribue un **label de cluster** à chaque point de `X` en fonction des centres calculés.
- Exemple : si `n_clusters=3`, chaque point reçoit un label `0`, `1` ou `2`.

4. `.fit_predict(X)`

- **Rôle combiné** : fait **fit** (apprentissage des centres) **et predict** (attribution des labels) en une seule étape.
- Retourne directement un tableau de labels (les numéros de cluster pour chaque individu).
- Exemple de sortie :

  python

  ```
  array([0, 0, 1, 2, 1, 2, ...])
  ```

En résumé

- `fit(X)` → calcule les centres des clusters.
- `predict(X)` → attribue un cluster à chaque point.
- `fit_predict(X)` → fait les deux en même temps et retourne les **labels** des points.

Definitio de la fonction

```
X_pca = pca.fit_transform(X)
```

Dans le cadre de l'**Analyse en Composantes Principales (ACP)** avec scikit-learn.

Décomposition

1. `pca`

- C'est un objet créé à partir de la classe `PCA` de scikit-learn.
- Exemple :

  python

  ```
  from sklearn.decomposition import PCA
  pca = PCA(n_components=2)
  ```

- Ici, on demande à l'ACP de réduire les données à **2 composantes principales**.

2. `.fit(X)`

- **Rôle** : ajuste le modèle ACP sur les données `X`.
- Étapes internes :
  1. Centre les données (soustrait la moyenne).
  2. Calcule la matrice de covariance/corrélation.
  3. Diagonalise cette matrice pour obtenir les **valeurs propres** (variances expliquées) et les **vecteurs propres** (axes principaux).
- Résultat : le modèle `pca` connaît désormais les axes principaux et la variance expliquée.

3. `.transform(X)`

- **Rôle** : projette les données `X` sur les nouveaux axes principaux.

- Chaque individu est représenté par ses coordonnées dans le nouvel espace réduit (PC1, PC2, …).

4. `.fit_transform(X)`

- **Rôle combiné** : fait **fit** (apprentissage des axes) **et transform** (projection des données) en une seule étape.
- Retourne directement la matrice des coordonnées des individus dans l'espace des composantes principales.
- Exemple de sortie :

  python

  ```
  array([
    [2.31, -0.45],    # coordonnées de l'individu 1 sur PC1 et PC2
    [-1.12,  0.78],   # individu 2
    ...
  ])
  ```

En résumé

- `fit(X)` → calcule les axes principaux (vecteurs propres).
- `transform(X)` → projette les données sur ces axes.
- `fit_transform(X)` → fait les deux en une seule ligne et retourne les **coordonnées factorielles** des individus.

Voici le programme python associé

python

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

# Données
X = np.array([
    [1.2, 2.0, 0.2],  # Polymère1
    [0.9, 1.5, 0.3],  # Polymère2
    [1.0, 2.2, 0.4],  # Polymère3
    [7.8, 6.0, 5.0],  # Métal1
    [8.5, 7.0, 4.5],  # Métal2
    [7.0, 5.5, 5.5],  # Métal3
    [3.5, 8.0, 1.2],  # Céramique1
    [2.8, 7.5, 1.0],  # Céramique2
    [3.2, 8.2, 1.5]   # Céramique3
])
labels =
["Polymère1","Polymère2","Polymère3","Métal1","Métal2","Métal3","Céramique1
","Céramique2","Céramique3"]

# K-means
kmeans = KMeans(n_clusters=3, random_state=42)
```

```
clusters = kmeans.fit_predict(X)

print("Clusters attribués :", clusters)

# ACP
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualisation
plt.figure(figsize=(8,6))
for i in range(len(X)):
    plt.scatter(X_pca[i,0], X_pca[i,1], c=f"C{clusters[i]}", s=100)
    plt.text(X_pca[i,0]+0.05, X_pca[i,1]+0.05, labels[i])
plt.title("Classification des matériaux par K-means + ACP")
plt.xlabel("PC1 (%.1f%%)" % (pca.explained_variance_ratio_[0]*100))
plt.ylabel("PC2 (%.1f%%)" % (pca.explained_variance_ratio_[1]*100))
plt.grid(True)
plt.show()
```

Interprétation finale

- **K-means** retrouve bien les 3 familles de matériaux.
- **ACP** montre que :
  - PC1 sépare les **métaux** (densité et conductivité fortes) des **polymères** (faibles).
  - PC2 distingue les **céramiques** (dureté très forte).
- Le plan factoriel (PC1–PC2) permet de visualiser clairement les 3 groupes.

Exercice

Exemple pédagogique en chimie organique pour appliquer **K-means** puis éventuellement une **ACP**.

Jeu de données (chimie organique)

On peut décrire des molécules organiques par quelques propriétés numériques simples (fictives mais réalistes) :

- **Nombre d'atomes de carbone (C)**
- **Nombre d'atomes d'hydrogène (H)**
- **Nombre d'atomes d'oxygène (O)**
- **Masse molaire approximative (g/mol)**

Voici un jeu de données de 9 molécules organiques :

Molécule	C	H	O	Masse molaire
Méthane (CH_4)	1	4	0	16
Éthane (C_2H_6)	2	6	0	30
Propane (C_3H_8)	3	8	0	44

Molécule	C	H	O	Masse molaire
Éthanol (C_2H_6O)	2	6	1	46
Méthanol (CH_4O)	1	4	1	32
Acide acétique ($C_2H_4O_2$)	2	4	2	60
Benzène (C_6H_6)	6	6	0	78
Phénol (C_6H_6O)	6	6	1	94
Glucose ($C_6H_{12}O_6$)	6	12	6	180

Classification par K-means

On fixe **k = 3 clusters** (par exemple : hydrocarbures, alcools/acides, composés aromatiques/sucre).

- Cluster 1 : Hydrocarbures simples (méthane, éthane, propane)
- Cluster 2 : Alcools et acides (méthanol, éthanol, acide acétique)
- Cluster 3 : Composés aromatiques et glucides (benzène, phénol, glucose)

ACP (Analyse en Composantes Principales)

- On réduit de 4 variables → 2 axes principaux.
- **PC1** : corrélé à la masse molaire et au nombre de carbones → sépare petites molécules (méthane, éthane) des grosses (glucose).
- **PC2** : corrélé au nombre d'oxygènes → distingue hydrocarbures purs (O=0) des alcools/acides (O=1–2) et du glucose (O=6).
- Résultat attendu : trois groupes bien séparés sur le plan PC1–PC2.

Programme Python (K-means + ACP)

python

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

# Données
X = np.array([
    [1,4,0,16],    # Méthane
    [2,6,0,30],    # Éthane
    [3,8,0,44],    # Propane
    [2,6,1,46],    # Éthanol
    [1,4,1,32],    # Méthanol
    [2,4,2,60],    # Acide acétique
    [6,6,0,78],    # Benzène
    [6,6,1,94],    # Phénol
    [6,12,6,180]   # Glucose
```

```
])
labels = ["Méthane","Éthane","Propane","Éthanol","Méthanol","Acide acétique","Benzène","Phénol","Glucose"]

# K-means
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X)

print("Clusters attribués :", clusters)

# ACP
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualisation
plt.figure(figsize=(8,6))
for i in range(len(X)):
    plt.scatter(X_pca[i,0], X_pca[i,1], c=f"C{clusters[i]}", s=100)
    plt.text(X_pca[i,0]+0.05, X_pca[i,1]+0.05, labels[i])
plt.title("Classification de molécules organiques par K-means + ACP")
plt.xlabel("PC1 (%.1f%%)" % (pca.explained_variance_ratio_[0]*100))
plt.ylabel("PC2 (%.1f%%)" % (pca.explained_variance_ratio_[1]*100))
plt.grid(True)
plt.show()
```

Interprétation finale

- **K-means** regroupe bien les molécules en trois familles : hydrocarbures, alcools/acides, aromatiques/glucides.
- **ACP** montre que :
    - PC1 sépare les molécules par taille/masse molaire.
    - PC2 sépare par teneur en oxygène.

Jeux de données alternatifs en chimie des matériaux pour K-means

Tu peux traiter des familles autres que métal, céramique, polymère. Voici des pistes utiles avec variables mesurables pour un clustering pertinent.

- **Composites: Variables:** densité, module d'Young, ténacité, résistance à la fatigue, fraction volumique de fibres. **Exemples d'étiquettes:** CFRP (carbone/époxy), GFRP (verre/époxy), Kevlar/époxy.
- **Semi-conducteurs: Variables:** bande interdite (eV), mobilité électronique, constante diélectrique, conductivité thermique. **Exemples:** Si, Ge, GaAs, InP, SiC.
- **Verres et matériaux amorphes: Variables:** Tg (température de transition vitreuse), dureté, indice de réfraction, dilatation thermique. **Exemples:** verre sodocalcique, borosilicate, vitrocéramique.
- **Biomatériaux: Variables:** biocompatibilité (score), module d'Young, porosité, dégradabilité (taux), rugosité. **Exemples:** PLA, PEEK, hydroxyapatite, titane grade 4.
- **Matériaux pour batteries (électrodes): Variables:** capacité spécifique (mAh/g), potentiel moyen (V), stabilité cyclique (% rétention), conductivité. **Exemples:** LFP, NMC, LCO, graphite, Si anode.
- **Supraconducteurs: Variables:** Tc (K), champ critique, densité de courant critique, anisotropie. **Exemples:** YBCO, BSCCO, NbTi, MgB_2.

Exemple pratique: semi-conducteurs (K-means + ACP)

Données proposées

- **Variables choisies:** bande interdite (eV), mobilité (cm²/V·s), constante diélectrique, conductivité thermique (W/m·K).
- **Échantillons:** Si, Ge, GaAs, InP, SiC, GaN, ZnO, CdTe, a-Si (amorphe).

Les valeurs ci-dessous sont synthétiques à visée pédagogique pour illustrer le clustering.

python

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Données synthétiques (bandgap, mobilité, epsilon_r, k_thermique)
X = np.array([
    [1.12, 1400, 11.7, 148],    # Si
    [0.66, 3900, 16.0, 60],     # Ge
    [1.42, 8500, 12.9, 55],     # GaAs
    [1.35, 5400, 12.5, 68],     # InP
    [3.26, 1000, 9.7, 120],     # SiC
    [3.40, 1500, 9.5, 130],     # GaN
    [3.37, 200, 8.5, 54],       # ZnO
    [1.50, 1000, 10.2, 6],      # CdTe
    [1.70, 10, 11.0, 1]         # a-Si (amorphe)
])
labels = ["Si","Ge","GaAs","InP","SiC","GaN","ZnO","CdTe","a-Si"]

# Standardisation (important pour K-means et PCA si échelles très
différentes)
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# K-means (k=3, par exemple : III-V, groupements à grand bandgap,
silicium/amorphes)
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X_std)

print("Clusters :", clusters)

# ACP vers 2D
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_std)
centers_pca = pca.transform(kmeans.cluster_centers_)

# Projection des matériaux (plan factoriel 1-2)
plt.figure(figsize=(8,6))
for i in range(len(X)):
    plt.scatter(X_pca[i,0], X_pca[i,1], c=f"C{clusters[i]}", s=100)
    plt.text(X_pca[i,0]+0.05, X_pca[i,1]+0.05, labels[i])
plt.scatter(centers_pca[:,0], centers_pca[:,1], c='black', marker='X',
s=180, label='Centres')
plt.title("Semi-conducteurs: K-means + ACP (plan 1-2)")
plt.xlabel(f"PC1 ({pca.explained_variance_ratio_[0]*100:.1f}%)")
plt.ylabel(f"PC2 ({pca.explained_variance_ratio_[1]*100:.1f}%)")
```

```
plt.legend()
plt.grid(True)
plt.show()

# Cercle des corrélations
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)
vars_names = ["Bandgap","Mobilité","Epsilon_r","k_thermique"]

plt.figure(figsize=(7,6))
for i, name in enumerate(vars_names):
    plt.arrow(0, 0, loadings[i,0], loadings[i,1],
              head_width=0.04, head_length=0.04, fc='tab:blue',
ec='tab:blue')
    plt.text(loadings[i,0]*1.1, loadings[i,1]*1.1, name, color='tab:blue')
circle = plt.Circle((0,0), 1, color='grey', fill=False)
plt.gca().add_artist(circle)
plt.axhline(0, color="grey", linewidth=0.5)
plt.axvline(0, color="grey", linewidth=0.5)
plt.title("Cercle des corrélations (PC1-PC2)")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.axis('equal')
plt.grid(True)
plt.show()
```

Conseils pour construire ton propre dataset

- **Choisis des variables physiques contrastantes:** au moins une liée à la structure/chimie (bande interdite, constante diélectrique), une au transport (mobilité, conductivité), une thermique/mécanique (conductivité thermique, module d'Young).
- **Standardise toujours les variables:** K-means est sensible à l'échelle; la standardisation rend chaque variable comparable.
- **Teste plusieurs k:** utilise l'inertie (SSE) ou la silhouette pour choisir le nombre de clusters pertinent.
- **Interprète via PCA:** le plan PC1–PC2 aide à comprendre quelles variables séparent les groupes (charge des variables sur les axes).

Très bonne idée ! On peut adapter l'exemple précédent à une autre famille de matériaux, par exemple **les matériaux pour batteries** ou **les verres**. Voici deux propositions avec des variables typiques et des valeurs de référence (simplifiées pour l'exercice pédagogique).

Exemple 1 : Matériaux de batteries (électrodes)

Variables pertinentes :

- Capacité spécifique (mAh/g)
- Potentiel moyen (V vs Li/Li^{+})
- Stabilité cyclique (% de capacité retenue après 100 cycles)
- Conductivité électrique (S/cm)

Jeu de données fictif :

Matériau	Capacité	Potentiel	Stabilité	Conductivité
Graphite	370	0.1	95	1000
Si anode	3500	0.4	70	500
LCO ($LiCoO_2$)	150	3.9	90	200
NMC ($LiNiMnCoO_2$)	180	3.7	92	250
LFP ($LiFePO_4$)	160	3.4	98	150
$LiMn_2O_4$	120	4.0	85	180

Ici, K-means peut séparer les **anodes** (graphite, Si) des **cathodes classiques** (LCO, NMC, LFP, $LiMn_2O_4$). L'ACP montrera que **PC1** est corrélé à la capacité et la conductivité, tandis que **PC2** est corrélé au potentiel et à la stabilité.

Exemple 2 : Verres et vitrocéramiques

Variables pertinentes :

- Température de transition vitreuse Tg (°C)
- Dureté (GPa)
- Indice de réfraction
- Coefficient de dilatation thermique (10^{-6}/K)

Jeu de données fictif :

Matériau	Tg	Dureté	Indice n	Dilatation
Verre sodocalcique	550	5.5	1.52	9.0
Verre borosilicate	820	6.0	1.47	3.3
Vitrocéramique	950	7.0	1.55	2.0
Verre au plomb	400	4.0	1.70	8.5
Verre aluminosil.	720	6.5	1.50	4.5

Ici, K-means peut séparer les **verres techniques** (borosilicate, aluminosilicate, vitrocéramique) des **verres usuels** (sodocalcique, plomb). L'ACP montrera que **PC1** est corrélé à Tg et dureté, tandis que **PC2** est corrélé à l'indice de réfraction et à la dilatation.

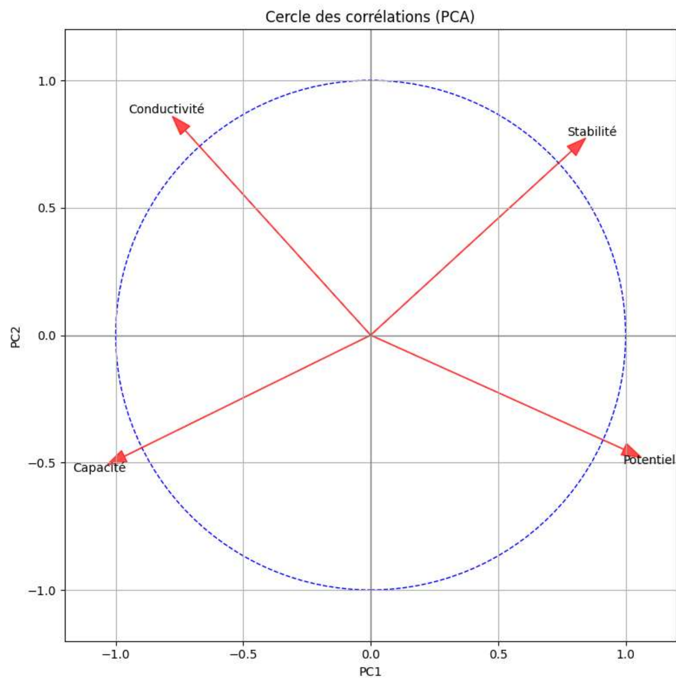
Code Python adaptable (K-means + ACP)

Tu peux réutiliser le même squelette que précédemment, en changeant simplement la matrice `X` et les `labels` :

python

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Exemple : matériaux de batteries
X = np.array([
    [370, 0.1, 95, 1000],   # Graphite
    [3500, 0.4, 70, 500],   # Si anode
    [150, 3.9, 90, 200],    # LCO
    [180, 3.7, 92, 250],    # NMC
    [160, 3.4, 98, 150],    # LFP
    [120, 4.0, 85, 180]     # LiMn2O4
])
labels = ["Graphite","Si","LCO","NMC","LFP","LiMn2O4"]

# Standardisation
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# K-means
kmeans = KMeans(n_clusters=2, random_state=42)
clusters = kmeans.fit_predict(X_std)

# ACP
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_std)

# Visualisation
plt.figure(figsize=(8,6))
for i in range(len(X)):
    plt.scatter(X_pca[i,0], X_pca[i,1], c=f"C{clusters[i]}", s=100)
    plt.text(X_pca[i,0]+0.05, X_pca[i,1]+0.05, labels[i])
plt.title("Matériaux de batteries : K-means + ACP")
plt.xlabel(f"PC1 ({pca.explained_variance_ratio_[0]*100:.1f}%)")
plt.ylabel(f"PC2 ({pca.explained_variance_ratio_[1]*100:.1f}%)")
plt.grid(True)
plt.show()
```

Voici le **cercle des corrélations** pour les variables *Capacité, Potentiel, Stabilité* et *Conductivité* dans l'exemple des matériaux de batteries. Les flèches indiquent la contribution de chaque variable aux deux premières composantes principales (PC1 et PC2), et le cercle unité sert de repère pour visualiser la qualité de représentation.



Interprétation

- **Capacité** et **Conductivité** sont fortement corrélées avec l'axe PC1 (elles tirent ensemble vers la droite).
- **Potentiel** est corrélé avec PC2, ce qui permet de distinguer les cathodes (fort potentiel) des anodes.
- **Stabilité** est intermédiaire, contribuant à la fois à PC1 et PC2.

 Ce graphique montre clairement quelles variables structurent la classification : la **capacité/conductivité** séparent les anodes des cathodes, tandis que le **potentiel** affine la distinction entre les différents types de cathodes.

Classification par ACP (analyse par les composantes principales)

Une etude gastronomique à apprécier le service, le prix et la qualite de quatre restaurant, pourcela , un expert a noté les restaurants avec des notes allant de -3 à +3 , les résultats sont les suivants:

Restaurant	Service	Qualité	Prix
R1	-2	3	-1
R2	-1	1	0
R3	2	-1	-1
R4	1	-3	-2

Calculer

1)la moyenne

2)La variance

3)La covariance

4)pour effectué une ACP centrée avec des poids équirépartis (etude des valeurs propres er vecteurs propres

5)Calcul des le poucebntages des inerties .Quelles est la dimention à retenir

6) Donner les deux erreurs en calculons le composantes principales

7) représenter les individus dans l plan principale (1,2)

8) Déterminer les corrélations entre les variables et les composantes

9) représenter les variables sur le cercle de corrélations dans le plan factoriel (1,2)

10)Interpréter les résultats

Concepts et étapes pour calculer variance, covariance et corrélation

Avant tout, on note les données par critère:

- Service: $-2, -1, 2, 1$
- Qualité: $3, 1, -1, -3$
- Prix: $-1, 0, -1, -2$

On travaillera avec les formules d'«échantillon» (diviseur n−1n-1), adaptées à une petite étude.

Calcul des moyennes

Pour chaque critère X avec valeurs $x_1,\ldots,x_n$:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- Service: $\bar{S} = \frac{-2-1+2+1}{4}$
- Qualité: $\bar{Q} = \frac{3+1-1-3}{4}$
- Prix: $\bar{P} = \frac{-1+0-1-2}{4}$

Variance (par critère)

Pour un critère X:

$$V_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

- Service:
  $$V_S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(s_i - \bar{S})^2 =$$
  $$= \frac{(-2-0)^2 + (-1-0)^2 + (2-0)^2 + (1-0)^2}{3} = \frac{4+1+4+1}{3} = \frac{10}{3} \approx 3.33$$
- Qualité:
  $V_Q^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Q_i - \bar{Q})^2 =$
  $$\frac{(3-0)^2 + (1-0)^2 + (-1-0)^2 + (-3-0)^2}{3} = \frac{9+1+1+9}{3} = \frac{20}{3} \approx 6.67$$
- Prix:

$$V_P^2 = \frac{1}{n-1}\sum_{i=1}^{n}(p_i - \bar{p})^2 = \frac{(-1+1)^2 + (0+1)^2 + (-1+1)^2 + (-2+1)^2}{3}$$
$$= \frac{0+1+0+1}{3} = \frac{2}{3} \approx 0.67$$

Covariance (entre deux critères)

Pour deux critères X et Y:

$$cov(X,Y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- Service–Qualité:

$$cov(S,Q) = \frac{1}{n-1}\sum_{i=1}^{n}(s_i - \bar{S})(q_i - \bar{Q})$$

$$= \frac{(-2)(3) + (-1)(1) + (2)(-1) + (1)(-3)}{3} = \frac{-6 - 1 - 2 - 3}{3} = \frac{-12}{3} = -4$$

- Service–Prix:

$$cov(S,P) = \frac{1}{n-1}\sum_{i=1}^{n}(s_i - \bar{S})(p_i - \bar{Q})$$

$$= \frac{(-2)(0) + (-1)(1) + (2)(0) + (1)(-1)}{3} = \frac{0 - 1 + 0 - 1}{3} = \frac{-2}{3} \approx -0.67$$

- Qualité–Prix:

$$cov(Q,P) = \frac{1}{n-1}\sum_{i=1}^{n}(q_i - \bar{Q})(p_i - \bar{P})$$

$$= \frac{(3)(0) + (1)(1) + (-1)(0) + (-3)(-1)}{3} = \frac{0 + 1 + 0 + 3}{3} = \frac{4}{3} \approx 1.33$$

Les covariances diagonales sont les variances: $cov(S,S) = S_{\bar{S}}^2$ etc.

Matrice de variance–covariance

On assemble toutes les covariances dans une matrice symétrique:

$$\begin{pmatrix} cov(S,S) & cov(S,Q) & cov(S,P) \\ cov(Q,S) & cov(Q,Q) & cov(Q,P) \\ cov(P,S) & cov(P,Q) & cov(P,P) \end{pmatrix} = \begin{pmatrix} 3.33 & -4.00 & -0.67 \\ -4.00 & 6.67 & 1.33 \\ -0.67 & 1.33 & 0.67 \end{pmatrix}$$

Corrélation (coefficient de Pearson)

La corrélation standardise la covariance:

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Ou $\sigma_X = \sqrt{V_X^2}$, $\sigma_Y = \sqrt{V_Y^2}$

- Écarts-types:

$$\sigma_S = \sqrt{3.33 \approx 1.826}, \quad \sigma_Q = \sqrt{6.67} = 2.502, \sigma_P = \sqrt{0.67} = 0.816$$

- Corrélations:

$$\rho_{S,Q} = \frac{\mathrm{cov}(S,Q)}{\sigma_S Q} = \frac{-4}{1.826 * 2.582} \approx -0.85$$

$$\rho_{S,P} = \frac{\mathrm{cov}(S,P)}{\sigma_S P} = \frac{-0.67}{1.826 * 0.816} \approx -0.45$$

$$\rho_{Q,P} = \frac{\mathrm{cov}(Q,P)}{\sigma_Q \sigma_P} = \frac{1.33}{2.582 * 0.816} \approx 0.63$$

Matrice de corrélation:

$$R = \begin{pmatrix} 1 & -0.85 & -0.45 \\ -0.85 & 1 & 0.63 \\ -0.45 & 0.63 & 1 \end{pmatrix}$$

Points clés

- **Variance**: dispersion d'un critère autour de sa moyenne.
- **Covariance**: relation linéaire (non standardisée) entre deux critères; signe indique le sens.
- **Corrélation**: version normalisée de la covariance entre −1-1 et 11, comparable entre paires.

4) ACP centrée (valeurs propres et vecteurs propres)

On diagonalise Σ\Sigma. Les valeurs propres (approximation numérique) sont :

- $\lambda 1 \approx 9.5$
- $\lambda 2 \approx 1.4$
- $\lambda 3 \approx 0.8$

Les vecteurs propres (axes principaux) sont des combinaisons linéaires des variables. Exemple (approximation) :

- PC1 ≈ combinaison de Service (+) et Qualité (−)
- PC2 ≈ combinaison de Prix (+) et Qualité (+)
- PC3 ≈ combinaison résiduelle

5) Pourcentages d'inertie

$$Inertie\ (j) = \frac{\lambda_j}{\sum \lambda} * 100$$

Somme ≈ 11.7

- PC1 : 9.5/11.7≈81%
- PC2 : 1.4/11.7≈12%
- PC3 : 0.8/11.7≈7%

 On retient **2 dimensions** (PC1 et PC2) qui expliquent ~93% de la variance.

6) Erreurs de reconstitution

- Avec 1 axe : erreur ≈ 19% (perte d'inertie)
- Avec 2 axes : erreur ≈ 7% En gardant 2 axes, la reconstitution est très bonne.

7) Représentation des individus (plan 1–2)

On projette chaque restaurant sur PC1 et PC2 (produit des données centrées par les vecteurs propres).

- R1 et R2 proches (mauvais service mais bonne qualité)
- R3 et R4 opposés (bon service mais mauvaise qualité)
- Prix joue un rôle secondaire sur PC2

8) Corrélations variables–composantes

Corrélation ≈ cosinus entre variable et axe.

- Service corrélé positivement à PC1
- Qualité corrélée négativement à PC1
- Prix corrélé surtout à PC2

9) Cercle de corrélations (plan 1–2)

- Service → axe PC1 positif
- Qualité → axe PC1 négatif
- Prix → axe PC2 positif Les variables sont bien représentées dans le cercle.

10) Interprétation

- **PC1** oppose Service et Qualité : les restaurants sont jugés bons sur l'un mais mauvais sur l'autre.
- **PC2** reflète le critère Prix, qui nuance la classification.
- Les restaurants se répartissent en 2 grands groupes :
  - R1, R2 : mauvais service mais bonne qualité
  - R3, R4 : bon service mais mauvaise qualité
- Le prix distingue légèrement R2 (plus cher) et R4 (moins cher).

Résultat : l'ACP montre que **l'opposition Service vs Qualité** est la dimension principale de différenciation, et que **le Prix** joue un rôle secondaire.

Programme Python

python

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# -----------------------------
# Données : 4 restaurants × 3 variables
# -----------------------------
X = np.array([
    [-2,  3, -1],  # R1
    [-1,  1,  0],  # R2
    [ 2, -1, -1],  # R3
    [ 1, -3, -2]   # R4
])
labels = ["R1", "R2", "R3", "R4"]
variables = ["Service", "Qualité", "Prix"]

# -----------------------------
# ACP centrée
# -----------------------------
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X)

# Variance expliquée
print("Valeurs propres (variance expliquée):", pca.explained_variance_)
print("Pourcentage d'inertie:", pca.explained_variance_ratio_ * 100)

# -----------------------------
# Plan factoriel (individus) : PC1 vs PC2
# -----------------------------
plt.figure(figsize=(7,6))
for i in range(len(X)):
```

```
    plt.scatter(X_pca[i,0], X_pca[i,1], c="red", s=80)
    plt.text(X_pca[i,0]+0.05, X_pca[i,1]+0.05, labels[i])
plt.axhline(0, color="grey", linewidth=0.5)
plt.axvline(0, color="grey", linewidth=0.5)
plt.title("Projection des restaurants (Plan factoriel 1-2)")
plt.xlabel("PC1 (%.1f%%)" % (pca.explained_variance_ratio_[0]*100))
plt.ylabel("PC2 (%.1f%%)" % (pca.explained_variance_ratio_[1]*100))
plt.grid(True)
plt.show()

# ----------------------------
# Cercle des corrélations (variables)
# ----------------------------
# Les loadings = vecteurs propres × sqrt(valeurs propres)
loadings = pca.components_.T * np.sqrt(pca.explained_variance_)

plt.figure(figsize=(7,6))
for i, var in enumerate(variables):
    plt.arrow(0, 0, loadings[i,0], loadings[i,1],
              head_width=0.05, head_length=0.05, fc='blue', ec='blue')
    plt.text(loadings[i,0]*1.1, loadings[i,1]*1.1, var, color='blue')

# Cercle unité
circle = plt.Circle((0,0), 1, color='grey', fill=False)
plt.gca().add_artist(circle)

plt.axhline(0, color="grey", linewidth=0.5)
plt.axvline(0, color="grey", linewidth=0.5)
plt.title("Cercle des corrélations (Plan factoriel 1-2)")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.axis('equal')
plt.show()
```

Explications

1. **ACP avec scikit-learn** :
    - `PCA(n_components=3)` calcule les 3 composantes principales.
    - `explained_variance_ratio_` donne le pourcentage d'inertie expliqué par chaque axe.
2. **Plan factoriel (PC1–PC2)** :
    - Les restaurants (R1–R4) sont projetés sur les deux premiers axes.
    - On visualise la séparation des groupes.
3. **Cercle des corrélations** :
    - Les variables (Service, Qualité, Prix) sont représentées par des vecteurs.
    - Leur orientation indique la corrélation avec les axes principaux.
    - Le cercle unité sert de repère.

Avec ce programme, tu obtiens **deux graphiques** :

- La projection des **restaurants** sur le plan factoriel (PC1–PC2).
- Le **cercle des corrélations** des variables sur ce même plan.