

```
    if u[i] == v[j]:
        l[i+1][j+1] = l[i][j] + 1
    else:
        l[i+1][j+1] = max(l[i][j+1],l[i+1][j])

return l[m][n]
```

4. Le code :

```
def plcs2(u, v):
    m = len(u)
    n = len(v)
    l = [[0 for __ in range(m+1)] for __ in range(n+1)]
    b = [[0 for __ in range(m+1)] for __ in range(n+1)]
    for i in range(m):
        for j in range(n):
            if u[i] == v[j]:
                l[i+1][j+1] = l[i][j] + 1
                b[i+1][j+1] = 0
            elif l[i][j+1] >= l[i+1][j]:
                l[i+1][j+1] = l[i][j+1]
                b[i+1][j+1] = 1
            else:
                l[i+1][j+1] = l[i+1][j]
                b[i+1][j+1] = 2
    path = []
    def build_plcs(b, u, i, j):
        nonlocal path
        if i*j == 0:
            return
        if b[i][j] == 0:
            build_plcs(b,u, i-1,j-1)
            path.append(u[i-1])
        elif b[i][j] == 1:
            build_plcs(b,u, i-1,j)
        else:
            build_plcs(b,u, i,j-1)

    build_plcs(b, u, m,n)

return l[m][n], "" .join(path)
```

Documentations complètes

Python3 : <https://docs.python.org/3/>
Numpy : <https://docs.scipy.org/doc/>
Matplotlib : <http://matplotlib.org/>
SQL : <http://sql.sh/>



Algorithmique pour l'intelligence artificielle

Sommaire

6.1	Introduction	91
6.2	Apprentissage supervisé et l'algorithme des k plus proches voisins	91
6.2.1	Des exemples jouets de classification	93
6.2.2	Un exemple plus sérieux	97
6.3	Apprentissage non-supervisé : Le clustering	100
6.3.1	Clustering ou partitionnement des données	100
6.3.2	L'algorithme des k -means de Forgy	101
6.3.3	La forte dépendance aux conditions initiales	101
6.3.4	Et k dans tout ça ?	106
6.3.5	Un jeu de données classique	110
6.3.6	Implémentation de k -means et k -means++	112

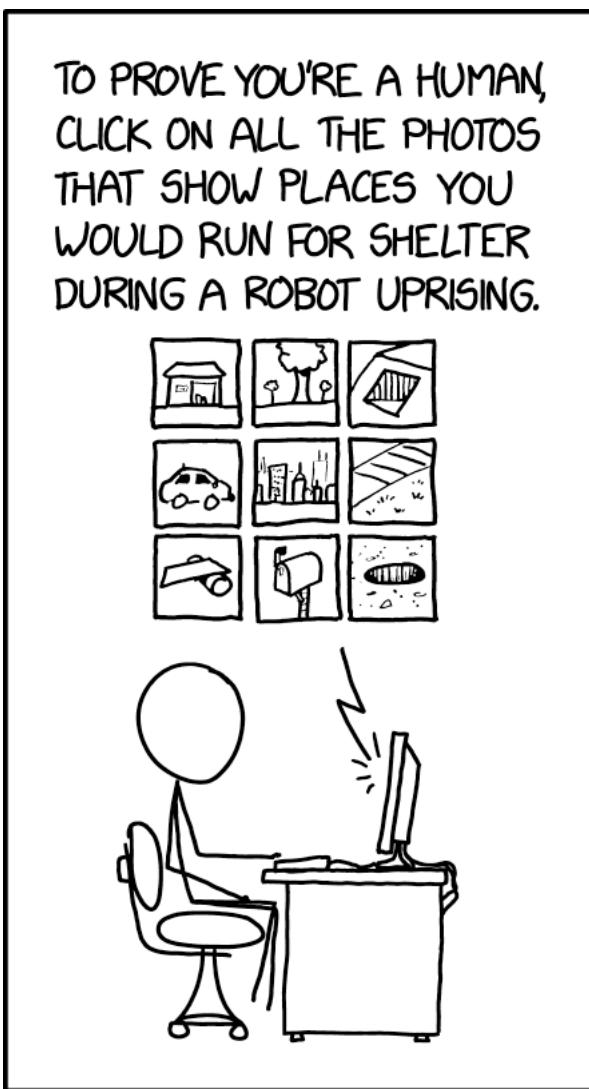


FIGURE 6.1 – From <https://xkcd.com/2228/>

6.1 Introduction

Le machine learning est un domaine de l'intelligence artificielle qui permet aux ordinateurs d'« apprendre » et de s'améliorer automatiquement à partir de données, plutôt que d'être explicitement programmés pour réaliser une tâche spécifique. Il utilise des algorithmes pour analyser des données et en extraire des modèles qui peuvent être utilisés pour effectuer des tâches telles que la reconnaissance de la parole, la reconnaissance d'images, la prévision de la demande et la détection de la fraude.

Il existe différents types de techniques de machine learning, comme l'apprentissage **supervisé**, l'apprentissage **non supervisé** et l'apprentissage **par renforcement**. Le machine learning est utilisé dans de nombreux domaines tels que la finance, la santé, la robotique et les médias pour résoudre des problèmes complexes et automatiser des processus.

Pour l'apprentissage supervisé l'algorithme est entraîné à partir de données étiquetées, c'est-à-dire que les données d'entraînement comportent des exemples avec des entrées et des sorties connues. L'objectif est qu'il puisse apprendre à prédire les étiquettes correspondantes pour des données non vues

Pour l'apprentissage non supervisé, le modèle est entraîné sur des données afin de découvrir les structures ou les relations dans les données. Les exemples courants incluent la segmentation de données (regroupement de données similaires), la réduction de dimensionnalité (réduction de la complexité des données) et l'agrégation de données (identification de tendances générales).

Enfin, pour l'apprentissage par renforcement un agent apprend à prendre des décisions en interagissant avec son environnement, en utilisant des récompenses pour maximiser ses performances. Les exemples courants incluent les systèmes de contrôle industriels, la robotique et les jeux vidéo.

Cette introduction a été écrite par ChatGPT, après plusieurs résultats étonnés.

6.2 Apprentissage supervisé et l'algorithme des k plus proches voisins

Il existe deux types d'apprentissage supervisé : la régression et la classification.

- La régression est utilisée pour prédire une variable continue, comme le prix d'une maison ou la température.
- La classification est utilisée pour prédire une variable discrète, comme le type d'une fleur ou la catégorie d'un article.

Pour entraîner un modèle d'apprentissage supervisé, on utilise généralement une technique d'optimisation pour ajuster les paramètres de l'algorithme de manière à minimiser l'erreur de prédiction sur les données d'entraînement. Ensuite, le modèle est testé sur des données de test pour évaluer sa performance et sa capacité à généraliser à des données inconnues.

Nous ne traiterons pas de régression dans ce cours, mais seulement de classification et de prédiction.

Le problème est le suivant : nous disposons d'un ensemble d'objets que l'on souhaite classer en différentes catégories. Les objets sont modélisés par des vecteurs de \mathbb{R}^d , dont chaque composante représente une caractéristique, on parle de **déscripteurs**. Par exemple on peut modéliser :

- un appartement par le vecteur de \mathbb{R}^8 :

(superficie_totale, nb_pièces, nb_salle_de_bain, nb_toilette, superficie_terrasse, distance_au_centre_ville, pri-

- une iris par le vecteur de \mathbb{R}^4 :

(longueur_sépale, largeur_sépale, longueur_pétale, largeur_pétale)

La méthode des k plus proches voisins (*KNN* ou *k-NN* pour *k* Nearest Neighbor) est un algorithme de **classification supervisée** utilisé pour prédire la classe d'un nouvel objet en se basant sur les existantes. Il fonctionne en comparant les caractéristiques de l'objet à prédire aux caractéristiques des objets dans la base de données d'apprentissage, et en sélectionnant les *k* objets les plus similaires (les *k* plus proches voisins) pour déterminer la classe de l'objet à prédire.

Pour utiliser l'algorithme *k-NN*, il faut d'abord définir la distance à utiliser pour calculer la similitude entre les objets. Cela peut être la distance Euclidienne, la distance de Manhattan, celle de Hamming, celle de Levenshtein ou toute autre distance définie par l'utilisateur. Ensuite, il faut définir la valeur de *k*, c'est-à-dire le nombre de voisins à considérer pour la prédiction. Plus la valeur de *k* est élevée, plus la prédiction sera basée sur un grand nombre de voisins.

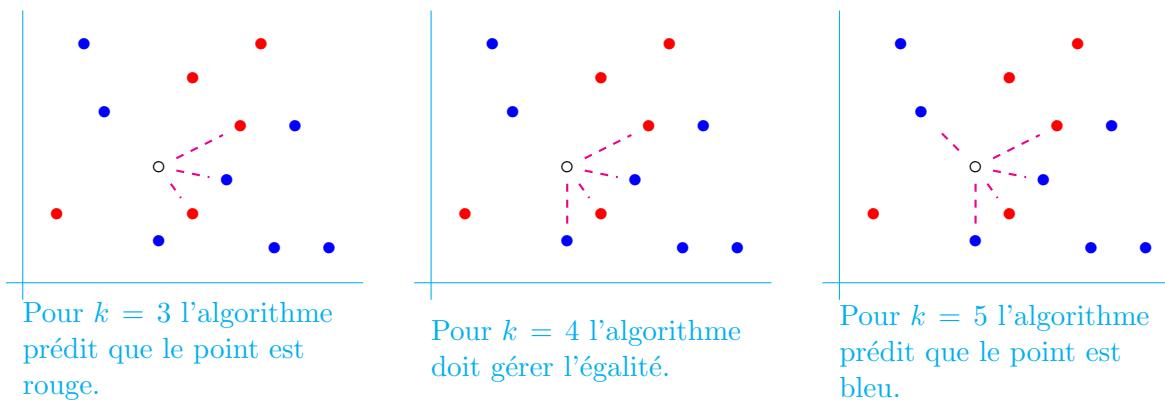


FIGURE 6.2 – Le principe de l'algorithme *k-NN*.

Voici quelques exemples d'applications courantes :

1. Classification d'images : *k-NN* peut être utilisé pour classifier des images en utilisant des caractéristiques telles que les couleurs et les textures. Par exemple, il peut être utilisé pour classifier des images de fruits en fonction de leurs couleurs et de leurs formes.
2. Recherche de similarité : *k-NN* peut être utilisé pour trouver des articles ou des produits similaires dans une base de données en utilisant des caractéristiques telles que les mots-clés et les caractéristiques de produit. Par exemple, il peut être utilisé pour trouver des produits similaires à un produit donné sur un site de commerce électronique.
3. Prédiction de la qualité de vin : *k-NN* peut être utilisé pour prédire la qualité d'un vin en utilisant des caractéristiques telles que les teneurs en sucre, en alcool et en acide.
4. Diagnostic médical : *k-NN* peut être utilisé pour diagnostiquer des maladies en utilisant des caractéristiques telles que les résultats des tests sanguins et les symptômes.
5. Prédiction des marchés financiers : *k-NN* peut être utilisé pour prédire les tendances des marchés financiers en utilisant des caractéristiques telles que les prix des actions et les indicateurs économiques.

Parmi les avantages de l'algorithme *k-NN* on trouve :

- Simplicité d'utilisation
- Peu de paramètres à ajuster

- Peut être utilisé pour des données à plusieurs dimensions

Parmi les inconvénients majeurs de l'algorithme k -NN on peut noter :

- Peut être lent pour des jeux de données volumineux
- Peut être influencé par des données bruitées ou aberrantes.

6.2.1 Des exemples jouets de classification

J'ai créé trois jeux de données de 400 points du plan de couleur rouge ou bleu de manière pseudo-aléatoire : un nuage, une croix et une couronne.

Chacun des jeux de données est découpé en deux sous ensembles :

- le `training_set` contenant 300 points, qui servira à entraîner le modèle
- le `testing_set` qui contient 100 points, qui servira à tester le modèle.

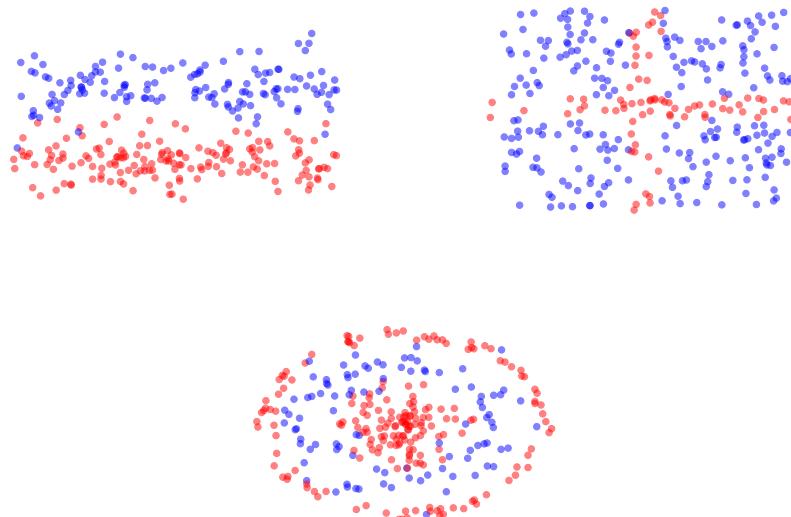


FIGURE 6.3 – Les `training_set` des jeux de données : nuage, croix et couronne.

J'ai utilisé l'algorithme k -NN pour prédire la couleur de chaque point du `testing_set` pour différentes valeurs de k , puis j'ai mesuré l'erreur commise par l'algorithme pour découvrir les points rouges.

Pour l'implémentation, les points sont représentés par des tuples (x, y, c) où (x, y) sont les coordonnées du point et c sa couleur.

```
def d2(pt1, pt2):
    return (pt1[0]-pt2[0])**2+(pt1[1]-pt2[1])**2

def k_nn(k, point, training_set, dist):
    kNN = sorted(training_set, key = lambda pt : dist(point, pt))[:k]
    colors = {}
    for nd in kNN:
        colors[nd[2]] = colors.get(nd[2], 0)+1
    return max(colors, key=lambda k: colors[k])
```

Les prédictions de l'algorithme peuvent être de quatre types :

- les vrais positifs (TP pour True Positive), ce sont les points rouges qui sont reconnus ;
- les faux positifs (FP pour False Positive), ce sont les points bleus pris pour des rouges ;
- les vrais négatifs (TN pour True Negative), les points bleus reconnus comme bleu ;
- les faux négatifs (FN pour False Negative), les points rouges pris pour des bleus.

En général, on représente les cardinaux de ces quatre catégories sous la forme d'une matrice, que l'on appelle **matrice de confusion** :

$$\begin{pmatrix} \text{TP} & \text{FN} \\ \text{FP} & \text{TN} \end{pmatrix}$$

Par exemple, voilà ce que j'ai obtenu comme matrice de confusion pour un jeu de données de la forme nuage :

$$\begin{array}{c} k = 3 \\ \begin{pmatrix} 31 & 23 \\ 14 & 32 \end{pmatrix} \end{array} \quad \begin{array}{c} k = 5 \\ \begin{pmatrix} 36 & 18 \\ 10 & 36 \end{pmatrix} \end{array} \quad \begin{array}{c} k = 9 \\ \begin{pmatrix} 39 & 15 \\ 9 & 37 \end{pmatrix} \end{array}$$

Il apparait que pour $k = 9$ l'algorithme fait le plus de vrais positifs, donc le choix de $k = 9$ semble être le meilleurs parmi ceux essayés. Mais suivant les cas, la seule information du nombre de vrais positifs n'est pas suffisante pour choisir la valeur de k .

Prenons l'exemple d'un diagnostic médical, si le nombre de faux positifs est aussi très élevé cela peut avoir des conséquences assez graves : annoncer à un patient qu'il est condamné à cause d'une maladie incurable alors qu'il n'en rien, peut le placer dans une détresse émotionnelle intense, sans compter qu'il peut se voir administrer un traitement lourd dont il n'a pas besoin.

La lecture de la matrice de confusion peut être facilité par l'usage de différentes métriques, qui répondent à des questions différentes, parmi lesquelles :

- l'**erreur** qui représente le ratio de toutes les mauvaises prédictions sur l'ensemble des prédictions :

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}.$$

Plus il est bas, mieux c'est.

- la **précision** qui indique le ratio entre les prévisions positives correctes et le nombre total de prévisions positives :

$$\frac{TP}{TP + FP}.$$

Ce paramètre répond donc à la question suivante : sur tous les enregistrements positifs prédits, combien sont réellement positifs ?

- le **rappel** qui permet de mesurer le nombre de prévisions positives correctes sur le nombre total de données positives :

$$\frac{TP}{TP + FN}.$$

Il permet de répondre à la question suivante : sur tous les enregistrements positifs, combien ont été correctement prédits ?

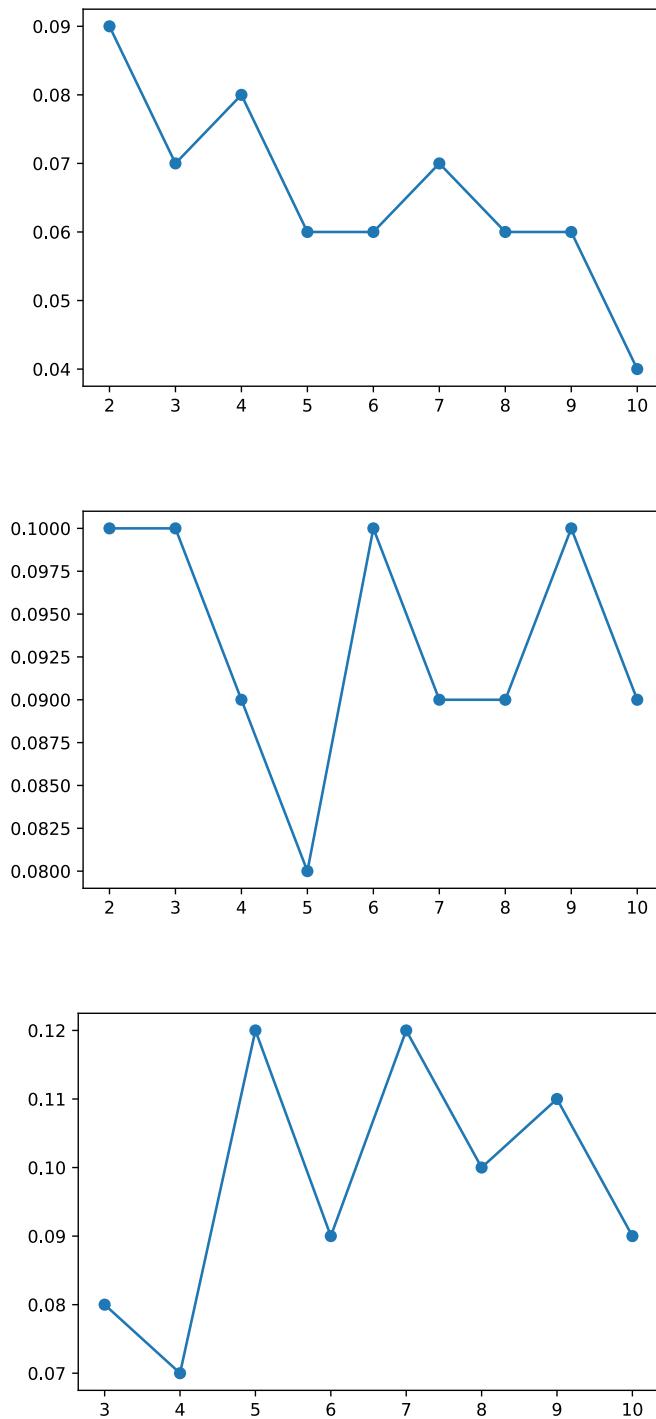


FIGURE 6.4 – Les erreurs en fonction de k pour les différents jeux de données : nuage, croix et couronne.

Dans la pratique, on ne testera que des valeurs impaires de k pour éviter les cas d'égalités.

Une fois la meilleure valeur de k déterminée, on peut utiliser l'algorithme pour prédire la couleur de l'ensemble de tous les points. On obtient alors un découpage par zone.

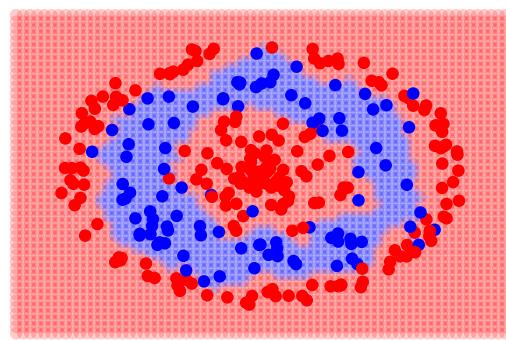
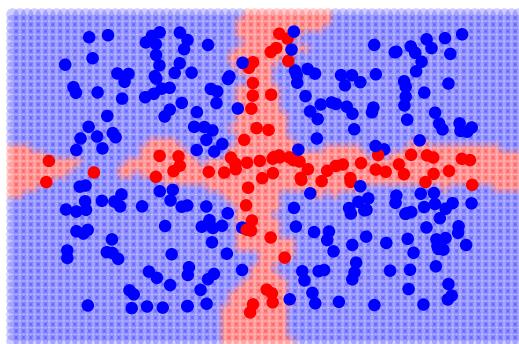
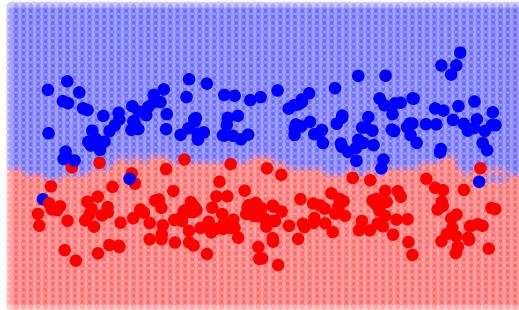


FIGURE 6.5 – Le zonage pour le nuage, la croix et la couronne.

Pour améliorer les résultats, on peut aussi pondérer l'importance de chacun des k plus proches voisins par l'inverse du carré de leur distance au point.

Bien entendu, on peut réaliser le même type d'étude avec des points de plus de deux couleurs. C'est-à-dire plus de deux catégories d'étiquettes.

6.2.2 Un exemple plus sérieux

Je dispose de la base données MNIST au format csv (vous pouvez la télécharger ici : <https://www.kaggle.com/datasets/oddrationale/mnist-in-csv>).

Cette base de données est très connue, elle rassemble 60 000 chiffres manuscrits sous forme d'images de 28×28 pixels, réparties en deux ensembles `mnist_train.csv` et `mnist_test.csv` qui contiennent respectivement 60 000 images d'entraînements et 10 000 images de tests.

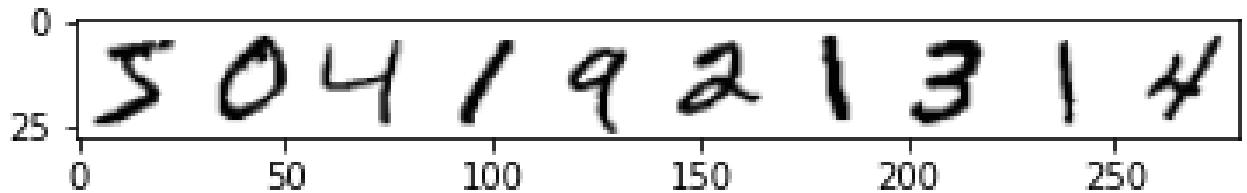


FIGURE 6.6 – Les dix premiers chiffres de la base MNIST.

Chaque donnée est représentée par un vecteur de \mathbb{R}^{784} , que l'on peut interpréter comme une image 28×28 pixels et chaque ligne du tableau `mnist_train.csv` est constituée de 785 colonnes, la première contient le `label` de l'image et les 784 autres codent l'image.

Afin d'utiliser l'algorithme des k -NN pour prédire le label d'une image, j'ai adapté mon code précédent pour m'adapter au format des données de MNIST :

```
def d2(pt1, pt2):
    return np.sqrt(np.sum((pt1[1:]-pt2[1:])**2))

def k_nn(k, point, training_set, dist):
    kNN = sorted(training_set, key = lambda pt : dist(point, pt))[:k]
    labels = {}
    for nd in kNN:
        labels[nd[0]] = labels.get(nd[0], 0) + 1
    m = max([labels[key] for key in labels.keys()])
    pred_labels = [key for key in labels.keys() if labels[key] == m]
    return pred_labels
```

Ce nouveau code tient compte des résultats incertains puisqu'il retourne la liste des prédictions de l'algorithme.

La figure (6.7) présentent des exemples de prédictions réalisées en utilisant l'algorithme avec $k = 5$ sur les dix premières images de l'ensemble de test et en utilisant seulement les cent premières images de l'ensemble d'entraînement. Il apparait que dans le cas du 4 l'algorithme hésite entre un 9 et un 7...

Pour déterminer quelle est la meilleure valeur de k , nous allons utiliser une mesure de précision, en calculant le ratio des prédictions correctes sur l'ensemble des prédictions.

```
def calculate_accuracy(testing_set, training_set, dist, k=5):
    predictions = []
    for test in testing_set:
        pred_label = k_nn(k, test, training_set, dist)
        predictions.append(pred_label[0])
```

Label : [7]

Label : [9, 7]

Label : [6]

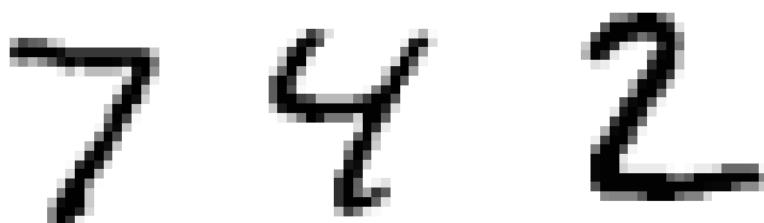
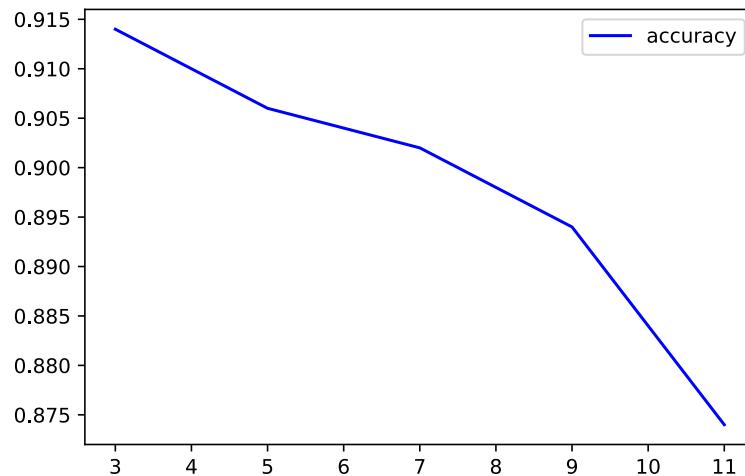


FIGURE 6.7 – Exemples de prédictions : une correcte, une indécise et une fausse.

```
predictions = np.array(predictions)
accuracy = (predictions == np.array(testing_set[:,0])).sum() / testing_set.shape[0]
return accuracy
```

Pour les 5 000 premières images de l'ensemble d'entraînement et les 500 premières images de l'ensemble de test j'obtiens la courbe suivante de la précision en fonction de k .

FIGURE 6.8 – Précision en fonction de k .

Visiblement pour $k = 3$ les résultats des prédictions sont plus précises.

Observons maintenant comment se comportent les prédictions pour différentes valeurs en fonction de la taille de l'ensemble d'entraînement.

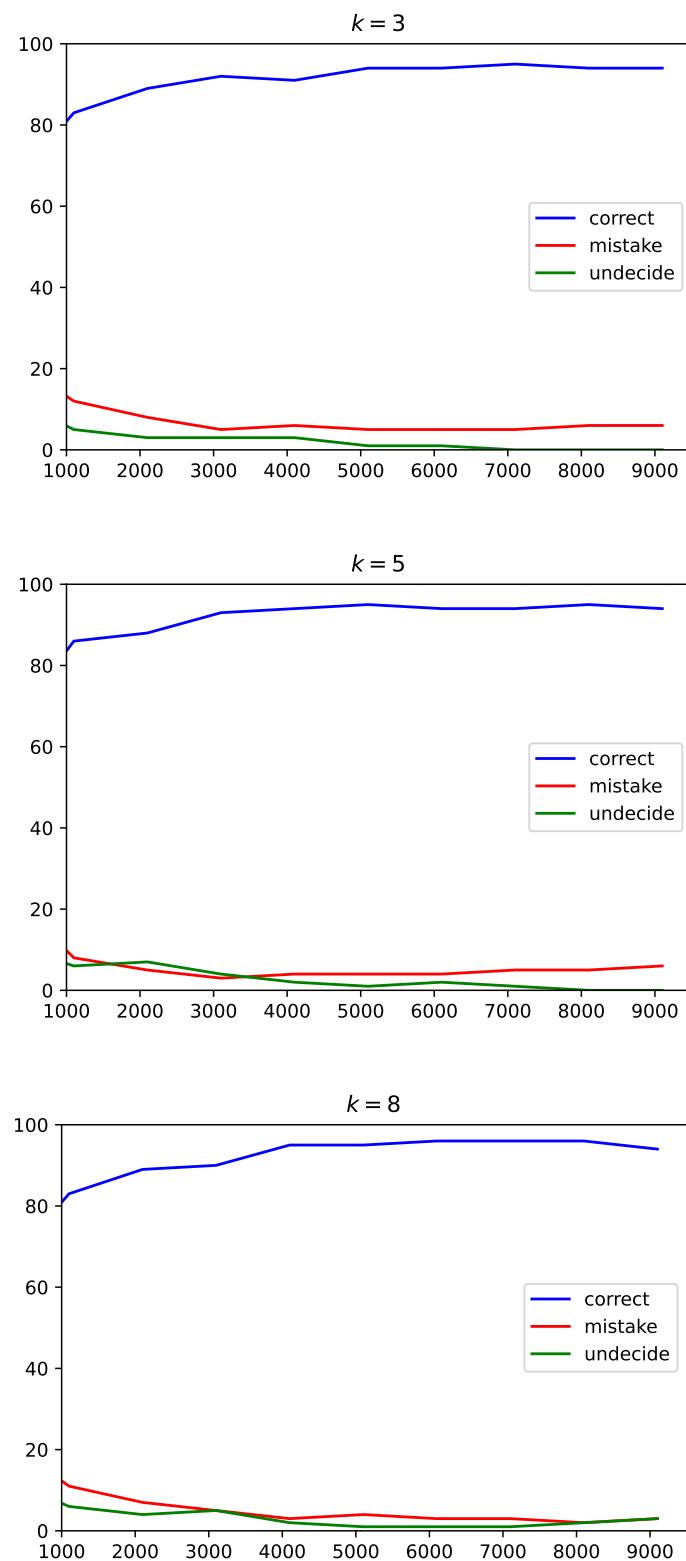


FIGURE 6.9 – Prédiction correctes, éronées et indécises en fonction de la taille de l'ensemble d'entraînement pour $k = 3$, $k = 5$ et $k = 8$.

Dans tous les cas le taux de réponses correctes converge vers 94 – 95%, ce qui change c'est le taux d'indécision, avec $k = 3$ il est bien meilleur.

6.3 Apprentissage non-supervisé : Le clustering

Dans cette partie nous allons nous intéresser aux problèmes d'apprentissage **non supervisé**. Ces méthodes répondent à des questions plus complexes que les méthodes supervisées puisqu'elles doivent « comprendre » les données sans avoir été entraînées au paravant sur des données étiquetées.

6.3.1 Clustering ou partitionnement des données

Le **clustering**, ou **partitionnement de données**, désigne un ensemble de techniques d'apprentissage automatique non supervisé qui consiste à diviser un ensemble de données en groupes (ou clusters) en fonction de leurs similitudes. L'objectif du clustering est de regrouper des observations similaires ensemble, de manière à ce que les observations dans le même cluster soient plus similaires les unes aux autres que les observations dans des clusters différents.

Le clustering peut être utilisé pour une variété d'applications, notamment la segmentation de marché, la reconnaissance de formes et la classification. Il existe différentes techniques de clustering, telles que *k*-means, Hierarchical Clustering, DBSCAN, etc.

La figure (6.10) illustre ce que l'on attend de la machine. Nous observons clairement que les données présentent quatre « regroupements » ou **clusters**, et nous attendons que la machine les reconnaissent.

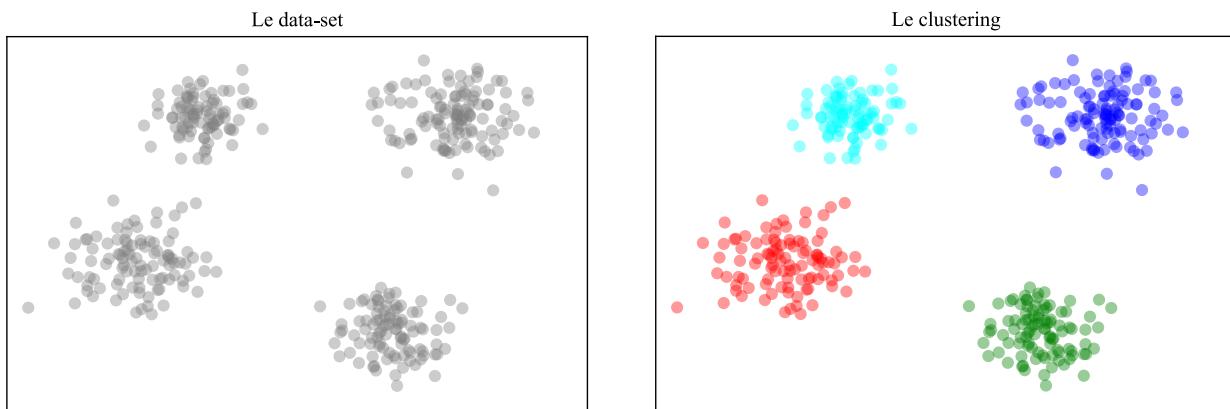


FIGURE 6.10 – Clustering d'un ensemble de données.

Derrière le nom de *k*-moyennes (*k*-means) se cache en fait toute une famille de déclinaison d'une idée développée par Lloyd et Steinhaus en 1957ⁱ, que l'on appelle les « nuées dynamiques ». Pour partitionner un ensemble, on commence par choisir le nombre *k* de classes que l'on souhaite obtenir, puis une métrique et enfin une fonction ϕ à minimiser pour former les classes autour de leurs centroïdes.ⁱⁱ

L'algorithme des nuées dynamiques se déroule alors de manière itérative :

1. On sélectionne aléatoirement *k* points dans l'ensemble des données.
2. On forme *k* classes C_1, \dots, C_k qui minimisent la fonction ϕ .
3. On calcule les centroïdes de chaque classe C_i .
4. On forme *k* nouvelles classes en fonction de ces centroïdes, qui minimisent ϕ

Les deux dernières étapes étant répétées jusqu'à ce que l'ensemble des centroïdes n'évolue plus.

Dans ce cours, nous allons étudier l'algorithme présenté par Forgy en 1965, dit des « centres mobiles », qui tente de répondre au problème suivant : on se donne n points de \mathbb{R}^d et un entier *k*, et l'on cherche *k* centroïdes tels

i. Publiée en 1982

ii. C'est un autre mot pour centre de masse, ou de gravité.

que la somme des carrés des distances entre chaque point et le centroïde le plus proche soit minimale.

Précisément soient $\mathcal{S} = \{x_1, \dots, x_n\}$ un ensemble de points de \mathbb{R}^d , il s'agit de trouver un ensemble $\mathcal{C} = \{c_1, \dots, c_k\}$ de k centroïdes telle que la somme :

$$\phi(\mathcal{C}) = \sum_{x \in S} \min_{c \in \mathcal{C}} d(x, c)^2,$$

soit minimale. Puis de construire partition $\{\mathcal{C}_1, \dots, \mathcal{C}_k\}$ de \mathcal{S} en k classes en assignant chaque point à la classe dont le centre lui est le plus proche.

Trouver la solution exacte de ce problème est un problème *NP*-difficile mais l'algorithme de Forgy permet de s'approcher de façon assez simple d'un minimum local, mais pas nécessairement global du problème.

6.3.2 L'algorithme des *k*-means de Forgy

La simplicité de cet algorithme explique qu'encore aujourd'hui il est très populaire alors qu'il n'offre aucune garantie de converger vers une bonne solution du problème...

Voici précisement comment se déroule l'algorithme :

1. On initialise l'ensemble $\mathcal{C} = \{c_1, \dots, c_k\}$, avec k points choisis au hasard dans l'ensemble de données \mathcal{S} .
2. Pour chaque $i \in \{1, \dots, k\}$, former le cluster C_i en regroupant les points qui sont plus proches de c_i que de toutes les autres c_j pour $j \neq i$.
3. Pour tout $i \in \{1, \dots, k\}$, déterminer le centroïde c_i de l'ensemble C_i en calculant $c_i = \frac{1}{\#C_i} \sum_{x \in C_i} x$.
4. Recommencer les étapes **2** et **3** jusqu'à ce que l'ensemble \mathcal{C} ne change plus.

L'idée de l'algorithme est que la répétition des étapes **2** et **3** garantit la décroissance de ϕ , de sorte que l'algorithme apporte des améliorations locales à un partitionnement arbitraire jusqu'à ce qu'il ne soit plus possible de le faire.

Ceux qui se demandent pourquoi l'étape **(3)** assure la décroissance de ϕ , il faut se rappeler du résultat suivant dont la preuve est laissée en exercice :

Théorème 6.1.

Si \mathcal{S} est un ensemble de points de l'espace euclidien \mathbb{R}^d , de centroïde μ , alors pour tout $y \in \mathbb{R}^d$:

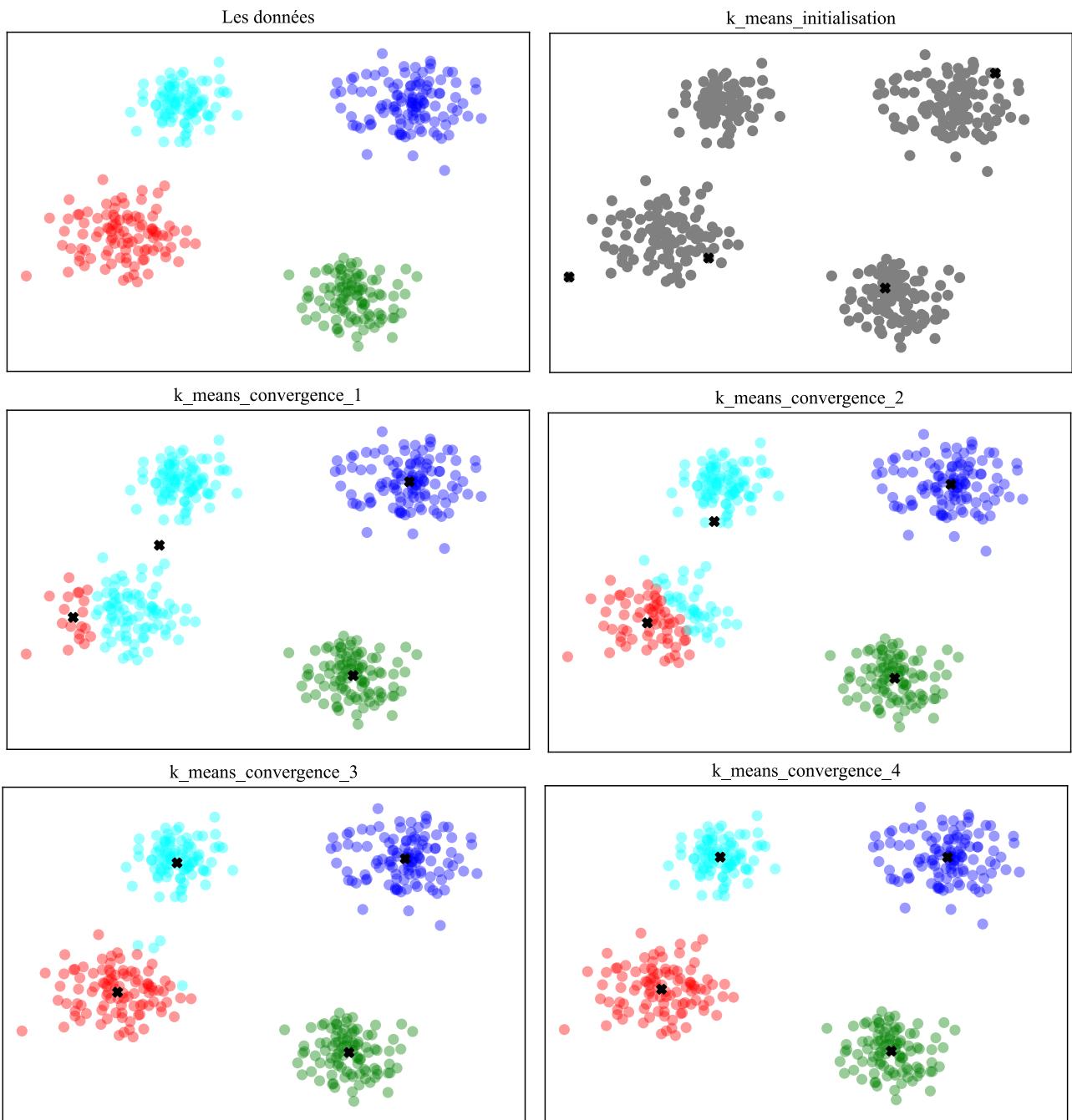
$$\sum_{x \in \mathcal{S}} \|x - y\|^2 - \sum_{x \in \mathcal{S}} \|x - \mu\|^2 = \#\mathcal{S} \cdot \|\mu - y\|^2.$$

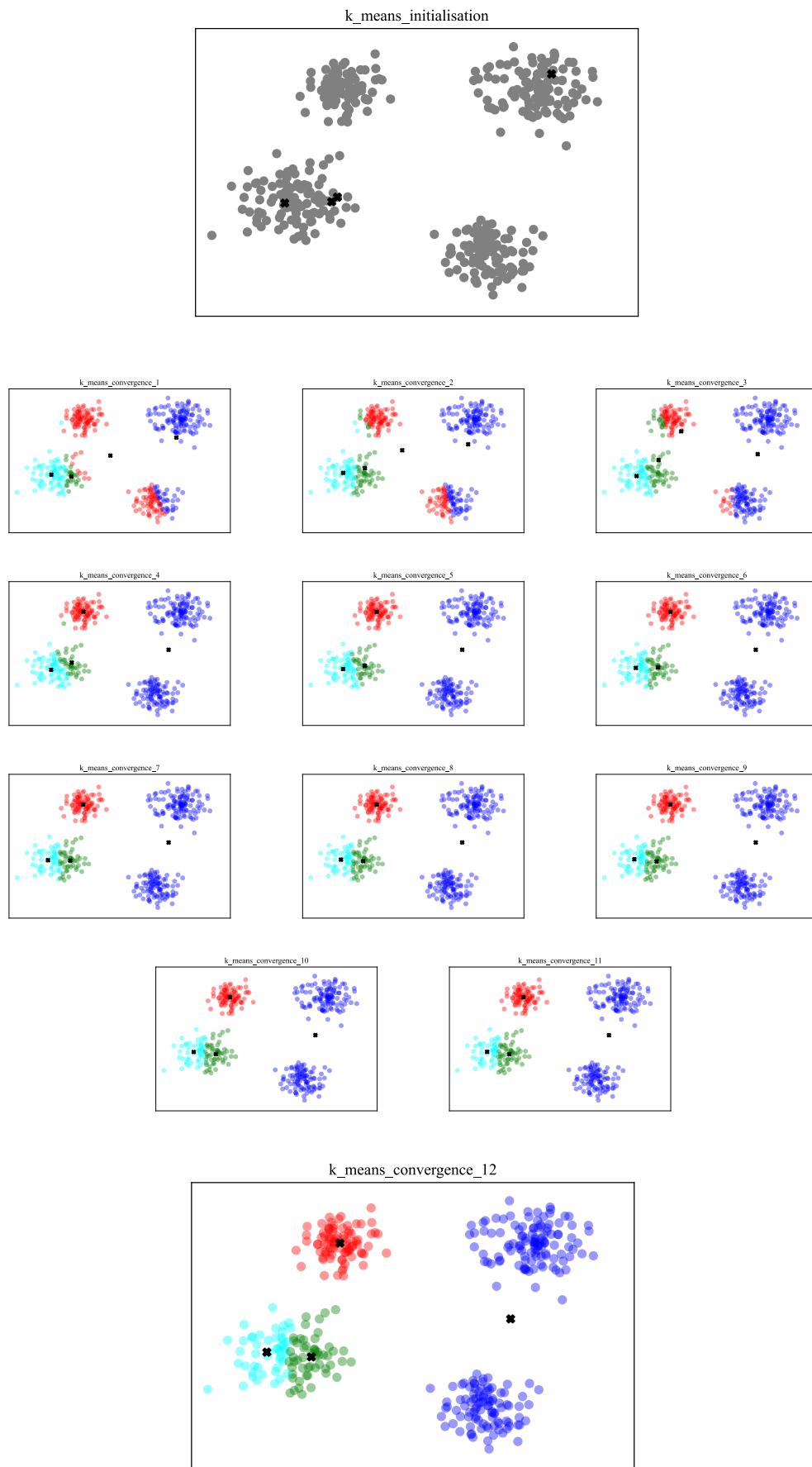
Une implémentation en Python de cet algorithme se trouve à la fin du cours.

La figure (6.11) montre la convergence de l'algorithme des *k*-means sur un jeu de données aléatoires de 400 points. Dans cet exemple l'algorithme converge vers la solution optimale, ce qui n'est évidemment pas toujours le cas.

6.3.3 La forte dépendance aux conditions initiales

Pour un jeu de données le partitionnement obtenu par l'algorithme de Forgy peut considérablement changé en fonctions des points qui ont servi à son initialisation. Les figures (6.12) et (6.13) montrent que des choix différents des points initiaux mènent à des partitionnements très différents de l'ensemble de données de l'exemple précédent.

FIGURE 6.11 – Convergence de k -means.



© J.Stiker - PT - Couffignal
FIGURE 6.12 – Dépendance à l'initialisation.

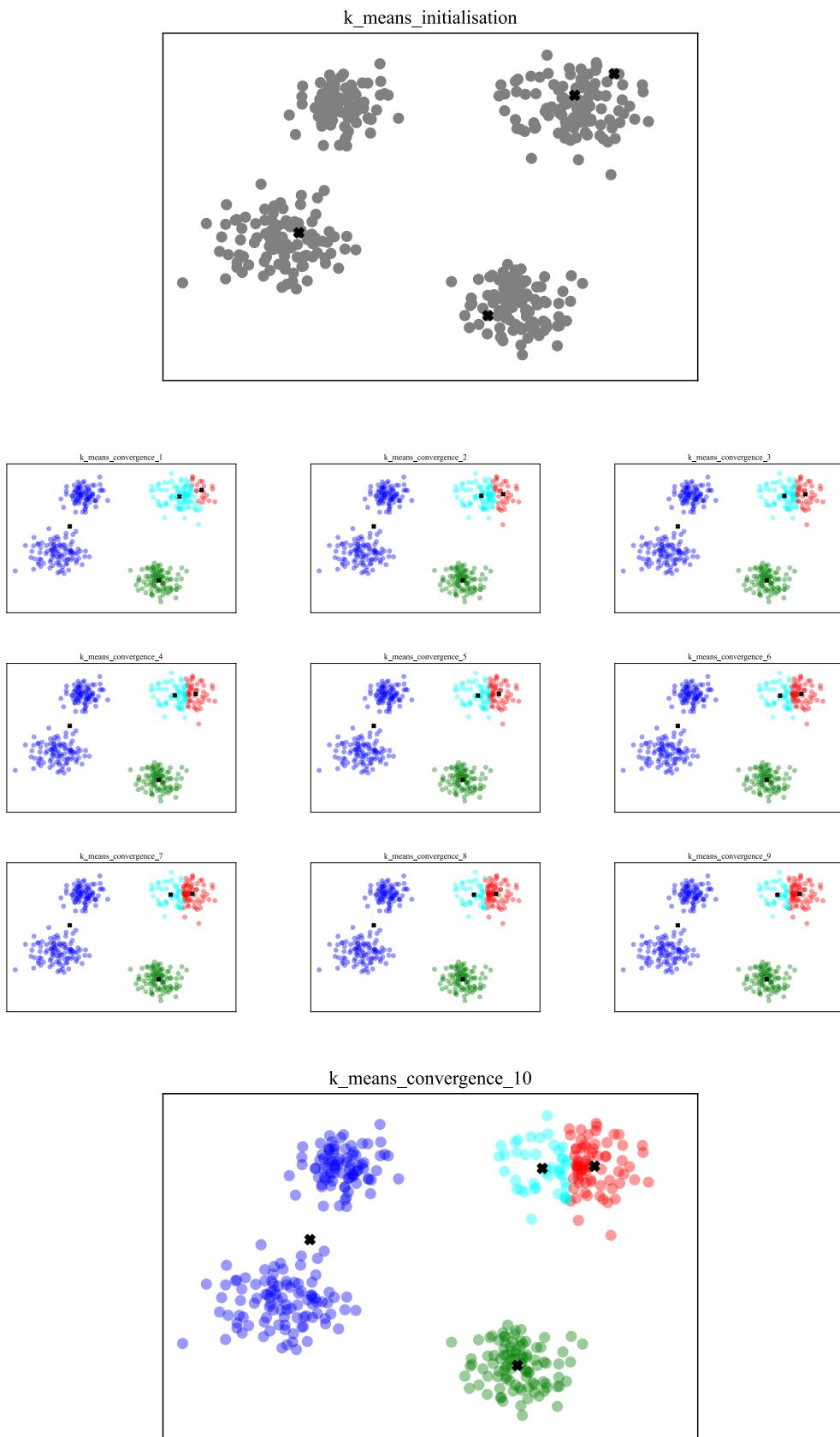


FIGURE 6.13 – Dépendance à l'initialisation.

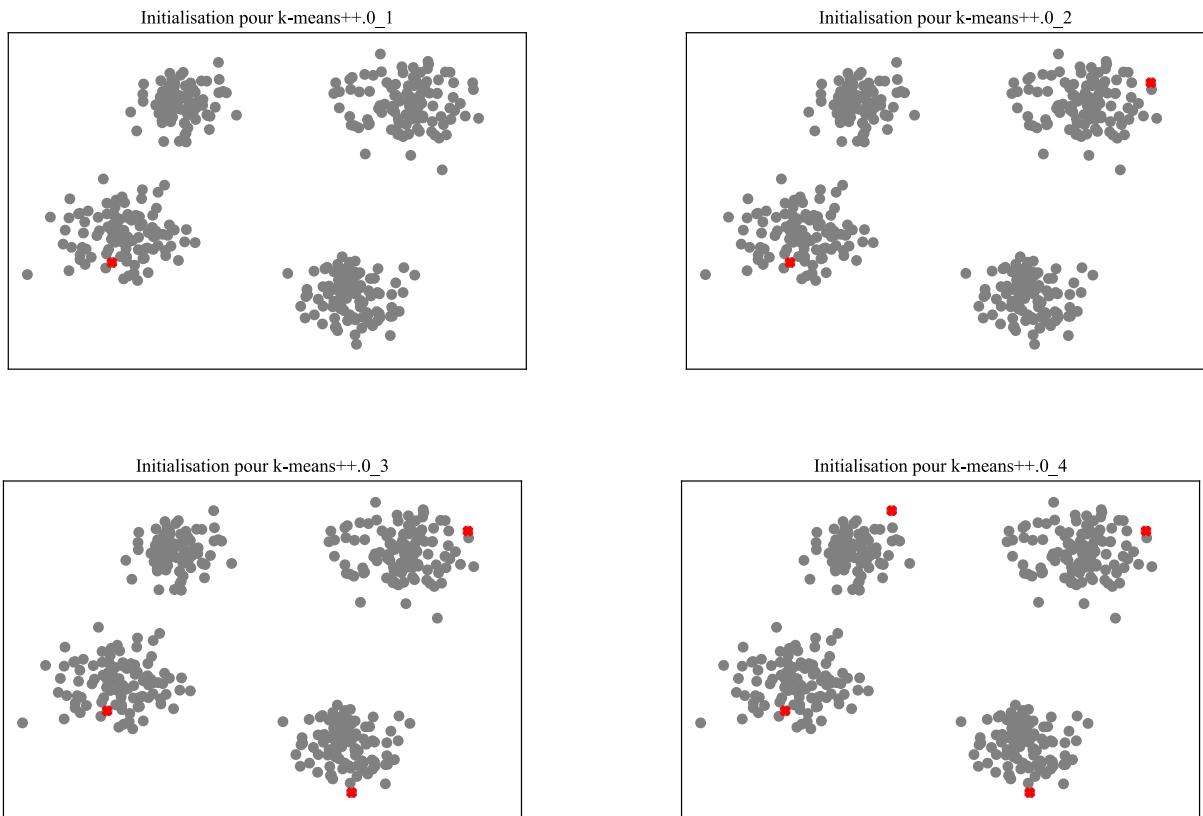


FIGURE 6.14 – Construction de l'ensemble des centres initiaux pas à pas.

On peut aussi observer que la vitesse de convergence de l'algorithme dépend des points initiaux.

La dépendance se comprend aisément, si lors de l'initialisation plusieurs points sont choisis dans un même ensemble assez dense ils auront du mal à en sortir.

En 2007, David Arthur and Sergei Vassilvitskii ont proposé un algorithme qui permet de palier à cette dépendance. Il consiste à construire un ensemble de centres pour initialiser l'algorithme k -means plutôt que de laisser ce choix au hasard puis de continuer k -means, on appelle cet algorithme k -means++.

Lors de cette initialisation, les centres sont choisis de manière itérative afin qu'ils soient les plus éloignés les uns des autres.

On note $D_{\mathcal{C}}(x)$ la distance du point x au centre $c \in \mathcal{C}$ dont il est le plus proche. L'initialisation des centres se fait alors en trois étapes :

1. Choisir un centre c_1 au hasard parmi les données de S . Initialiser $\mathcal{C} = \{c_1\}$.
2. Choisir un nouveau centre c_i dans S parmi les points de S tels que $\frac{D_{\mathcal{C}}(x)^2}{\sum_{y \in S} D_{\mathcal{C}}(y)^2}$ soit maximal, et l'ajouter à \mathcal{C} .
3. Répéter l'étape (2) jusqu'à ce que \mathcal{C} contienne k points.

Une implémentation de k -means++ se trouve à la fin du cours.

La figure (6.14) montre deux exemples de résultats de cette méthode pour l'initialisation des points.

6.3.4 Et k dans tout ça ?

C'est certainement la question que vous vous posez : et k dans tout ça ? On parle de méthode d'apprentissage non-supervisé alors que nous imposons le nombre de clusters que la machine doit découvrir. Or, dans la pratique l'exploration d'un grand nombre de données en grande dimension se fait à l'« aveugle ». On ne peut pas représenter les donnéesⁱⁱⁱ pour déterminer à l'œil nu le nombre de clusters.

Il existe plusieurs méthodes pour déterminer le nombre optimal de clusters à utiliser, qui se base sur différentes métriques définies sur les partitions d'un ensemble. Il faut garder à l'esprit que nous souhaitons partitionner des données, plus les ensembles de la partition seront resserrés sur eux-même et plus ils seront éloignés les uns des autres plus la classification sera précise.

La méthode elbow

La variance ou distortion, d'une partition $\{C_1, \dots, C_k\}$ de \mathcal{S} est définie par :

$$V(C_1, \dots, C_k) = \sum_{i=1}^k \sum_{x \in C_i} d(x, c_i)^2.$$

Dans la pratique, on peut lancer k -mean pour différentes valeurs de k et mesurer la variance de la partition obtenue, ensuite on représente cette variance en fonction de k , et généralement on obtient une courbe qui ressemble à celle de la figure (6.15). Cette courbe a, du moins est sensé avoir, une forme de bras, le point le plus haut est l'épaule et à l'autre extrémité on trouve la main. Le nombre optimal de clusters se situe alors au « coude », elbow en anglais. Ce point est celui qui « articule » la baisse de la variance, en effet à sa gauche la baisse entre chaque point est beaucoup plus importante qu'entre deux points successifs à droite de 4, ce qui signifie que la création de nouveaux clusters avant 4 permet de fortement séparer les données alors qu'après le coude ajouter un cluster de sépare plus réellement les données.

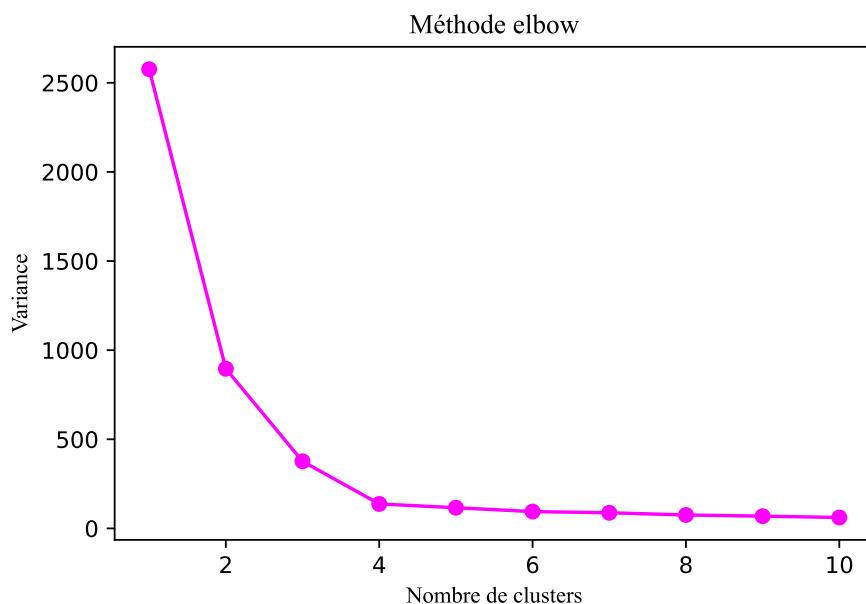


FIGURE 6.15 – Méthode elbow de recherche du nombre optimal de clusters.

iii. Il existe des techniques pour projeter les données dans des espaces de dimensions inférieures.

Coefficient de silhouette

Si la première méthode est réellement utilisée et ne demande que peu d'effort de calcul, on lui préfère celle du **coefficent de silhouette**.

Étant donné une partition de \mathcal{S} en k classes C_1, \dots, C_k , et $x \in \mathcal{C}_j$, le coefficient de silhouette de x permet de savoir s'il a été affecté au bon cluster, autrement dit de répondre aux questions : est-il proche des autres points de son cluster et est-il éloigné des autres points ?

Afin de répondre à la première question, on calcule $a(x)$ la distance moyenne du point x aux autres points du cluster auquel il appartient. On a alors :

$$a(x) = \frac{1}{\#C_i - 1} \sum_{y \in C_i \setminus \{x\}} d(x, y) = \frac{1}{\#C_i - 1} \sum_{y \in C_i} d(x, y).$$

Et pour répondre à la seconde question, on calcule $b(x)$ la plus petite valeur des distances moyennes du point x aux autres points des différents clusters :

$$b(x) = \min_{j \neq i} \left\{ \frac{1}{\#C_j} \sum_{y \in C_j} d(x, y) \right\}.$$

Si lors du partitionnement C_i était bien le meilleur choix de cluster auquel assigner x , nous devrions avoir $a(x) < b(x)$. On définit alors son coefficient de silhouette $s(x)$ par :

$$s(x) = \frac{b(x) - a(x)}{\max\{a(x), b(x)\}}.$$

Ce coefficient appartient à l'intervalle $[-1, 1]$, et plus il est proche de 1 plus le choix de l'assignation de x à C_i était pertinente.

Enfin, on définit le coefficient de silhouette de la partition en calculant la moyenne des coefficients de silhouette de chacun des points.

La méthode consiste alors à calculer ce coefficient moyen pour différentes partitions obtenues pour des valeurs de k différentes, le choix optil de k étant celui pour lequel le coefficient est le plus proche de 1.

La figure (6.16), montre que pour notre jeu de données les deux méthodes préconisent un partitionnement en $k = 4$ clusters.

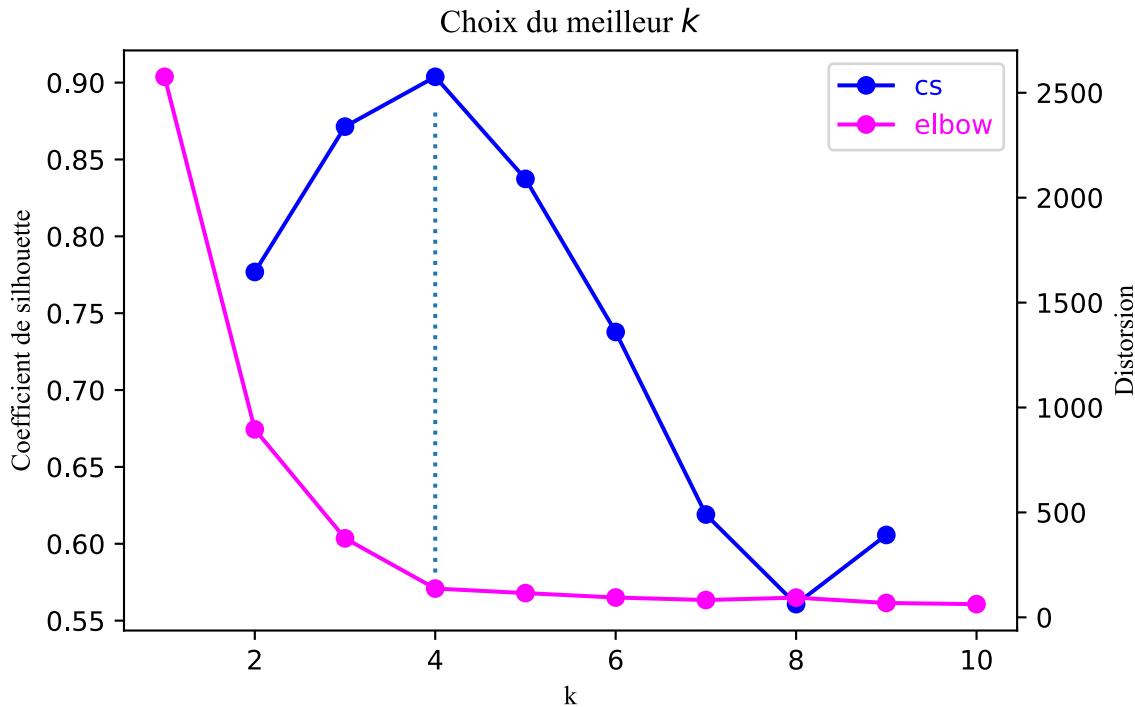


FIGURE 6.16 – La méthode du coefficient de silhouette pour notre jeu de données.

J'ai testé les deux méthodes pour déterminer le nombre optimal de clusters pour partitionner différents ensemble de données. Sur la figure (6.17), il apparaît clairement que la méthode du coefficient de silhouette donne un résultat plus lisible, même si la méthode elbow donne de bons résultats.

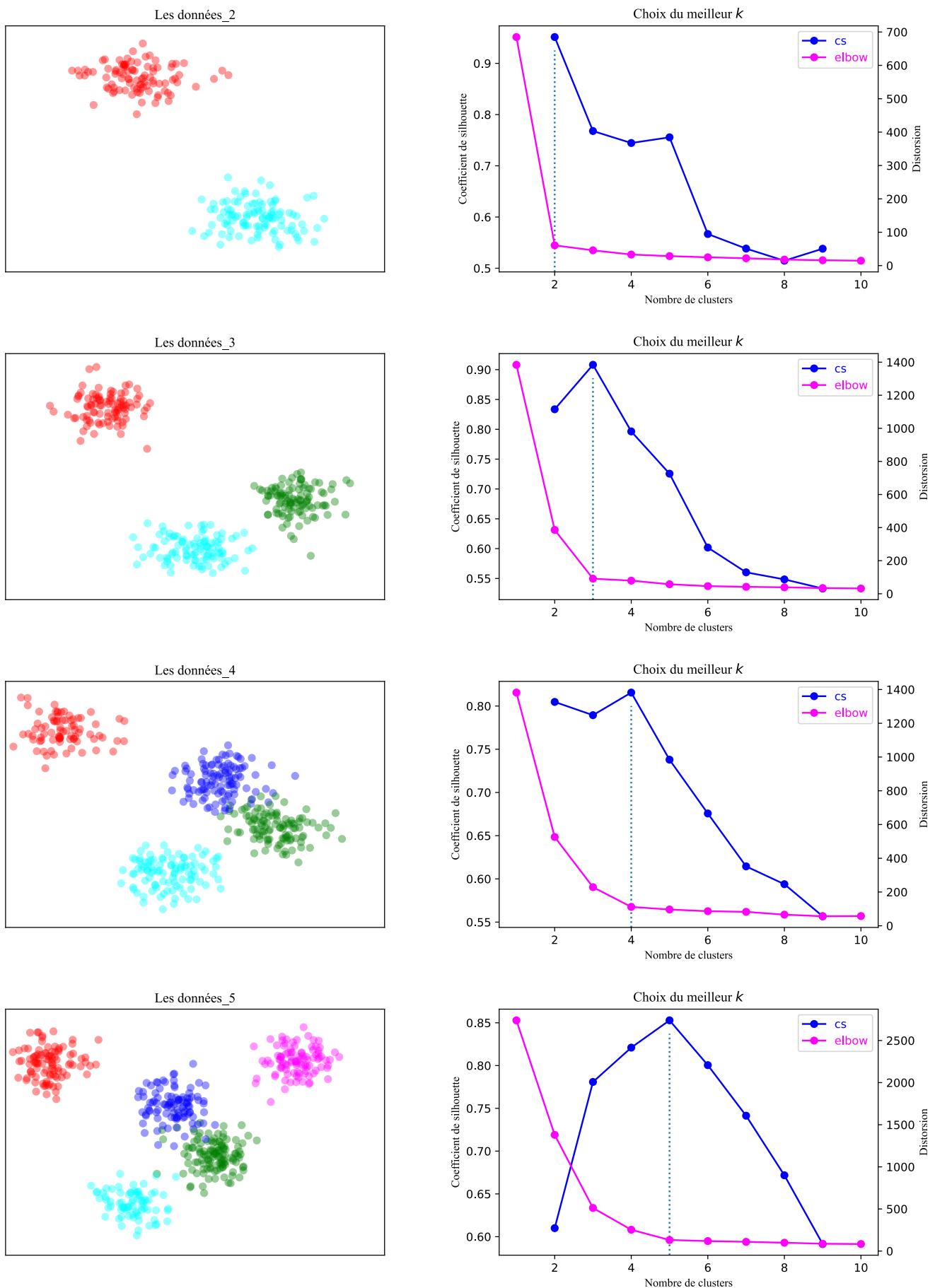


FIGURE 6.17 – Prévision du nombre optimal de clusters par les deux méthodes.

6.3.5 Un jeu de données classique

Je dispose du jeu de données Iris de Fisher, ou d'Anderson suivant les sources. C'est un jeu de données datant de 1936 qui compile quatre caractéristiques : la longueur et la largeur des sépales et des pétales, en centimètres, des trois espèces d'iris (Iris setosa, Iris virginica et Iris versicolor). Il contient 50 échantillons de chacune des trois.

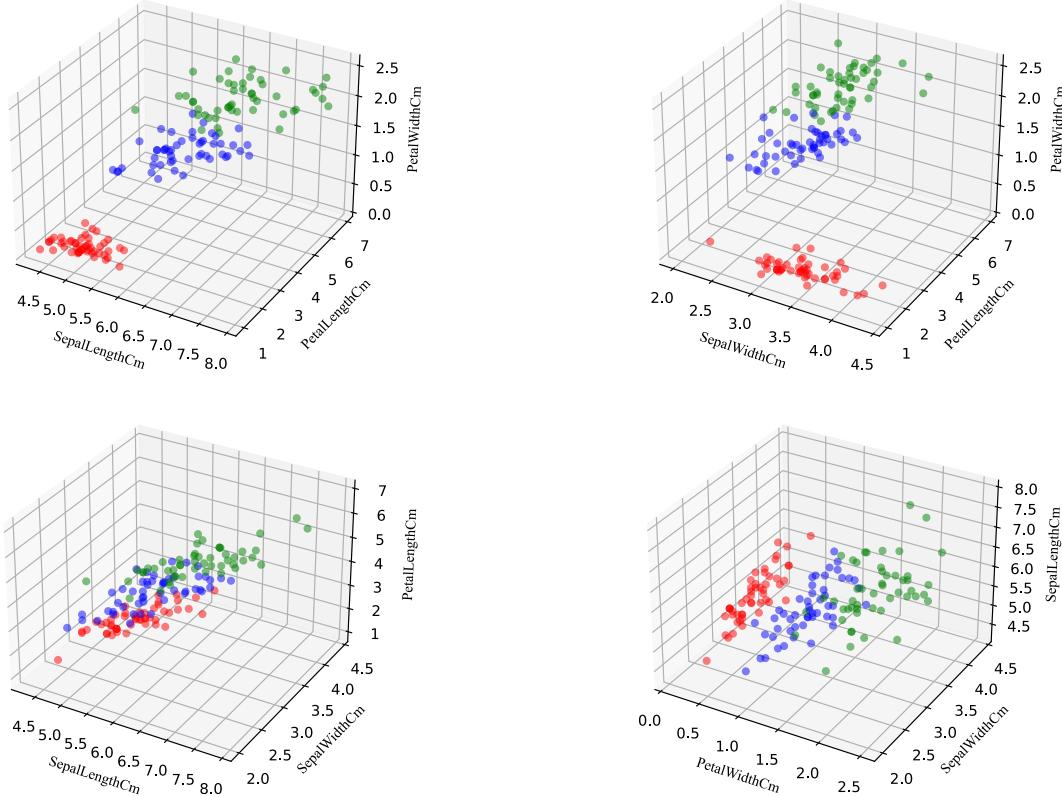


FIGURE 6.18 – Représentations partielles des données du jeu Iris dans \mathbb{R}^3 .

J'ai demandé à k -means et à k -means++ de classifier ces données. Le jeu de données étant assez petit la forte dépendance de k -means aux centroïdes initiaux se fait largement ressentir, pour deux passages différents je trouve :

```
{'Iris-setosa': 28, 'Iris-versicolor': 3}
{'Iris-setosa': 22}
{'Iris-versicolor': 47, 'Iris-virginica': 50}

{'Iris-versicolor': 3, 'Iris-virginica': 36}
{'Iris-versicolor': 47, 'Iris-virginica': 14}
{'Iris-setosa': 50}
```

Alors, qu'avec k -means++ les classes sont très stables, en particulier celle des setosa qui sont toujours reconnues à 100%.

```
{'Iris-versicolor': 48, 'Iris-virginica': 14}
{'Iris-versicolor': 2, 'Iris-virginica': 36}
{'Iris-setosa': 50}
```

Si les résultats sont satisfaisants, il semblerait qu'il y ait une erreur incompréhensible. Une des raisons réside dans le fait que le nombre de descripteurs, 4 ici, des données est élevé devant la taille de l'ensemble des données,

seulement 150. On appelle ça le « fléau de la dimension », dans la pratique on utilise des techniques pour « réduire » la dimension. Une des plus utilisée est l'Analyse en Composante Principale.

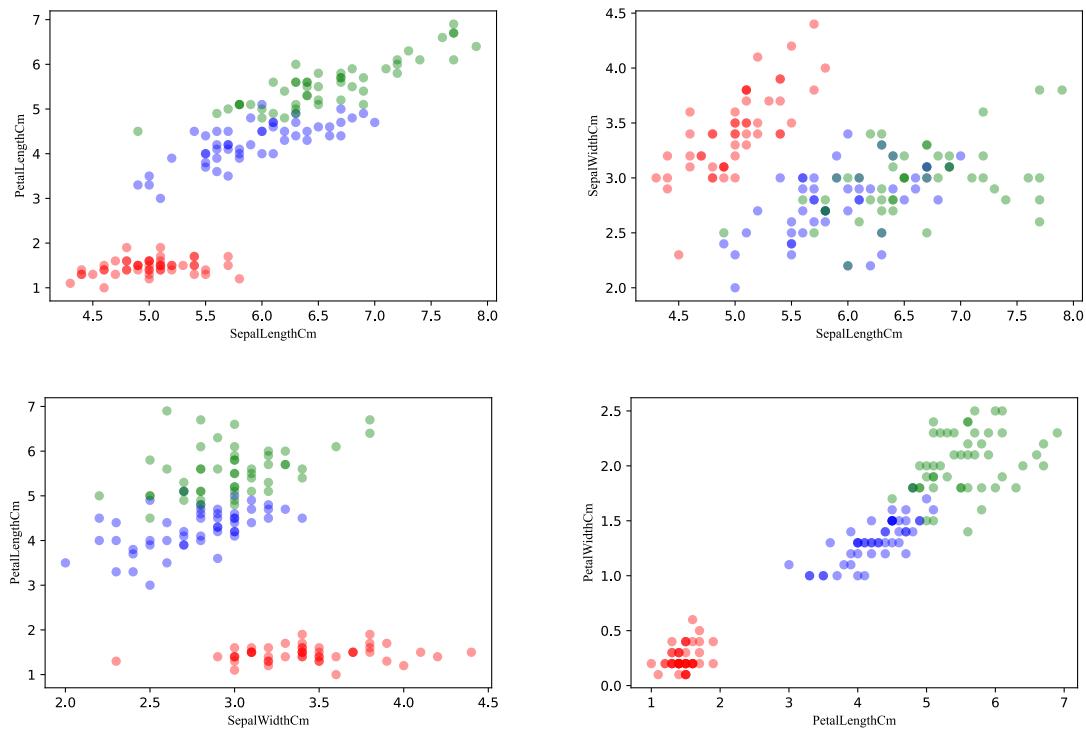


FIGURE 6.19 – Projection du jeu de données dans \mathbb{R}^2 .

En observant la figure (6.19), on voit que la quatrième projection des données dans \mathbb{R}^2 sépare assez bien les données. L'algorithme k -means++ donne d'ailleurs de meilleurs résultats en utilisant seulement la troisième et la quatrième coordonnée des points.

```
{'Iris-setosa': 50}
{'Iris-versicolor': 2, 'Iris-virginica': 44}
{'Iris-versicolor': 48, 'Iris-virginica': 6}
```

En diminuant la dimension, on réduit aussi grandement le nombre de calculs.

6.3.6 Implémentation de *k-means* et *k-means++*

Voici des implémentations des algorithmes en python. Les données sont représentées par des listes [x,y,c], où (x, y) sont les coordonnées du point et c est un chaîne de caractères qui indique la couleur. Je n'ai pas utilisé de tuples pour pouvoir modifier la couleur du point.

La fonction `init` est celle qui permet d'implémenter *k-means++*.

C'est assez simple de faire en sorte que ces algorithmes s'appliquent en dimension supérieures à 2.

```
def center(cluster: list):
    x = [pt[0] for pt in cluster]
    y = [pt[1] for pt in cluster]
    return sum(x)/len(x), sum(y)/len(y)

def d2(pt1, pt2):
    return (pt1[0]-pt2[0])**2+(pt1[1]-pt2[1])**2

def k_means(data, k , dist):
    inf = float('inf')
    pts = data[0]
    #Initialisation des centres avec des points du data\set
    centroids = [(pt[0], pt[1]) for pt in sample(pts, k)]
    end = False
    while not end:
        clusters = [[] for __ in range(k)]
        # Pour chaque on détermine sa distance aux différents centroïdes
        # Et on affecte le point à la classe du centroïde le plus proche
        for pt in pts:
            d_min = inf
            for i in range(k):
                d = dist(pt, centroids[i])
                if d < d_min:
                    d_min = d
                    idx_min = i
            clusters[idx_min].append(pt)
        # Mise à jour des centroïdes après calcul des centres de gravité de
        # chaque classe
        news_centroids = [center(clusters[i]) for i in range(k)]
        # Si les centroïdes n'évoluent plus
        # c'est fini.
        if news_centroids == centroids:
            end = True
        centroids = news_centroids

    return centroids, clusters

def init(data, k, dist):
    centroids = []
    # On initialise l'ensemble des centroïdes
    # avec un point des données
    centroids.append( data[0][randint(0,len(data[0])-1) ] )

    pts = data[0]
    # On utilise le flottant infini
    inf = float('inf')
```

```
for c in range(k - 1):

    # On initialise un dictionnaire pour stocker
    # les distances des points au centre le plus proche
    dists = {}
    for i in range(len(pts)):
        # On calcule les distance du point à chaque centroïde déjà sélectionné
        # et on stocke la distance minimale

        d = inf
        for j in range(len(centroids)):
            dist_tmp = dist(pts[i], centroids[j])
            d = min(d, dist_tmp)
        dists[i] = d

    # On sélectionne parmi les données le point de distance maximale
    idx = max(dists, key = lambda k : dists[k])
    next_centroid = data[0][idx]
    centroids.append(next_centroid)
    dists = {}

return centroids
```

