CYCLE 5

RESOLUTION DE PROBLEMES PAR UTILISATION DE L'INGENIERIE NUMERIQUE OU L'APPRENTISSAGE AUTOMATISE

TD1 - PSI

TD 1

ALGORITHMES D'INTELLIGENCE ARTIFICIELLE

L'objectif de ce TD est d'implémenter et de tester les algorithmes vus dans le cours en Python

1 PRELIMINAIRES

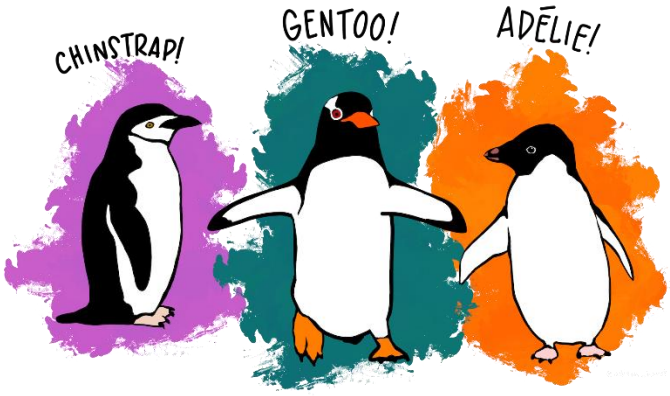
Dans cette partie pratique nous allons utiliser un jeu de données décrivant des manchots, le jeu Palmer Penguins.

Ces données sont mises à disposition par la base antarctique Palmer sous licence CC-0.

Le jeu de données contient des informations décrivant un certain nombre de manchots appartenant à trois espèces :

- Manchot d'Adélie (*Pygoscelis adeliae*), Adelie dans les données
- Manchot papou (*Pygoscelis papua*), Gentoo dans les données
- Manchot à jugulaire (*Pygoscelis antarcticus*), Chinstrap dans les données.

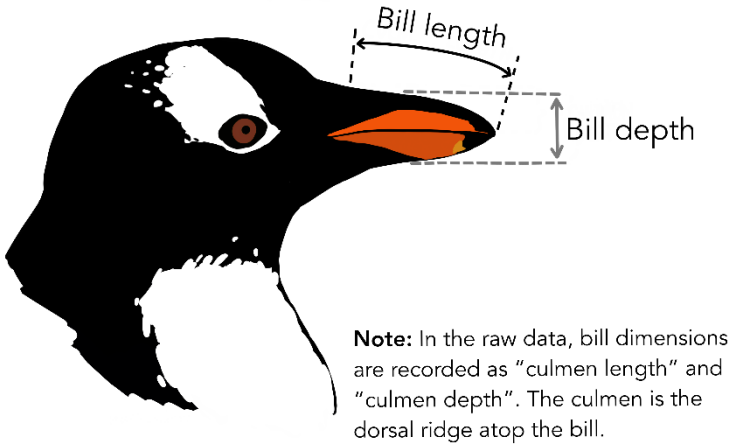
Les trois espèces de manchots: (illustration par @allison_horst)



Pour chacun des manchots, le jeu de données contient

- la longueur de son bec en mm (bill_length_mm)
- la hauteur de son bec en mm (bill_depth_mm)
- la longueur de ses palettes natatoires en mm (flipper_length_mm)
- son poids en g (body_mass_g)

Caractéristiques du bec : (illustration par @allison_horst)



1.1 Chargement des librairies

Numpy et matplotlib

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

Pandas

La librairie pandas nous permet de charger et manipuler les données.

```
import pandas as pd
```

1.2 Chargement des données

Les données sont disponibles au format csv ("comma-separated values") dans le fichier data/penguins.csv. Nous allons les charger grâce à la librairie pandas.

```
penguins = pd.read_csv("data/penguins.csv") #mettre votre chemin
```

Nous pouvons maintenant visualiser la table des données :

```
print(penguins)
```

Question : Combien d'échantillons comporte notre jeu de données ?

Réponse : La table de données contient 333 lignes, c'est donc le nombre de manchots i.e. d'échantillons. On peut aussi y accéder grâce à `penguins.shape`

1.3 Problème 1 : prédire le poids d'un manchot

Le premier problème que nous allons étudier dans cette session va être de prédire le poids d'un manchot à partir de la taille (longueur et hauteur) de son bec ainsi que de la longueur de ses nageoires.

Question : De quel type de problème s'agit-il ?

Visualisation :

Nous allons représenter ici le poids d'un manchot en fonction de chacune des trois variables :

```
fig = plt.figure(figsize=(15, 5))

# Figure 1 dans une grille (1 x 3)
ax = plt.subplot(1, 3, 1)
ax.scatter(penguins["bill_length_mm"], penguins["body_mass_g"])
plt.xlabel("Longueur du bec (mm)")
plt.ylabel("Poids (g)")

# Figure 2 dans une grille (1 x 3)
ax = plt.subplot(1, 3, 2)
ax.scatter(penguins["bill_depth_mm"], penguins["body_mass_g"])
plt.xlabel("Hauteur du bec (mm)")
plt.ylabel("Poids (g)")

# Figure 3 dans une grille (1 x 3)
```

```
ax = plt.subplot(1, 3, 3)
ax.scatter(penguins["flipper_length_mm"], penguins["body_mass_g"])
plt.xlabel("Longueur de la palette natatoire (mm)")
plt.ylabel("Poids (g)")
```

Extraction des données

Nous avons formulé nos algorithmes de *machine learning* de sorte à considérer une matrice de données X (autant de lignes que d'observations, autant de colonnes que de variables) et un vecteur d'étiquettes y (autant d'entrées que d'observations).

Question : Quelles sont les dimensions de X et y dans le cas du Problème 1 ?

(bill_length_mm, bill_depth_mm, flipper_length_mm).

Nous allons maintenant créer les arrays numpy correspondant à X et y.

```
X = np.array( penguins[["bill_length_mm", "bill_depth_mm", "flipper_length_mm"]])
y_regress = np.array( penguins["body_mass_g"])
```

Vérifions que nos array ont les bonnes dimensions :

```
print(X.shape)
print(y_regress.shape)
```

1.4 Problème 2 : prédire le sexe d'un manchot

Le deuxième problème que nous allons étudier dans cette session va être de prédire si un manchot est mâle ou femelle à partir de la taille (longueur et hauteur) de son bec ainsi que de la longueur de ses nageoires.

Question : De quel type de problème s'agit-il ?

```
plt.ion() #A enlever si l'affichage plante
fig = plt.figure(figsize=(15, 5))
coul={"male":"b", "female":"r"} #dico pour les couleurs

# Figure 1 dans une grille (1 x 3)
ax = plt.subplot(1, 3, 1)
for penguin_sex in ["male", "female"]:
    ax.scatter(penguins.loc[penguins["sex"] == penguin_sex]["bill_length_mm"],
               penguins.loc[penguins["sex"] == penguin_sex]["bill_depth_mm"],
               c=coul[penguin_sex],
               label=penguin_sex)
plt.xlabel("Longueur du bec (mm)")
plt.ylabel("Hauteur du bec (mm)")

# Figure 2 dans une grille (1 x 3)
ax = plt.subplot(1, 3, 2)
for penguin_sex in ["male", "female"]:
    ax.scatter(penguins.loc[penguins["sex"] == penguin_sex]["bill_length_mm"],
               penguins.loc[penguins["sex"] ==
penguin_sex]["flipper_length_mm"],
               c=coul[penguin_sex],
               label=penguin_sex)
plt.xlabel("Longueur du bec (mm)")
plt.ylabel("Longueur de la palette natatoire (mm)")

# Figure 3 dans une grille (1 x 3)
ax = plt.subplot(1, 3, 3)
for penguin_sex in ["male", "female"]:
    ax.scatter(penguins.loc[penguins["sex"] == penguin_sex]["bill_depth_mm"],
               penguins.loc[penguins["sex"] ==
penguin_sex]["flipper_length_mm"],
               c=coul[penguin_sex],
               label=penguin_sex)
plt.xlabel("Hauteur du bec (mm)")
plt.ylabel("Longueur de la palette natatoire (mm)")
plt.legend()
```

Extraction des données

Question : Quelles sont les dimensions de X et y dans le cas du Problème 2 ?
Nous allons maintenant créer les arrays numpy correspondant à X et y. En particulier, nous allons transformer les étiquettes female et male en 0 et 1, respectivement.

```
y_classif = pd.Categorical(penguins["sex"]).codes
```

Vérifions que nos array ont les bonnes dimensions :

```
print(y_classif.shape)
y_classif
```

2 REGRESSION LINEAIRE

2.1 Formalisme

Nous disposons donc d'un jeu d'entraînement de n observations en p dimensions, représentés par une matrice $X \in R^{n*p}$ (ici, $p=3$), et d'un vecteur d'étiquettes $y \in R^n$.
Le but ici d'une régression linéaire est d'apprendre une fonction linéaire :

$$f : x \in \mathbb{R}^P \rightarrow \beta_0 + \beta^T x$$

Où $\beta_0 \in \mathbb{R}$ et $\beta \in \mathbb{R}^P$

Pour ce faire, on **minimise le risque empirique**, calculé avec l'**erreur quadratique**

$$(y, f(x)) \mapsto \big(y - f(x)\big)^2$$

Comme une fonction de perte.
Il s'agit donc de trouver β_0, β qui minimisent

$$J(\beta_0, \beta) = \frac{1}{n}\sum_{i=1}^{n}\big(y_i - (\beta_0 + \beta^T x_i)\big)^2$$

Récupérons les valeurs de n (n_samples) et p (n_features) :

```
n_samples, n_features = X.shape
print("Nous avons n=%d observations et p=%d variables." % (n_samples,
n_features))
```

2.2 Régression linéaire avec scikit-learn

Nous allons maintenant utiliser scikit-learn pour trouver β_0, β qui minimisent J.
En machine learning, on parle d'**apprendre** le modèle f.
Les modèles linéaires sont implémentés dans le module linear_model. La régression linéaire elle-même est implémentée dans la classe linear_model.LinearRegression.

```
from sklearn import linear_model
```

L'entraînement d'un modèle d'apprentissage suit toujours la même logique dans scikit-learn :

1. On instancie un objet de la classe de modèle qui nous intéresse, ici LinearRegression

```
linreg = linear_model.LinearRegression()
```

2. Puis on entraîne cet objet sur les données avec la méthode fit :

```
linreg.fit(X, y_regress)
```

3. On peut accéder au modèle entraîné via des attributs de cet objet. Ici, les paramètres du modèle appris sont accessibles via les attributs intercept_ (pour β_0) et coef_ (pour β) :

```
print("Le poids (g) d'un manchot est prédit par %.2f " %
linreg.intercept_, end='')

print("+ %.2f x bill_length_mm + %.2f x bill_depth_mm + %.2f x
flipper_length_mm" % (tuple(linreg.coef_)))
```

4. Enfin, on peut utiliser le modèle entraîné pour faire des prédictions en utilisant la méthode predict :

```
y_pred = linreg.predict(X)
```

2.3 Performance du modèle

On peut maintenant s'intéresser à la qualité du modèle. Commençons par visualiser la corrélation entre les valeurs réelles et les valeurs prédites :

```
plt.scatter(y_regress, y_pred)

plt.xlabel("Poids réel (g)")
plt.ylabel("Poids prédit (g)")
plt.show()
```

scikit-learn nous permet aussi de quantifier la qualité du modèle grâce à différents scores, que l'on trouve dans le module metrics.

```
from sklearn import metrics
```

Nous allons utiliser ici la racine carrée de l'erreur quadratique moyenne, ou *RMSE* pour *Root Mean Squared Error*, implémentée dans mean_squared_error :

```
print("La RMSE de notre modèle est %.2f g" %
(metrics.mean_squared_error(y_regress, y_pred, squared=False)))
```

Attention ! Une erreur d'estimation de 400g n'a pas le même sens selon que l'on parle d'individus pesant plutôt 500g, 5kg ou 50kg.

Nous pouvons aussi regarder la corrélation, donnée par le coefficient de détermination R^2, implémenté dans r2_score.

```
print("Le coefficient de détermination de notre modèle est R2 = %.2f" %
(metrics.r2_score(y_regress, y_pred)))
```

2.4 Utilisation du jeu de test

Les mesures de performance calculées ci-dessus nous permettent de savoir à quel point le modèle « colle » bien aux données.

Cependant, elles ne nous informent pas sur la **capacité du modèle à généraliser**, c'est-à-dire à prédire le poids d'un manchot qui ne fasse pas partie de ce jeu de données. C'est cependant ce qui nous intéresse en apprentissage.

L'évaluation que nous venons de faire est analogue à poser un examen contenant uniquement des exercices déjà traités en classe : les élèves peuvent réussir cet examen et être tout à fait incapables de résoudre un nouveau problème faisant appel aux mêmes notions.

C'est pourquoi on préfère évaluer la performance d'un modèle sur des données qu'il n'a jamais vues auparavant. Pour ce faire, nous allons mettre de côté une partie de nos données, appelées le **jeu de test**, que nous utiliserons uniquement pour l'évaluation. Nous entraînerons le modèle sur le reste des données, appelé **jeu d'entraînement**.

2.5 Séparation des données en jeu d'entraînement et jeu de test

Le module model_selection de scikit-learn propose de nombreuses fonctionalités pour séparer des jeux de données en jeu d'entraînement et jeu de test (ainsi que des modes d'évaluation plus élaborés).

Ici nous utilisons train_test_split pour séparer les données en un jeu d'entrainement (X_train, y_train) contenant 70% des données et un jeu de test (X_test, y_test) contenant les 30% restant.

```
from sklearn import model_selection
(X_train, X_test, y_train, y_test) = model_selection.train_test_split(X,
y_regress, test_size=0.3, random_state=25)
```

Le paramètre random_state nous permet de fixer la graine du générateur de nombres aléatoires. La valeur choisie n'a aucun intérêt, mais permet de garantir que si nous réutilisons cette graine à un autre endroit de notre session, nous obtiendrons de nouveau la même séparation en jeu d'entraînement et jeu de test.

```
print(y_test)
```

Nous pouvons vérifier le nombre d'observations et de variables dans chacun de ces jeux :

```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
print("Le jeu d'entraînement contient %d observations et le jeu de test %d
observations." % (X_train.shape[0], X_test.shape[0]))
```

2.6 Régression linéaire sur le jeu d'entraînement

Nous pouvons maintenant entraîner une régression linéaire sur le jeu d'entraînement uniquement :

```
linreg.fit(X_train, y_train)
```

2.7 Performance sur le jeu de test

Nous évaluons maintenant la qualité de ce modèle sur le jeu de test :

```
y_test_pred = linreg.predict(X_test)
plt.scatter(y_test, y_test_pred)

plt.xlabel("Poids réel (g)")
plt.ylabel("Poids prédit (g)")

plt.title("Prédictions de la régression linéaire sur le jeu de test")
plt.show()

print("La RMSE de notre modèle est %.2f g" % (metrics.mean_squared_error(y_test,
y_test_pred, squared=False)))

print("Le coefficient de détermination de notre modèle est R2 = %.2f" %
(metrics.r2_score(y_test, y_test_pred)))
```

Ces valeurs sont très proches de celles obtenues sur le jeu d'entraînement ; nous pouvons en conclure qu'il n'y a vraisemblablement pas de sur-apprentissage.

3 REGRESSION LOGISTIQUE

Nous allons ici travailler avec le problème de **classification (chap.1.4)**

```
y_classif = pd.Categorical(penguins["sex"]).codes
```

3.1 Formalisme

Nous disposons toujours de n observations en p dimensions, représentés par une matrice $X \in R^{n*p}$ (ici, $p=3$), et d'un vecteur d'étiquettes $y \in \{0,1\}^n$.

Dans une régression logistique, on modélise la probabilité qu'une observation appartienne à la classe positive, autrement dit ait l'étiquette 1, par une transformation logistique d'une fonction linéaire des variables. Plus précisément, en notant X un vecteur alétaoire réel p-dimensionnel qui modélise un manchot et Y une variable aléatoire discrète à valeurs dans $\{0,1\}$ qui modélise son étiquette, on modélise $P(Y=1|X=x)$ par $f(x)$, avec

$$f : x \in \mathbb{R}^P \to \sigma(\beta_0 + \beta^T x)$$

Où $\beta_0 \in \mathbb{R}$ et $\beta \in \mathbb{R}^P$

Pour trouver β_0, β on a de nouveau recours à la **minimisation du risque empirique**, calculé avec la **perte logistique**

$$\big(y, f(x)\big) \mapsto y \log(f(x) - (1-y)\log(1-f(x)))$$

Comme fonction de perte.

Il s'agit donc de trouver β_0, β qui minimisent

$$J(\beta_0, \beta) = \frac{1}{n}\sum_{i=1}^{n} -y_i \log\big(\sigma(\beta_0 + \beta^T x_i)\big) - (1-y_i)\log\big(1-\sigma(\beta_0 + \beta^T x_i)\big)$$

3.2 Scikit-learn

La régression logistique est implémentée dans la classe LogisticRegression de linear_model de scikit-learn.

Nous utilisons une régression logistique « classique », sans pénalisation/régularisation. Il nous faut donc fixer l'argument penalty à none.

Nous suivons exactement les mêmes étapes que pour la régression linéaire :

1. Instancions un objet de la classe de modèle qui nous intéresse, ici LogisticRegression

```
logreg = linear_model.LogisticRegression(penalty="l2",tol=0.01, solver="saga")
```

2. Entraînons cet objet sur les données d'entraînement avec la méthode fit :

```
y_train=y_train.ravel()

logreg.fit(X_train, y_train)
print("La probabilité qu'un manchot soit mâle est prédite par sigma (%.2f " %
logreg.intercept_[0], end='')

print("+ %.2f x bill_length_mm + %.2f x bill_depth_mm + %.2f x
flipper_length_mm)" % (tuple(logreg.coef_[0])))
```

4. Enfin, prédisons les étiquettes des données du jeu de test en utilisant la méthode predict :

```
y_test_pred = logreg.predict(X_test)
```

3.3 Performance

Pour évaluer la performance d'un algorithme de classification, on peut regarder la **matrice de confusion** des prédictions :

```
print("%d manchots mâles ont été incorrectement prédits femelle." %
metrics.confusion_matrix(y_test, y_test_pred)[1, 0])
print("%d manchots femelles ont été incorrectement prédits mâles." %
metrics.confusion_matrix(y_test, y_test_pred)[0, 1])

disp=metrics.ConfusionMatrixDisplay(metrics.confusion_matrix(y_test,
y_test_pred))
disp.plot()
plt.show()
```

La performance globale du modèle peut aussi être mesurée par son **accuracy**, qui est la proportion de prédictions correctes.

Le terme *accuracy* est généralement traduit par « précision », mais « précision » est aussi la traduction de *precision*, qui est le ratio de prédictions correctes parmi les prédictions positives, c'est-à-dire ici la proportion de manchots mâles parmi les manchots que le modèle a prédit être mâles.

```
print("%.f%% des prédictions du modèle sur le jeu de test sont correctes." %
(100*metrics.accuracy_score(y_test, y_test_pred)))
```

Attention ! Si le nombre d'observations n'est pas le même dans chacune des classes, ce nombre peut être trompeur. Par exemple, si seulement 1% des observations appartiennent à la classe négative, un modèle naïf qui assigne toutes les observations à la classe positive aura une *accuracy* de 99%.

4 K PLUS PROCHES VOISINS

4.1 Séparation des données en jeu d'entraînement et jeu de test

Comme précédemment dans le **paragraphe 2.5**, nous utilisons le module model_selection

```
from sklearn import model_selection
(X_train, X_test, y_train, y_test) = model_selection.train_test_split(X,
y_regress, test_size=0.3, random_state=25)
```

4.2 Formalisme

Nous disposons toujours d'un jeu d'entraînement de n observations en p dimensions, représentés par une matrice $X \in R^{n*p}$ (ici, $p=3$), et d'un vecteur d'étiquettes $y \in R^n$.

L'étiquette d'une observation x est prédite comme la moyenne des étiquettes des k éléments du jeu d'entraînement les plus proches de x :

$$f(x) = \frac{1}{k} \sum_{i:x_i \in N_k(x)} y_i$$

où $N_k(x)$ désigne les k éléments de $\{x_1,x_2,...,x_n\}$ les plus proches de x (au sens de la distance euclidienne).

Une particularité de cet algorithme est qu'il n'a pas de phase d'apprentissage du modèle ; on parle parfois d'apprentissage paresseux, ou *lazy learning*.

4.3 kNN avec scikit-learn

Les algorithmes de plus proches voisins sont implémentés dans le module neighbors de scikit-learn. Pour la régression, nous utilisons la classe KNeighborsRegressor.

Nous allons fixer ici le nombre de plus proches voisins k à 7.

```
from sklearn import neighbors
```

Nous suivons toujours les étapes habituelles :

1. Instancions un objet de la classe KNeighborsRegressor

```
knnreg = neighbors.KNeighborsRegressor(n_neighbors=7)
```

2. Entraînons cet objet sur les données d'entraînement avec la méthode fit :

Ici, l'entraînement consiste uniquement à définir l'ensemble des observations étiquetées parmi lesquelles chercher les voisins d'une observation.

```
knnreg.fit(X_train, y_train)
```

3. Enfin, prédisons les étiquettes des données du jeu de test en utilisant la méthode predict :

```
y_test_pred = knnreg.predict(X_test)
```

4.4 Performances du modèle

On peut maintenant s'intéresser à la qualité du modèle. Commençons par visualiser la corrélation entre les valeurs réelles et les valeurs prédites :

```
plt.scatter(y_test, y_test_pred)
plt.plot(y_test,y_test,'r')

plt.xlabel("Poids réel (g)")
plt.ylabel("Poids prédit (g)")
plt.show()
```

```
from sklearn import metrics
print("La RMSE de notre modèle est %.2f g" % (metrics.mean_squared_error(y_test,
y_test_pred, squared=False)))
print("Le coefficient de détermination de notre modèle est R2 = %.2f" %
(metrics.r2_score(y_test, y_test_pred)))
```

4.5 Plus proches voisins pour la classification

On repart ici de `y-classif`

```
(X_train, X_test, y_train, y_test) = model_selection.train_test_split(X,
y_classif, test_size=0.3, random_state=25, stratify=y_classif)
```

4.6 Formalisme

Plutôt que d'utiliser la *moyenne* des étiquettes des k plus proches voisins, dans le cas de la classification, on utilise la *classe majoritaire* parmi les étiquettes des k plus proches voisins.

Nous disposons toujours d'un jeu d'entraînement de n observations en p dimensions, représentés par une matrice $X \in R^{n*p}$ (ici, $p=3$), et d'un vecteur d'étiquettes $y \in \{0,1\}^n$.
L'étiquette d'une observation x est prédite comme :

$$f(x) = \arg\max_{c \in \{0,1\}} \sum_{i : x_i \in N_k(x)} \delta(y_i, c)$$

où $\delta(y_i, c)$ est l'indicatrice qui vaut 1 si $y_i = c$ et 0 sinon.

4.7 kNN avec scikit-learn

Pour la classification, nous utilisons la classe KNeighborsClassifier.

1. Instancions un objet de la classe KNeighborsRegressor.

```
knnclass = neighbors.KNeighborsClassifier(n_neighbors=7)
```

2. Entraînons cet objet sur les données d'entraînement avec la méthode fit :

```
knnclass.fit(X_train, y_train)
```

3. Enfin, prédisons les étiquettes des données du jeu de test en utilisant la méthode predict :

```
y_test_pred = knnclass.predict(X_test)
```

4.8 Performances

Regardons la **matrice de confusion** des prédictions

```
print("%d manchots mâles ont été incorrectement prédits femelle." %
metrics.confusion_matrix(y_test, y_test_pred)[1, 0])
print("%d manchots femelles ont été incorrectement prédits mâles." %
metrics.confusion_matrix(y_test, y_test_pred)[0, 1])
print("%.f%% des prédictions du modèle sur le jeu de test sont correctes." %
(100*metrics.accuracy_score(y_test, y_test_pred)))

disp=metrics.ConfusionMatrixDisplay(metrics.confusion_matrix(y_test,
y_test_pred))
disp.plot()
plt.show()
```