

Algorithmes d'apprentissage

CSI 4506 - automne 2026

Marcel Turcotte

Version : 15 sept. 2026 16h04

Préambule

Message du jour



« [Misleading Metaphors and Real Risks](#) », Melanie Mitchell, 2026-09-10.

L'IA fait l'objet de beaucoup de battage médiatique et de bruit, tant dans les médias traditionnels qu'en ligne. Les récits spectaculaires attirent l'attention, et les entreprises d'IA ont des produits et des visions particulières de l'avenir à promouvoir. Il faut donc aller au-delà des titres, examiner les éléments probants et se demander qui formule chaque affirmation.

Le débat public comprend un large éventail de points de vue. Des chercheurs comme Gary Marcus, Geoffrey Hinton, Stuart Russell et Yoshua Bengio mettent souvent l'accent sur des limites ou des risques sérieux, bien que leurs arguments et les réponses qu'ils proposent diffèrent. Melanie Mitchell adopte généralement une perspective plus mesurée, fondée sur les données scientifiques. Elle distingue souvent ce qu'un système d'IA a manifestement accompli du langage anthropomorphique employé pour le décrire, tout en reconnaissant les risques techniques et sociaux réels.

Dans cet article, Mitchell examine les descriptions d'agents d'IA qui auraient « échappé à tout contrôle », « pris la fuite » ou formé un « essaim ». Selon elle, ces métaphores suggèrent des intentions humaines et une perte de contrôle littérale que les événements rapportés n'établissent pas. Elle relève néanmoins des risques concrets liés à une cybersécurité déficiente, à une supervision insuffisante, aux agents persistants d'apprentissage par renforcement et au détournement de récompense. Plus généralement, les métaphores trompeuses peuvent masquer les décisions humaines, techniques et politiques qui exigent notre attention.

Devant un récit alarmiste ou triomphaliste sur l'IA, il est souvent utile de vérifier si Mitchell en a parlé. Il ne s'agit pas de considérer une personne comme l'autorité définitive, mais plutôt de profiter de ses analyses, qui présentent souvent des interprétations concurrentes et relient les affirmations aux données scientifiques.

Brève biographie. Melanie Mitchell est professeure James B. Alley Jr. au Santa Fe Institute. Ses travaux relient l'intelligence artificielle, les sciences cognitives et les systèmes complexes. Elle s'intéresse notamment à l'abstraction, à l'analogie et au raisonnement chez les humains et les machines. Elle est l'autrice de plusieurs livres, dont *Complexity: A Guided Tour* et *Artificial Intelligence: A Guide for Thinking Humans*.

Ressources supplémentaires :

- [Site Web de Melanie Mitchell](#)
- [The Nature of Intelligence](#), une saison de six épisodes du balado *Complexity* du Santa Fe Institute, publiée de septembre à décembre 2024
- [Are We Thinking Correctly About AI Intelligence?](#), *The Joy of Why*, 2026-08-20

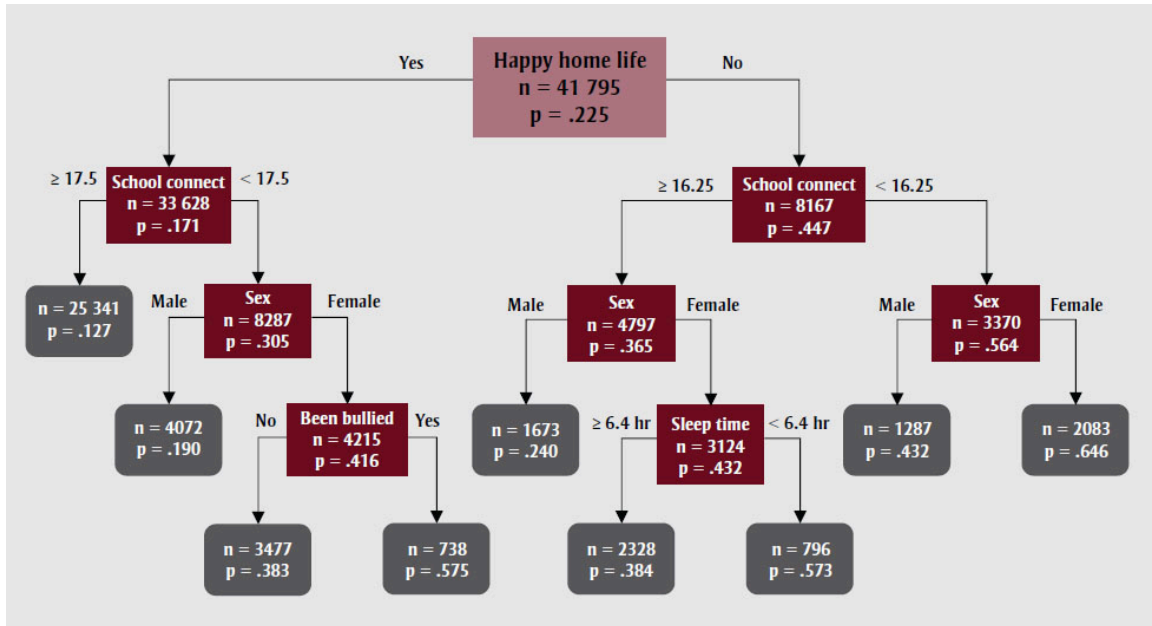
Objectifs d'apprentissage

- **Distinguer** le modèle, la fonction objectif et l'optimiseur dans un algorithme d'apprentissage.
- **Décrire** les arbres de décision et **appliquer** l'entropie pour choisir une division.
- **Expliquer** KNN pour la classification et la régression, y compris la prédiction uniforme et pondérée par la distance.
- **Interpréter** les frontières de décision et le concept de séparabilité linéaire.

Comme je l'ai indiqué dans le cours d'introduction, je souhaite présenter une série de concepts menant à l'apprentissage profond. La régression linéaire constituerait un point de départ logique. Il est toutefois tout aussi important d'acquérir une compréhension générale d'autres algorithmes d'apprentissage, car leurs structures de modèles et leurs processus d'entraînement diffèrent considérablement. Nous étudierons donc les arbres de décision et les k plus proches voisins avant d'introduire la régression linéaire.

Arbre de décision

Interprétable



Attribution : [Agence de la santé publique du Canada](#)

Les arbres de décision sont utiles parce qu'ils exposent clairement les règles apprises par le modèle. L'arbre ci-dessus illustre les résultats d'une étude sur les symptômes d'anxiété cliniquement significatifs. Dans ce cas, le premier facteur était la présence déclarée d'un milieu familial positif (Battista et al. 2023).

Qu'est-ce qu'un arbre de décision ?

- Un arbre de décision est une **structure hiérarchique enracinée** utilisée pour des tâches de **classification** et de **régression**.
- Chaque **nœud interne** est un nœud de décision qui effectue un test binaire sur un attribut particulier (j), par exemple pour déterminer si le nombre de liens à l'école dépasse un seuil donné.
- Chaque **feuille** produit une prédiction : une étiquette ou des probabilités de classes en classification, ou une valeur numérique en régression.

La **structure** de l'arbre est **inférée (apprise)** à partir des **données d'entraînement**.

Les arbres de décision peuvent dépasser les divisions binaires. Par exemple, des algorithmes comme **ID3** permettent aux nœuds d'avoir plusieurs enfants.

Classifier de nouvelles instances (inférence)

- Commencer à la **racine** de l'arbre. Répondre ensuite à une suite de **questions binaires** jusqu'à atteindre une **feuille**. L'étiquette associée à cette feuille constitue la **classification** de l'instance.
- Certains algorithmes conservent plutôt une **distribution de probabilités** dans la feuille, représentant la proportion des exemples d'entraînement appartenant à chacune des classes k .

Lorsqu'un **arbre de décision** résout une tâche de **régression**, chaque **feuille** conserve une **valeur prédite**. Plus précisément :

$$\hat{y}_{\text{leaf}} = \frac{1}{N_{\text{leaf}}} \sum_{i \in \text{leaf}} y_i,$$

où N_{leaf} est le nombre d'exemples d'entraînement qui ont atteint cette feuille et les y_i sont leurs valeurs cibles.

Frontière de décision

Manchots de Palmer

```
In [1]: # Chargement du jeu de données

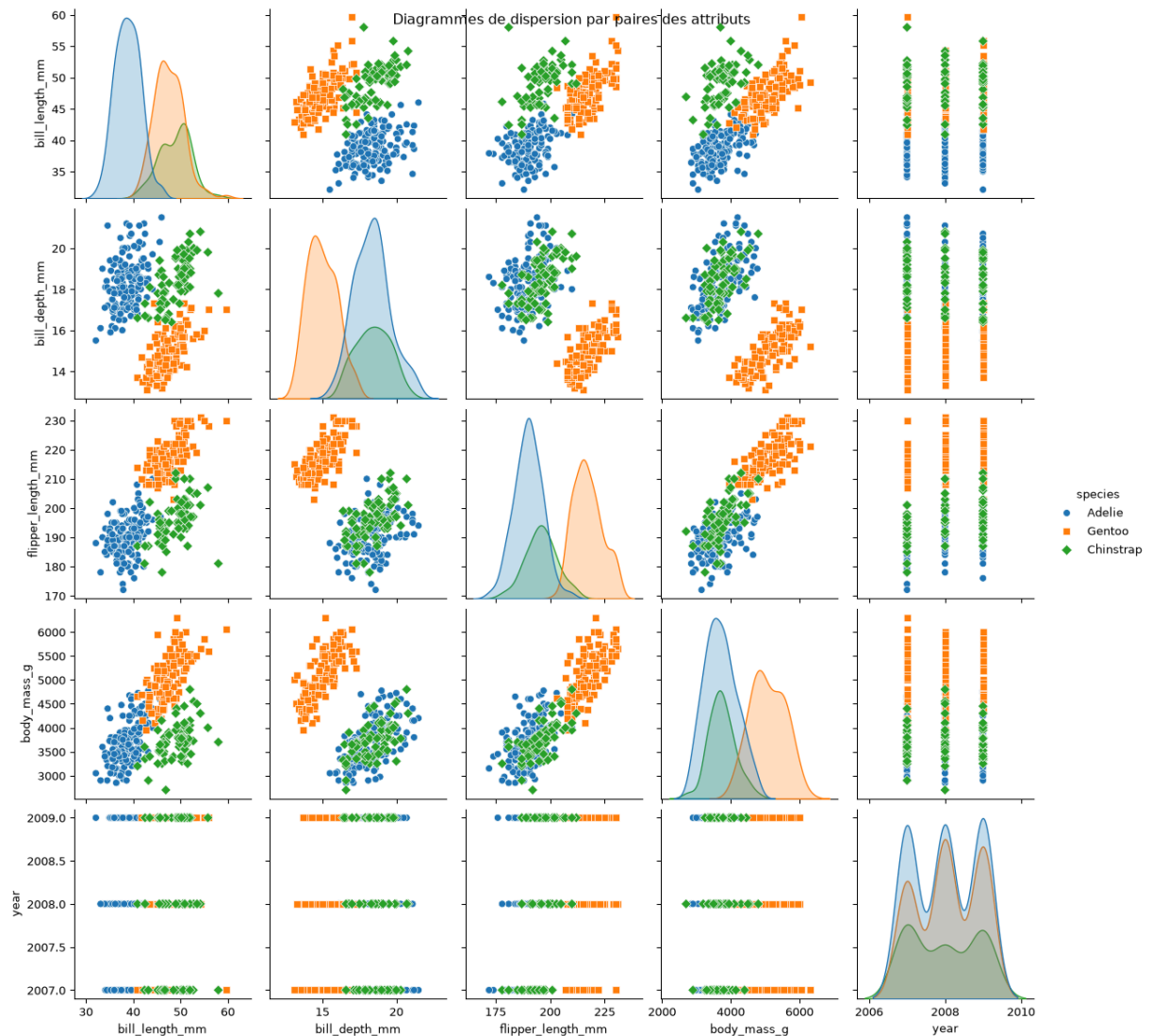
try:
    from palmerpenguins import load_penguins
except:
    ! pip install palmerpenguins
    from palmerpenguins import load_penguins

penguins = load_penguins()

# Diagrammes par paires avec seaborn

import matplotlib.pyplot as plt
import seaborn as sns

sns.pairplot(penguins, hue='species', markers=["o", "s", "D"])
plt.suptitle("Diagrammes de dispersion par paires des attributs")
plt.show()
```

Problème de classification binaire

- Plusieurs diagrammes de dispersion révèlent un regroupement distinct des instances **Gentoo**.
- Pour illustrer le prochain exemple, nous proposons un problème de **classification binaire** : **Gentoo** ou **non-Gentoo**.
- Notre analyse portera sur deux attributs : la **masse corporelle** et la **profondeur du bec**.

Définition

Une **frontière de décision** (*decision boundary*) est une «frontière» qui divise l'espace des attributs en **régions** correspondant à **différentes étiquettes de classes**.

Le terme **frontière** sera précisé dans les prochaines diapositives.

Frontière de décision

La frontière de décision entre ces attributs peut être représentée par une **droite**.

```
In [2]: # Importer les bibliothèques nécessaires
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

try:
    from palmerpenguins import load_penguins
except:
    ! pip install palmerpenguins
    from palmerpenguins import load_penguins

# Charger le jeu de données des manchots de Palmer
df = load_penguins()

# Conserver seulement les attributs nécessaires
features = ['bill_depth_mm', 'body_mass_g']
df = df[features + ['species']]

# Retirer les lignes contenant des valeurs manquantes
df.dropna(inplace=True)

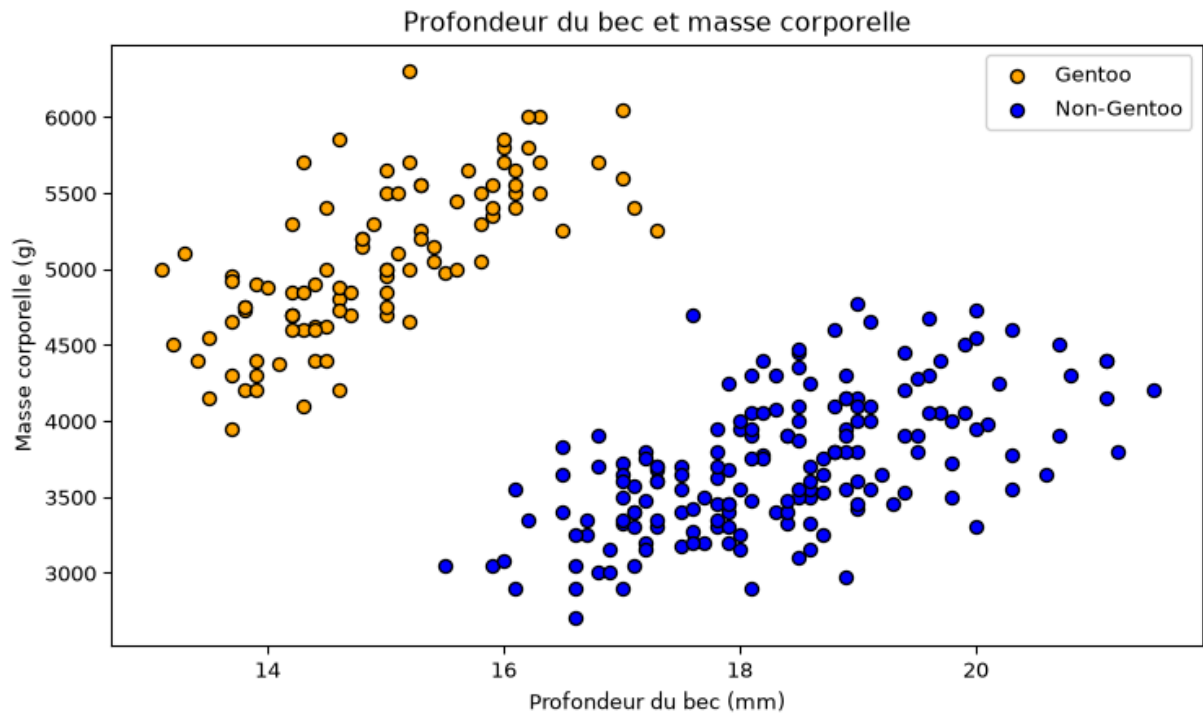
# Créer un problème binaire : « Gentoo » ou « non-Gentoo »
df['species_binary'] = df['species'].apply(lambda x: 1 if x == 'Gentoo' else 0)

# Définir la matrice d'attributs X et le vecteur cible y
X = df[features].values
y = df['species_binary'].values

# Diviser les données en jeux d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Tracer le diagramme de dispersion initial
def plot_scatter(X, y):
    plt.figure(figsize=(9, 5))
    plt.scatter(X[y == 1, 0], X[y == 1, 1], color='orange', edgecolors='k',
               plt.scatter(X[y == 0, 0], X[y == 0, 1], color='blue', edgecolors='k', ma
    plt.xlabel('Profondeur du bec (mm)')
    plt.ylabel('Masse corporelle (g)')
    plt.title('Profondeur du bec et masse corporelle')
    plt.legend()
    plt.show()

# Afficher le diagramme de dispersion initial
plot_scatter(X_train, y_train)
```



Frontière de décision

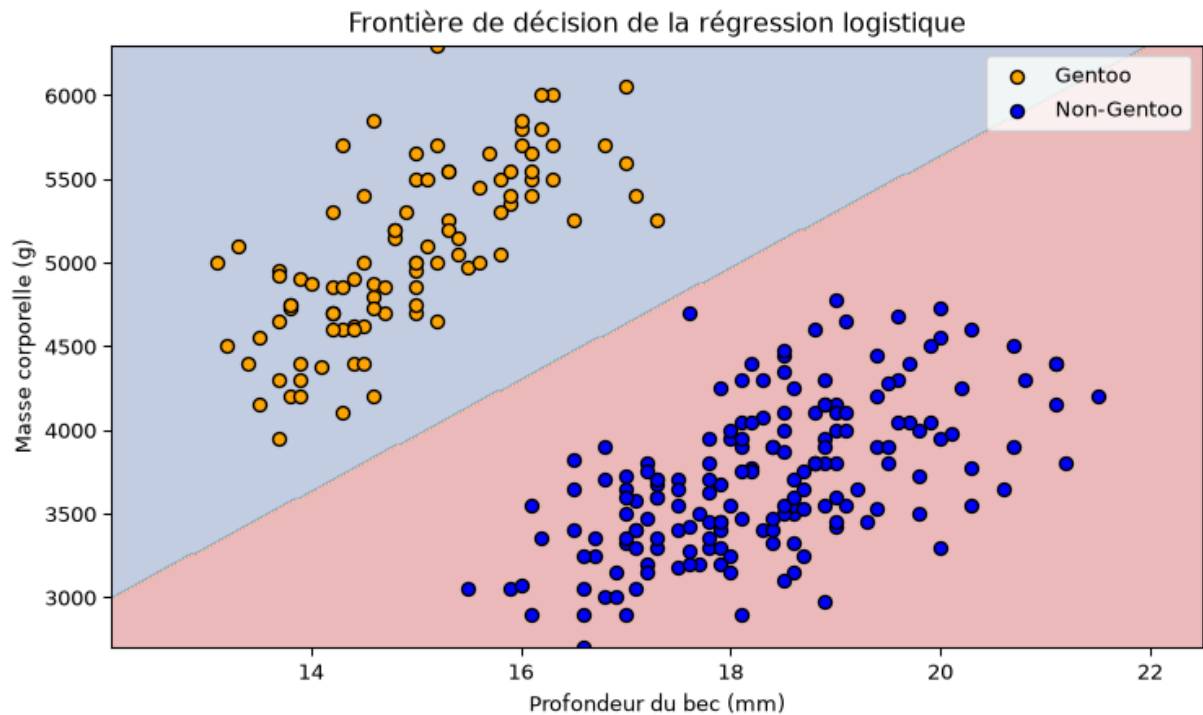
La frontière de décision entre ces attributs peut être représentée par une **droite**.

```
In [3]: # Entraîner un modèle de régression logistique
model = LogisticRegression()
model.fit(X_train, y_train)

# Tracer la frontière de décision
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, 300),
        np.linspace(y_min, y_max, 300)
    )
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(9, 5))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='RdYlBu')
    plt.scatter(X[y == 1, 0], X[y == 1, 1], color='orange', edgecolors='k',
                plt.scatter(X[y == 0, 0], X[y == 0, 1], color='blue', edgecolors='k', ma
    plt.xlabel('Profondeur du bec (mm)')
    plt.ylabel('Masse corporelle (g)')
    plt.title('Frontière de décision de la régression logistique')
    plt.legend()
    plt.show()

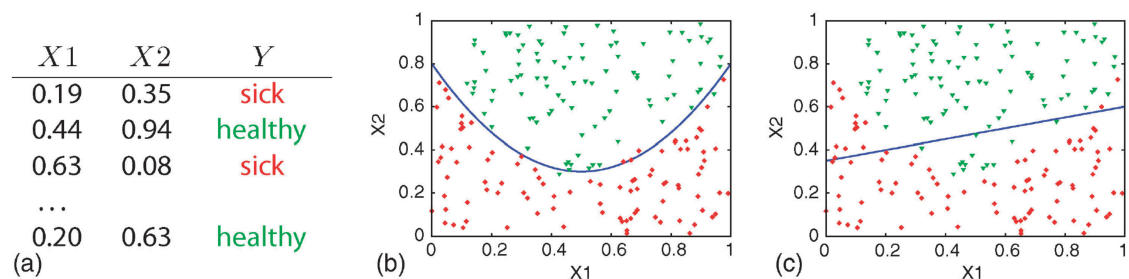
# Afficher la frontière de décision sur le jeu d'entraînement
plot_decision_boundary(X_train, y_train, model)
```



Définition

Les données sont **linéairement séparables** lorsque deux classes peuvent être parfaitement séparées par une **seule frontière linéaire**, comme une **droite** dans un **espace bidimensionnel** ou un **hyperplan en dimension supérieure**.

Frontière de décision simple



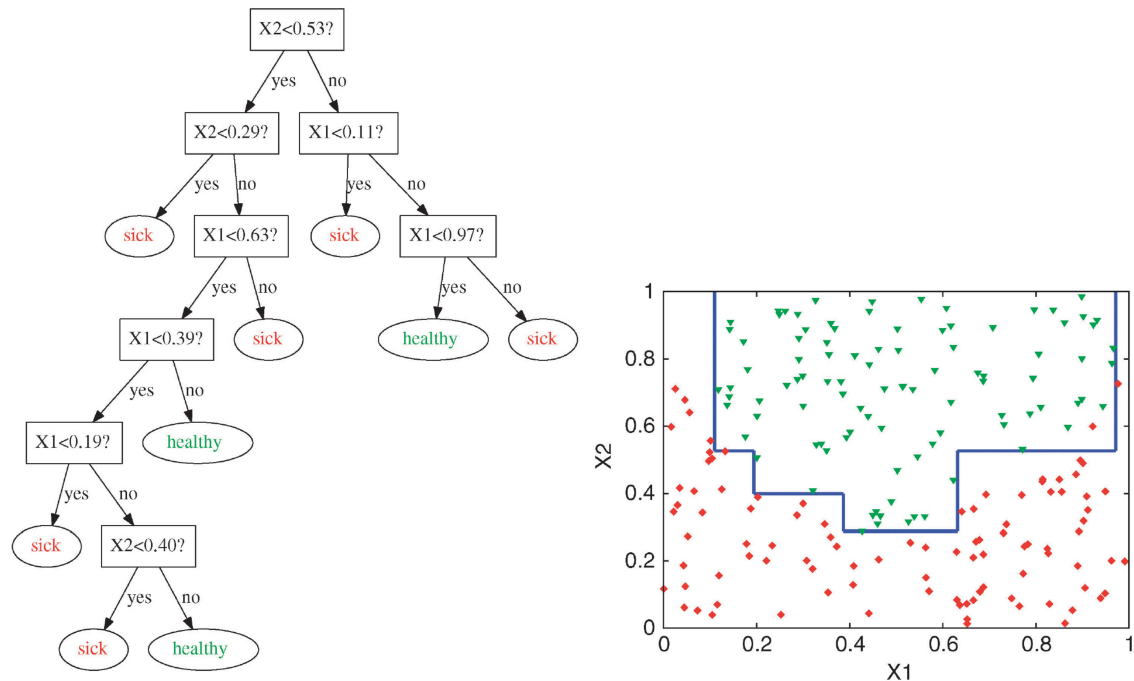
(a) données d'entraînement, (b) courbe quadratique et (c) fonction linéaire.

Attribution : (Geurts et al. 2009)

Le tableau de gauche présente des données d'entraînement pour une tâche hypothétique de **classification binaire** en médecine. Les deux attributs, X_1 et X_2 , servent à prédire la cible y , qui prend deux valeurs : *malade* ou *en santé*. On peut imaginer que X_1 et X_2 représentent la pression artérielle et la fréquence cardiaque, ou encore les taux de cholestérol et de glucose.

La **régression logistique** (c) utilise une **frontière de décision linéaire**. Ici, cette frontière est une droite. Son emploi entraîne plusieurs erreurs de classification : des points rouges au-dessus de la droite, qui devraient être classés « malades », sont prédits « en santé », et inversement pour certains points verts sous la droite.

Frontière de décision complexe



Les **arbres de décision** peuvent produire des frontières de décision **irrégulières** et **non linéaires**.

Attribution : *ibidem*.

Assurez-vous de comprendre la correspondance entre les huit règles de décision de l'arbre et les neuf segments de droite du diagramme de dispersion.

Définition (révisée)

Une **frontière de décision** est une hypersurface qui divise l'espace des attributs en **régions** correspondant à **différentes étiquettes de classes**.

Arbre de décision (suite)

Construire un arbre de décision

- Comment **construire** (**apprendre**) un arbre de décision ?

- Certains arbres sont-ils « **meilleurs** » que d'autres ?
- Est-il possible de construire efficacement un **arbre de décision optimal** ?

(Hyafil et Rivest 1976)

Optimalité

- Soit $X = \{x_1, \dots, x_n\}$ un ensemble fini d'**objets**.
- Soit $\mathcal{T} = \{T_1, \dots, T_t\}$ un ensemble fini de **tests**.
- Pour chaque objet et chaque test :
 - $T_i(x_j)$ est **vrai** ou **faux**.
- Un arbre **optimal** identifie complètement tous les objets de X en employant un nombre **minimal** de tests.

(Hyafil et Rivest 1976)

Hyafil et Rivest (1976) ont montré que la construction d'arbres de décision binaires optimaux est un problème NP-complet.

Construire un arbre de décision

- **Construction itérative** : commencer par un **arbre vide**, puis ajouter progressivement des **nœuds** à partir du **jeu d'entraînement**, jusqu'à ce que les exemples soient **complètement classés** ou qu'un autre critère d'arrêt, comme la **profondeur maximale**, soit atteint.

L'**apprentissage** est le processus qui construit l'arbre à partir des données d'entraînement.

Construire un arbre de décision

- **Construction du premier nœud** :
 - pour établir la racine, évaluer chacun des D attributs;
 - pour chaque attribut, évaluer plusieurs seuils dérivés des valeurs observées dans le jeu d'entraînement.

Construire un arbre de décision

- Pour un **attribut numérique**, l'algorithme considère les **points de division possibles** (seuils) dans l'étendue de l'attribut.
- Ces points sont généralement les **milieux** entre deux valeurs uniques consécutives de l'attribut, après les avoir **triées**.

À quel point un nœud est-il mélangé ?

- Un **nœud pur** contient des exemples d'une seule classe.
- Un **nœud mélangé** contient des exemples de plusieurs classes.
- Une division utile produit des enfants **moins mélangés** que leur parent.

Pour le nœud i , soit $p_{i,k}$ la proportion d'exemples de la classe k .

Il nous faut un score numérique pour comparer les divisions candidates. Ce score devrait être nul pour un nœud pur et augmenter à mesure que les proportions des classes s'équilibrent.

Entropie

L'**entropie** du nœud i est

$$H_i = - \sum_{k=1}^K p_{i,k} \log_2 p_{i,k}.$$

- $H_i = 0$ lorsque le nœud est **pur**.
- H_i augmente lorsque les classes sont représentées de façon **plus équilibrée**.
- Une valeur **plus petite** est donc préférable.

Nous utilisons la convention $0 \log_2 0 = 0$.

Il n'est pas nécessaire de connaître la théorie de l'information ici. Dans ce contexte, l'entropie est simplement une mesure numérique de l'incertitude qui subsiste au sujet de l'étiquette de classe dans un nœud.

La base du logarithme change l'unité et redimensionne toutes les valeurs d'entropie, mais elle ne change pas la division candidate qui obtient le plus petit score. La base 2 donne l'unité intuitive du *bit*.

Entropie binaire

Si p est la proportion de manchots Gentoo, alors

$$H(p) = -p \log_2 p - (1 - p) \log_2 (1 - p).$$

```
In [4]: import matplotlib.pyplot as plt
import numpy as np

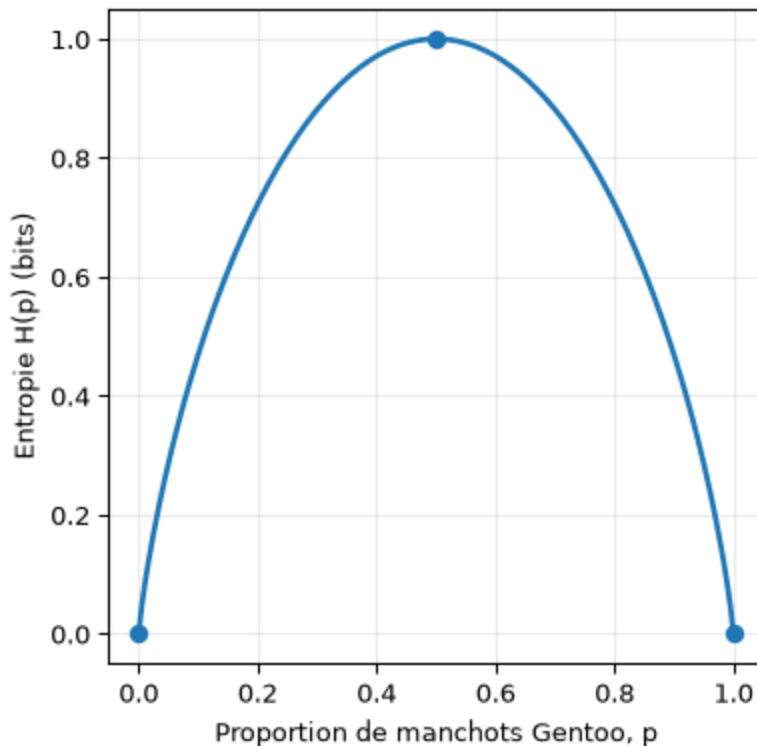
p = np.linspace(0, 1, 201)
h = np.zeros_like(p)
inside = (p > 0) & (p < 1)
h[inside] = -(
```

```

    p[inside] * np.log2(p[inside])
    + (1 - p[inside]) * np.log2(1 - p[inside])
)

fig, ax = plt.subplots(
    figsize=(4, 4),
    constrained_layout=True,
)
ax.plot(p, h, linewidth=2)
ax.scatter([0, 0.5, 1], [0, 1, 0], zorder=3)
ax.set_box_aspect(1)
ax.set(xlabel="Proportion de manchots Gentoo, p",
       ylabel="Entropie H(p) (bits)", ylim=(-0.05, 1.05))
ax.grid(alpha=0.25)
plt.show()

```



L'entropie est maximale lorsque les deux classes sont **équiprobables**.

À $p = 0$ ou $p = 1$, la classe est certaine et l'entropie est nulle. À $p = 0,5$, aucune classe n'est plus probable et l'entropie atteint son maximum d'un bit.

Cette courbe donne une interprétation intuitive du score d'un nœud sans exiger une étude plus générale de la théorie de l'information.

Entropie en Python

```

In [5]: import numpy as np

def class_probabilities(y, classes):
    """Retourner la proportion d'exemples de chaque classe."""

```

```

return np.array([np.mean(y == label) for label in classes])

def entropy(y, classes):
    """Mesurer le mélange des classes; zéro correspond à un nœud pur."""
    probabilities = class_probabilities(y, classes)
    probabilities = probabilities[probabilities > 0]
    if len(probabilities) == 1:
        return 0.0
    return float(-np.sum(probabilities * np.log2(probabilities)))

```

Cette implémentation utilise directement les étiquettes de classes. Les probabilités nulles sont retirées avant l'application du logarithme, ce qui met en œuvre la convention $0 \log_2 0 = 0$.

La fonction accepte deux classes ou plus. Le tableau `classes` fixe l'ordre des probabilités retournées et sera ensuite conservé par le classificateur.

Évaluer une division

Une division utilisant l'**attribut** j et le **seuil** t produit deux enfants :

$$X_{\text{left}} = \{x : x^{(j)} \leq t\}, \quad X_{\text{right}} = \{x : x^{(j)} > t\}.$$

Son score est l'**entropie pondérée** :

$$J(j, t) = \frac{N_{\text{left}}}{N_{\text{parent}}} H_{\text{left}} + \frac{N_{\text{right}}}{N_{\text{parent}}} H_{\text{right}}.$$

Choisir l'attribut et le seuil qui **minimisent** $J(j, t)$.

Plus tard, l'**entropie croisée** utilisera la même idée logarithmique pour évaluer les probabilités prédites par un modèle.

Les poids sont importants. Isoler un seul exemple pur ne devrait pas compenser le fait de laisser presque tous les exemples dans un enfant très mélangé. L'influence de chaque enfant est donc proportionnelle au nombre d'exemples qu'il contient.

Comme le parent est le même pour toutes les divisions candidates, nous pouvons comparer directement l'entropie pondérée de leurs enfants.

Trois divisions candidates

Supposons que le parent contienne **5 Gentoo** et **5 non-Gentoo**.

Candidate	Enfant gauche	H_{gauche}	Enfant droit	H_{droit}
Désordonnée	3 G, 2 non-G	0,971	2 G, 3 non-G	0,971
Isole un exemple	1 G, 0 non-G	0	4 G, 5 non-G	0,991

Candidate	Enfant gauche	H_{gauche}	Enfant droit	H_{droit}
Utile	4 G, 1 non-G	0,722	1 G, 4 non-G	0,722

L'enfant isolé est pur — mais il ne contient qu'un seul exemple.

Les trois candidates divisent le même parent, dont l'entropie vaut un bit. Dans la division désordonnée, les deux enfants demeurent presque aussi incertains que leur parent.

La deuxième candidate crée un enfant parfaitement pur, mais seulement en isolant un exemple. La plupart des exemples restent regroupés dans un enfant très mélangé. La division utile rend les deux enfants nettement plus purs.

Pourquoi pondérer les enfants ?

Candidate	Moyenne non pondérée	Entropie pondérée J
Désordonnée	$(0,971 + 0,971)/2$ $= 0,971$	$\frac{5}{10}(0,971)$ $+ \frac{5}{10}(0,971)$ $= 0,971$
Isole un exemple	$(0 + 0,991)/2$ $= \mathbf{0,496}$	$\frac{1}{10}(0)$ $+ \frac{9}{10}(0,991)$ $= 0,892$
Utile	$(0,722 + 0,722)/2$ $= 0,722$	$\frac{5}{10}(0,722)$ $+ \frac{5}{10}(0,722)$ $= \mathbf{0,722}$

Sans pondération, isoler un exemple pur semble préférable. Avec la pondération, le grand enfant mélangé reçoit l'influence qui lui revient.

La moyenne non pondérée traite un enfant contenant un seul exemple comme s'il était aussi important qu'un enfant en contenant neuf. Elle choisirait donc la candidate qui isole un exemple. L'entropie pondérée préfère correctement la division utile, qui améliore la classification de plusieurs exemples.

Les poids n'interdisent pas les divisions déséquilibrées. Ils garantissent que la contribution d'un enfant est proportionnelle au nombre d'exemples qu'il contient.

Évaluer une division en Python

```
In [6]: def weighted_entropy(y_left, y_right, classes):
        """Retourner l'entropie pondérée produite par une division."""
        n_left = len(y_left)
```

```

n_right = len(y_right)
n_parent = n_left + n_right

return (
    n_left / n_parent * entropy(y_left, classes)
    + n_right / n_parent * entropy(y_right, classes)
)

def candidate_thresholds(values):
    """Retourner les milieux entre les valeurs distinctes consécutives."""
    values = np.unique(values)
    return (values[:-1] + values[1:]) / 2

```

Pour un attribut numérique, les seuils candidats sont les milieux entre les valeurs distinctes consécutives. Tout seuil situé entre ces deux valeurs produirait la même partition; le milieu fournit un représentant déterministe.

Recherche gloutonne d'une division

Pour les exemples qui atteignent le nœud courant :

1. Pour chaque **attribut**, générer ses seuils candidats.
2. Pour chaque **seuil**, répartir les exemples entre les enfants gauche et droit.
3. Rejeter les divisions qui créent un enfant plus petit que `min_samples_leaf`.
4. Calculer l'**entropie pondérée**.
5. Conserver l'attribut et le seuil qui donnent le score le **plus faible**.

Cette recherche est gloutonne : elle choisit la meilleure division immédiate sans réexaminer les décisions précédentes. Elle ne garantit donc pas de trouver le meilleur arbre possible.

Cette implémentation didactique recalcule les masques et les proportions de classes pour chaque candidate. Dans un nœud contenant n exemples et D attributs, elle effectue jusqu'à $D(n - 1)$ évaluations, chacune exigeant un autre parcours des données. Le coût est donc approximativement $\mathcal{O}(Dn^2)$ dans ce nœud.

Recherche gloutonne en Python

```

In [7]: def find_best_split(X, y, classes, min_samples_leaf):
    best_split = None
    best_score = entropy(y, classes)
    n_candidates = 0

    for feature in range(X.shape[1]):
        for threshold in candidate_thresholds(X[:, feature]):
            go_left = X[:, feature] <= threshold
            n_left = np.sum(go_left)
            n_right = len(y) - n_left

```



```

        if min(n_left, n_right) < min_samples_leaf:
            continue

        n_candidates += 1
        score = weighted_entropy(
            y[go_left], y[~go_left], classes
        )
        if score < best_score:
            best_score = score
            best_split = (feature, float(threshold), go_left)

    return best_split, n_candidates

```

Le code suit directement les cinq étapes de la diapositive précédente. Le tableau booléen `go_left` indique les exemples affectés à l'enfant gauche; son complément sélectionne l'enfant droit.

`best_score` est initialisé avec l'entropie du nœud non divisé. La fonction retourne donc `None` si aucune candidate ne réduit le score.

Implémentation complète

La classe complète rassemble les opérations précédentes. La validation des entrées et la mise en forme du texte sont du code auxiliaire; elles ne font pas partie de l'algorithme central.

[Lire le tutoriel](#) · [Télécharger le notebook](#) · [Ouvrir dans Colab](#)

```

In [8]: from dataclasses import dataclass

@dataclass
class Node:
    probabilities: np.ndarray
    n_samples: int
    loss: float
    feature: int | None = None
    threshold: float | None = None
    left: "Node | None" = None
    right: "Node | None" = None

    @property
    def is_leaf(self):
        return self.feature is None

class SimpleDecisionTreeClassifier:
    """Un classificateur didactique avec une petite interface de type scikit

    def __init__(
        self,
        max_depth=None,
        min_samples_split=2,

```

```

        min_samples_leaf=1,
    ):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf

    def fit(self, X, y):
        X = np.asarray(X, dtype=float)
        y = np.asarray(y)
        self._validate_training_data(X, y)

        self.classes_ = np.unique(y)
        self.n_features_in_ = X.shape[1]
        self.n_candidate_splits_ = 0
        self.tree_ = self._grow_tree(X, y, depth=0)
        return self

    def _grow_tree(self, X, y, depth):
        probabilities = class_probabilities(y, self.classes_)
        node = Node(
            probabilities=probabilities,
            n_samples=len(y),
            loss=entropy(y, self.classes_),
        )

        depth_limit = self.max_depth is not None and depth >= self.max_depth
        pure_node = np.count_nonzero(probabilities) == 1
        too_small = len(y) < self.min_samples_split

        if pure_node or depth_limit or too_small:
            return node

        split, n_candidates = find_best_split(
            X, y, self.classes_, self.min_samples_leaf
        )
        self.n_candidate_splits_ += n_candidates
        if split is None:
            return node

        node.feature, node.threshold, go_left = split
        node.left = self._grow_tree(X[go_left], y[go_left], depth + 1)
        node.right = self._grow_tree(X[~go_left], y[~go_left], depth + 1)
        return node

    def _find_leaf(self, x):
        node = self.tree_
        while not node.is_leaf:
            if x[node.feature] <= node.threshold:
                node = node.left
            else:
                node = node.right
        return node

    def predict_proba(self, X):
        X = self._validate_prediction_data(X)
        return np.vstack([self._find_leaf(x).probabilities for x in X])

```

```

def predict(self, X):
    probabilities = self.predict_proba(X)
    return self.classes_[np.argmax(probabilities, axis=1)]

def score(self, X, y):
    return float(np.mean(self.predict(X) == np.asarray(y)))

def export_text(self, feature_names=None):
    if feature_names is None:
        feature_names = [f"x[{j}]" for j in range(self.n_features_in_)]
    if len(feature_names) != self.n_features_in_:
        raise ValueError("feature_names doit correspondre au nombre d'at

lines = []

def visit(node, indent):
    if node.is_leaf:
        prediction = self.classes_[np.argmax(node.probabilities)]
        if isinstance(prediction, np.generic):
            prediction = prediction.item()
        probabilities = np.round(node.probabilities, 3)
        lines.append(
            f"{indent}prédire {prediction!r} "
            f"(p={probabilities}, n={node.n_samples})"
        )
        return

    name = feature_names[node.feature]
    lines.append(f"{indent}si {name} <= {node.threshold:.3f} :")
    visit(node.left, indent + "    ")
    lines.append(f"{indent}sinon :")
    visit(node.right, indent + "    ")

visit(self.tree_, "")
return "\n".join(lines)

def _validate_training_data(self, X, y):
    if X.ndim != 2 or y.ndim != 1 or len(X) != len(y):
        raise ValueError("X doit être 2D et y doit être 1D, avec le même
    if len(y) == 0:
        raise ValueError("le jeu d'entraînement ne peut pas être vide")
    if not np.isfinite(X).all():
        raise ValueError("les valeurs manquantes ou non finies ne sont p
    if self.max_depth is not None and self.max_depth < 0:
        raise ValueError("max_depth doit être positif ou nul, ou None")
    if self.min_samples_split < 2:
        raise ValueError("min_samples_split doit être au moins 2")
    if self.min_samples_leaf < 1:
        raise ValueError("min_samples_leaf doit être au moins 1")

def _validate_prediction_data(self, X):
    X = np.asarray(X, dtype=float)
    if X.ndim == 1:
        X = X.reshape(1, -1)
    if X.ndim != 2 or X.shape[1] != self.n_features_in_:

```

```

        raise ValueError("X n'a pas le bon nombre d'attributs")
    if not np.isfinite(X).all():
        raise ValueError("les valeurs manquantes ou non finies ne sont p
    return X

```

Le code affiché par défaut dans les diapositives précédentes constitue le cœur conceptuel. Les fonctions de validation défensive permettent au notebook de produire des messages clairs lorsqu'il reçoit des entrées non prises en charge, mais il n'est pas nécessaire de les présenter en classe.

L'implémentation accepte les attributs numériques, les divisions binaires par seuil et deux classes ou plus. Elle omet volontairement les valeurs manquantes, les attributs catégoriels, la pondération des exemples, la recherche optimisée des divisions et l'élagage.

Entraînement sur les manchots de Palmer

```

In [9]: from palmerpenguins import load_penguins
        from sklearn.model_selection import train_test_split

        feature_names = ["bill_depth_mm", "body_mass_g"]
        penguins_tree = load_penguins()
        penguins_tree = penguins_tree[
            feature_names + ["species"]
        ].dropna().copy()

        X = penguins_tree[feature_names].to_numpy()
        y = np.where(
            penguins_tree["species"].to_numpy() == "Gentoo",
            "Gentoo",
            "Pas Gentoo",
        )
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.2, random_state=42, stratify=y
        )

        model = SimpleDecisionTreeClassifier(max_depth=2, min_samples_leaf=5)
        model.fit(X_train, y_train)

```

<__main__.SimpleDecisionTreeClassifier at 0x11b8237a0>

L'implémentation elle-même accepte plusieurs classes, mais cet exemple conserve la question binaire du cours : Gentoo ou non-Gentoo. L'emploi de seulement deux attributs nous permet aussi de représenter les régions de décision apprises.

`max_depth=2` limite le modèle à deux décisions sur tout chemin.

`min_samples_leaf=5` empêche une division de créer une feuille contenant moins de cinq exemples d'entraînement.

Qu'a appris l'arbre ?

```
In [10]: print(f"Exactitude sur le jeu de test : {model.score(X_test, y_test):.3f}")
print(f"Divisions candidates évaluées : {model.n_candidate_splits}")
print(f"Ordre des classes : {model.classes_}")
print("\nRègles apprises :\n")
print(model.export_text(feature_names))
```

```
Exactitude sur le jeu de test : 1.000
Divisions candidates évaluées : 312
Ordre des classes : ['Gentoo' 'Pas Gentoo']
```

Règles apprises :

```
si bill_depth_mm <= 16.450 :
    si body_mass_g <= 3750.000 :
        prédire 'Pas Gentoo' (p=[0. 1.], n=5)
    sinon :
        prédire 'Gentoo' (p=[1. 0.], n=92)
sinon :
    si body_mass_g <= 5100.000 :
        prédire 'Pas Gentoo' (p=[0. 1.], n=170)
    sinon :
        prédire 'Gentoo' (p=[1. 0.], n=6)
```

Le vecteur de probabilités de chaque feuille suit l'ordre affiché par `model.classes_`. Avec cette division des données, le modèle de profondeur deux classe correctement tous les exemples de test. Il s'agit du résultat d'un petit jeu de données et d'une seule division entraînement-test, et non d'une garantie générale.

Le nombre de candidates évaluées rend visible le coût de la recherche exhaustive. Les bibliothèques optimisées peuvent apprendre le même type de modèle beaucoup plus efficacement.

Régions de décision

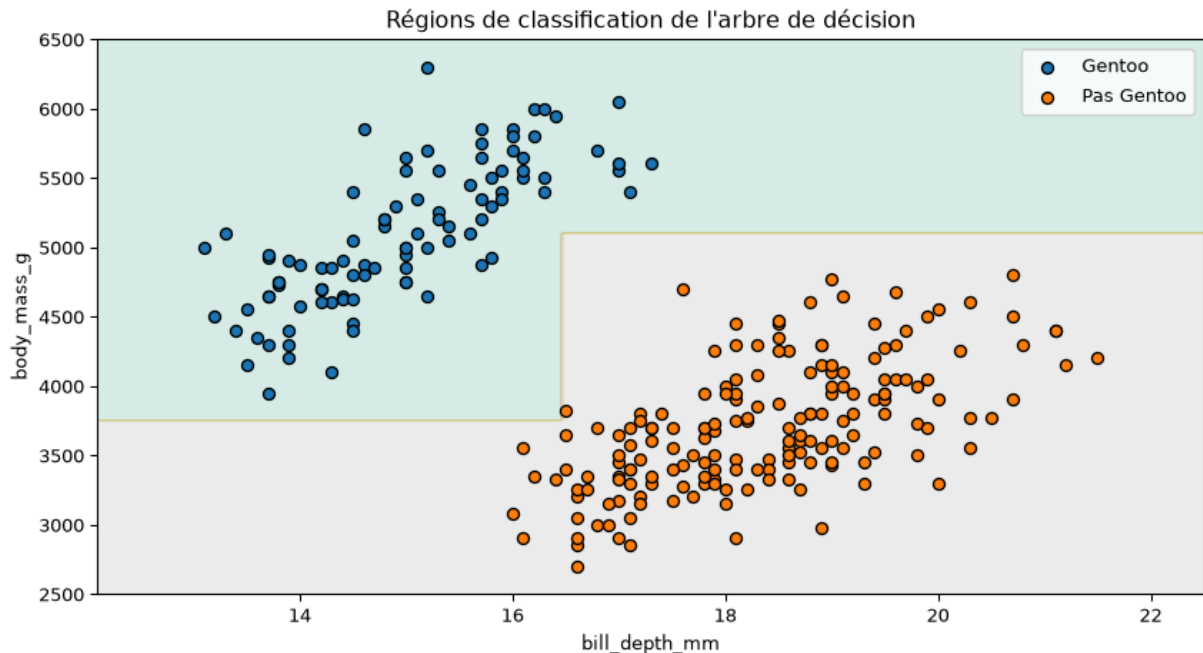
```
In [11]: def plot_tree_boundary(X, y, model, feature_names):
    x0 = np.linspace(X[:, 0].min() - 1, X[:, 0].max() + 1, 300)
    x1 = np.linspace(X[:, 1].min() - 200, X[:, 1].max() + 200, 300)
    xx0, xx1 = np.meshgrid(x0, x1)

    grid = np.column_stack([xx0.ravel(), xx1.ravel()])
    regions = np.argmax(
        model.predict_proba(grid), axis=1
    ).reshape(xx0.shape)

    plt.figure(figsize=(9, 5))
    plt.contourf(xx0, xx1, regions, alpha=0.25, cmap="Set2")
    for label in model.classes_:
        selected = y == label
        plt.scatter(
            X[selected, 0], X[selected, 1],
            edgecolor="black", label=label
        )
```

```
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.title("Régions de classification de l'arbre de décision")
plt.legend()
plt.tight_layout()
plt.show()
```

```
plot_tree_boundary(X_train, y_train, model, feature_names)
```



Chaque décision compare un attribut à un seuil. En deux dimensions, chaque décision correspond donc à une droite verticale ou horizontale. La combinaison de plusieurs décisions produit des régions de classification rectangulaires et une frontière de décision globale non linéaire.

Exemple complet

https://www.youtube.com/watch?v=_L39rN6gz7Y

Decision and Classification Trees, Clearly Explained!!!, (18 m 7s) [StatQuest](#), 2021-04-26

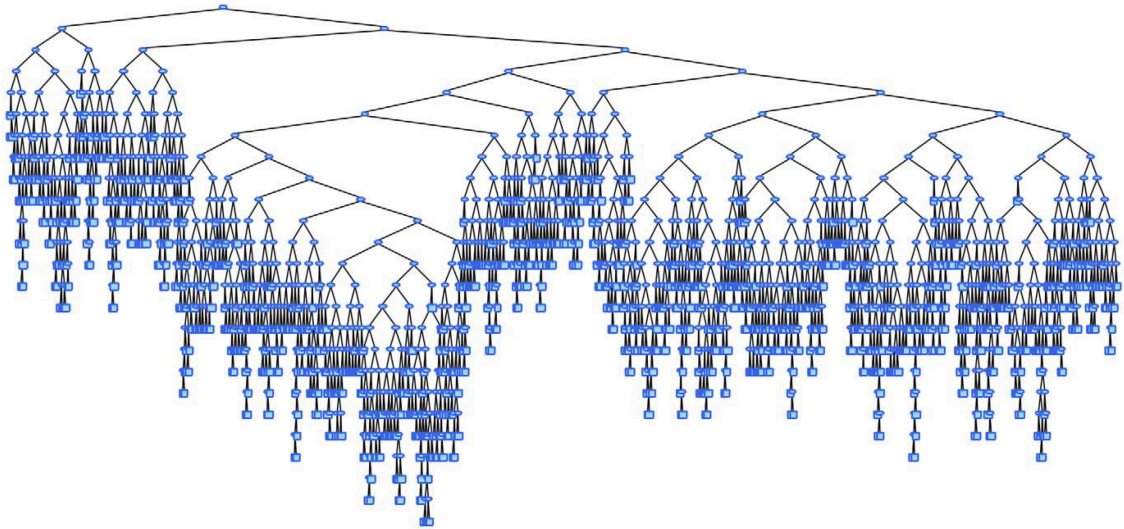
Critères d'arrêt

- Tous les exemples d'un nœud appartiennent à la **même classe**.
- La profondeur de l'arbre dépasserait **max_depth**.
- Le nœud contient moins de **min_samples_split** exemples.
- Aucune division ne réduit suffisamment l'impureté (**min_impurity_decrease**).
- Consulter la documentation pour d'autres critères.

Limites

- Peut produire des **arbres de grande taille**
 - Difficiles à **interpréter**
 - **Surapprentissage**
- **Algorithme glouton**, sans garantie de trouver l'arbre optimal (Hyafil et Rivest 1976)
- De **petites modifications** du jeu de données peuvent produire des **arbres très différents**

Arbres de grande taille



(Stiglic et al. 2012)

Sensibilité aux petites modifications

```
In [12]: from sklearn import tree
from sklearn.metrics import classification_report, accuracy_score

# Chargement du jeu de données

X, y = load_penguins(return_X_y = True)

target_names = ['Adelie', 'Chinstrap', 'Gentoo']

# Division en jeux d'entraînement et de test

for seed in (4, 7, 90, 96, 99, 2):

    print(f'Graine : {seed}')

    # Créer de nouveaux jeux selon une graine différente

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=seed, stratify=y
    )
```

```

# Créer un nouveau classificateur

clf = tree.DecisionTreeClassifier(
    criterion="entropy", random_state=seed
)

# Entraînement

clf.fit(X_train, y_train)

# Effectuer les prédictions

y_pred = clf.predict(X_test)

# Tracer l'arbre

tree.plot_tree(clf,
                feature_names = X.columns,
                class_names = target_names,
                filled = True)
plt.show()

# Évaluer le modèle

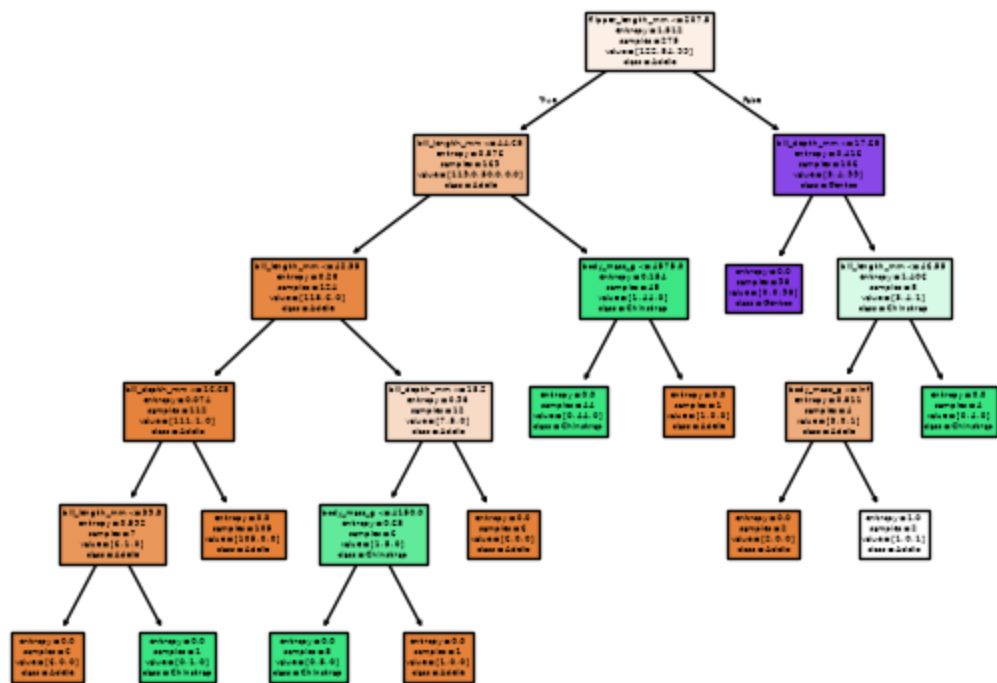
accuracy = accuracy_score(y_test, y_pred)

report = classification_report(y_test, y_pred, target_names=target_names)

print(f'Exactitude : {accuracy:.2f}')
print('Rapport de classification :')
print(report)

```

Graine : 4



Exactitude : 0.94

Rapport de classification :

	precision	recall	f1-score	support
Adelie	0.93	0.93	0.93	30
Chinstrap	0.88	1.00	0.93	14
Gentoo	1.00	0.92	0.96	25
accuracy			0.94	69
macro avg	0.94	0.95	0.94	69
weighted avg	0.95	0.94	0.94	69

Graine : 90



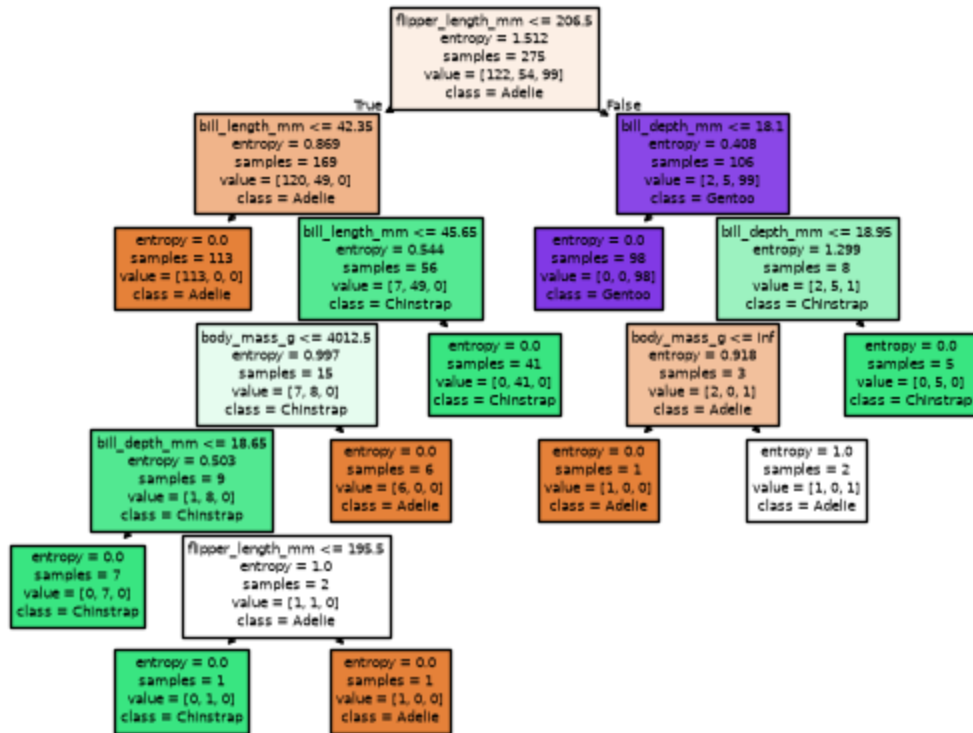


Exactitude : 0.96

Rapport de classification :

	precision	recall	f1-score	support
Adelie	0.97	0.93	0.95	30
Chinstrap	0.88	1.00	0.93	14
Gentoo	1.00	0.96	0.98	25
accuracy			0.96	69
macro avg	0.95	0.96	0.95	69
weighted avg	0.96	0.96	0.96	69

Graine : 99



Exactitude : 0.93

Rapport de classification :

	precision	recall	f1-score	support
Adelie	0.96	0.90	0.93	30
Chinstrap	0.81	0.93	0.87	14
Gentoo	0.96	0.96	0.96	25
accuracy			0.93	69
macro avg	0.91	0.93	0.92	69
weighted avg	0.93	0.93	0.93	69

Graine : 2



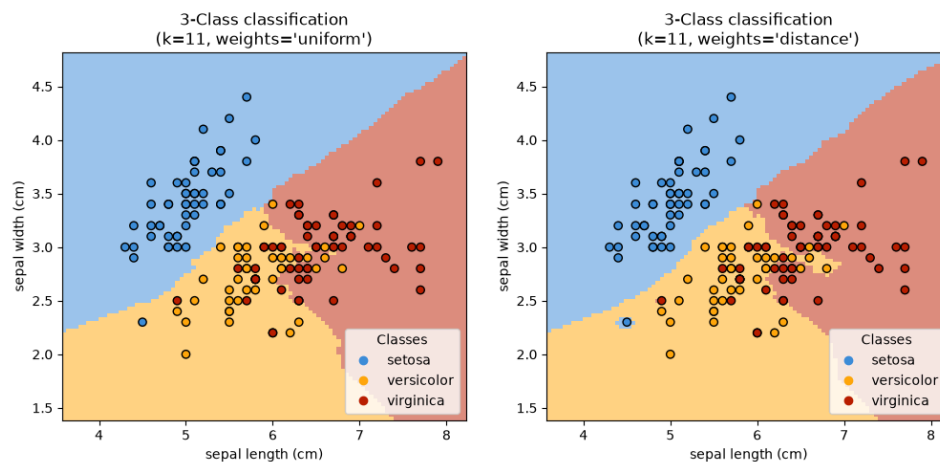
Exactitude : 0.97

Rapport de classification :

	precision	recall	f1-score	support
Adelie	1.00	0.93	0.97	30
Chinstrap	0.88	1.00	0.93	14
Gentoo	1.00	1.00	1.00	25
accuracy			0.97	69
macro avg	0.96	0.98	0.97	69
weighted avg	0.97	0.97	0.97	69

KNN

k plus proches voisins (KNN)



- **Fondé sur les instances** : les exemples d'entraînement constituent le modèle.
- **Non paramétrique** : le modèle ne possède pas un nombre fixe de paramètres appris.
- La prédiction suppose que les exemples voisins tendent à avoir des cibles semblables.

Attribution : [Nearest Neighbors Classification](#), scikit-learn.

KNN offre un contraste utile avec les arbres de décision. Un arbre construit une hiérarchie de règles pendant l'apprentissage. KNN ne construit aucune règle et n'estime aucun coefficient : il conserve les exemples et reporte la majeure partie du calcul au moment où une prédiction est demandée.

Non paramétrique ne signifie pas sans hypothèse. KNN suppose que la distance choisie donne un sens à la proximité et que des exemples voisins ont tendance à avoir des cibles semblables.

Apprentissage : mémoriser les données

```
def fit(self, X, y):
    self.X_train_ = X.copy()
    self.y_train_ = y.copy()
    return self
```

Aucun arbre n'est construit et aucun coefficient n'est estimé.

KNN est souvent qualifié d'algorithme d'*apprentissage paresseux*. L'étape d'apprentissage du classificateur se réduit à mémoriser le jeu d'entraînement. En conserver une copie demande $\mathcal{O}(ND)$ mémoire pour N exemples comportant D attributs.

La préparation des données et le choix des hyperparamètres peuvent néanmoins comprendre un apprentissage. Par exemple, une étape de standardisation estime des moyennes et des écarts-types à partir des données d'entraînement. Le point important est que le classificateur KNN lui-même n'ajuste pas de paramètres.

Trouver les plus proches voisins

Pour une requête x et un exemple d'entraînement x_i , la distance euclidienne est

$$d(x, x_i) = \sqrt{\sum_{j=1}^D \left(x^{(j)} - x_i^{(j)} \right)^2}.$$

```
In [13]: import numpy as np
```

```
def nearest_neighbors(X_train, x, n_neighbors):
    distances = np.sqrt(np.sum((X_train - x) ** 2, axis=1))
    indices = np.argsort(distances, kind="stable")[:n_neighbors]
    return indices, distances[indices]
```

L'implémentation directe avec tri complet coûte $\mathcal{O}(ND + N \log N)$ par requête.

Le calcul de toutes les distances coûte $\mathcal{O}(ND)$ et leur tri coûte $\mathcal{O}(N \log N)$. Une implémentation de production peut sélectionner les éléments les plus proches sans effectuer un tri complet et employer parfois des structures comme un arbre kd ou un arbre métrique (*ball tree*).

Le tri stable donne un comportement déterministe lorsque deux exemples se trouvent à la même distance. Il s'agit d'un choix d'implémentation simple et non d'un élément essentiel de KNN.

Voter et calculer une moyenne

Après avoir sélectionné les k exemples les plus proches :

- **Classification** : chaque voisin vote pour sa classe.
- **Régression** : calculer la moyenne des cibles numériques des voisins.
- **Poids uniformes** : chaque voisin exerce la même influence.
- **Poids selon la distance** : les voisins plus proches reçoivent le poids $w_i = 1/d_i$.

Pour la régression pondérée par la distance,

$$\hat{y}(x) = \frac{\sum_{i=1}^k w_i y_i}{\sum_{i=1}^k w_i}.$$

La recherche du voisinage est identique pour la classification et la régression; seule l'étape d'agrégation change. La classification additionne les votes par classe, tandis que la régression calcule la moyenne de valeurs numériques.

La pondération par la distance donne plus d'influence aux exemples les plus proches. Si une requête correspond exactement à un ou plusieurs exemples d'entraînement, l'implémentation laisse seulement ces correspondances exactes voter. Cette règle évite la division par zéro et possède une interprétation intuitive.

Vote en Python

```
In [14]: def voting_weights(distances, mode):
    if mode == "uniform":
        return np.ones(len(distances))

    exact_matches = distances == 0
```



```

    if np.any(exact_matches):
        return exact_matches.astype(float)

    return 1 / distances

def vote_probabilities(labels, weights, classes):
    scores = np.array([
        np.sum(weights[labels == label])
        for label in classes
    ])
    return scores / np.sum(scores)

```

Les comparaisons `labels == label` sélectionnent les voisins appartenant à chaque classe. Leurs poids sont additionnés, puis normalisés afin que la somme des scores obtenus soit égale à un.

Avec des poids uniformes, il s'agit simplement des proportions des classes dans le voisinage. Avec des poids selon la distance, il s'agit de votes pondérés normalisés.

Implémentation complète

[Lire le tutoriel](#) · [Télécharger le notebook](#) · [Ouvrir dans Colab](#)

```

In [15]: class SimpleKNeighborsClassifier:
    """Un classificateur didactique avec une petite interface de type scikit

    def __init__(self, n_neighbors=5, weights="uniform"):
        self.n_neighbors = n_neighbors
        self.weights = weights

    def fit(self, X, y):
        X = np.asarray(X, dtype=float)
        y = np.asarray(y)
        self._validate_training_data(X, y)

        self.X_train_ = X.copy()
        self.y_train_ = y.copy()
        self.classes_ = np.unique(y)
        self.n_features_in_ = X.shape[1]
        return self

    def _predict_proba_one(self, x):
        indices, distances = nearest_neighbors(
            self.X_train_, x, self.n_neighbors
        )
        labels = self.y_train_[indices]
        weights = voting_weights(distances, self.weights)
        return vote_probabilities(labels, weights, self.classes_)

    def predict_proba(self, X):
        X = self._validate_prediction_data(X)
        return np.vstack([self._predict_proba_one(x) for x in X])

```

```

def predict(self, X):
    probabilities = self.predict_proba(X)
    return self.classes_[np.argmax(probabilities, axis=1)]

def score(self, X, y):
    return float(np.mean(self.predict(X) == np.asarray(y)))

def _validate_training_data(self, X, y):
    if X.ndim != 2 or y.ndim != 1 or len(X) != len(y):
        raise ValueError(
            "X doit être 2D et y doit être 1D, avec le même nombre de li
        )
    if len(y) == 0:
        raise ValueError("le jeu d'entraînement ne peut pas être vide")
    if not np.isfinite(X).all():
        raise ValueError(
            "les valeurs manquantes ou non finies ne sont pas prises en
        )
    if not isinstance(self.n_neighbors, (int, np.integer)):
        raise ValueError("n_neighbors doit être un entier")
    if not 1 <= self.n_neighbors <= len(y):
        raise ValueError("n_neighbors doit être compris entre 1 et len(y)
    if self.weights not in {"uniform", "distance"}:
        raise ValueError("weights doit être 'uniform' ou 'distance'")

def _validate_prediction_data(self, X):
    if not hasattr(self, "X_train_"):
        raise ValueError("appelez fit avant d'effectuer des prédictions")
    X = np.asarray(X, dtype=float)
    if X.ndim == 1:
        X = X.reshape(1, -1)
    if X.ndim != 2 or X.shape[1] != self.n_features_in_:
        raise ValueError("X n'a pas le bon nombre d'attributs")
    if not np.isfinite(X).all():
        raise ValueError(
            "les valeurs manquantes ou non finies ne sont pas prises en
        )
    return X

```

Lorsque les votes pondérés sont à égalité, `np.argmax` choisit la première classe du tableau trié `classes_`. Les logiciels de production peuvent offrir d'autres règles pour départager les égalités.

KNN sur les manchots de Palmer

```

In [16]: from palmerpenguins import load_penguins
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

knn_features = ["bill_length_mm", "bill_depth_mm"]
penguins_knn = load_penguins()
penguins_knn = penguins_knn[
    knn_features + ["species"]

```

```

].dropna().copy()

X_knn = penguins_knn[knn_features].to_numpy()
y_knn = penguins_knn["species"].to_numpy()
X_knn_train, X_knn_test, y_knn_train, y_knn_test = train_test_split(
    X_knn, y_knn, test_size=0.2, random_state=42, stratify=y_knn
)

knn_scaler = StandardScaler()
X_knn_train_scaled = knn_scaler.fit_transform(X_knn_train)
X_knn_test_scaled = knn_scaler.transform(X_knn_test)

for mode in ("uniform", "distance"):
    knn = SimpleKNeighborsClassifier(n_neighbors=5, weights=mode)
    knn.fit(X_knn_train_scaled, y_knn_train)
    print(f"weights='{mode}' : {knn.score(X_knn_test_scaled, y_knn_test):.3f}

```

```

weights='uniform' : 0.957
weights='distance' : 0.957

```

Cet exemple conserve les trois espèces de manchots. La longueur et la profondeur du bec permettent de mieux distinguer les frontières de décision que le problème précédent, Gentoo ou non-Gentoo, qui était presque séparable.

KNN est sensible à l'échelle des attributs, car ses prédictions dépendent directement de la distance. Le transformateur est ajusté uniquement sur les données d'entraînement, ce qui empêche l'information du jeu de test d'influencer le prétraitement.

Frontières de décision de KNN

```

In [17]: import matplotlib.pyplot as plt
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline

configurations = [
    (1, "uniform"),
    (5, "uniform"),
    (25, "uniform"),
    (5, "distance"),
]

weight_names = {"uniform": "uniforme", "distance": "distance"}
classes, y_encoded = np.unique(y_knn_train, return_inverse=True)

fig, axes = plt.subplots(2, 2, figsize=(11, 7), sharex=True, sharey=True)
for ax, (n_neighbors, weights) in zip(axes.ravel(), configurations):
    sklearn_knn = make_pipeline(
        StandardScaler(),
        KNeighborsClassifier(n_neighbors=n_neighbors, weights=weights),
    )
    sklearn_knn.fit(X_knn_train, y_encoded)
    DecisionBoundaryDisplay.from_estimator(
        sklearn_knn,

```

```

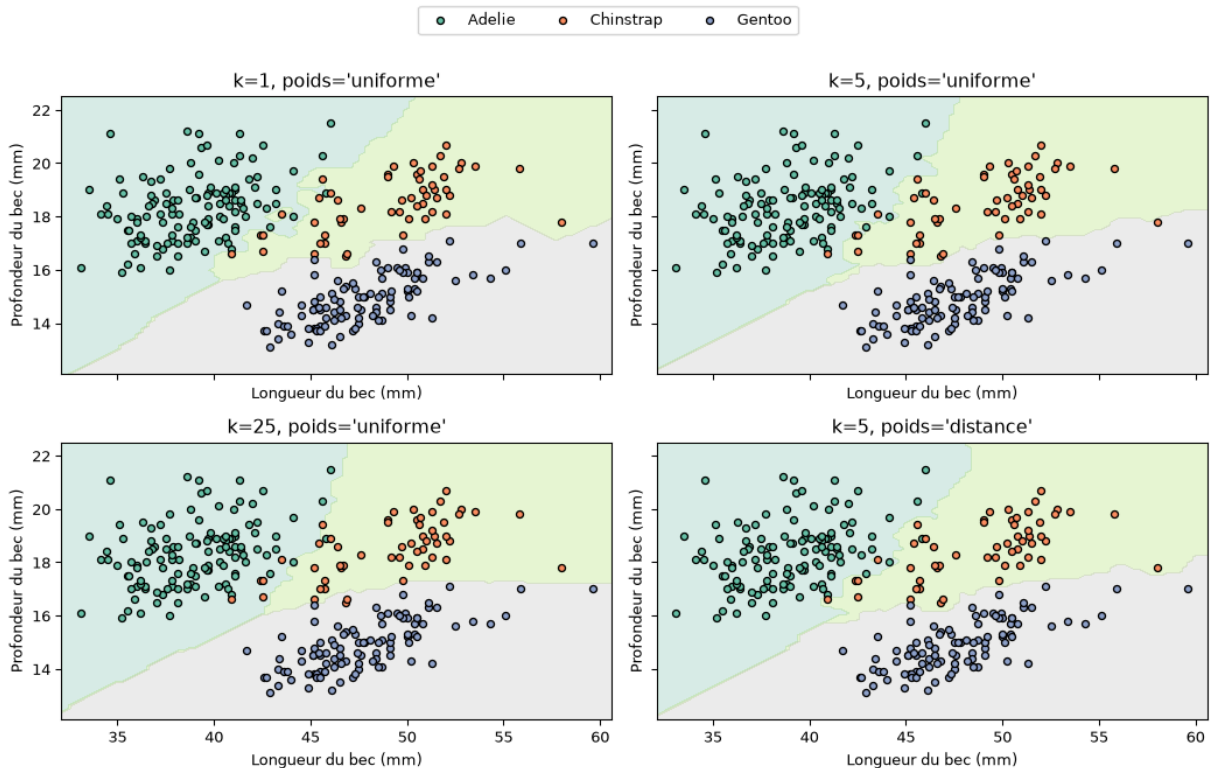
X_knn_train,
response_method="predict",
multiclass_colors="Set2",
alpha=0.25,
ax=ax,
)

for label, name in enumerate(classes):
    selected = y_encoded == label
    ax.scatter(
        X_knn_train[selected, 0],
        X_knn_train[selected, 1],
        color=plt.get_cmap("Set2")(label),
        edgecolor="black",
        s=20,
        label=name,
    )

ax.set_title(f"k={n_neighbors}, poids='{weight_names[weights]}'")
ax.set_xlabel("Longueur du bec (mm)")
ax.set_ylabel("Profondeur du bec (mm)")

handles, labels = axes[0, 0].get_legend_handles_labels()
fig.legend(handles, labels, loc="upper center", ncols=len(classes))
fig.tight_layout(rect=(0, 0, 1, 0.93))
plt.show()

```



Avec $k = 1$, chaque observation d'entraînement contrôle une petite région autour d'elle. La frontière est donc irrégulière et sensible aux exemples individuels. Augmenter k lisse la frontière, car un voisinage plus grand doit s'accorder.

La pondération par la distance donne plus d'influence aux observations les plus proches. Avec $k = 5$, sa frontière peut donc conserver plus de détails locaux que le vote uniforme employant le même k .

Le diagramme utilise délibérément scikit-learn plutôt que d'alourdir l'implémentation didactique avec du code de visualisation. Le pipeline standardise les attributs à l'interne tout en conservant leurs unités originales sur les axes.

Limites de KNN

- La prédiction conserve et parcourt les données d'entraînement.
- Les distances sont sensibles à la **mise à l'échelle des attributs** et à la métrique choisie.
- Les voisinages deviennent moins informatifs dans les espaces de grande dimension.
- Un petit k peut être sensible au bruit; un grand k peut masquer la structure locale.
- Le déséquilibre des classes peut dominer le vote d'un voisinage.

La perte de contraste utile entre les distances dans les espaces de grande dimension est communément appelée la *malédiction de la dimensionnalité*. KNN peut aussi demander beaucoup de mémoire et de temps de prédiction, car son modèle effectif grandit avec le jeu d'entraînement.

En régression, les prédictions sont des moyennes des cibles observées dans le voisinage; elles n'extrapolent donc pas au-delà de l'étendue observée des cibles.

Exercices et ressources sur KNN

Expérimentez en modifiant k , la pondération des votes, la distance et la mise à l'échelle des attributs :

- [Nearest Neighbors Classification](#)
- [Nearest Neighbors Regression](#)
- [Importance of Feature Scaling](#)

Les exemples de scikit-learn peuvent être téléchargés sous forme de notebooks Jupyter. Vous pouvez également modifier le notebook exécutable produit à partir de ce cours, par exemple en changeant les quatre configurations de la cellule consacrée aux frontières de décision.

Résumé

- Le cours a présenté les **arbres de décision** et les **k plus proches voisins (KNN)**, et utilisé la **régression logistique** pour illustrer une frontière de décision linéaire.

- Nous avons construit un arbre de décision qui minimise gloutonnement l'entropie pondérée et un classificateur KNN qui **mémore, cherche et vote**.
- Les exemples des manchots de Palmer ont illustré comment différents algorithmes d'apprentissage produisent différentes **frontières de décision**.

Conclusion

Ressources

- [Plot the decision surface of decision trees trained on the iris dataset](#), `sklearn`
- [Decision trees](#), Jan Kirenz, professeur à la HdM Stuttgart
- [CS 320 Apr12-2021 \(Part 2\) - Decision Boundaries](#), Tyler Caraza-Harter, professeur à UW-Madison

Références

Battista, Katelyn, Liqun Diao, Karen A Patte, Joel A Dubin, et Scott T Leatherdale. 2023. « Examining the use of decision trees in population health surveillance research: an application to youth mental health survey data in the COMPASS study ». *Health Promotion and Chronic Disease Prevention in Canada* 43 (2): 73-86.

<https://doi.org/10.24095/hpcdp.43.2.03>.

Geurts, Pierre, Alexandre Irtuthum, et Louis Wehenkel. 2009. « Supervised learning with decision tree-based methods in computational and systems biology ». *Molecular bioSystems* 5 (12): 1593-605. <https://doi.org/10.1039/b907946g>.

Hyafil, Laurent, et Ronald L. Rivest. 1976. « Constructing Optimal Binary Decision Trees is NP-Complete ». *Inf. Process. Lett.* 5 (1): 15-17. [https://doi.org/10.1016/0020-0190\(76\)90095-8](https://doi.org/10.1016/0020-0190(76)90095-8).

Russell, Stuart, et Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4^e éd. Pearson. <http://aima.cs.berkeley.edu/>.

Stiglic, Gregor, Simon Kocbek, Igor Pernek, et Peter Kokol. 2012. « Comprehensive decision tree models in bioinformatics ». *PLoS ONE* 7 (3): e33812. <https://doi.org/10.1371/journal.pone.0033812>.

Prochain cours

- Entraîner un modèle linéaire

Marcel.Turcotte@uOttawa.ca

École de science informatique et de génie électrique (**SIGE**)

Université d'Ottawa