

Université Paris 8 Vincennes - Saint Denis  
Institut d'enseignement à distance  
Master 1 big data

EC : Programmation concurrente

Patrice Blanchard - 18904701

Jeudi 9 mars 2023

# Table des matières

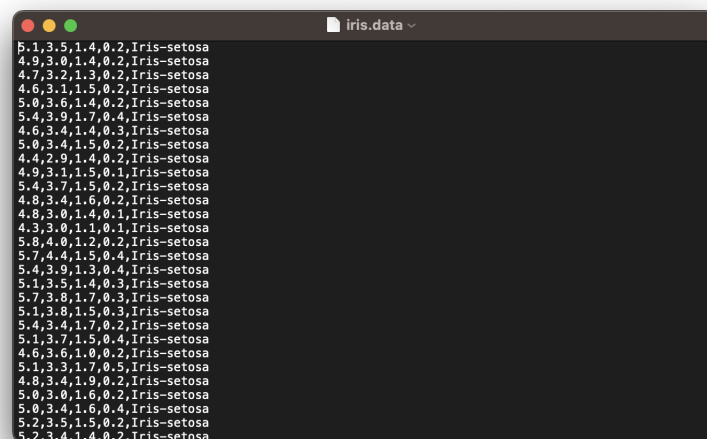
<b>1</b>	<b>Projet n°1 : les mesures de tendance centrale et de dispersion</b>	<b>3</b>
1.1	Les mesures à utiliser : la moyenne arithmétique, la variance, l'écart-type, la covariance, le coefficient de corrélation . . . . .	4
1.2	L'implémentation du projet 1 : . . . . .	6
1.3	L'interprétation des résultats . . . . .	13
1.3.1	La moyenne de chaque colonne . . . . .	13
1.3.2	La variance de chaque colonne . . . . .	13
1.3.3	L'écart-type de chaque colonne . . . . .	13
1.3.4	Les coefficients de corrélation entre toutes les variables 2 à 2 . . . . .	14
<b>2</b>	<b>Projet n°2 : l'algorithme K-means</b>	<b>15</b>
2.1	La mesure à utiliser : la distance euclidienne . . . . .	17
2.2	L'implémentation du projet 2 . . . . .	19
2.3	L'importation de la bibliothèque graphique plotly pour Almond . . . . .	24
2.4	La représentation du nombre d'éléments pour chaque classe . . . . .	25
2.5	La représentation de la longueur et de la largeur des sépales pour chaque classe . . . .	26
2.6	La représentation de la longueur et de la largeur des pétales pour chaque classe . . . .	29
2.7	En conclusion : . . . . .	32

# 1 Projet n°1 : les mesures de tendance centrale et de dispersion

Dans ce projet, vous aurez à implémenter **les mesures de tendance centrale et de dispersion** ainsi que la mesure de distance adéquate. Le but est d'analyser la distribution des données de la base IRIS fournie dans le fichier **iris.data** à la page <https://archive.ics.uci.edu/ml/datasets/iris>.

Les données utilisées pour trois espèces d'iris : iris setosa, iris versicolor et iris virginica sont les mesures en centimètres des caractères suivants :

- longueur du sépale (sepal.length);
- largeur du sépale (sepal.width);
- longueur du pétale (petal.length);
- largeur du pétale (petal.width).



```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
5.4,3.7,1.5,0.2,Iris-setosa
4.8,3.4,1.6,0.2,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
5.4,3.4,1.7,0.2,Iris-setosa
5.1,3.7,1.5,0.4,Iris-setosa
4.6,3.6,1.0,0.2,Iris-setosa
5.1,3.3,1.7,0.5,Iris-setosa
4.8,3.4,1.9,0.2,Iris-setosa
5.0,3.0,1.6,0.2,Iris-setosa
5.0,3.4,1.6,0.4,Iris-setosa
5.2,3.5,1.5,0.2,Iris-setosa
5.2,3.4,1.4,0.2,Iris-setosa
```

Figure 1 – Les premières lignes du dataset iris

Ces caractères sont étudiés sur une population de 150 individus (données en lignes). La dernière colonne représente la classe réelle (la catégorie d'iris correspondante). Elle nous servira juste pour visualiser et évaluer nos résultats, elle n'est pas à inclure dans l'analyse.

Sur cette base :

- extraire le fichier correspondant, idéalement stocker les données dans une matrice de taille (150 lignes et 5 colonnes, la dernière colonne représentant la classe réelle);
- calculer la moyenne, la variance, l'écart type pour chaque variable étudiée (caractère), commenter vos résultats;
- Calculer le coefficient de corrélation entre toutes les variables 2 à 2, projeter et commenter.

## 1.1 Les mesures à utiliser : la moyenne arithmétique, la variance, l'écart-type, la covariance, le coefficient de corrélation

Soit le vecteur (ensemble de données)  $x$  de coordonnées  $(x_1, \dots, x_n)$  et le vecteur  $y$  de coordonnées  $(y_1, \dots, y_n)$ .

**La moyenne** est définie par la formule suivante :

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

La moyenne par défaut est dite arithmétique. Elle notée  $\bar{x}$ . Cet outil de calcul consiste à résumer une liste de valeurs numériques en un seul nombre réel. À cette fin, l'addition de l'ensemble des éléments observés est réalisée pour que ce résultat soit ensuite divisé par le nombre total d'éléments.

**La variance** est définie par la formule suivante :

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

En statistique et en théorie des probabilités, la variance mesure le carré des écarts par rapport à la moyenne. La variance n'est pas une mesure de dispersion absolue. Elle mesure la quantité moyenne d'information contenue dans une distribution autrement dit une mesure globale de la variation d'une distribution.

**L'écart-type** est défini par la formule suivante :

$$\sigma(x) = \sqrt{\sigma^2(x)}$$

L'écart type ou écart-type est noté généralement avec la lettre grec  $\sigma$ . L'écart-type est le paramètre de dispersion le plus utilisé en statistique. Cet outil de calcul est défini comme la racine carrée de la variance ou comme la moyenne quadratique des écarts par rapport à la moyenne. La moyenne quadratique est la racine carrée de la moyenne arithmétique des carrés de ces écarts par rapport à la moyenne. Ce résultat permet de synthétiser les résultats numériques d'une expérience répétée. Néanmoins, il est important de se souvenir des propriétés suivantes lorsque l'on utilise l'écart-type [1] :

- ce calcul est sensible aux valeurs aberrantes. Une seule valeur aberrante peut accroître l'écart-type et par le même fait, déformer la valeur de la dispersion;
- pour deux ensembles de données ayant la même moyenne, celui dont l'écart type est le plus grand est celui dans lequel les données sont les plus dispersées par rapport à l'autre;
- l'écart-type est égal à zéro dans le cas où toutes les valeurs d'un ensemble de données sont les mêmes. Ces données seront aussi égale à la moyenne.

Les écarts-types sont rencontrés lors de l'usage des probabilités et la statistique, particulièrement dans le domaine des sondages, en physique, en biologie ou dans la finance.

**La covariance** est définie par la formule suivante :

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

La covariance est une mesure de l'association ou du lien linéaire qui existe entre deux variables. La covariance est une extension de la notion de variance. La covariance est la moyenne de la somme du produit des écarts des valeurs de deux variables par rapport à leur moyenne arithmétique. La création d'une matrice de covariance permet de quantifier la variation de chaque variable par rapport à chacune des autres.

**Le coefficient de corrélation** est défini par la formule suivante :

$$r(x, y) = \frac{COV(x, y)}{\sigma(x)\sigma(y)}$$

La corrélation entre deux variables ou statistiques numériques permet d'étudier l'intensité de la liaison existant entre ces variables. Ce coefficient de corrélation est égale au rapport de leur covariance et du produit non nul de leurs écarts types. Ce coefficient est compris entre -1 et 1.

## 1.2 L'implémentation du projet 1 :

L'implémentation de la consigne demandée, à savoir :

- l'extraction du fichier iris.data dans une matrice de taille 150 lignes et 5 colonnes;
- Le calcul de la moyenne, l'écart-type, la variance de chaque variable;
- Le calcul du coefficient de corrélation.

Pour implémenter le projet 1, j'ai utilisé une version de jupyter notebook spécialement dédiée au langage scala. Il porte le nom de almond, voici l'adresse de la page officielle : <https://almond.sh/>

Le code utilisé dans le cadre du projet 1 :

```
// Importation de bibliothèques :
import scala.io.Source
// Lecture de fichiers
import scala.math.{sqrt,pow}
// Effectuer des calculs mathématiques : racine carré et puissance

class IrisData {
  private val columnNames = Array(
    "Sepal.Length",
    "Sepal.Width",
    "Petal.Length",
    "Petal.Width")
  // Un tableau de chaînes de caractères représentant
  // le nom des colonnes du dataset iris.data
  private var data: Array[Array[Double]] = Array.empty
  // Une matrice stockant les données du fichier iris.data
  private var means: Array[Double] = Array.empty
  // Un tableau de double stockant les moyennes
  private var variances: Array[Double] = Array.empty
  // Un tableau de double stockant les variances
  private var stdDevs: Array[Double] = Array.empty
  // un tableau de double stockant les écarts-types
  private var covariances: Array[Double] = Array.empty
  // Un tableau de double stockant les covariances
  private var nbColumns = 4
  // Le nombre de colonnes qui seront enregistrées dans la matrice

  def getData(): Array[Array[Double]] = {
    // Cette méthode affiche le tableau data
    data
  }

  def readData(filename: String): Unit = {
    // Cette méthode lit les données à partir du nom de fichier
    // intégré dans la variable filename et stocke les valeurs
  }
}
```

```

// de type double dans la matrice data
val lines = scala.io.Source.fromFile(filename).getLines.toArray
// Créer lines, un tableau de string intégrant les lignes du fichier
data = Array.ofDim[Double](lines.length, nbColumns)
// Création de data, un tableau de double de dimension (lines.length, nbColumns)
for (i <- 0 until lines.length) {
  // Pour tout i de lines.length
  val fields = lines(i).split(",")
  // Tokeniser les i de lines pour les intégrer
  // dans un tableau fields
  for (j <- 0 until 4) {
    // Pour tout j de 0 à 3
    data(i)(j) = fields(j).toDouble
    // Convertir en double et enregistrer les j fields dans les i,j data
  }
}

}

def printData(): Unit = {
  // Afficher les données stockées dans la matrice data
  for (i <- 0 until data.length) {
    // Pour tout i de data.length
    for (j <- 0 until data(i).length) {
      // Pour tout j de data(i).length
      print(data(i)(j) + "_")
      // Afficher les i, j de data
    }
    println()
    // Afficher un retour à la ligne
  }
}

def calculateMeans(): Unit = {
  // Calculer la moyenne de chaque colonne et stocker
  // les valeurs dans un tableau dénommé means
  means = Array.ofDim[Double](4)
  // Créer un tableau de double de taille 4
  for (j <- 0 until 4) {
    // Pour tout j de 0 à 3
    var sum = 0.0
    // Initialiser sum à la valeur 0.0
    for (i <- 0 until data.length) {
      // Pour tout i de data.length
      sum += data(i)(j)
      // Faire la somme des i, j de data
    }
    means(j) = sum / data.length
    // Enregistrer les j means : le résultat de sum divisé par data.length
  }
}

```

```

    }
}

def calculateVariances(): Unit = {
    // Calculer la variance de chaque colonne et stocker
    // les valeurs dans un tableau dénommé variances
    variances = Array.ofDim[Double](4)
    // Créer variances un tableau de double de taille 4
    for (j <- 0 until 4) {
        // Pour tout j allant de 0 à 3
        var sum = 0.0
        // Initialiser sum à la valeur 0.0
        for (i <- 0 until data.length) {
            // Pour tout i de data.length
            sum += pow(data(i)(j) - means(j), 2)
            // Calculer sum : la somme des carrés des i, j de data soustraient
            // par j de means
        }
        variances(j) = sum / data.length
        // Enregistrer les j variances : le résultat de sum divisé par data length
    }
}

def calculateStdDevs(): Unit = {
    // Calculer l'écart type de chaque colonne et stocker
    // les valeurs dans un tableau dénommé stdDevs
    stdDevs = Array.ofDim[Double](4)
    // Créer stdDevs un tableau de double de taille 4
    for (j <- 0 until 4) {
        // Pour tout j allant de 0 à 3
        stdDevs(j) = sqrt(variances(j))
        // Enregistrer les j stdDevs : la racine carrée des j variances
    }
}

def printMeans(): Unit = {
    // Afficher la moyenne de chaque colonne :
    for (i <- 0 until means.length) {
        // Pour tout i de means.length
        println(f"Means_of_{columnNames(i)}: {means(i)}%1.2f")
        // Afficher les moyennes
    }
}

def printVariances(): Unit = {
    // Afficher la variance de chaque colonne :

```



```

    for (i <- 0 until variances.length){
        // Pour tout i de variances.length
        println(f"Variances_of_{columnNames(i)}:_{variances(i)}%1.2f")
        // Afficher les variances
    }
}

```

```

def printStdDevs(): Unit = {
    // Afficher l'écart-type de chaque colonne :
    for (i <- 0 until stdDevs.length){
        // Pour tout i de stdDevs.length
        println(f"StdDevs_of_{columnNames(i)}:_{stdDevs(i)}%1.2f")
        // Afficher les écarts-types
    }
}

```

```

def printMinColumn(){
    // Afficher le minimum de chaque colonne :
    for (i <- 0 until 4){
        // Pour tout i de 0 à 3
        println(s"${i}:_{iris.getData().map(_(i)).min}")
        // Afficher les valeurs minimums
    }
}

```

```

def printMaxColumn(){
    // Afficher le maximum de chaque colonne :
    for (i <- 0 until 4){
        // Pour tout i de 0 à 3
        println(s"${i}:_{iris.getData().map(_(i)).max}")
        // Afficher les valeurs maximums
    }
}

```

```

def printCorrelations(): Unit = {
    // Afficher les coefficients de corrélation
    covariances = Array.ofDim[Double](4 * 4)
    // Créer covariances un tableau de double de dimension (4, 4)
    for (i <- 0 until 4) {
        // Pour tout i de 0 à 3
        for (j <- 0 until 4) {
            // Pour tout j de 0 à 3
            if (i != j) {
                // Si i est différent de j
                var sum = 0.0
                // Initialiser sum à la valeur 0.0
                for (k <- 0 until data.length) {

```

```

        // Pour tout k de data.length
        sum += (data(k)(i) - means(i)) * (data(k)(j) - means(j))
        // Enregistrer sum : le résultat des k, i de data -
        // les i de means multiplié par les k, j de data -
        // les j de means
    }
    covariances(i * 4 + j) = sum / data.length
    // Enregistrer les i, j covariances : le résultat de sum
    // divisé par data.length
    covariances(j * 4 + i) = covariances(i * 4 + j)
    // Enregistrer les j, i covariances égale aux i, j covariances
}
}
}

for (i <- 0 until 4) {
    // Pour tout i de 0 à 3
    for (j <- 0 until 4) {
        // Pour tout j de 0 à 3
        if (i != j) {
            // Si i est différent de j
            val correlation = covariances(i * 4 + j) / (stdDevs(i) * stdDevs(j))
            // Enregistrer correlation = les i, j covariances divisées par
            // les i stdDevs multipliés par les j stdDevs
            println("Correlation_between_variable_:")
            println(s"${columnNames(i)}_and_${columnNames(j)}:_$correlation")
            // Afficher les corrélations
        }
    }
}
}
}
}

```

```

// Exemple d'utilisation de la classe :
val iris = new IrisData()
// Créer une instance de la classe IrisData()
iris.readData("iris.data")
// Exécuter la méthode readData

```

```

iris.calculateMeans()
iris.printMeans()
// Affichage des moyennes :

```

```

iris.calculateVariances()
iris.printVariances()
// Affichage des variances :

```

```

iris.calculateStdDevs()

```

```

iris.printStdDevs()
// Affichage des écart-types :

iris.printCorrelations()
// Affichage des coefficients de corrélation :

iris.printData()

// Pour chaque colonne :
// Sepal.length
// Sepal.width
// Petal.length
// Petal.width

//Afficher les données présentes dans data :

iris.printMinColumn()

// Pour chaque colonne :
// 0 : Sepal.length
// 1 : Sepal.width
// 2 : Petal.length
// 3 : Petal.width

// Afficher le minimum de chaque colonne :

iris.printMaxColumn()

// Pour chaque colonne :
// 0 : Sepal.length
// 1 : Sepal.width
// 2 : Petal.length
// 3 : Petal.width

// Afficher le maximum de chaque colonne :

```

## Une capture d'écran des résultats :

```
In [2]: 1 iris.calculateMeans()
2 iris.printMeans()
3 // Affichage des moyennes :

Means of Sepal.Length: 5,84
Means of Sepal.Width: 3,05
Means of Petal.Length: 3,76
Means of Petal.Width: 1,20

In [3]: 1 iris.calculateVariances()
2 iris.printVariances()
3 // Affichage des variances :

Variances of Sepal.Length : 0,68
Variances of Sepal.Width : 0,19
Variances of Petal.Length : 3,09
Variances of Petal.Width : 0,58

In [4]: 1 iris.calculateStdDevs()
2 iris.printStdDevs()
3 // Affichage des écart-types :

StdDevs of Sepal.Length : 0,83
StdDevs of Sepal.Width : 0,43
StdDevs of Petal.Length : 1,76
StdDevs of Petal.Width : 0,76

In [5]: 1 iris.printCorrelations()
2 // Affichage des coefficients de corrélation :

Correlation between variable :
Sepal.Length and Sepal.Width: -0.10936924995064935
Correlation between variable :
Sepal.Length and Petal.Length: 0.8717541573048718
Correlation between variable :
Sepal.Length and Petal.Width: 0.8179536333691633
Correlation between variable :
Sepal.Width and Sepal.Length: -0.10936924995064935
Correlation between variable :
Sepal.Width and Petal.Length: -0.42051609640115495
Correlation between variable :
Sepal.Width and Petal.Width: -0.3565440896138055
```

Figure 2 – L’affichage de la moyenne, de la variance de l’écart-type pour chaque variable ainsi que le coefficient de corrélation entre les variables 2 à

2

```
1 iris.printMinColumn()
2
3 // Pour chaque colonne :
4 // 0 : Sepal.length
5 // 1 : Sepal.width
6 // 2 : Petal.length
7 // 3 : Petal.width
8
9 // Afficher le minimum de chaque colonne :
```

```
0 : 4.3
1 : 2.0
2 : 1.0
3 : 0.1
```

```
1 iris.printMaxColumn()
2
3 // Pour chaque colonne :
4 // 0 : Sepal.length
5 // 1 : Sepal.width
6 // 2 : Petal.length
7 // 3 : Petal.width
8
9 // Afficher le maximum de chaque colonne :
```

```
0 : 7.9
1 : 4.4
2 : 6.9
3 : 2.5
```

Figure 3 – La vue des minimums et des maximums pour chaque colonne

## 1.3 L'interprétation des résultats

### 1.3.1 La moyenne de chaque colonne

- La première colonne est dédiée à une série de longueur de sépales comprenant des valeurs évoluant entre 4.3 et 7.9 avec une moyenne de 5.84;
- la seconde colonne est consacrée à une série de largeur de sépales intégrant des valeurs allant de 2.0 à 4.4 avec une moyenne de 3.05;
- la troisième colonne correspond à une série de longueur de pétales. La valeur minimale de cette série est 1.0, la valeur maximale est de 6.9. La moyenne est égale à 3.76;
- La quatrième colonne se rapporte aux valeurs d'une série de largeur de pétales. La valeur maximale est 2.5, la valeur minimale est 0.1. La valeur moyenne est égale à 1.20.

### 1.3.2 La variance de chaque colonne

Il s'agit de la moyenne des carrés des écarts à la moyenne.

Colonne	Moyenne	variance	% variance/moyenne
Longueur du sépale	5.84	0.68	11.64
Largeur du sépale	3.05	0.19	6.22
Longueur du pétale	3.76	3.09	82.18
Largeur du pétale	1.20	0.58	48.3

On peut donc remarquer que la colonne dénommée "Longueur du pétale" présente la variance la plus élevée par rapport à la moyenne. Les valeurs de la colonne "Largeur du pétale" vient en seconde position, puis la colonne "Longueur du sépale". Enfin, les valeurs de la colonne "Largeur du sépale" révèle la variance la plus faible, la moyenne des carrées des écarts est donc assez proche de la moyenne.

### 1.3.3 L'écart-type de chaque colonne

L'écart-type est la racine carrée de la variance. Une distribution avec une valeur d'écart-type élevée sera plus étalée, alors qu'une distribution avec une valeur d'écart-type faible sera très resserrée autour de la moyenne.

Colonne	Moyenne	Écart-type	% Écart-type/moyenne
Longueur du sépale	5.84	0.83	14,21
Largeur du sépale	3.05	0.43	14,09
Longueur du pétale	3.76	1.76	46.8
Largeur du pétale	1.20	0.76	63.3

Nous pouvons donc remarquer que la distribution la plus dispersée se trouve dans la colonne dénommée "Largeur du pétale". La seconde la plus dispersée par rapport à la moyenne est celle dénommée "Longueur du pétale". Les distributions nommées "Longueur du sépale" et "Largeur du sépale" sont les plus proches de la moyenne. Par ailleurs, nous pouvons conclure que l'interprétation de l'écart-type présente une meilleure précision de la dispersion d'une distribution que celle de la variance.

### 1.3.4 Les coefficients de corrélation entre toutes les variables 2 à 2

Le coefficient de corrélation est la mesure spécifique qui quantifie la force de la relation linéaire entre deux variables d'une analyse de corrélation. La formule permet de mesurer la distance de chaque point de données depuis la moyenne de la variable et l'utilise pour indiquer dans quelle mesure la relation entre les variables suit une ligne imaginaire tracée par les données. C'est pourquoi les corrélations concernent les relations linéaires [2].

Cette analyse ne détecte pas :

- les valeurs aberrantes (l'analyse sera donc biaisée dans cette situation);
- les relations curvilinéaires (caractérisés par des lignes courbes)

Le coefficient de corrélation est compris entre -1 et 1 sachant que :

- plus le coefficient est proche de zéro, plus la relation linéaire est faible;
- si le coefficient est positif, les valeurs de deux variables tendent à augmenter ensemble;
- si le coefficient est négatif, les valeurs d'une variable tendent à augmenter tandis les valeurs de l'autre variable diminuent;
- les valeurs 1 et -1 représentent des corrélation "parfaites" désignant le fait que les variables évoluent ensemble à une vitesse fixe. La relation est dite linéaire.

Longueur du sépal	largeur du sépal	-0.109
Longueur du sépal	longueur du pétal	0.8717
Longueur du sépal	largeur du pétal	0.8179
Largeur du sépal	longueur du pétal	-0.4205
Largeur du sépal	largeur du pétal	-0.3565
Longueur du pétal	largeur du pétal	0.9627

Les relations "Longueur du sépal - Longueur du pétale", "Longueur du sépal - Largeur du pétale", et "Longueur du pétale - Largeur du pétale" sont proches de 1. La relation entre ces variables est proche d'être linéaire surtout concernant la relation "Longueur du pétale - largeur du pétale" présentant un coefficient de 0.962.

Les relations "Longueur du sépal - Largeur du sépal", "Largeur du sépal - Longueur du pétale", "Largeur du sépal - Largeur du pétale" sont négatives et proches de 0, la linéarité dans ces trois relations est faible.

## 2 Projet n°2 : l'algorithme K-means

En deuxième partie de ce projet, nous souhaitons implémenter un algorithme de classification : la méthode des K-means, afin de classer de manière non supervisée les différentes catégories d'iris. Faites un choix aléatoire pour la partition initiale et utiliser la distance euclidienne comme mesure de distance.

Un descriptif de l'algorithme des K-means est donné ci-contre et disponible sur internet :

Algorithme des K-means

- K est initialisé aléatoirement;
- Étant donné k, et une distance entre chaque paire d'éléments;
- Construire une partition aléatoire comportant k classes non vides;
- Répéter :
  - Calculer le centre de chaque classe de la partition (moyenne des éléments de la classe)
  - Assigner chaque élément à la classe dont le centre est le plus proche;
- Jusqu'à ce que la partition soit stable.

La visualisation des résultats du clustering peut être obtenue via <https://vega.github.io/vega/examples/>. N'hésitez pas à prendre des initiatives dans ce projet. Visualiser les résultats est le meilleur moyen d'évaluer votre travail.

**K-means** est un algorithme d'apprentissage de type non supervisé. Son but est de regrouper des données dans différents ensembles dénommés cluster en français regroupement afin de les labelliser.

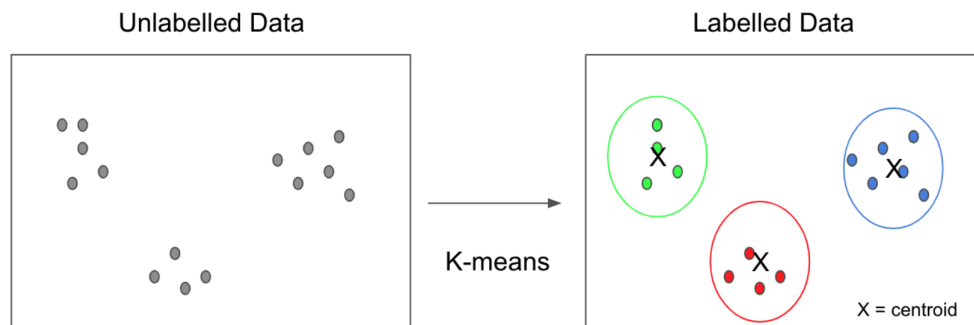


Figure 4 – Un schéma spécifiant la force de k-means en terme de labellisation de données [3]

Ce regroupement peut être effectué pour des données ayant une taille, une forme, une mesure, une caractéristique similaire. Ainsi, l'algorithme K-means permet de découvrir des groupes qui n'ont pas été explicitement définis.

K-means entre dans une des deux catégories d'algorithme de type clustering :

- la première catégorie de clustering se dénomme **la classification hiérarchique**. Il s'agit d'une classification au moyen d'une structure arborescente. Ce type de classification est également subdivisé en deux catégories :
  - la première catégorie de classification hiérarchique correspond au **regroupement agglomératif** où l'approche arborescente est ascendante. Autrement dit, chaque élément de l'ensemble de données constitue un cluster séparé qui sera ensuite fusionné en cluster successivement plus massifs ;
  - la seconde catégorie de classification hiérarchique est un **regroupement dit de division** où l'approche arborescente est descendante. En d'autres mots, l'ensemble de données est divisé en groupes successivement plus petit.
- La seconde catégorie de clustering est le **partitionnement clustering**. Ce dernier est également subdivisé en deux sous-types :
  - La première catégorie se nomme le regroupement K-means qui consiste, comme nous l'avons déjà vu, à diviser les données en K clusters ;
  - la seconde catégorie correspond au **regroupement fuzzy C-means**. Tout comme K-means, le fuzzy C-means en français le clustering flou regroupe des données ayant des caractéristiques similaires. La différence se ressent par le fait que les données peuvent appartenir à plus d'un regroupement.

Il est à noter que la complexité de la distribution hiérarchique est généralement d'au moins  $O(n^2)$ , ce qui rend ces techniques trop lentes pour les grands ensembles de données. Les algorithmes non hiérarchiques présentent une complexité linéaire  $O(n)$  et donc un gain de vitesse de traitement non négligeable.



## 2.1 La mesure à utiliser : la distance euclidienne

Soit le vecteur (ensemble de données)  $x$  de coordonnées  $(x_1, \dots, x_n)$  et le vecteur  $y$  de coordonnées  $(y_1, \dots, y_n)$ .

**La distance euclidienne** définie par la distance entre deux points dans un espace euclidien est la longueur d'un segment de droite entre ces deux points. En général, pour des points donnés par des coordonnées cartésiennes dans n espace euclidien à plusieurs dimensions, la distance est :

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + \dots + (x_n - y_n)^2}$$

**La distance euclidienne pondérée** (par le nombre de cluster K) est définie par :

$$d_k(x, y) = \frac{1}{\sqrt{K}} d(x, y)$$

L'algorithme de clustering K-means est une approche simple et élégante pour partitionner un ensemble de données K clusters distincts et sans chevauchement. L'algorithme k-means commence par initialiser K centres de cluster de façon aléatoire pour ensuite regrouper les données en essayant de séparer les échantillons en k groupes ou clusters. Pour ce faire, chaque point de données est attribué à un centroïde le plus proche en mesurant la distance euclidienne entre tous les points et le centroïde d'un des k clusters. Les centres des clusters sont ensuite mis à jour pour devenir les « centres de gravité » de tous les points qui leurs sont assignés. Cela se fait en recalculant les centres de cluster comme la moyenne des points dans chaque cluster respectif.

Choisir un nombre de cluster K n'est pas une tâche aisée si ce nombre n'est pas connu à l'avance. Effectivement, une valeur K élevée a pour conséquence un fractionnement important des données. Ces nombreux partitionnements limitent la possibilité de découvrir des structures intéressantes. Par ailleurs, un faible nombre de clusters engendre l'agrégation d'un volume important de données engendrant également la même problématique d'interprétabilité des résultats. Il est donc important d'utiliser la méthode des essais et erreurs en modifiant la valeur de K afin de trouver un point d'équilibre.

La méthode Elbow en français coude est l'une des méthodes les plus populaires pour trouver le nombre optimal de clusters. Elle consiste dans un premier temps à lancer K-means avec différentes valeurs de K. Dans un second temps la variance des différents clusters est calculée, voici sa formule :

$$V = \sum_j \sum_{x_i \rightarrow c_j} D(c_j, x_i)^2$$

Où :

- $c_j$  : le centroïde du cluster;
- $x_i$  : la ième observation dans le cluster ayant pour centroïde  $c_j$ ;
- $D(c_j, x_i)$  : La distance (euclidienne ou autre) entre le centre du cluster et le point  $x_i$ .

Comme nous pouvons le remarquer grâce à la figure 3 en page 18, la représentation graphique d'un intervalle de nombres de clusters K en fonction de la variance prend la forme d'un bras.

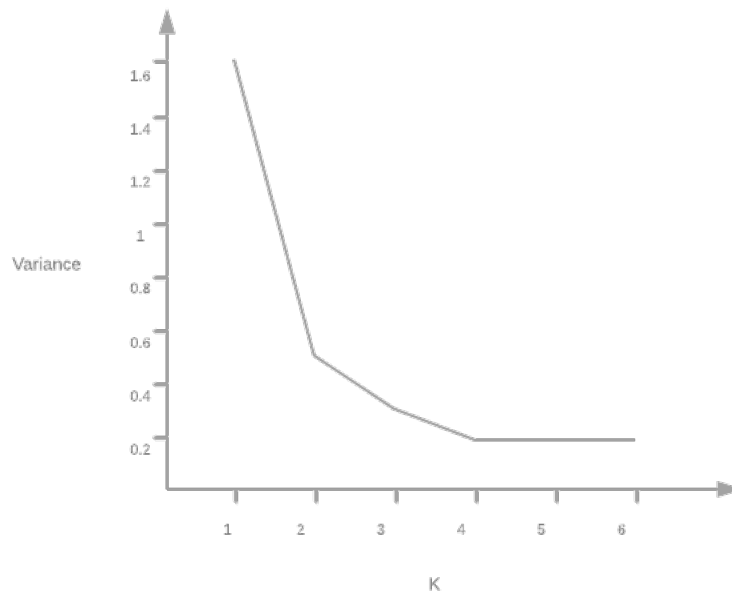


Figure 5 – Déterminer k à l'aide de la méthode Elbow

La valeur de K la plus élevée testée est le chiffre 6. La plus basse est 1. Le nombre optimal de clusters est le point représentant un coude donc 2 ou 3 dans ce cas présent.

En ce qui concerne son utilisation, l'algorithme k-means est largement utilisé dans l'analyse de données pour résoudre différents problèmes tels que [4] :

- La segmentation de la clientèle afin de regrouper les clients les plus adaptés à certains produits ou services ;
- Le regroupement de textes, de documents ou de résultats de recherche ;
- le regroupement d'images ou les couleurs similaires dans la compression d'images ;
- la détection d'anomalies : trouver des dissemblances, des écarts dans les regroupements ;
- l'apprentissage semi-supervisé en combinant les clusters à un ensemble de données étiquetées et à l'apprentissage supervisé.

Néanmoins, il est important de noter que l'algorithme k-means peut ne pas fonctionner correctement si les données ont des formes complexes ou si les données sont bruyantes.

## 2.2 L'implémentation du projet 2

Pour implémenter le projet 2, j'ai utilisé une version de jupyter notebook spécialement dédiée au langage scala. Il porte le nom de almond, voici l'adresse de la page officielle : <https://almond.sh/>

Le code utilisé dans le cadre du projet 2 :

```
import scala.io.Source
import scala.util.Random

class KMeans(dataFile: String, k: Int, maxIterations: Int) {

  private val data: Array[Array[Double]] = readData(dataFile)
  // Initialiser du tableau à deux dimensions data à l'aide des données
  // du fichier data.iris
  private val nbColumns: Int = 4
  // Initialisation de la variable nbColumns qui va prendre la valeur 4
  private val nbLines = data.length
  // Initialisation de la variable NbLines qui va prendre la valeur 150
  private var centroids: Array[Array[Double]] = initCentroids(k)
  // Initialiser de manière aléatoire k centroïdes distincts

  def readData(filename: String): Array[Array[Double]] = {
    // Cette méthode lit les données à partir du nom de fichier
    // intégré dans la variable filename et stocke les valeurs
    // de type double dans la matrice data
    val lines = scala.io.Source.fromFile(filename).getLines.toArray
    // Création de lines, un tableau de string intégrant les lignes
    // du fichier filename
    val data = Array.ofDim[Double](lines.length, 4)
    // Création de data, un tableau de double de dimension (lines.lenght, 4)

    for (i <- 0 until lines.length) {
      // Pour i lignes de lines
      val fields = lines(i).split(",")
      // Tokeniser les éléments de tableau lines dans fields
      for (j <- 0 until 4) {
        // Pour i allant de 0 à 3
        data(i)(j) = fields(j).toDouble
        // Enregistrer fields(j) dans data(i)(j)
      }
    }
    data
    // Retourner data
  }

  private def initCentroids(k: Int): Array[Array[Double]] = {
    // Cette méthode initialise aléatoirement k centroïdes
  }
```

```

    val centroids = new Array[Array[Double]](k)
    // Créer un tableau de double à deux dimension de dimension (1, k)
    for (i <- 0 until k) {
        // Pour tout i de k
        centroids(i) = data(Random.nextInt(data.length))
        // Enregistrer i centroïdes dans le tableau centroids
    }
    centroids
    // Retourner centroids
}

private def euclideanDistance(arr1: Array[Double], arr2: Array[Double]): Double = {
    // Cette méthode calcule la distance euclidienne
    // entre les deux tableaux arr1 et arr2
    var sum = 0.0
    for (i <- 0 until arr1.length) {
        // Pour tout i de arr1.length
        sum += math.pow(arr1(i) - arr2(i), 2)
        // Somme des i arr1 - les i arr2 au carré
    }
    math.sqrt(sum)
    // Retourner la racine carré de la somme totale
}

private def assignToClusters(): Array[Int] = {
    // Cette méthode attribue chaque point de données à un centroïde
    // le plus proche à l'aide de la distance euclidienne pour mesurer
    // la similarité entre chaque point de données et chaque centroïde
    val clusterAssignments = new Array[Int](data.length)
    for (i <- 0 until data.length) {
        // Pour chaque i de data :
        var minDistance = Double.MaxValue
        // Initialiser minDistance à la valeur maximale
        var closestCenterIndex = -1
        // Initialiser de closestCenterIndex à -1
        for (j <- 0 until centroids.length) {
            // Pour tout j de centroids
            val distance = euclideanDistance(data(i), centroids(j))
            // Calculer la distance euclidienne entre un point d'indice i dans data
            // et un centroïde d'indice j dans centroids
            if (distance < minDistance) {
                // Si la distance euclidienne est inférieure
                // à la distance minimale enregistrée
                minDistance = distance
                // alors la distance est enregistrée dans mindistance
                closestCenterIndex = j
                // et l'indice j est enregistré dans closestCenterIndex
            }
        }
    }
}

```

```

    }
    clusterAssignments(i) = closestCenterIndex
    // Le tableau clusterAssignments permet d'enregistrer
    // les indices des centroïdes les plus proches pour chaque i
  }
  clusterAssignments
  // Retourner clusterAssignments
}

// Calcul des nouveaux centres de chaque cluster
private def calculateCenters(clusterAssignments: Array[Int]): Array[Array[Double]] = {
  // Cette méthode calcule les nouveaux centres grâce
  // à la moyenne des données assignées à chaque cluster
  val newCenters = Array.ofDim[Double](k, data(o).length)
  // Création de newCenters, un tableau de double de dimension (k, 4)
  val counts = new Array[Int](k)
  // Création de counts, un tableau d'entier de dimension k

  for (i <- data.indices) {
    // Pour tout i de data.length :
    val clusterId = clusterAssignments(i)
    // Récupérer la valeur du cluster enregistrée dans les i clusterAssignments
    for (j <- data(i).indices) {
      // Pour tout j de data(i).length :
      newCenters(clusterId)(j) += data(i)(j)
      // Calculer la somme des i, j data et enregistrer le résultat
      // dans les clusterId, j de newCenters

    }
    counts(clusterId) += 1
    // Enregistrer le nombre de points de chaque cluster
  }
  for (i <- centroids.indices) {
    // Pour tout i de centroids.indices :
    for (j <- centroids(i).indices) {
      // Pour tout j de centroids(i).indices :
      newCenters(i)(j) /= counts(i)
      // Diviser les i, j newCenters par les i counts
      // et enregistrer le résultat dans les i, j newCenters
    }
  }
  newCenters
  //Retourner newCenters
}

def run(): Array[Int] = {
  // Cette méthode exécute l'algorithme k-means
  var clusterIds = assignToClusters()

```

```

// Attribuer chaque point du dataset à un centroïde initialement aléatoirement
var newCenters = calculateCenters(clusterIds)
// Calculer le nouveau centroïde
while (centroids.sameElements(newCenters)) {
    // Tant que centroids est différent de newCenters :
    centroids = newCenters
    // Enregistrer newCenters dans centroids
    clusterIds = assignToClusters()
    // Assigner les points de data au nouveau centroïde
    newCenters = calculateCenters(clusterIds)
    // Calculer de nouveaux centroïdes newCenters
}
clusterIds
// Retourner ClusterIds
}

def getCentroids(): Array[Array[Double]] = {
    // Assesseur en lecture concernant le tableau centroids
    return centroids
    // Retourner centroids
}

def predict(point: Array[Double]): Int = {
    // Cette méthode renvoie l'indice du cluster
    // auquel un point donné en argument appartient
    var minDistance = Double.MaxValue
    // Initialiser minDistance à la valeur maximale
    var clusterIndex = -1
    // Initialiser clusterIndex à -1
    for (i <- centroids.indices) {
        // Pour tout i de centroids.indices :
        val distance = euclideanDistance(point, centroids(i))
        // Calculer la distance euclidienne entre le point et les centroïdes
        if (distance < minDistance) {
            // Si la distance est inférieure à minDistance
            minDistance = distance
            // minDistance prend la valeur de distance
            clusterIndex = i
            // Ajouter la valeur de i dans clusterIndex
        }
    }
    clusterIndex
    // Retourner clusterIndex
}

}

val kMeans = new KMeans("iris.data", 3, 1000)
// Créer une instance de la classe KMeans

```

```
val clusters = kMeans.run()
// Exécuter la méthode run implémentant l'algorithme K-Means

println(s"Final clusters: ${clusters.mkString(", ")}")
// Afficher les indices des clusters après exécution de K-Means
```

La capture d'écran des résultats :

[illegible]

Figure 6 – La vue des résultats de l’algorithme k-means appliqué au dataset iris

L'affichage de l'ensemble des indices des clusters met en lumière que certaines données des clusters 0 et 2 sont mélangées. Les données récoltées via le fichier iris.data n'ont pas été mélangées. Dans ce fichier, les cinquante premières données sont des iris setosa, les cinquante suivantes sont les iris versicolor et les cinquante dernières sont les iris virginica. À première vue, le cluster 0 correspond donc aux iris virginica, le 1 aux iris setosa et le cluster 2 aux iris versicolor.

L'affichage des trois centroïdes :

```
1 val cent = kMeans.getCentroids()
2 // Renvoie les valeurs des centroïdes

cent: Array[Array[Double]] = Array(
  Array(6.3, 2.5, 4.9, 1.5),
  Array(5.0, 3.5, 1.6, 0.6),
  Array(6.6, 3.0, 4.4, 1.4)
)
```

Figure 7 – La vue de l'exécution de la méthode `kMeans.getCentroids()`

Les trois centroïdes sont les points :

- (6.3, 2.5, 4.9, 1.5);
- (5.0, 3.5, 1.6, 0.6);
- (6.6, 3.0, 4.4, 1.4).

L'affichage de la prédiction d'un point :

```
1 val indiceCluster = kMeans.predict(Array[Double](5, 3, 1, 0.5))  
indiceCluster: Int = 1
```

Figure 8 – La vue d'un cas d'utilisation de la méthode `kMeans.predict(Array[Double](5, 3, 1, 0.5))`

Le point (5, 3, 1, 0.5) fait partie du cluster 1 donc des iris setosa.

## 2.3 L'importation de la bibliothèque graphique plotly pour Almond

```
import $ivy.'org.plotly-scala::plotly-almond:0.8.2'
// Téléchargement la version de 0.8.2 de plotly-scala pour almond

import plotly._
// Importer tous les sous-modules de Plotly
import plotly.element._
// Importer les définitions pour différents éléments graphiques
// comme les couleurs, les bordures, etc
import plotly.layout._
// Importer la définition de la mise en page
// des graphiques : les titres, les légendes, les axes, etc
import plotly.Almond._
// Importer une extension de plotly pour almond sous jupyter notebook
```



## 2.4 La représentation du nombre d'éléments pour chaque classe

```
val data = scala.io.Source.fromFile("iris.data").getLines.map(_.split(",")).toSeq
// Tokenisation des lignes du fichier iris.data

val irisSetosaCount = data.filter(_(4) == "Iris-setosa").size
// Compter le nombre de lignes contenant la chaîne de caractères : "Iris-setosa"
val irisVersicolorCount = data.filter(_(4) == "Iris-versicolor").size
// Compter le nombre de lignes contenant la chaîne de caractères : "Iris-versicolor"
val irisVirginicaCount = data.filter(_(4) == "Iris-virginica").size
// Compter le nombre de lignes contenant la chaîne de caractères : "Iris-virginica"

val data_iris = Seq(
  Bar(
    // Créer une séquence de barres
    Seq("Iris-setosa", "Iris-versicolor", "Iris-virginica"),
    // pour Iris-setosa, Iris-versicolor, Iris-virginica
    Seq(irisSetosaCount, irisVersicolorCount, irisVirginicaCount)
    // Montrer le nombre d'échantillons pour chaque type d'iris
  )
)

plot(data_iris)
// Afficher le graphique à barres
```

La capture d'écran du graphe à barres :

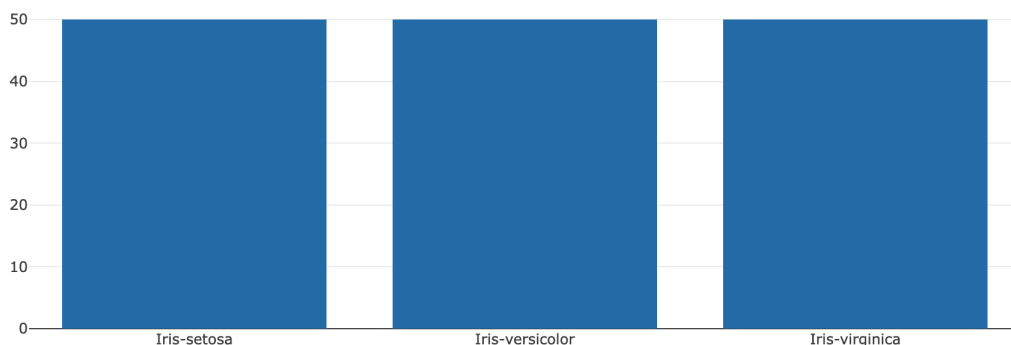


Figure 9 – La vue du graphe à barres indiquant le nombre de données pour chaque classe d'iris

## 2.5 La représentation de la longueur et de la largeur des sépales pour chaque classe

```
val irisSetosa_x = data.filter(_(4) == "Iris-setosa").map(x => x(0).toDouble)
// Extraire les valeurs correspondant à la première caractéristique
// de l'ensemble des lignes dédiées aux iris setosa
val irisSetosa_y = data.filter(_(4) == "Iris-setosa").map(x => x(1).toDouble)
// Extraire les valeurs correspondant à la seconde caractéristique
// de l'ensemble des lignes dédiées aux iris setosa

val irisVersicolor_x = data.filter(_(4) == "Iris-versicolor").map(x => x(0).toDouble)
// Extraire les valeurs correspondant à la première caractéristique
// de l'ensemble des lignes dédiées aux iris versicolor
val irisVersicolor_y = data.filter(_(4) == "Iris-versicolor").map(x => x(1).toDouble)
// Extraire les valeurs correspondant à la seconde caractéristique
// de l'ensemble des lignes dédiées aux iris versicolor

val irisVirginica_x = data.filter(_(4) == "Iris-virginica").map(x => x(0).toDouble)
// Extraire les valeurs correspondant à la première caractéristique
// de l'ensemble des lignes dédiées aux iris virginica
val irisVirginica_y = data.filter(_(4) == "Iris-virginica").map(x => x(1).toDouble)
// Extraire les valeurs correspondant à la seconde caractéristique
// de l'ensemble des lignes dédiées aux iris virginica

val trace1 = Scatter(
  // Créer une instance de la classe Scatter dans le but d'afficher
  // deux tableaux sous la forme d'un graphique
  irisSetosa_x,
  // Coordonnées x du graphe : un tableau intégrant
  // la première caractéristique de l'ensemble des iris setosa
  irisSetosa_y,
  // Coordonnées y du graphe : un tableau intégrant
  // la seconde caractéristique de l'ensemble des iris setosa
  mode = ScatterMode(ScatterMode.Markers),
  // Utiliser des marqueurs
  name = "Iris Setosa",
  // Le nom du graphe
  marker = Marker(
    // Couleur et taille des marqueurs et de la ligne
    // de sélection de chaque marqueur
    color = Color.RGBA(255, 0, 0, 0.8),
    size = 10,
    line = Line(
      color = Color.RGBA(255, 0, 0, 1),
      width = 1
    )
  )
)
```

```

val trace2 = Scatter(
    // Créer une instance de la classe Scatter dans le but
    // d'afficher deux tableaux sous la forme d'un graphique
    irisVersicolor_x,
    // Coordonnées x du graphe : un tableau intégrant
    // la première caractéristique de l'ensemble des iris versicolor
    irisVersicolor_y,
    // Coordonnées y du graphe : un tableau intégrant
    // la seconde caractéristique de l'ensemble des iris versicolor
    mode = ScatterMode(ScatterMode.Markers),
    // Utiliser des marqueurs
    name = "Iris Versicolor",
    // Le nom du graphe
    marker = Marker(
        // Couleur et taille des marqueurs et de la ligne
        // de sélection de chaque marqueur
        color = Color.RGBA(0, 255, 0, 0.8),
        size = 10,
        line = Line(
            color = Color.RGBA(0, 255, 0, 1),
            width = 1
        )
    )
)

val trace3 = Scatter(
    // Créer une instance de la classe Scatter dans le but
    // d'afficher deux tableaux sous la forme d'un graphique
    irisVirginica_x,
    // Coordonnées x du graphe : un tableau intégrant
    // la première caractéristique de l'ensemble des iris virginica
    irisVirginica_y,
    // Coordonnées y du graphe : un tableau intégrant
    // la seconde caractéristique de l'ensemble des iris virginica
    mode = ScatterMode(ScatterMode.Markers),
    // Utiliser des marqueurs
    name = "Iris Virginica",
    // Le nom du graphe
    marker = Marker(
        color = Color.RGBA(0, 0, 255, 0.8),
        // Couleur et taille des marqueurs et de la ligne
        // de sélection de chaque marqueur
        size = 10,
        line = Line(
            color = Color.RGBA(0, 0, 255, 1),
            width = 1
        )
    )
)

```

```

)
)

val layout = Layout(
    // Créer une instance Layout dédiée aux propriétés du graphe
    title = "Iris Data",
    // Le titre du graphe
    xaxis = Axis(
        title = "Sepal Length (cm)"
        // Le titre de l'axe des x
    ),
    yaxis = Axis(
        title = "Sepal width (cm)"
        // Le titre de l'axe des y
    )
)

plot(Seq(trace1, trace2, trace3), layout)
// Afficher le graphe et sa mise en page

```

La capture d'écran du graphe de la longueur et de la largeur des sépales pour chaque classe :

Cette représentation graphique affiche un nuage de points pour les longueurs et les largeurs de sépales, où les 3 espèces d'iris représentées pour une couleur distincte. On peut remarquer un écart distinct dans les distributions de mesures entre les setosa et les deux autres : versicolor et virginica. En outre, ces deux dernières s'amalgament très nettement.

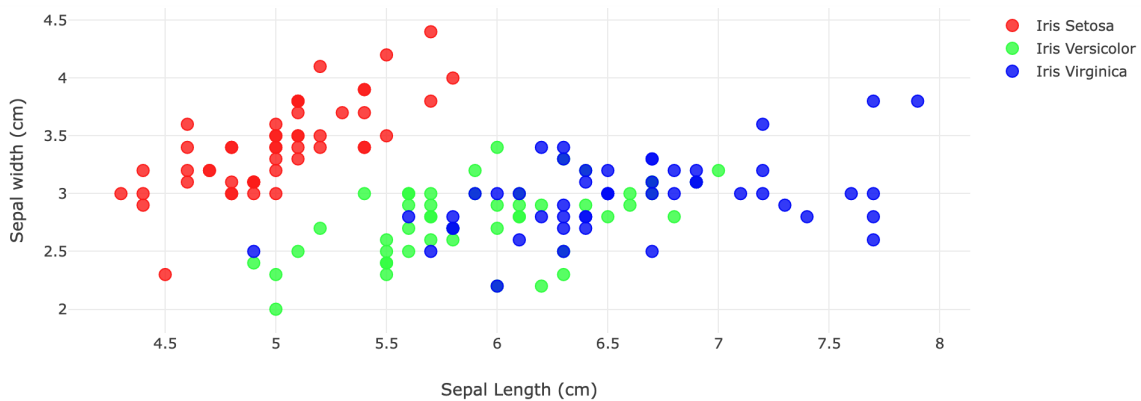


Figure 10 – La vue de la représentation de la longueur et de la largeur des sépales pour chaque classe

## 2.6 La représentation de la longueur et de la largeur des pétales pour chaque classe

```
val irisSetosa_x = data.filter(_(4) == "Iris-setosa").map(x => x(2).toDouble)
// Extraire les valeurs correspondant à la troisième caractéristique
// de l'ensemble des lignes dédiées aux iris setosa
val irisSetosa_y = data.filter(_(4) == "Iris-setosa").map(x => x(3).toDouble)
// Extraire les valeurs correspondant à la quatrième caractéristique
// de l'ensemble des lignes dédiées aux iris setosa

val irisVersicolor_x = data.filter(_(4) == "Iris-versicolor").map(x => x(2).toDouble)
// Extraire les valeurs correspondant à la troisième caractéristique
// de l'ensemble des lignes dédiées aux iris versicolor
val irisVersicolor_y = data.filter(_(4) == "Iris-versicolor").map(x => x(3).toDouble)
// Extraire les valeurs correspondant à la quatrième caractéristique
// de l'ensemble des lignes dédiées aux iris versicolor

val irisVirginica_x = data.filter(_(4) == "Iris-virginica").map(x => x(2).toDouble)
// Extraire les valeurs correspondant à la troisième caractéristique
// de l'ensemble des lignes dédiées aux iris virginica
val irisVirginica_y = data.filter(_(4) == "Iris-virginica").map(x => x(3).toDouble)
// Extraire les valeurs correspondant à la quatrième caractéristique
// de l'ensemble des lignes dédiées aux iris virginica

val trace1 = Scatter(
  // Créer une instance de la classe Scatter dans le but
  // d'afficher deux tableaux sous la forme d'un graphique
  irisSetosa_x,
  // Coordonnées x du graphe : un tableau intégrant la troisième caractéristique
  // de l'ensemble des iris virginica
  irisSetosa_y,
  // Coordonnées y du graphe : un tableau intégrant la quatrième caractéristique
  // de l'ensemble des iris virginica
  mode = ScatterMode(ScatterMode.Markers),
  name = "Iris Setosa",
  // Le nom du graphe
  marker = Marker(
    // Utiliser des marqueurs
    color = Color.RGBA(255, 0, 0, 0.8),
    // Couleur et taille des marqueurs et de la ligne de sélection de chaque marqueur
    size = 10,
    line = Line(
      color = Color.RGBA(255, 0, 0, 1),
      width = 1
    )
  )
)
```

```

val trace2 = Scatter(
    // Créer une instance de la classe Scatter dans le but
    // d'afficher deux tableaux sous la forme d'un graphique
    irisVersicolor_x,
    // Coordonnées x du graphe : un tableau intégrant la troisième caractéristique
    // de l'ensemble des iris versicolor
    irisVersicolor_y,
    // Coordonnées y du graphe : un tableau intégrant la quatrième caractéristique
    // de l'ensemble des iris versicolor
    mode = ScatterMode(ScatterMode.Markers),
    name = "Iris Versicolor",
    // Le nom du graphe
    marker = Marker(
        // Utiliser des marqueurs
        color = Color.RGBA(0, 255, 0, 0.8),
        // Couleur et taille des marqueurs et de la ligne de sélection de chaque marqueur
        size = 10,
        line = Line(
            color = Color.RGBA(0, 255, 0, 1),
            width = 1
        )
    )
)

val trace3 = Scatter(
    // Créer une instance de la classe Scatter dans le but d'afficher
    // deux tableaux sous la forme d'un graphique
    irisVirginica_x,
    // Coordonnées x du graphe : un tableau intégrant la troisième caractéristique
    // de l'ensemble des iris virginica
    irisVirginica_y,
    // Coordonnées y du graphe : un tableau intégrant la quatrième caractéristique
    // de l'ensemble des iris virginica
    mode = ScatterMode(ScatterMode.Markers),
    name = "Iris Virginica",
    // Le nom du graphe
    marker = Marker(
        // Utiliser des marqueurs
        color = Color.RGBA(0, 0, 255, 0.8),
        // Couleur et taille des marqueurs et de la ligne de sélection de chaque marqueur
        size = 10,
        line = Line(
            color = Color.RGBA(0, 0, 255, 1),
            width = 1
        )
    )
)

```

```

val layout = Layout(
  // Créer une instance Layout dédiée aux propriétés du graphe
  title = "Iris Data",
  // Le titre du graphe
  xaxis = Axis(
    title = "Petal Length (cm)"
    // Le titre l'axe des x
  ),
  yaxis = Axis(
    title = "Petal width (cm)"
    // Le titre l'axe des y
  )
)

plot(Seq(trace1, trace2, trace3), layout)
// Afficher le graphe et sa mise en page

```

La capture d'écran du graphe de la longueur et de la largeur des pétales pour chaque classe

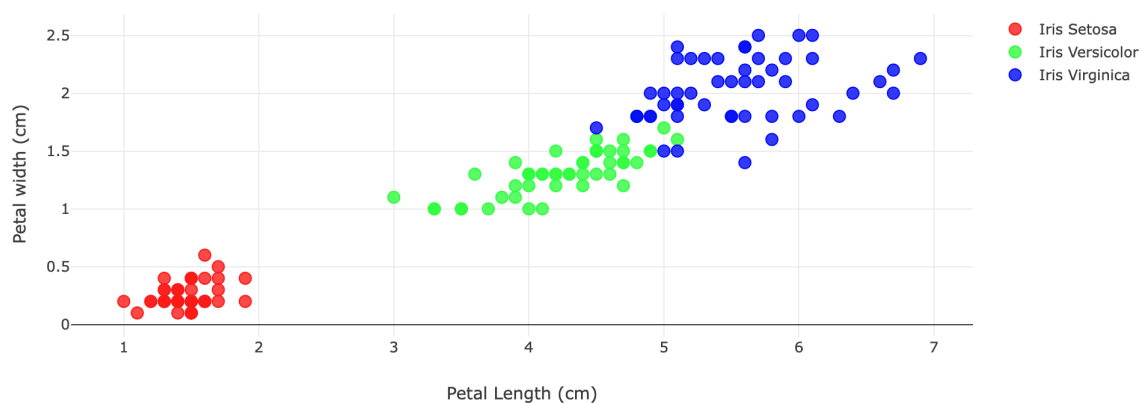


Figure 11 – La vue de la représentation de la longueur et de la largeur des pétales pour chaque classe

Cette représentation graphique affiche un nuage de points pour les longueurs et largeurs de pétales. La dispersion des classes est nettement plus marquée dans cette représentation. Cependant le regroupement des iris versicolor et celui des iris virginica sont très rapprochés. Concernant ces dernières, quelques données ont tendances à se mélanger.

## 2.7 En conclusion :

K-means est un algorithme non supervisé uniquement dédié aux données numériques. Cette implémentation est simple et très efficace pour interpréter des clusters par son centre de gravité. Néanmoins, il nécessite une sélection préalable du nombre de clusters. Certaines méthodes peuvent aider comme la méthode elbow mise en lumière dans ce projet.

En ce qui concerne les points négatifs de k-means, il arrive parfois que ce dernier converge vers un minimum local. Cette dernière expression signifie que cet algorithme est sensible à l'initialisation des clusters de départ et aux regroupements de clusters comme nous avons pu le remarquer en visualisant la longueur et la largeur des sépales et des pétales ou encore le regroupement des clusters 0 et 2 dans la figure 3 du projet. Dans ces différentes visualisations, le cluster iris versicolor et celui des iris virginica sont proches voire mélangés. Par conséquent, K-means donne des résultats médiocres pour les données qui ne sont pas linéairement séparables.

En définitive, l'utilisation de K-means nécessite que les clusters soient sphériques, de taille similaire et de densité similaire ce qui n'est pas toujours le cas des jeux de données présentant des valeurs aberrantes voire des données bruyantes. Si les bonnes conditions d'utilisation sont réunies, cette méthode non supervisée est idéale. Effectivement, il est à noter que K-means est une des méthodes de clustering les plus utilisées dans de nombreux domaines scientifiques.



## Références

- [1] Statistique Canada. Mesures de la dispersion. *Calculer la variance et l'écart-type*, 2021. <https://www150.statcan.gc.ca/n1/edu/power-pouvoir/ch12/5214891-fra.htm>.
- [2] Corrélation (statistiques). *Wikipédia*. [https://fr.wikipedia.org/wiki/Corr%C3%A9lation\\_\(statistiques\)](https://fr.wikipedia.org/wiki/Corr%C3%A9lation_(statistiques)).
- [3] Data Science Team. K signifie regroupement (k-means clustering). *Apprentissage automatique*, 2020. <https://datascience.eu/fr/apprentissage-automatique/k-means-le-clustering-dans-lapprentissage-machine>.
- [4] Juvénal JVC. K-means : fonctionnement et utilisation dans un projet de clustering. <https://www.data-transitionnumerique.com/k-means>.