# Rendu : Conformal Prediction

January 15, 2025

Ce notebook a été crée dans le cadre de rendu de devoir pour le cours de Prédiction conforme, dispensé par Monsieur Rémi Vaucher.

Auteure : Noa THEBAUT

## 1 Introduction

Face à la pression croissante sur les ressources naturelles et la biodiversité, la gestion durable devient plus cruciale que jamais. Selon la FAO, en 2019, 35.4% des stocks halieutiques mondiaux étaient exploités à un niveau biologiquement non durable, contre 10% en 1974, une réalité alarmante pour les écosystèmes marins (source de l'article juste ici. Mais et si la science pouvait offrir une solution en combinant technologie et écologie ? Ce notebook plonge dans l'utilisation de la prédiction conforme, une méthode statistique exploitable, je l'espère, pour transformer la gestion des ressources naturelles.

En s'appuyant sur des techniques avancées comme la régression quantile et la Split Conformal Prediction, nous montrons comment cette approche peut fournir non seulement des prédictions plus précises mais aussi des intervalles de confiance, permettant ainsi de mieux anticiper et gérer les risques liés à l'exploitation des ressources naturelles.

À travers deux études de cas concrètes, nous verrons comment :

- Optimiser la gestion durable des ormeaux : En prédisant la quantité de chair dans les ormeaux selon leurs caractéristiques, nous visons à aider les pêcheurs à éviter la capture d'individus non suffisamment développés, contribuant ainsi à la préservation de cette ressource marine essentielle.
- Améliorer le tri industriel des haricots secs : Grâce à la classification précise des variétés de haricots, nous permettons une réduction des erreurs de tri, avec des impacts significatifs sur l'efficacité industrielle et la réduction des déchets.

Cet outil statistique ne se contente pas de prédire, il quantifie l'incertitude, donnant aux décideurs les moyens de faire des choix plus éclairés et de mieux gérer les risques. Un pas de plus vers une exploitation responsable et optimisée des ressources naturelles, en alliant innovation scientifique et préservation de notre planète.

## 2 Imports nécessaires avant de se lancer dans la suite

[5]:
```
pip install ucimlrepo
```

```
Requirement already satisfied: ucimlrepo in /usr/local/lib/python3.10/dist-
packages (0.0.7)
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/dist-
packages (from ucimlrepo) (2.2.2)
Requirement already satisfied: certifi>=2020.12.5 in
/usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2024.12.14)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-
packages (from pandas>=1.0.0->ucimlrepo) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-
packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.8.2->pandas>=1.0.0->ucimlrepo) (1.17.0)
```

[6]:
```
from ucimlrepo import fetch_ucirepo
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

## 3 Regression quantile

Les ormeaux, ou abalones en anglais, sont des mollusques gastéropodes, connus pour leur coquille plate et arrondie, ainsi que pour leur chair délicate et savoureuse. Ils sont considérés comme un mets délicat et raffiné. De ce fait, ils sont classés comme une espèce vulnérable, notamment en raison de la surpêche. Pour contrer ce phénomène, plusieurs restrictions sont mises en place en France et dans plusieurs pays, notamment en Bretagne, où, par exemple, il est interdit de prélever plus de 20 ormeaux par jour et par personne.

Le lien vers la base de données se trouve ici. Il est tiré du site UCI Machine LEarning Repository.

Dans cette partie, nous chercherons à répondre à la problématique métier suivante :

***Comment la prédiction conforme peut-elle aider à la gestion durable des ormeaux, en guidant les décisions de pêche ?***

### 3.1 Description des données

Ce jeu de données comprend 4177 observations, et 9 caractéristiques (dont une varibale cible). Voici une description brève de ces dernières.

- Sex : variable catégorielle qui représente le sexe de l'ormeau, M (male), F (female), I (infant).

- Length : variable continue qui représente la longueur de la coquille en mm.
- Diameter : variable continue qui représente le diamètre de la coquille en mm.
- Height : variable continue qui représente la hauteur de la coquille en mm.
- Whole_weight : variable continue représente le poids total de l'ormeau en grammes.
- Shucked_weight : variable continue qui représente le poids de la chair seule en grammes (donc mesuré après ouverture de la coquille).
- Viscera_weight : variable continue qui représente le poids des viscères après saignée en grammes (donc mesuré après ouverture de la coquille).
- Shell_weight : variable continue qui représente le poids de la coquille après séchage en grammes (donc mesuré après ouverture de la coquille).
- Rings : variable discrète qui représente le nombre d'anneaux (variable cible dans le cadre de prédiction de l'âge de l'ormeaux).

L'âge des ormeaux est déterminé en comptant le nombre d'anneaux présents sur leur coquille. Cependant, ce processus est destructif et coûteux. C'est dans cet objectif principal que ce jeu de données a été établi : créer un modèle prédictif capable d'estimer l'âge d'un ormeau sans avoir à détruire sa coquille, en se basant uniquement sur des caractéristiques mesurables.

En effet, l'âge du coquillage peut être estimé en utilisant la formule suivante : Âge estimé = Rings + 1.5.

Cependant, nous allons utiliser ce jeu de données dans un autre objectif. En effet, si initialement il était utilisé pour prédire l'âge du coquillage, nous nous placerons dans le contexte de la prédiction de la quantité de chair présente dans la coquille, et ce, sans devoir l'ouvrir.

Ce processus présente des avantages sur plusieurs plans : il permet de s'assurer que l'ormeau est suffisamment charnu pour être consommé et, dans le cas contraire, de le relâcher. Cela offre un double bénéfice, à la fois économique et écologique.

[30]:
```
# fetch dataset
abalone = fetch_ucirepo(id=1)

# data (as pandas dataframes)
X = abalone.data.features
y = abalone.data.targets
```

[31]:
```
# Combiner les features et la variable cible pour avoir un DataFrame complet
df = pd.concat([X, y], axis=1)
```

### 3.1.1 Statistiques descriptives

Jettons un oeil aux statistiques descriptives de nos variables.

[32]:
```
# Obtenir des statistiques descriptives générales
print("\nStatistiques descriptives du dataset :")
print(df.describe())

# Statistiques pour chaque colonne
for column in df.columns:
    print(f"\nStatistiques pour la colonne '{column}' :")
```

```
    print(df[column].describe())
```

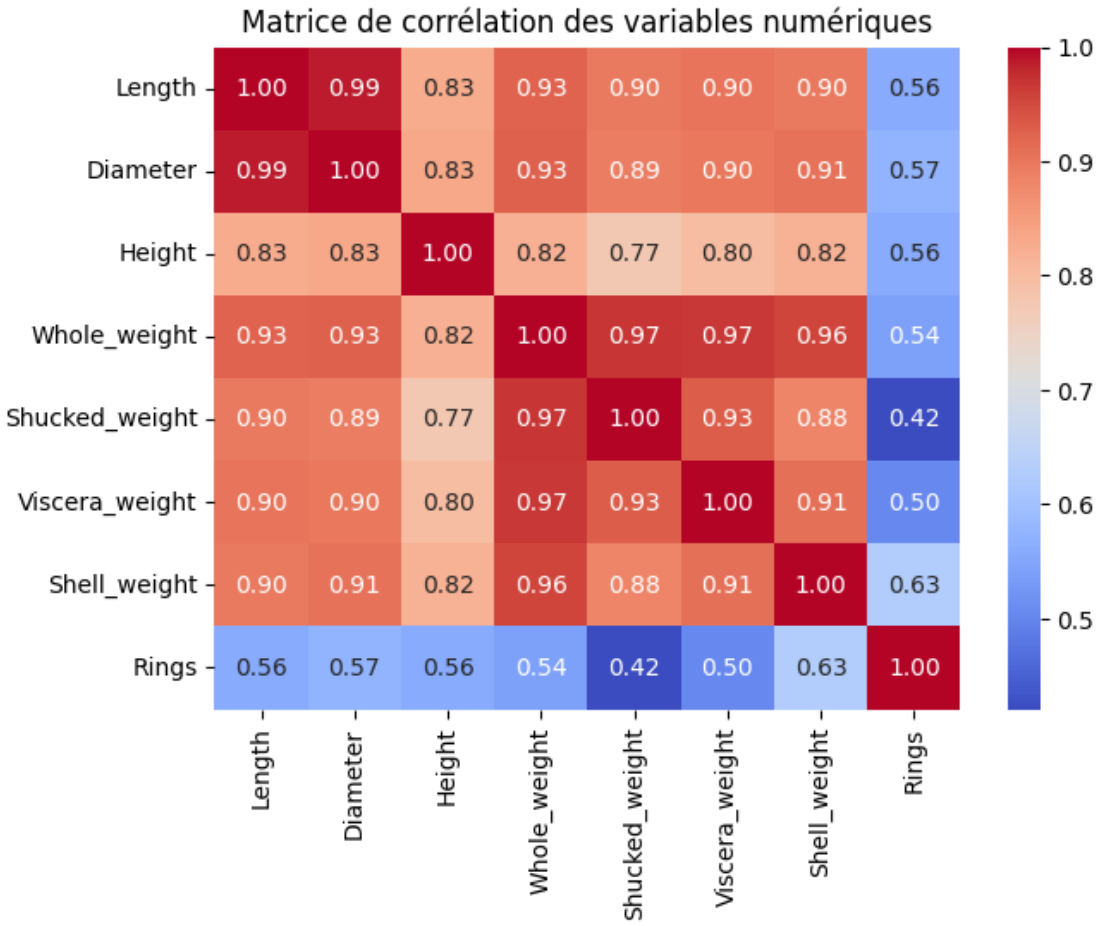
```
Statistiques descriptives du dataset :
            Length     Diameter       Height  Whole_weight  Shucked_weight  \
count  4177.000000  4177.000000  4177.000000   4177.000000     4177.000000
mean      0.523992     0.407881     0.139516      0.828742        0.359367
std       0.120093     0.099240     0.041827      0.490389        0.221963
min       0.075000     0.055000     0.000000      0.002000        0.001000
25%       0.450000     0.350000     0.115000      0.441500        0.186000
50%       0.545000     0.425000     0.140000      0.799500        0.336000
75%       0.615000     0.480000     0.165000      1.153000        0.502000
max       0.815000     0.650000     1.130000      2.825500        1.488000

       Viscera_weight  Shell_weight        Rings
count     4177.000000   4177.000000  4177.000000
mean         0.180594      0.238831     9.933684
std          0.109614      0.139203     3.224169
min          0.000500      0.001500     1.000000
25%          0.093500      0.130000     8.000000
50%          0.171000      0.234000     9.000000
75%          0.253000      0.329000    11.000000
max          0.760000      1.005000    29.000000

Statistiques pour la colonne 'Sex' :
count     4177
unique       3
top          M
freq      1528
Name: Sex, dtype: object

Statistiques pour la colonne 'Length' :
count    4177.000000
mean        0.523992
std         0.120093
min         0.075000
25%         0.450000
50%         0.545000
75%         0.615000
max         0.815000
Name: Length, dtype: float64

Statistiques pour la colonne 'Diameter' :
count    4177.000000
mean        0.407881
std         0.099240
min         0.055000
```

```
25%         0.350000
50%         0.425000
75%         0.480000
max         0.650000
Name: Diameter, dtype: float64

Statistiques pour la colonne 'Height' :
count    4177.000000
mean        0.139516
std         0.041827
min         0.000000
25%         0.115000
50%         0.140000
75%         0.165000
max         1.130000
Name: Height, dtype: float64

Statistiques pour la colonne 'Whole_weight' :
count    4177.000000
mean        0.828742
std         0.490389
min         0.002000
25%         0.441500
50%         0.799500
75%         1.153000
max         2.825500
Name: Whole_weight, dtype: float64

Statistiques pour la colonne 'Shucked_weight' :
count    4177.000000
mean        0.359367
std         0.221963
min         0.001000
25%         0.186000
50%         0.336000
75%         0.502000
max         1.488000
Name: Shucked_weight, dtype: float64

Statistiques pour la colonne 'Viscera_weight' :
count    4177.000000
mean        0.180594
std         0.109614
min         0.000500
25%         0.093500
50%         0.171000
75%         0.253000
max         0.760000
```

```
Name: Viscera_weight, dtype: float64

Statistiques pour la colonne 'Shell_weight' :
count    4177.000000
mean        0.238831
std         0.139203
min         0.001500
25%         0.130000
50%         0.234000
75%         0.329000
max         1.005000
Name: Shell_weight, dtype: float64

Statistiques pour la colonne 'Rings' :
count    4177.000000
mean        9.933684
std         3.224169
min         1.000000
25%         8.000000
50%         9.000000
75%        11.000000
max        29.000000
Name: Rings, dtype: float64
```

### 3.1.2 Corrélation entre les variables : matrice de corrélation

[33]:
```
# Sélectionner uniquement les colonnes numériques (donc pas Sex, variable␣
↪catégorielle)
numeric_df = df.select_dtypes(include=['number'])

# Calcul de la matrice de corrélation
correlation_matrix = numeric_df.corr()

# Visualiser la matrice de corrélation
plt.figure(figsize=(7, 5))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm',␣
↪cbar=True)
plt.title("Matrice de corrélation des variables numériques")
plt.show()
```




Pour rappel, la matrice de corrélation montre les relations linéaires entre les variables de notre dataset. Les coefficients sont compris entre -1 (corrélation parfaitement négative, quand une variable augmente l'autre diminue proportionnellement), et 1 (corrélation parfaitement positive, les deux variables augmentent ensemble proportionnellement), en passant par 0 qui reflète aucune corrélation linéaire ou une relation non significative.

Plus un coefficient est proche de 1 ou -1 plus la relation décrite mérite une attention particulière.

Dans notre cas, on voit que les relations entre le poids, la taille et le diamètre sont tres fortes entre elles. Elles mesurent probablement des aspects similaires des dimensions physiques (cela va de soi). Si on utilise ces relations ensemble dans un modèle cela pourrait créer une redondance.

La variable Shucked_weight (poids de la chair seule de l'ormeau) est fortement corrélée avec Whole_weight, Viscera_weight et Shell_weight. En effet, cela semble relativement logique. Cependant, ces trois dernieres variables sont mesurées après ouverture de la coquille, elles ne seront donc pas utilisables dans notre problèmatique de prédiction conforme dans le but de pêche moins intensive des ormeaux avec une volonté de pêcher uniquement lorsque la chair contenue dans le coquillage est assez conséquente.

De ce fait, on peut se pencher sur les autres variables : les dimensions du coquillage montrent ausis

une corrélation raisonnables avec Shucked_weight (de 0.77 à 0.90). On va donc s'en servir comme principales variables explicatives pour prédire le poids de la viande sans ouvrir l'ormeau.

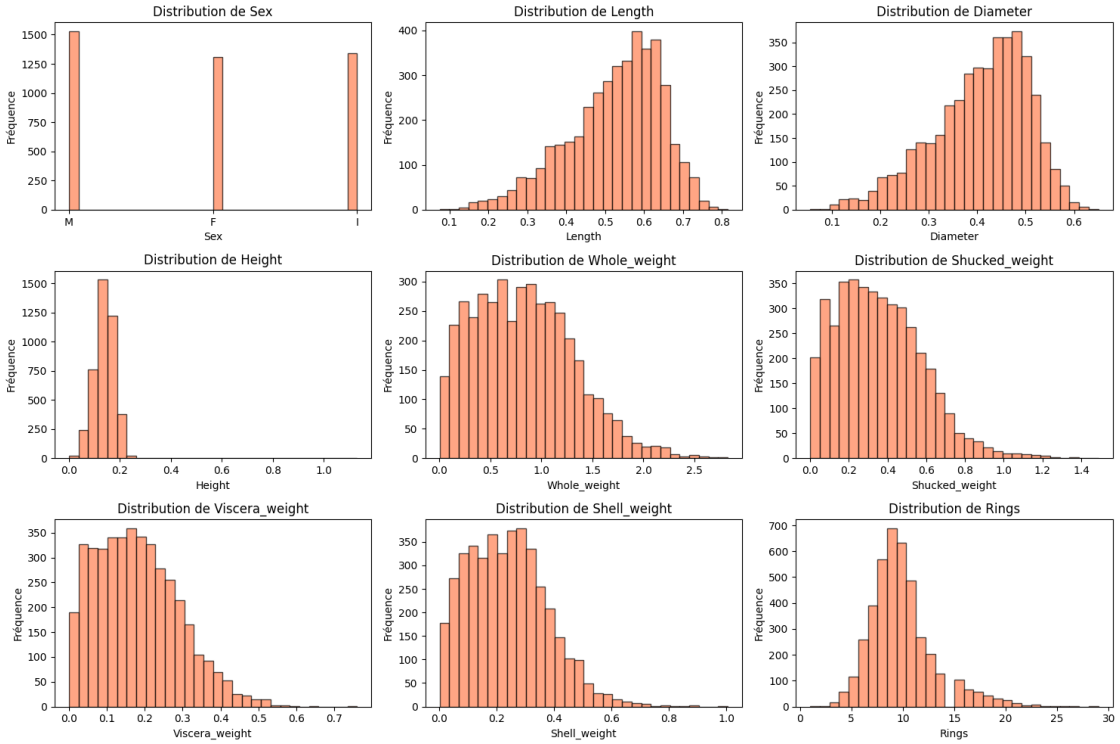
Comme abordé precedemment, la longueur et le diamètre sont très corrélés (0.99), on peut enviager de ne garder qu'une seule de ces deux varibales pour éviter la redondance dans nos modèles.

Pour finir, on peut remarquer que le poids de la chair et le nombre d'anneaux sont peu corrélés linéairement (0.42, coefficient le plus faible). Seulement, cela ne siginifie pas que les deux ne sont pas liés, mais qu'ils ne sont pas liés linéairement.

### 3.1.3 Histogrammes des variables

On peut afficher un histogrammes par variables pour avoir plus de visibilité sur celles-ci.

[34]:
```
plt.figure(figsize=(15, 10))  # Taille du graphique
for i, feature in enumerate(df.columns, 1):
    plt.subplot(3, 3, i)
    plt.hist(df[feature], bins=30, alpha=0.7, color='coral', edgecolor='black')
    plt.xlabel(feature)
    plt.ylabel('Fréquence')
    plt.title(f"Distribution de {feature}")
plt.tight_layout()
plt.show()
```

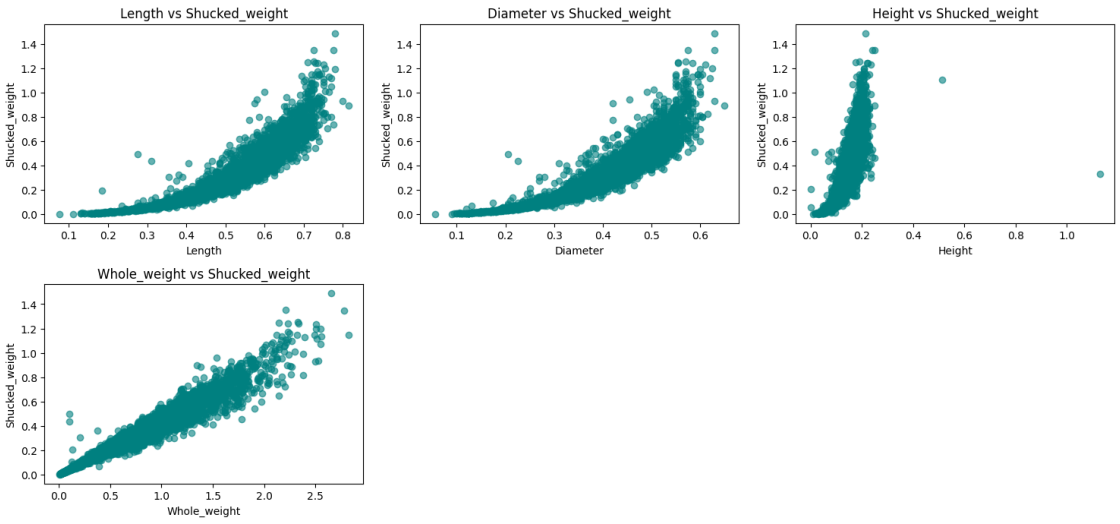


### 3.1.4 Nuages de points des variables pertinentes avec la variable cible Shucked_weight

Pour rappel, comme énoncé précedemment, nous devons dans notre étude ne garder que certaines variables, visibles/exploitables sans avoir à ouvrir la coquille, donc en l'occurence la longueur, le diamètre, la hauteur et le poids entier du coquillage avant ouverture.

Dans ce contexte, nous affichons des nuages de points entre ces variables et la variable que l'on cherche à prédire.

[35]:
```
# Variable cible
target_variable = 'Shucked_weight'

# Exclure certaines variables explicatives
excluded_features = ['Shucked_weight', 'Viscera_weight', 'Shell_weight',␣
 ↪'Rings', 'Sex']
feature_columns = [col for col in df.columns if col not in excluded_features]

# Tracer les nuages de points
plt.figure(figsize=(15, 10))  # Taille globale du graphique
for i, feature in enumerate(feature_columns, 1):
    plt.subplot(3, 3, i)  # Organiser les graphiques en grille
    plt.scatter(df[feature], df[target_variable], alpha=0.6, color='teal')
    plt.xlabel(feature)
    plt.ylabel(target_variable)
    plt.title(f"{feature} vs {target_variable}")

plt.tight_layout()
plt.show()
```




Avec les nuages de points, on a accès à une description graphique des relations entre Shucked_weight

et 4 autres varibles pernientes.

Voici ce que l'on peut déduire des quatres nuages de points :

- **Relation avec Lenght** : relation croissante (assez logique puisque plus l'ormeau est long, plus on s'attend à ce qu'il y ait de la chair dedans), pas compltement linéaire surtout pour les valeurs extrêmes.
- **Relation avec Diameter** : relation similaire que celle avec Lenght.
- **Relation avec Height** : relation moins marquée comparé à celle de la longueur et du diamètre, présence de valeurs aberrante qui pourraient fausser le résultat.
- **Relation avec Whole_weight** : relation très forte, quasi-linéaire, logique puis le poids de l'ormeau est directement lié au poids de sa chair.

Dans un soucis de practicité, on va s'intéresser à la relation entre le poids de la chair et la longueur de l'ormeau, cela semble être la variable la plus facile à mesurer lors de la pêche.

### 3.2 Régression

On rappel que l'objectif de cette étude est de prédire la quantité de viande dans l'ormeau à partir de la taille de celui-ci, afin d'évaluer sur cela vaut la peine de le pêcher ou non.

On va premièrement effectuer une régression polynomiale classique, pour établir une estimation centrale, un modèle simple avec des prédictions moyennes.

Par la suite, on va mettre en place une régression quantile, mais pourquoi ? La régression quantile va nous permettre d'estimer différents points de notre distribution (en l'occurence ici le quantile inférieur (10%), le quantile médian (50%), et le quantle supérieur (90%), on a pris $\beta = 0.1$ pour obtenir une couverture d'ordre $1 - \beta = 0.9$).

Avec ceci on va pouvoir avoir des informations avec une incertitude autour de nos prédictions. De plus, on va aussi pouvoir évaluer des scénarios optimistes et prudents avec les quantiles supéireurs et inférieurs.

[44]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression, QuantileRegressor
```

[45]:
```
# Variables spécifiques
X = df['Length'].values
y = df['Shucked_weight'].values

# Séparation des données en train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,␣
 ↪random_state=42)
```

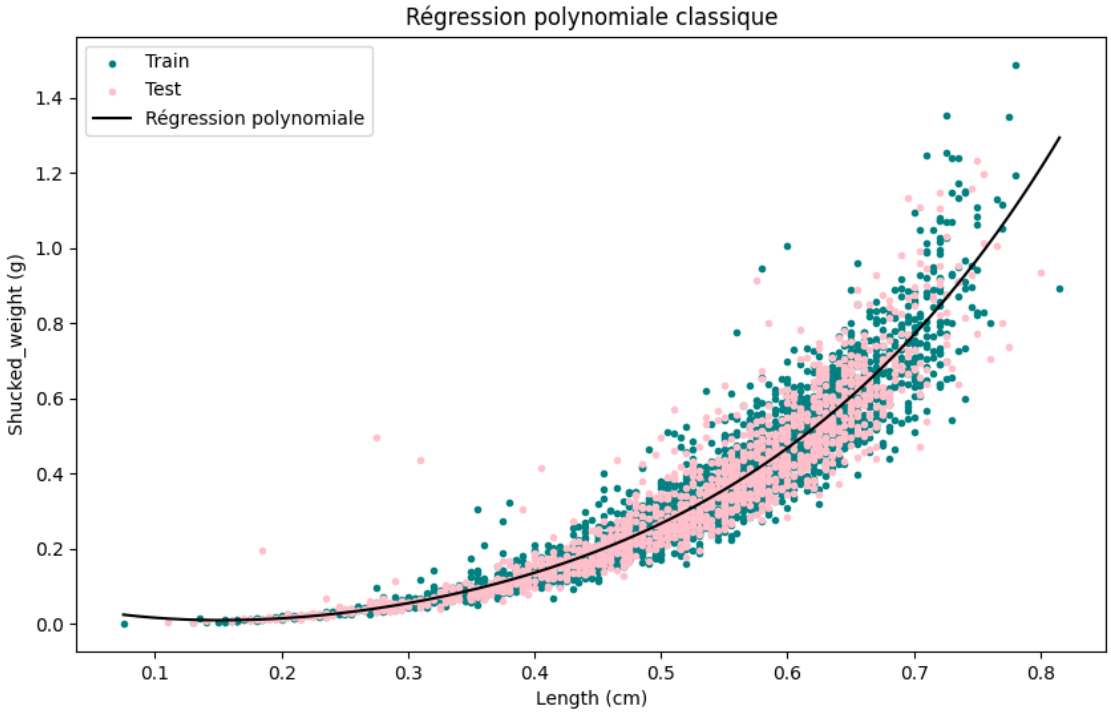
[46]:
```
poly = PolynomialFeatures(4)
Xpoly_train = poly.fit_transform(X_train[:, np.newaxis])
Xpoly_test = poly.fit_transform(X_test[:, np.newaxis])
```

[47]:
```
# Régression polynomiale classique (médiane)
linear_reg = LinearRegression(fit_intercept=False)
```

```
linear_reg.fit(Xpoly_train, y_train)
```

[47]:
```
LinearRegression(fit_intercept=False)
```

[48]:
```
# Visualisation de la régression polynomiale classique
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, marker='.', color='teal', label="Train")
plt.scatter(X_test, y_test, marker='.', color='pink', label="Test")

aux = np.linspace(X.min(), X.max(), len(X))
auxpoly = poly.transform(aux[:, np.newaxis])
plt.plot(aux, linear_reg.predict(auxpoly), color='black', label="Régression␣
 ↪polynomiale")
plt.legend()
plt.xlabel('Length (cm)')
plt.ylabel('Shucked_weight (g)')
plt.title("Régression polynomiale classique")
plt.show()
```



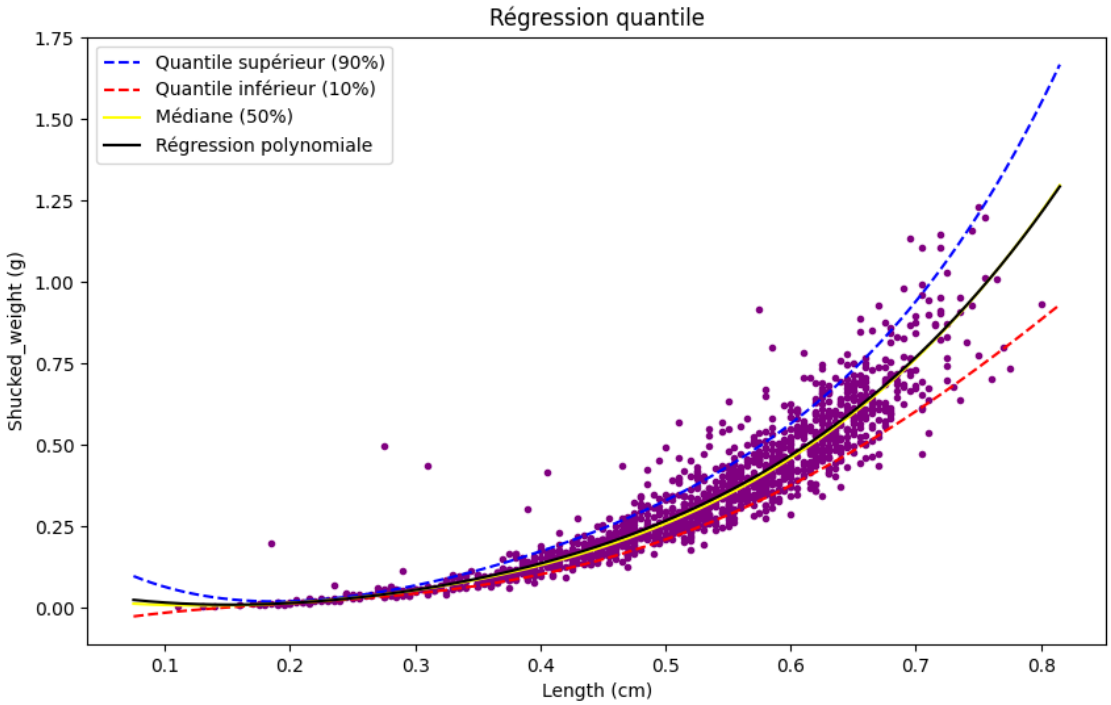

Maintenant, on va mettre en place notre regression quantile.

[49]:
```
# Régression quantile
solver = "highs"
beta = 0.2
```

```
# Intervalle de confiance : 90% (alpha = 10%)
qr_down = QuantileRegressor(quantile=beta / 2, alpha=0, solver=solver)
qr_up = QuantileRegressor(quantile=1 - beta / 2, alpha=0, solver=solver)
qr_med = QuantileRegressor(quantile=0.5, alpha=0, solver=solver)

# Ajustement des modèles
qr_down.fit(Xpoly_train, y_train)
qr_up.fit(Xpoly_train, y_train)
qr_med.fit(Xpoly_train, y_train)
```

[49]:
```
QuantileRegressor(alpha=0)
```

[50]:
```
# Visualisation des régressions quantiles
plt.figure(figsize=(10, 6))
plt.scatter(X_test, y_test, marker='.', color='purple')
plt.plot(aux, qr_up.predict(auxpoly), '--', color='blue', label="Quantile␣
 ↪supérieur (90%)")
plt.plot(aux, qr_down.predict(auxpoly), '--', color='red', label="Quantile␣
 ↪inférieur (10%)")
plt.plot(aux, qr_med.predict(auxpoly), '-', color='yellow', label="Médiane␣
 ↪(50%)")
plt.plot(aux, linear_reg.predict(auxpoly), '-', color='black',␣
 ↪label="Régression polynomiale")
plt.legend()
plt.xlabel('Length (cm)')
plt.ylabel('Shucked_weight (g)')
plt.title("Régression quantile")
plt.show()
```




Le modèle de regression quantile semble bien adapté à notre problèmatique. Cependant, avant d'appliquer une prédiction conforme, je vais tout de même vérifier le pourcentage de couverture empiruqe pour m'assurer que ce dernier est proche de $1-\beta = 0.9$. Si la couverture est loin de 90%, cela peut indiquer que le modèle est mal choisit, ou que les intervalles sont mal estimés (à cause de résidus non conformes par exemple).

[51]:
```
# Prédictions
y_pred = linear_reg.predict(Xpoly_test)

# 3. Prédiction conforme avec les résidus
residuals = np.abs(y_train - linear_reg.predict(Xpoly_train))
quantile = np.quantile(residuals, 1 - 0.1)  # Pour 90%
# Intervalles prédits
lower_bound = y_pred - quantile
upper_bound = y_pred + quantile

# 4. Calcul de la couverture
coverage = np.mean((y_test >= lower_bound) & (y_test <= upper_bound))

print(f"Couverture empirique : {coverage:.2%} (cible : 90%)")
```

```
Couverture empirique : 89.56% (cible : 90%)
```

On tombe à 89.56%, très proche de notre cible, on peut donc passer à la prédiction conforme en utilisant notre régression quantile.

### 3.3 Prédiction conforme

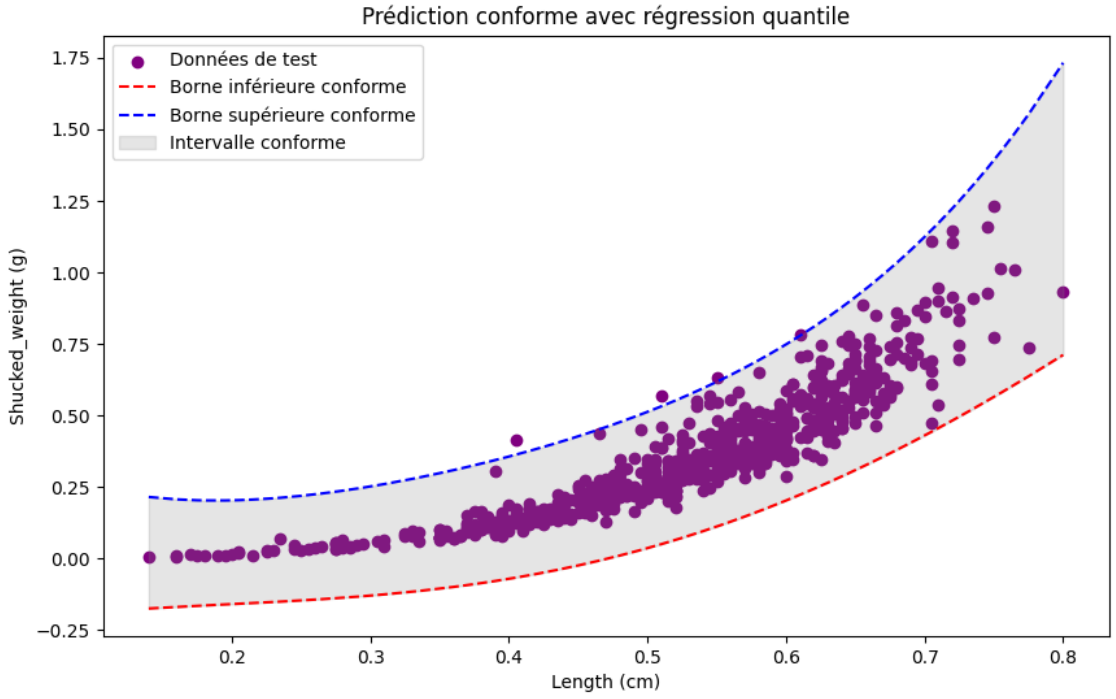
Dans notre cas de régression, plusieurs possibilité s'offre à nous en terme de choix d'algorithme de prédiction conforme.

Étant donné que nous avons mis en place une régression quantile à l'étape précédente, nous allons donc pencher vers une Conformal Quantile Regression (CQR), plutôt qu'une "simple" Split Conformal Prediction (SCP). Nos intervalles vont donc être asymétriques, en se basant sur les quantile supérieurs et inférieurs, on aura donc un modèle adaptatif.

[56]:
```
# On sépare les données pour la calibration et le test
X_calib, X_test_calib, y_calib, y_test_calib = train_test_split(X_test, y_test,␣
 ↪test_size=0.5, random_state=42)

# On transforme les données pour calibration
Xpoly_calib = poly.transform(X_calib[:, np.newaxis])
Xpoly_test_calib = poly.transform(X_test_calib[:, np.newaxis])

# On prédit les quantiles inférieur et supérieur sur l'ensemble de calibration
y_calib_down = qr_down.predict(Xpoly_calib)
y_calib_up = qr_up.predict(Xpoly_calib)

# Calcul des résidus (différence entre les bornes et les vraies valeurs)
residuals_down = y_calib - y_calib_down
residuals_up = y_calib_up - y_calib

# Calcul du seuil Q pour garantir (1-alpha) de couverture
alpha = 0.1  # Intervalle de confiance de 90%
q_low = np.quantile(residuals_down, 1 - alpha)
q_high = np.quantile(residuals_up, 1 - alpha)

# Calcul des bornes prédictives sur l'ensemble de test
y_test_down = qr_down.predict(Xpoly_test_calib) - q_low
y_test_up = qr_up.predict(Xpoly_test_calib) + q_high
```

[57]:
```
# Visualisation des intervalles conformes
plt.figure(figsize=(10, 6))
plt.scatter(X_test_calib, y_test_calib, color="purple", label="Données de test")

# Trier les données pour une meilleure visualisation de l'intervalle
idx = np.argsort(X_test_calib)
X_test_calib_sorted = X_test_calib[idx]
y_test_down_sorted = y_test_down[idx]
y_test_up_sorted = y_test_up[idx]

plt.plot(X_test_calib_sorted, y_test_down_sorted, '--', color='red',␣
 ↪label="Borne inférieure conforme")
```

```
plt.plot(X_test_calib_sorted, y_test_up_sorted, '--', color='blue',␣
 ↪label="Borne supérieure conforme")
plt.fill_between(X_test_calib_sorted, y_test_down_sorted, y_test_up_sorted,␣
 ↪color="gray", alpha=0.2, label="Intervalle conforme")
plt.xlabel("Length (cm)")
plt.ylabel("Shucked_weight (g)")
plt.title("Prédiction conforme avec régression quantile")
plt.legend()
plt.show()
```




On rappelle que notre but inital est de pouvoir prédire la quantité de viande en fonction de la taille de la coquille afin de déterminer à partir de quelle taille on peut être suffisamment confiant que l'ormeau est charnu. C'est grâce à la prédiction conforme que l'on pourra assurer, avec une certitude de tant de pourcent, que pour une taille de coquille donnée, la quantité de viande est dans tel l'intervalle.

Pour cela, on va se fixer un seuil à partir duquel on dira que l'ormeau est charnu. Un seuil bas garantit qu'un plus grand nombre d'ormeaux seront considérés comme charnus, au détriment de la qualité, ce qui est positif économiquement mais pas écologiquement. Au contraire, un seuil trop élevé selectionnera uniquement les ormeaux très charnus, limitant la sur-pêche mais réduisant fortement la population utilisable.

On jette un coup d'oeil à la répartition pour la chair afin de déterminer quel seuil on peut choisir :

[ ]:
```
print(f"Valeurs statistiques pour 'Shucked_weight':")
print(f"- Minimum : {y.min():.2f}")
print(f"- 1er Quartile (25%) : {np.percentile(y, 25):.2f}")
print(f"- Médiane (50%) : {np.median(y):.2f}")
print(f"- 3ème Quartile (75%) : {np.percentile(y, 75):.2f}")
print(f"- Maximum : {y.max():.2f}")
```

```
Valeurs statistiques pour 'Shucked_weight':
- Minimum : 0.00
- 1er Quartile (25%) : 0.19
- Médiane (50%) : 0.34
- 3ème Quartile (75%) : 0.50
- Maximum : 1.49
```

On affiche un petit apperçu des tailles des coquillages en fonction de différents seuils :

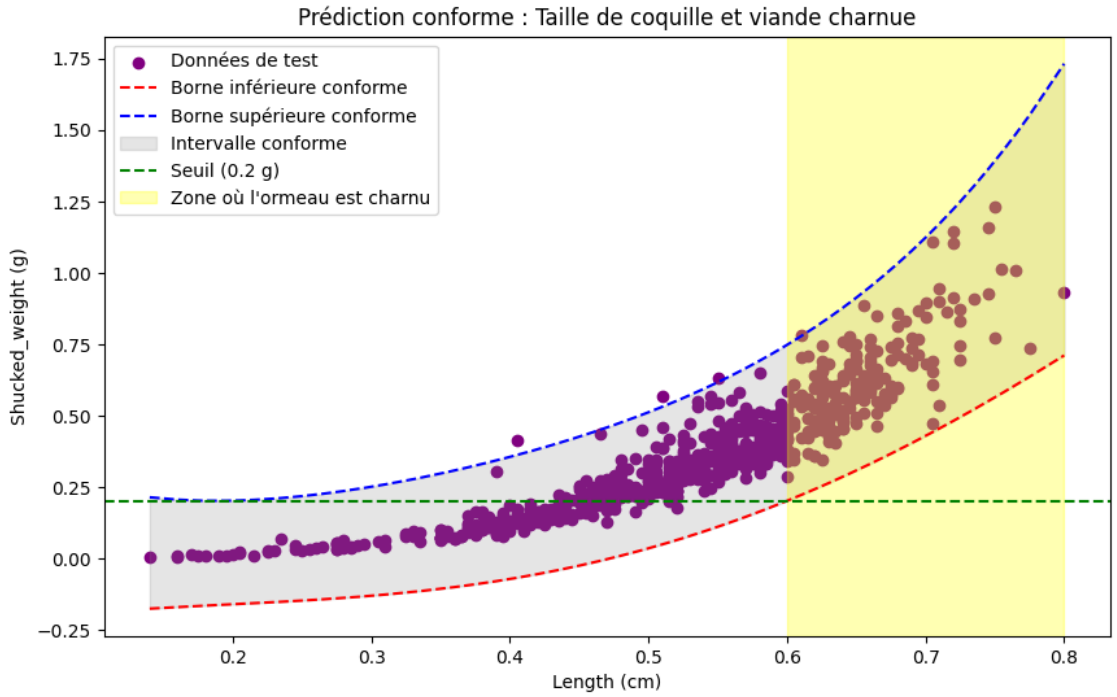
[ ]:
```
thresholds = [0.1, 0.2, 0.3]
for t in thresholds:
    print(f"Analyse pour un seuil de {t} g :")
    above_threshold = X_test_calib[y_test_down > t]
    if len(above_threshold) > 0:
        print(f"- Taille minimale : {above_threshold.min():.2f} cm")
        print(f"- Taille maximale : {above_threshold.max():.2f} cm")
    else:
        print("- Aucun ormeau charnu trouvé pour ce seuil.")
```

```
Analyse pour un seuil de 0.1 g :
- Taille minimale : 0.56 cm
- Taille maximale : 0.80 cm
Analyse pour un seuil de 0.2 g :
- Taille minimale : 0.62 cm
- Taille maximale : 0.80 cm
Analyse pour un seuil de 0.3 g :
- Taille minimale : 0.67 cm
- Taille maximale : 0.80 cm
```

Un bon compromis semble de choisir 0.2g.

[ ]:
```
# Définir le seuil pour considérer un ormeau comme "charnu"
meat_threshold = 0.2

# Visualisation des intervalles conformes et du seuil
plt.figure(figsize=(10, 6))
plt.scatter(X_test_calib, y_test_calib, color="purple", label="Données de test")

# Trier les données pour une meilleure visualisation de l'intervalle
idx = np.argsort(X_test_calib)
X_test_calib_sorted = X_test_calib[idx]
y_test_down_sorted = y_test_down[idx]
```

```
y_test_up_sorted = y_test_up[idx]

plt.plot(X_test_calib_sorted, y_test_down_sorted, '--', color='red',␣
 ↪label="Borne inférieure conforme")
plt.plot(X_test_calib_sorted, y_test_up_sorted, '--', color='blue',␣
 ↪label="Borne supérieure conforme")
plt.fill_between(X_test_calib_sorted, y_test_down_sorted, y_test_up_sorted,␣
 ↪color="gray", alpha=0.2, label="Intervalle conforme")

# Ajouter une ligne horizontale représentant le seuil
plt.axhline(meat_threshold, color="green", linestyle="--", label=f"Seuil␣
 ↪({meat_threshold} g)")

# Annoter les zones où la borne inférieure dépasse le seuil
above_threshold = X_test_calib_sorted[y_test_down_sorted > meat_threshold]  #␣
 ↪Utiliser les données triées
if len(above_threshold) > 0:
    plt.axvspan(above_threshold.min(), above_threshold.max(), color="yellow",␣
 ↪alpha=0.3,
                label="Zone où l'ormeau est charnu")

# Ajouter des légendes et titres
plt.xlabel("Length (cm)")
plt.ylabel("Shucked_weight (g)")
plt.title("Prédiction conforme : Taille de coquille et viande charnue")
plt.legend()
plt.show()
```




*Remarque : pour un soucis d'affichage, on a pas appliqué de limitation à 0 en ordonné, mais cela va de soit que le gramme de la chair ne peut être négative…*

Au final, grâce à notre prédiction conforme, on peut assurer la chose suivante :

[ ]:
```
# Définir le seuil de chair
meat_threshold = 0.2  # En grammes

# Identifier les ormeaux "charnus",
above_threshold_indices = np.where(y_test_down > meat_threshold)[0]

# Déterminer la longueur minimale de la coquille
min_length = X_test_calib[above_threshold_indices].min()

print(f"À partir d'une longueur de coquille de {min_length:.2f} cm, on a 90% de␣
↪chances d'avoir un ormeau avec au moins {meat_threshold} g de chair.")
```

```
À partir d'une longueur de coquille de 0.60 cm, on a 90% de chances d'avoir un
ormeau avec au moins 0.2 g de chair.
```

PROBLÈME ! C'est bien entendu après avoir pris le temps de mettre en place ma CQR que j'ai décidé de vérifier mon nombre de données dans l'ensemble de calibration, qui doit être d'au moins 1000 selon le cours. Nous sommes à 689 ce qui semble un peu juste…

Moi qui voulait mettre en place un algorithme de prédiction conforme uniquement pour les regressions, me voila embêtée (cela dit, je vais tout de même le mettre dans mon rendu pour montrer le

travail effectué).

[ ]:
```
num_calibration_data = len(X_calib)
print(f"Nombre de données dans l'ensemble de calibration:␣
  ↪{num_calibration_data}")
```

```
Nombre de données dans l'ensemble de calibration: 689
```

*Fun fact et moment culture, après m'être renseignée auprès de pêcheurs d'ormeaux bretons, en réalité la loi française implique une taille minimale du coquillage fixée à 9cm, or nous n'avons même pas de coquillage de cette taille dans notre jeu de données (max 8cm)…*

# 4 Classification

Après avoir exploré l'utilisation de la prédiction conforme pour évaluer la comestibilité des ormeaux en fonction de leur taille et la quantité de chair qu'ils contiennent, nous allons étendre cette méthodologie à un problèmme similaire : déterminer la variété de haricots secs en utilisant des caractéristiques extraites d'images de grains.

Dans le cadre de la classification, nous allons nous intéresser au dataset des haricots secs, toujours tiré du site UCI, qui propose une classification entre 7 espèces d'haricots secs. Ce dataset est disponible ici.

Dans cette partie, nous essaierons de répondre à la problématique métier suivante :

***Comment la quantification d'incertitude, en fournissant des intervalles de prédiction plutôt qu'une seule et unique prédiction, peut-elle améliorer la classification des variétés de haricots secs dans un système de vision par ordinateur ?***

## 4.1 Description et traitement des données

[7]:
```
# fetch dataset
dry_bean = fetch_ucirepo(id=602)

# data (as pandas dataframes)
X = dry_bean.data.features
y = dry_bean.data.targets
```

Ce jeu de données comprends 13611 instances, avec 16 caractéristiques qui comprennent des mesures comme la forme, la taille, la texture et d'autres propriétés morphologiques des haricots secs. La classe cible indique le type de haricoct (7 classes différentes).

Voici une brève description des différentes variables :

- Area : variable continue qui représente l'aire de la graine mesurée en pixels.
- Perimeter : variable continue qui représente le périmètre de la graine mesuré en pixels.
- Major_axis_length : variable continue qui représente la longueur de l'axe majeur de la graine, mesurée en pixels.
- Minor_axis_length : variable continue qui représente la longueur de l'axe mineur de la graine, mesurée en pixels.

- Convex_area : variable continue qui représente l'aire convexe minimale entourant la graine, mesurée en pixels.
- Equiv_diameter : variable continue qui représente le diamètre équivalent d'un cercle ayant la même aire que la graine, mesuré en pixels.
- Solidity : variable continue qui représente le ratio entre l'aire réelle de la graine et l'aire de sa forme convexe.
- Eccentricity : variable continue qui mesure l'excentricité de la graine, indiquant à quel point elle est allongée.
- Aspect_ratio : variable continue qui représente le ratio entre la longueur de l'axe majeur et l'axe mineur de la graine.
- Roundness : variable continue qui mesure la rondeur de la graine, calculée comme un indicateur de sa compacité.
- Compactness : variable continue qui mesure la compacité de la graine, en fonction de la forme géométrique globale.
- Shape_factor_1 : variable continue qui mesure la forme de la graine, calculée en fonction de l'aire et du périmètre.
- Shape_factor_2 : variable continue qui mesure un autre aspect géométrique de la graine, en complément du Shape_factor_1.
- Class : variable catégorielle qui représente le type de haricot, avec 7 classes différentes à prédire (les différentes variétés de haricots secs).

L'objectif est de classifier les haricots secs en différentes variétés sur la base de leurs caractéristiques mesurables. Le modèle devra être capable de différencier les types de haricots, ce qui peut être utile pour diverses applications commerciales telles que la classification automatique dans l'industrie alimentaire, le contrôle qualité, et l'optimisation de la production de haricots secs.

En intégrant la quantification de l'incertitude, il devient possible de fournir non seulement une prédiction de la classe pour un haricot donné, mais aussi un intervalle de confiance qui exprime la certitude associée à cette classification. Cela permettra aux producteurs et aux experts d'avoir une meilleure visibilité sur la fiabilité du modèle, notamment dans des situations où les données sont incertaines ou ambiguës.

### 4.1.1 Statistiques descriptives et distribution des classes

Voici un résumé statistique des variables continues :

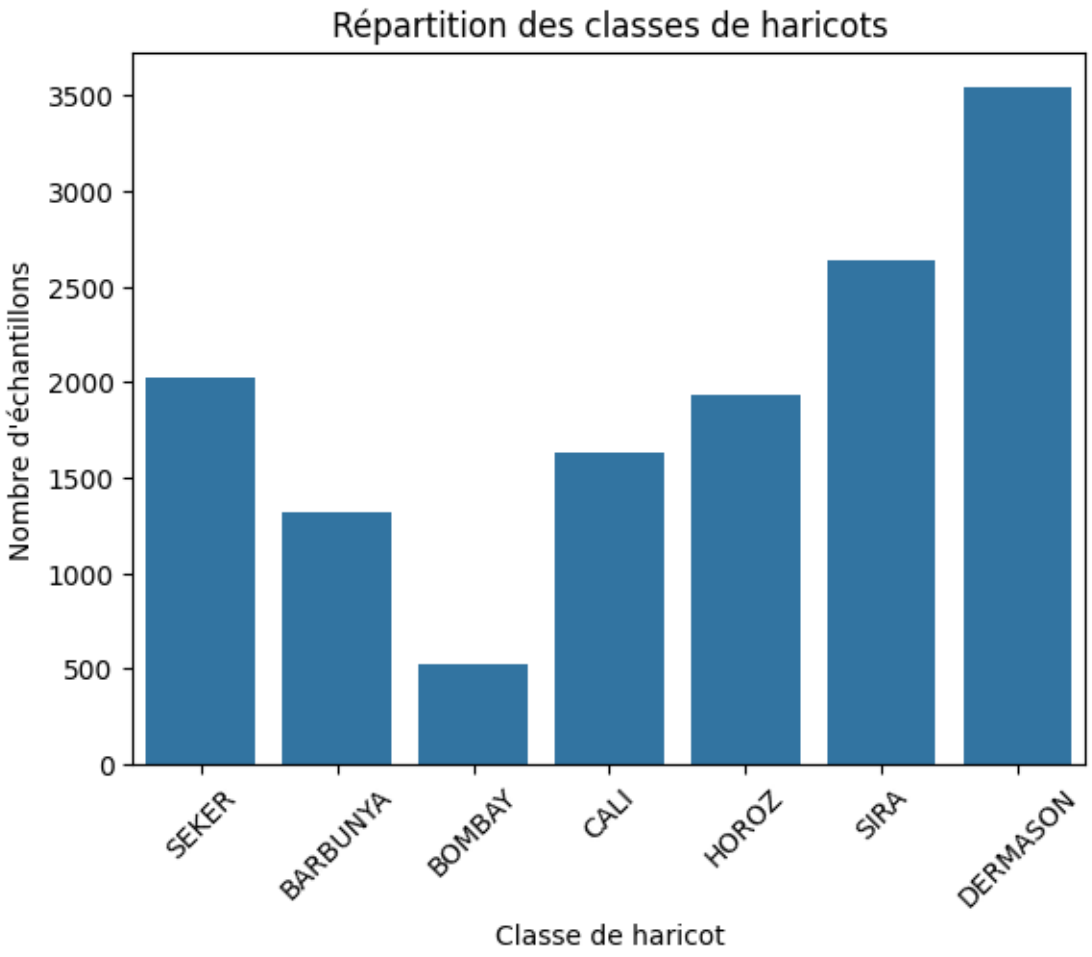
[8]:
```
# Statistiques descriptives des variables continues
print(X.describe())
```

```
                Area     Perimeter  MajorAxisLength  MinorAxisLength  \
count   13611.000000  13611.000000     13611.000000     13611.000000
mean    53048.284549    855.283459       320.141867       202.270714
std     29324.095717    214.289696        85.694186        44.970091
min     20420.000000    524.736000       183.601165       122.512653
25%     36328.000000    703.523500       253.303633       175.848170
50%     44652.000000    794.941000       296.883367       192.431733
75%     61332.000000    977.213000       376.495012       217.031741
max    254616.000000   1985.370000       738.860154       460.198497
```

```
        AspectRatio  Eccentricity     ConvexArea  EquivDiameter        Extent  \
count  13611.000000  13611.000000   13611.000000   13611.000000  13611.000000
mean       1.583242      0.750895   53768.200206     253.064220      0.749733
std        0.246678      0.092002   29774.915817      59.177120      0.049086
min        1.024868      0.218951   20684.000000     161.243764      0.555315
25%        1.432307      0.715928   36714.500000     215.068003      0.718634
50%        1.551124      0.764441   45178.000000     238.438026      0.759859
75%        1.707109      0.810466   62294.000000     279.446467      0.786851
max        2.430306      0.911423  263261.000000     569.374358      0.866195

           Solidity     Roundness   Compactness  ShapeFactor1  ShapeFactor2  \
count  13611.000000  13611.000000  13611.000000  13611.000000  13611.000000
mean       0.987143      0.873282      0.799864      0.006564      0.001716
std        0.004660      0.059520      0.061713      0.001128      0.000596
min        0.919246      0.489618      0.640577      0.002778      0.000564
25%        0.985670      0.832096      0.762469      0.005900      0.001154
50%        0.988283      0.883157      0.801277      0.006645      0.001694
75%        0.990013      0.916869      0.834270      0.007271      0.002170
max        0.994677      0.990685      0.987303      0.010451      0.003665

       ShapeFactor3  ShapeFactor4
count  13611.000000  13611.000000
mean       0.643590      0.995063
std        0.098996      0.004366
min        0.410339      0.947687
25%        0.581359      0.993703
50%        0.642044      0.996386
75%        0.696006      0.997883
max        0.974767      0.999733
```

On peut aussi jeter un coup d'oeil sur la distribution des classes :

[9]:
```
y = y.squeeze()  #on transforme y en series pour l'utiliser dans nos␣
 ↪visualisations
sns.countplot(x=y)
plt.title("Répartition des classes de haricots")
plt.xlabel("Classe de haricot")
plt.ylabel("Nombre d'échantillons")
plt.xticks(rotation=45)
plt.show()
```




[10]:
```
# Fréquence des classes
print(y.value_counts(normalize=True))
```

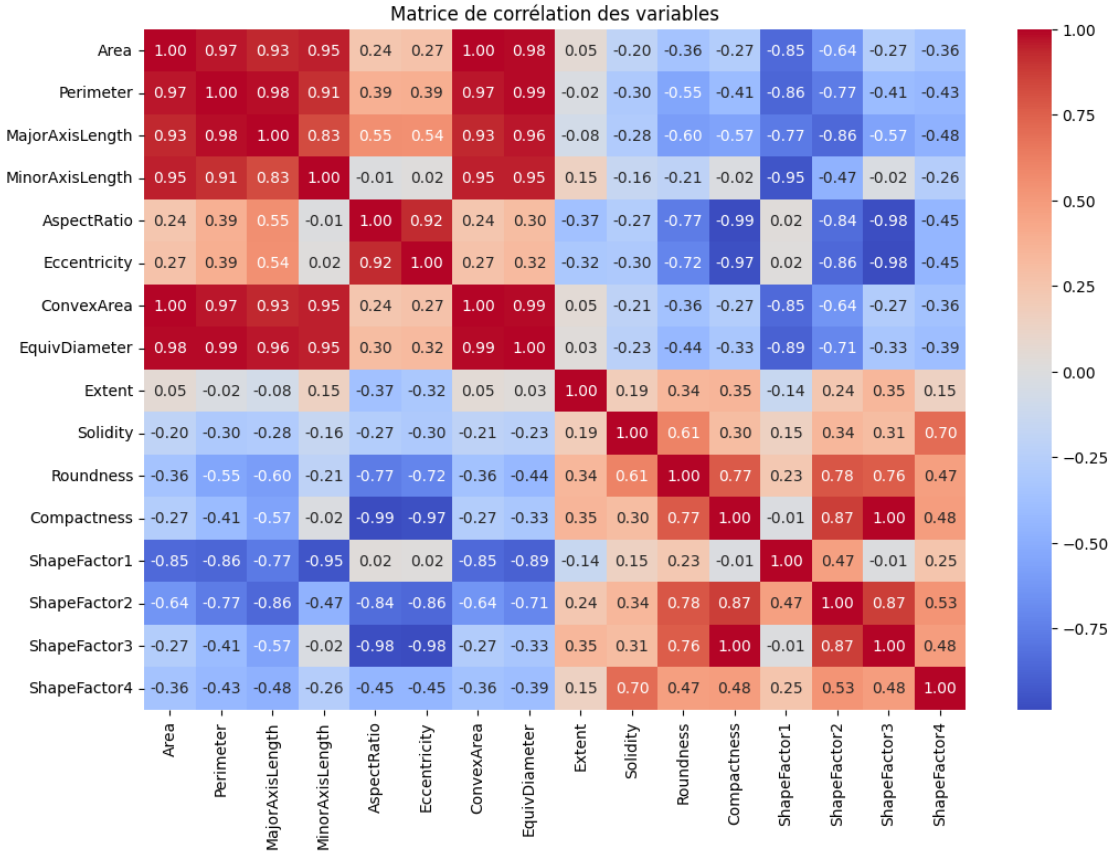
```
Class
DERMASON    0.260525
SIRA        0.193667
SEKER       0.148924
HOROZ       0.141650
CALI        0.119756
BARBUNYA    0.097127
BOMBAY      0.038351
Name: proportion, dtype: float64
```

On voit que notre dataset présente un déséquilibre de classes, avec une forte proportion de données pour la classe DERMASON et SIRA, et une faible pour la BARBUNYA et la BOMBAY. Ce déséquilibre peut influencer la performance de notre modèle en favorisant les classes plus fréquentes. On peut envisager de régler ce problème avec une stratégie d'échantillonnage (over-sampling des classes minoritaire ou under-sampling des classes majoritaires), ou utiliser des méthodes de classifi-

cation qui gèrent bien le déséquilibre (forêts aléatoires ou réseaux de neurones avec perte pondérée). Cela dit, ce n'est pas notre objectif ici donc nous garderons les données telles qu'elles sont.

### 4.1.2 Corrélation entre les variables et réduction de dimension

[11]:
```
# Calcul de la matrice de corrélation
corr_matrix = X.corr()

# Visualisation avec une heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', cbar=True)
plt.title("Matrice de corrélation des variables")
plt.show()
```



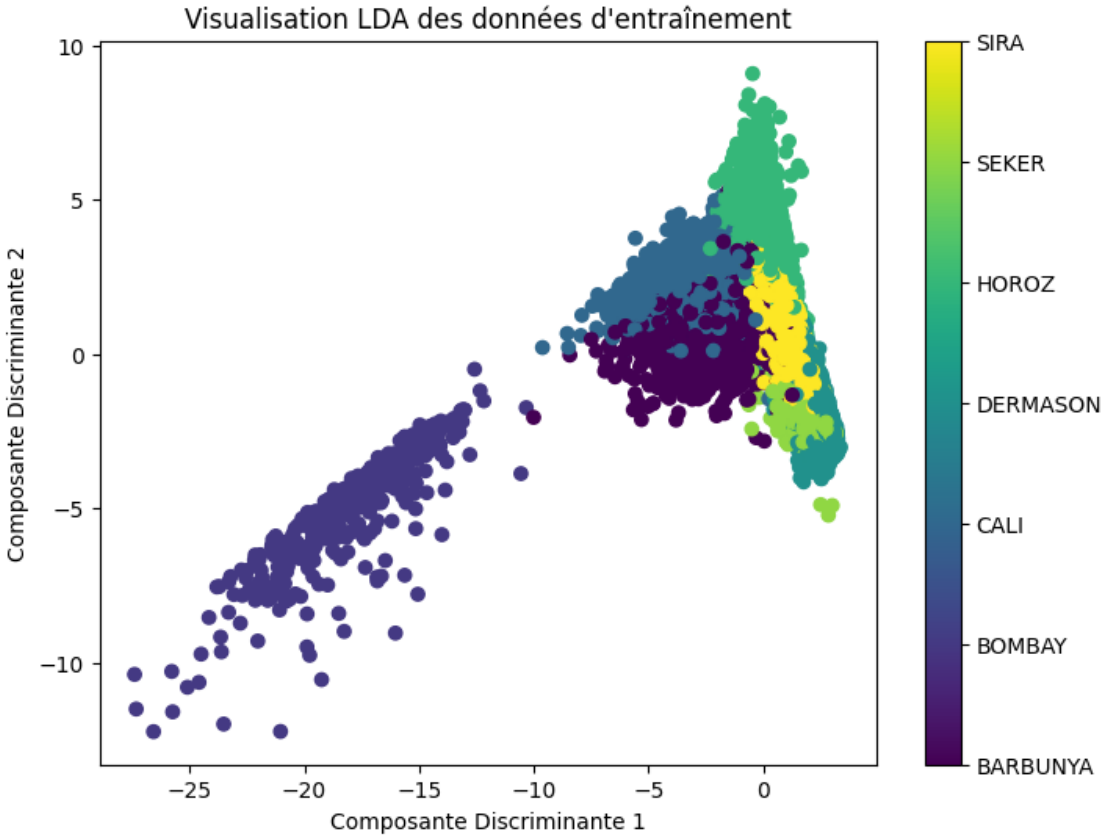

On voit grâce, à la matrice de corrélation, que certaines variables sont fortement corrélées (forte corrélation positive entre Area et Perimeter par exemple). On peut donc penser à appliquer une méthode de réduction de dimension avant notre modèle de classification pour améliorer les résultats de ce dernier.

Pour ce faire, on peut appliquer une LDA (Analyse Discriminante Linéaire), qui cherche à maximiser la séparation entre les classes, en créant de nouvelles variables qui sont les combinaisons linéaires

des variables originales. Le nombre de composantes discriminantes est limité par le nombre de classes - 1 (donc dans notre cas 6).

[12]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# 1. Diviser les données en ensembles 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)

# 2. Standardiser les données d'entraînement
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Utiliser le même scaler pour les␣
 ↪données de test

# 3. Appliquer la LDA
lda = LinearDiscriminantAnalysis(n_components=6)  # Choisir le nombre de␣
 ↪composantes (max = n_classes - 1)
X_train_lda = lda.fit_transform(X_train_scaled, y_train)
X_test_lda = lda.transform(X_test_scaled)  # Transformer les données de test
```

[13]:
```
plt.figure(figsize=(8, 6))

# Conversion en valeurs numériques
unique_labels = np.unique(y_train)
label_mapping = {label: i for i, label in enumerate(unique_labels)}
color_values = [label_mapping[label] for label in y_train]

plt.scatter(X_train_lda[:, 0], X_train_lda[:, 1], c=color_values,␣
 ↪cmap='viridis')
plt.xlabel("Composante Discriminante 1")
plt.ylabel("Composante Discriminante 2")
plt.title("Visualisation LDA des données d'entraînement")

# Ajout de la légende colorée
cbar = plt.colorbar()
cbar.set_ticks(np.arange(len(unique_labels)))
cbar.set_ticklabels(unique_labels)

plt.show()
```




On peut voir se distinguer la classe BOMBAY bien séparée, elle possède surement des caractéristiques spécifiques (par ailleurs, c'est la classe la plus sous-représentée dans notre jeu de données). Les autres classes, en haut à droite, sont regroupées et semblent partager des traits similaires.

### 4.2 Classification

Pour la classification, on va établir un reseau de neurones avec keras (en se basant sur celui fourni dans le notebook créé pour la SCP avec les données MNIST).

[14]:
```
import tensorflow as tf
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.utils import to_categorical

# Préparation des données
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # Standardiser les caractéristiques

# Encodage de la variable cible
label_encoder = LabelEncoder()
```

```
y_encoded = label_encoder.fit_transform(y)  # Conversion des classes en entiers
y_categorical = to_categorical(y_encoded)  # Conversion en one-hot encoding

# Division en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_categorical,␣
 ↪test_size=0.2, random_state=42)

# Définition du modèle
model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),  # Régularisation pour éviter le surapprentissage
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(y_train.shape[1], activation='softmax')  # Softmax pour la␣
 ↪classification multi-classes
])

# Compilation du modèle
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entraînement
history = model.fit(X_train, y_train,
                    validation_split=0.2,
                    epochs=50,
                    batch_size=32,
                    verbose=1)

# Évaluation sur l'ensemble de test
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Perte sur l'ensemble de test : {test_loss:.4f}")
print(f"Précision sur l'ensemble de test : {test_accuracy:.4f}")
```

```
Epoch 1/50
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
273/273             3s 4ms/step -
accuracy: 0.6509 - loss: 0.9749 - val_accuracy: 0.9141 - val_loss: 0.2564
Epoch 2/50
273/273             1s 3ms/step -
accuracy: 0.8802 - loss: 0.3394 - val_accuracy: 0.9206 - val_loss: 0.2255
Epoch 3/50
```

```
273/273            1s 4ms/step -
accuracy: 0.9022 - loss: 0.2845 - val_accuracy: 0.9275 - val_loss: 0.2188
Epoch 4/50
273/273            1s 3ms/step -
accuracy: 0.9084 - loss: 0.2556 - val_accuracy: 0.9279 - val_loss: 0.2124
Epoch 5/50
273/273            1s 3ms/step -
accuracy: 0.9130 - loss: 0.2500 - val_accuracy: 0.9224 - val_loss: 0.2170
Epoch 6/50
273/273            1s 3ms/step -
accuracy: 0.9152 - loss: 0.2379 - val_accuracy: 0.9238 - val_loss: 0.2120
Epoch 7/50
273/273            2s 5ms/step -
accuracy: 0.9172 - loss: 0.2272 - val_accuracy: 0.9252 - val_loss: 0.2098
Epoch 8/50
273/273            2s 4ms/step -
accuracy: 0.9179 - loss: 0.2250 - val_accuracy: 0.9288 - val_loss: 0.2091
Epoch 9/50
273/273            1s 3ms/step -
accuracy: 0.9199 - loss: 0.2256 - val_accuracy: 0.9279 - val_loss: 0.2066
Epoch 10/50
273/273            1s 4ms/step -
accuracy: 0.9221 - loss: 0.2209 - val_accuracy: 0.9252 - val_loss: 0.2115
Epoch 11/50
273/273            1s 3ms/step -
accuracy: 0.9185 - loss: 0.2309 - val_accuracy: 0.9238 - val_loss: 0.2072
Epoch 12/50
273/273            1s 3ms/step -
accuracy: 0.9195 - loss: 0.2168 - val_accuracy: 0.9302 - val_loss: 0.2007
Epoch 13/50
273/273            1s 3ms/step -
accuracy: 0.9276 - loss: 0.2019 - val_accuracy: 0.9293 - val_loss: 0.2090
Epoch 14/50
273/273            1s 3ms/step -
accuracy: 0.9255 - loss: 0.2125 - val_accuracy: 0.9293 - val_loss: 0.2070
Epoch 15/50
273/273            1s 3ms/step -
accuracy: 0.9164 - loss: 0.2278 - val_accuracy: 0.9284 - val_loss: 0.2102
Epoch 16/50
273/273            1s 3ms/step -
accuracy: 0.9252 - loss: 0.2130 - val_accuracy: 0.9302 - val_loss: 0.2023
Epoch 17/50
273/273            1s 3ms/step -
accuracy: 0.9272 - loss: 0.2040 - val_accuracy: 0.9302 - val_loss: 0.1993
Epoch 18/50
273/273            1s 4ms/step -
accuracy: 0.9203 - loss: 0.2129 - val_accuracy: 0.9298 - val_loss: 0.2012
Epoch 19/50
```

```
273/273            2s 6ms/step -
accuracy: 0.9285 - loss: 0.2014 - val_accuracy: 0.9298 - val_loss: 0.2019
Epoch 20/50
273/273            3s 6ms/step -
accuracy: 0.9310 - loss: 0.1946 - val_accuracy: 0.9265 - val_loss: 0.2066
Epoch 21/50
273/273            2s 4ms/step -
accuracy: 0.9284 - loss: 0.2104 - val_accuracy: 0.9293 - val_loss: 0.1996
Epoch 22/50
273/273            1s 3ms/step -
accuracy: 0.9263 - loss: 0.2022 - val_accuracy: 0.9298 - val_loss: 0.2011
Epoch 23/50
273/273            1s 3ms/step -
accuracy: 0.9218 - loss: 0.2072 - val_accuracy: 0.9293 - val_loss: 0.1973
Epoch 24/50
273/273            1s 3ms/step -
accuracy: 0.9291 - loss: 0.1943 - val_accuracy: 0.9320 - val_loss: 0.1944
Epoch 25/50
273/273            1s 3ms/step -
accuracy: 0.9235 - loss: 0.2053 - val_accuracy: 0.9288 - val_loss: 0.1985
Epoch 26/50
273/273            1s 3ms/step -
accuracy: 0.9214 - loss: 0.2157 - val_accuracy: 0.9275 - val_loss: 0.1990
Epoch 27/50
273/273            1s 3ms/step -
accuracy: 0.9284 - loss: 0.1974 - val_accuracy: 0.9298 - val_loss: 0.1992
Epoch 28/50
273/273            1s 4ms/step -
accuracy: 0.9292 - loss: 0.1974 - val_accuracy: 0.9288 - val_loss: 0.2013
Epoch 29/50
273/273            1s 4ms/step -
accuracy: 0.9308 - loss: 0.1949 - val_accuracy: 0.9320 - val_loss: 0.2004
Epoch 30/50
273/273            2s 5ms/step -
accuracy: 0.9275 - loss: 0.2049 - val_accuracy: 0.9302 - val_loss: 0.2065
Epoch 31/50
273/273            1s 3ms/step -
accuracy: 0.9264 - loss: 0.2050 - val_accuracy: 0.9330 - val_loss: 0.1988
Epoch 32/50
273/273            1s 3ms/step -
accuracy: 0.9251 - loss: 0.1959 - val_accuracy: 0.9325 - val_loss: 0.1953
Epoch 33/50
273/273            1s 3ms/step -
accuracy: 0.9274 - loss: 0.2017 - val_accuracy: 0.9307 - val_loss: 0.2003
Epoch 34/50
273/273            1s 3ms/step -
accuracy: 0.9318 - loss: 0.1848 - val_accuracy: 0.9311 - val_loss: 0.1959
Epoch 35/50
```

```
273/273            1s 3ms/step -
accuracy: 0.9330 - loss: 0.1789 - val_accuracy: 0.9343 - val_loss: 0.1977
Epoch 36/50
273/273            1s 3ms/step -
accuracy: 0.9237 - loss: 0.2008 - val_accuracy: 0.9275 - val_loss: 0.2006
Epoch 37/50
273/273            1s 3ms/step -
accuracy: 0.9300 - loss: 0.1830 - val_accuracy: 0.9288 - val_loss: 0.1972
Epoch 38/50
273/273            1s 3ms/step -
accuracy: 0.9298 - loss: 0.1895 - val_accuracy: 0.9293 - val_loss: 0.1971
Epoch 39/50
273/273            1s 3ms/step -
accuracy: 0.9319 - loss: 0.1920 - val_accuracy: 0.9298 - val_loss: 0.1959
Epoch 40/50
273/273            1s 3ms/step -
accuracy: 0.9315 - loss: 0.1770 - val_accuracy: 0.9320 - val_loss: 0.1975
Epoch 41/50
273/273            1s 4ms/step -
accuracy: 0.9284 - loss: 0.1897 - val_accuracy: 0.9357 - val_loss: 0.1967
Epoch 42/50
273/273            1s 4ms/step -
accuracy: 0.9278 - loss: 0.1927 - val_accuracy: 0.9348 - val_loss: 0.1955
Epoch 43/50
273/273            1s 5ms/step -
accuracy: 0.9273 - loss: 0.1988 - val_accuracy: 0.9288 - val_loss: 0.1966
Epoch 44/50
273/273            2s 3ms/step -
accuracy: 0.9330 - loss: 0.1861 - val_accuracy: 0.9293 - val_loss: 0.1939
Epoch 45/50
273/273            1s 3ms/step -
accuracy: 0.9282 - loss: 0.1934 - val_accuracy: 0.9320 - val_loss: 0.1997
Epoch 46/50
273/273            1s 3ms/step -
accuracy: 0.9296 - loss: 0.1968 - val_accuracy: 0.9348 - val_loss: 0.1954
Epoch 47/50
273/273            1s 3ms/step -
accuracy: 0.9311 - loss: 0.1875 - val_accuracy: 0.9343 - val_loss: 0.1947
Epoch 48/50
273/273            1s 3ms/step -
accuracy: 0.9305 - loss: 0.1911 - val_accuracy: 0.9279 - val_loss: 0.2057
Epoch 49/50
273/273            1s 3ms/step -
accuracy: 0.9302 - loss: 0.1923 - val_accuracy: 0.9330 - val_loss: 0.1960
Epoch 50/50
273/273            1s 3ms/step -
accuracy: 0.9275 - loss: 0.1877 - val_accuracy: 0.9330 - val_loss: 0.1964
86/86            0s 2ms/step -
```

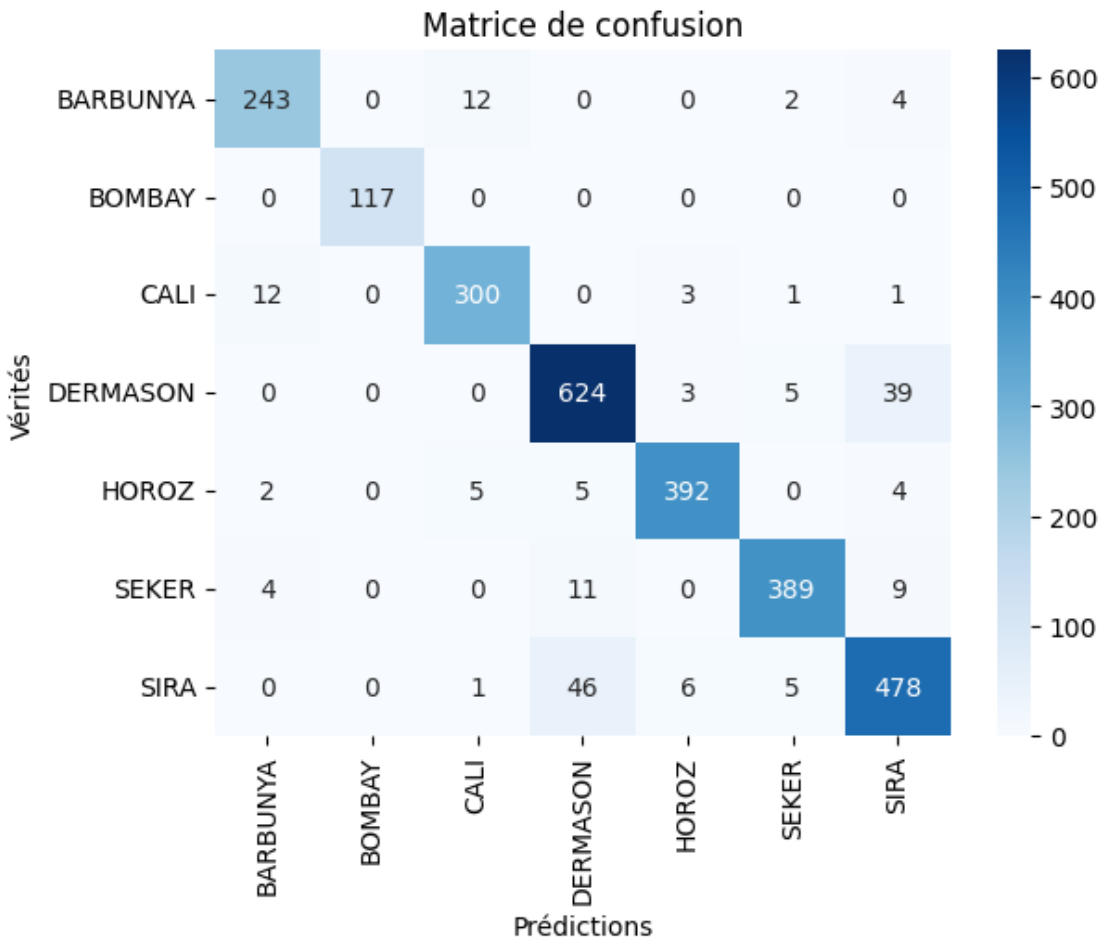
```
accuracy: 0.9396 - loss: 0.1756
Perte sur l'ensemble de test : 0.1885
Précision sur l'ensemble de test : 0.9339
```

Comme sortie d'entrainement, on a une perte sur l'ensemble de test de 0.1926, et une précision de 0.9306

Pour visualiser les résultats, on utilie une matrice de confusion, aisni que des métriques tels que la precision, le recall, le f1-score et le support.

[15]:
```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

y_pred = np.argmax(model.predict(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)

cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.
 ↪classes_, yticklabels=label_encoder.classes_)
plt.xlabel('Prédictions')
plt.ylabel('Vérités')
plt.title('Matrice de confusion')
plt.show()

print(classification_report(y_true, y_pred, target_names=label_encoder.
 ↪classes_))
```

```
86/86             0s 2ms/step
```




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| BARBUNYA | 0.93 | 0.93 | 0.93 | 261 |
| BOMBAY | 1.00 | 1.00 | 1.00 | 117 |
| CALI | 0.94 | 0.95 | 0.94 | 317 |
| DERMASON | 0.91 | 0.93 | 0.92 | 671 |
| HOROZ | 0.97 | 0.96 | 0.97 | 408 |
| SEKER | 0.97 | 0.94 | 0.95 | 413 |
| SIRA | 0.89 | 0.89 | 0.89 | 536 |
| accuracy | | | 0.93 | 2723 |
| macro avg | 0.95 | 0.94 | 0.94 | 2723 |
| weighted avg | 0.93 | 0.93 | 0.93 | 2723 |

Interprétation des résultats :

- Accuracy : 93% une précision globale haute pour un problème de classification.
- Précision, rappel et F1-score : on remarque que pour BOMBAY, la classe la plus séparée sur

notre graphique suite à la LDA, on atteint 100% (modèle parfait pour cette classe). La classe la moins bien prédite est SIRA, avec 89% de F1-score, et un rappel et précision un peu plus faibles (mais qui restent satisfaisants).

- Macro average : 94%, la moyenne des performance par classe montre une bonne homogénéité dans les prédictions, même pour les classes minoritaires.
- Weighted average : 93%, la moyenne pondérée par le support de chaque classe est similaire de l'accuracy et donc reflète le fait que le modèle fonctionne bien pour les classes minoritaires comme majoritaires.

### 4.3 Prédiction conforme

Notre modèle de reseau de neurones a généré des probabilités pour chaque classe (avec l'activation de softmax avec notre classification multi-classe). Notre but est d'intégrer ce modèle afin de produire un ensemble prédictif qui contient plusieurs classes si nécessaire. On cherche donc un modèle de prédiction conforme adapté à cette nature probabiliste, où chaque échantillon peut potentiellement appartenir à plusieurs classes avec une certaine probabilité. C'est dans ce contexte que nous allons mettre en place une SCP (Split Conformal Prediction), qui va nous permettre de quantifier l'incertitude des prédictions de notre réseau de neurones.

[16]:
```
# Division de l'ensemble de test pour calibration et validation finale
X_calib, X_val, y_calib, y_val = train_test_split(X_test, y_test, test_size=0.
 ↪5, random_state=42)
```

[17]:
```
def score(data, truth):
    # Prédire les probabilités d'appartenance
    probs = model.predict(data)[0]  # TensorFlow retourne une liste de␣
 ↪probabilités pour chaque classe
    labels = label_encoder.classes_  # Classes d'origine

    # Calcul du score
    sorted_indices = np.argsort(probs)[::-1]  # Indices des probabilités triées
    cumulative_score = 0
    for idx in sorted_indices:
        cumulative_score += probs[idx]
        if labels[idx] == truth:
            break
    return cumulative_score
```

[18]:
```
def calibration(calib_data, calib_targets, alpha):
    scores = []
    for i in range(len(calib_data)):
        # Access calib_targets using array indexing
        truth = label_encoder.inverse_transform([np.
 ↪argmax(calib_targets[i])])[0]
        scores.append(score(calib_data[i:i+1], truth))
    return np.quantile(scores, 1 - alpha)
```

[19]:
```
# Optimisation : réduire la taille de l'ensemble de calibration car temps de complétion beaucoup trop long (exemple : 20%)
X_calib_reduced, _, y_calib_reduced, _ = train_test_split(X_calib, y_calib, test_size=0.8, random_state=42)

# Calcul de q_hat sur l'ensemble de calibration réduit
q_hat = calibration(X_calib_reduced, y_calib_reduced, alpha=0.1)
```

```
1/1             0s 21ms/step
1/1             0s 21ms/step
1/1             0s 20ms/step
1/1             0s 22ms/step
1/1             0s 21ms/step
1/1             0s 20ms/step
1/1             0s 27ms/step
1/1             0s 23ms/step
1/1             0s 26ms/step
1/1             0s 26ms/step
1/1             0s 32ms/step
1/1             0s 22ms/step
1/1             0s 26ms/step
1/1             0s 31ms/step
1/1             0s 21ms/step
1/1             0s 21ms/step
1/1             0s 21ms/step
1/1             0s 21ms/step
1/1             0s 23ms/step
1/1             0s 24ms/step
1/1             0s 28ms/step
1/1             0s 21ms/step
1/1             0s 21ms/step
1/1             0s 25ms/step
1/1             0s 25ms/step
1/1             0s 24ms/step
1/1             0s 23ms/step
1/1             0s 22ms/step
1/1             0s 27ms/step
1/1             0s 21ms/step
1/1             0s 23ms/step
1/1             0s 22ms/step
1/1             0s 26ms/step
1/1             0s 24ms/step
1/1             0s 35ms/step
1/1             0s 22ms/step
1/1             0s 25ms/step
1/1             0s 23ms/step
1/1             0s 22ms/step
```

```
1/1             0s 21ms/step
1/1             0s 27ms/step
1/1             0s 23ms/step
1/1             0s 39ms/step
1/1             0s 32ms/step
1/1             0s 41ms/step
1/1             0s 41ms/step
1/1             0s 34ms/step
1/1             0s 31ms/step
1/1             0s 30ms/step
1/1             0s 30ms/step
1/1             0s 44ms/step
1/1             0s 34ms/step
1/1             0s 33ms/step
1/1             0s 30ms/step
1/1             0s 31ms/step
1/1             0s 33ms/step
1/1             0s 33ms/step
1/1             0s 44ms/step
1/1             0s 35ms/step
1/1             0s 35ms/step
1/1             0s 45ms/step
1/1             0s 37ms/step
1/1             0s 42ms/step
1/1             0s 32ms/step
1/1             0s 33ms/step
1/1             0s 36ms/step
1/1             0s 43ms/step
1/1             0s 34ms/step
1/1             0s 34ms/step
1/1             0s 37ms/step
1/1             0s 35ms/step
1/1             0s 32ms/step
1/1             0s 22ms/step
1/1             0s 22ms/step
1/1             0s 23ms/step
1/1             0s 22ms/step
1/1             0s 32ms/step
1/1             0s 33ms/step
1/1             0s 25ms/step
1/1             0s 24ms/step
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 26ms/step
1/1             0s 29ms/step
1/1             0s 26ms/step
1/1             0s 24ms/step
1/1             0s 22ms/step
```

```
1/1             0s 29ms/step
1/1             0s 21ms/step
1/1             0s 22ms/step
1/1             0s 44ms/step
1/1             0s 38ms/step
1/1             0s 38ms/step
1/1             0s 38ms/step
1/1             0s 112ms/step
1/1             0s 65ms/step
1/1             0s 39ms/step
1/1             0s 83ms/step
1/1             0s 65ms/step
1/1             0s 77ms/step
1/1             0s 53ms/step
1/1             0s 51ms/step
1/1             0s 60ms/step
1/1             0s 34ms/step
1/1             0s 37ms/step
1/1             0s 38ms/step
1/1             0s 35ms/step
1/1             0s 110ms/step
1/1             0s 85ms/step
1/1             0s 31ms/step
1/1             0s 45ms/step
1/1             0s 65ms/step
1/1             0s 128ms/step
1/1             0s 35ms/step
1/1             0s 33ms/step
1/1             0s 39ms/step
1/1             0s 48ms/step
1/1             0s 96ms/step
1/1             0s 52ms/step
1/1             0s 44ms/step
1/1             0s 45ms/step
1/1             0s 51ms/step
1/1             0s 36ms/step
1/1             0s 30ms/step
1/1             0s 53ms/step
1/1             0s 68ms/step
1/1             0s 44ms/step
1/1             0s 83ms/step
1/1             0s 59ms/step
1/1             0s 41ms/step
1/1             0s 100ms/step
1/1             0s 103ms/step
1/1             0s 78ms/step
1/1             0s 131ms/step
1/1             0s 65ms/step
```

```
1/1             0s 127ms/step
1/1             0s 57ms/step
1/1             0s 57ms/step
1/1             0s 163ms/step
1/1             0s 151ms/step
1/1             0s 152ms/step
1/1             0s 113ms/step
1/1             0s 136ms/step
1/1             0s 101ms/step
1/1             0s 31ms/step
1/1             0s 50ms/step
1/1             0s 34ms/step
1/1             0s 35ms/step
1/1             0s 31ms/step
1/1             0s 60ms/step
1/1             0s 66ms/step
1/1             0s 45ms/step
1/1             0s 46ms/step
1/1             0s 139ms/step
1/1             0s 33ms/step
1/1             0s 36ms/step
1/1             0s 34ms/step
1/1             0s 32ms/step
1/1             0s 33ms/step
1/1             0s 38ms/step
1/1             0s 33ms/step
1/1             0s 31ms/step
1/1             0s 109ms/step
1/1             0s 59ms/step
1/1             0s 158ms/step
1/1             0s 55ms/step
1/1             0s 40ms/step
1/1             0s 87ms/step
1/1             0s 60ms/step
1/1             0s 73ms/step
1/1             0s 53ms/step
1/1             0s 52ms/step
1/1             0s 57ms/step
1/1             0s 61ms/step
1/1             0s 39ms/step
1/1             0s 113ms/step
1/1             0s 37ms/step
1/1             0s 76ms/step
1/1             0s 39ms/step
1/1             0s 31ms/step
1/1             0s 41ms/step
1/1             0s 72ms/step
1/1             0s 59ms/step
```

```
1/1             0s 52ms/step
1/1             0s 117ms/step
1/1             0s 78ms/step
1/1             0s 37ms/step
1/1             0s 49ms/step
1/1             0s 58ms/step
1/1             0s 43ms/step
1/1             0s 69ms/step
1/1             0s 73ms/step
1/1             0s 58ms/step
1/1             0s 46ms/step
1/1             0s 43ms/step
1/1             0s 44ms/step
1/1             0s 29ms/step
1/1             0s 37ms/step
1/1             0s 39ms/step
1/1             0s 33ms/step
1/1             0s 31ms/step
1/1             0s 39ms/step
1/1             0s 32ms/step
1/1             0s 31ms/step
1/1             0s 32ms/step
1/1             0s 30ms/step
1/1             0s 33ms/step
1/1             0s 33ms/step
1/1             0s 34ms/step
1/1             0s 34ms/step
1/1             0s 43ms/step
1/1             0s 39ms/step
1/1             0s 38ms/step
1/1             0s 44ms/step
1/1             0s 52ms/step
1/1             0s 47ms/step
1/1             0s 37ms/step
1/1             0s 47ms/step
1/1             0s 48ms/step
1/1             0s 38ms/step
1/1             0s 36ms/step
1/1             0s 36ms/step
1/1             0s 29ms/step
1/1             0s 25ms/step
1/1             0s 34ms/step
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 22ms/step
1/1             0s 23ms/step
1/1             0s 24ms/step
1/1             0s 25ms/step
```

```
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 24ms/step
1/1             0s 23ms/step
1/1             0s 30ms/step
1/1             0s 25ms/step
1/1             0s 22ms/step
1/1             0s 22ms/step
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 24ms/step
1/1             0s 26ms/step
1/1             0s 24ms/step
1/1             0s 24ms/step
1/1             0s 21ms/step
1/1             0s 29ms/step
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 23ms/step
1/1             0s 25ms/step
1/1             0s 24ms/step
1/1             0s 23ms/step
1/1             0s 22ms/step
1/1             0s 23ms/step
1/1             0s 24ms/step
1/1             0s 32ms/step
1/1             0s 25ms/step
1/1             0s 25ms/step
1/1             0s 24ms/step
1/1             0s 26ms/step
1/1             0s 31ms/step
1/1             0s 35ms/step
1/1             0s 24ms/step
1/1             0s 25ms/step
1/1             0s 23ms/step
1/1             0s 28ms/step
1/1             0s 22ms/step
1/1             0s 22ms/step
1/1             0s 24ms/step
1/1             0s 33ms/step
```

[20]:
```python
def ens_pred(x, q_hat):
    probs = model.predict(x)[0]
    labels = label_encoder.classes_
    idx = np.argsort(probs)[::-1]
    p = 0
```

```
    i = 0
    ens = {}
    while p < q_hat:
        ens[labels[idx[i]]] = probs[idx[i]]
        p += probs[idx[i]]
        i += 1
    return ens
```

[21]:
```
import random

# Sélection de quelques exemples aléatoires
indices = random.sample(range(X_test.shape[0]), 5)
# Convert X_test and y_test to DataFrames if they are NumPy arrays
X_test_df = pd.DataFrame(X_test)
y_test_df = pd.DataFrame(y_test)
# Use .iloc to select rows by their integer location
test_samples = X_test_df.iloc[indices] # Use the DataFrame for indexing
test_labels = y_test_df.iloc[indices] # Use the DataFrame for indexing


for i, sample in enumerate(test_samples.values): # Get the values from the␣
 ↪selected rows
    truth = label_encoder.inverse_transform([np.argmax(test_labels.
 ↪iloc[i])])[0]  # Classe réelle
    ens = ens_pred(sample.reshape(1, -1), q_hat)
    print(f"Exemple {i+1} - Classe réelle : {truth}")
    print(f"Ensemble prédit : {ens}")
    print("------")
```

```
1/1             0s 27ms/step
Exemple 1 - Classe réelle : DERMASON
Ensemble prédit : {'DERMASON': 0.99998116, 'SIRA': 1.4799389e-05}
------
1/1             0s 25ms/step
Exemple 2 - Classe réelle : DERMASON
Ensemble prédit : {'DERMASON': 0.9085134, 'SIRA': 0.08447985, 'HOROZ':
0.006995922, 'SEKER': 1.0242857e-05}
------
1/1             0s 27ms/step
Exemple 3 - Classe réelle : SEKER
Ensemble prédit : {'SEKER': 0.9964449, 'DERMASON': 0.003054629, 'SIRA':
0.00047128074, 'CALI': 1.9568482e-05, 'BARBUNYA': 9.474231e-06}
------
1/1             0s 29ms/step
Exemple 4 - Classe réelle : SEKER
Ensemble prédit : {'SEKER': 0.99441445, 'DERMASON': 0.0054564215, 'SIRA':
0.00012730087}
```

```
------
1/1             0s 22ms/step
Exemple 5 - Classe réelle : HOROZ
Ensemble prédit : {'HOROZ': 0.999889, 'CALI': 8.816821e-05, 'SIRA':
2.1787419e-05}
------
```

Sur nos 5 exemples, toutes nos prédictions sont exactes. Je vais quand même générer un nouveau code qui m'affiche un exemple par classe de haricot.

[22]:
```python
def generate_examples_per_class(X_test, y_test, q_hat, label_encoder):
    """
    Génère des exemples de prédictions pour chaque classe de haricot.

    Args:
        X_test (ndarray): Données de test.
        y_test (ndarray): Étiquettes de test (encodées).
        q_hat (float): Seuil de calibration pour la prédiction conforme.
        label_encoder (LabelEncoder): Encodeur d'étiquettes.

    Returns:
        None (Affiche les exemples et leurs prédictions).
    """
    examples = {}  # Dictionnaire pour stocker un exemple par classe

    # Décoder les étiquettes à partir de y_test
    decoded_labels = label_encoder.inverse_transform(np.argmax(y_test, axis=1))

    for i, sample in enumerate(X_test):
        class_label = decoded_labels[i]
        if class_label not in examples:
            # Ajoute un exemple pour cette classe si non encore sélectionné
            truth = class_label
            ens = ens_pred(sample.reshape(1, -1), q_hat)  # Prédiction conforme
            examples[class_label] = {
                "sample": sample,
                "truth": truth,
                "prediction": ens
            }

    # Affichage des exemples et de leurs prédictions
    for class_label, data in examples.items():
        print(f"Classe : {class_label}")
        print(f"  - Vérité terrain : {data['truth']}")
        print(f"  - Prédiction conforme : {data['prediction']}")
        print("------")
```

```
# Appel de la fonction pour générer les exemples
generate_examples_per_class(X_test, y_test, q_hat, label_encoder)
```

```
1/1             0s 22ms/step
1/1             0s 22ms/step
1/1             0s 22ms/step
1/1             0s 22ms/step
1/1             0s 30ms/step
1/1             0s 25ms/step
1/1             0s 29ms/step
Classe : SEKER
  - Vérité terrain : SEKER
  - Prédiction conforme : {'SEKER': 0.9997985, 'DERMASON': 0.0001739099, 'SIRA':
2.4073825e-05}
------
Classe : BARBUNYA
  - Vérité terrain : BARBUNYA
  - Prédiction conforme : {'BARBUNYA': 0.9997795, 'HOROZ': 0.00016521582,
'CALI': 5.0911585e-05}
------
Classe : DERMASON
  - Vérité terrain : DERMASON
  - Prédiction conforme : {'DERMASON': 0.9938685, 'SIRA': 0.005977713, 'HOROZ':
0.00015267165}
------
Classe : CALI
  - Vérité terrain : CALI
  - Prédiction conforme : {'CALI': 0.99997056, 'HOROZ': 2.3212466e-05}
------
Classe : BOMBAY
  - Vérité terrain : BOMBAY
  - Prédiction conforme : {'BOMBAY': 1.0}
------
Classe : SIRA
  - Vérité terrain : SIRA
  - Prédiction conforme : {'SIRA': 0.9797436, 'HOROZ': 0.017147725, 'CALI':
0.0015241128, 'DERMASON': 0.0009001592, 'SEKER': 0.00036952642, 'BARBUNYA':
0.00031485155}
------
Classe : HOROZ
  - Vérité terrain : HOROZ
  - Prédiction conforme : {'HOROZ': 0.99695945, 'SIRA': 0.0028865933, 'CALI':
0.00015316923}
------
```

Encore mieux ! Pour chaque exemple on a une bonne classification !

### 4.3.1 Prédiction conforme au service du tri industriel

Après avoir mis en place notre prédiction conforme sur les différentes classes de haricots, on va tout de même vouloir l'utiliser dans un but technique et pratique.

Dans un contexte industriel, il serait intéressant de pouvoir garantir une classification précise pour minimiser les erreurs et optimiser l'efficacité de l'usine. Cependant, certaines observations peuvent être ambiguës et nécessiter une intervention humaine. C'est la que la prédiction conforme peut jouer un rôle, en permettant de quantifier cette incertitude en fournissant un ensemble prédictif, plutôt qu'une classe unique, pour chaque observation.

En identifiant les cas où l'incertitude est élevée, on peut :

- Réduire le bseoin de tri manuel en automatisant les décisions.
- Signaler les cas ambigus pour une vérification humaine.

On appliquerait donc une approche de tri hybride, combianant la rapidité d'un système automatique avec la précision d'une supervision humaine pour les cas difficiles.

C'est dans ce contexte que nous avons mis en place les codes suivants.

On a définit nos critères pour détecter l'incertitude selon la largeur de l'ensemble prédictif (plus de deux classes), et la confiance cumulative (probabilités cumulées inférieures à 80%).

Au final, on a un retour de fonction qui renvoie l'ensemble prédictif et signale les cas nécessitant une intervention humaine.

[23]:
```
def ens_pred_with_uncertainty(x, q_hat, threshold_width=2,␣
↪threshold_confidence=0.8):
    """
    Prédiction conforme avec détection d'incertitude.

    Args:
        x (array): Exemple à prédire.
        q_hat (float): Seuil de calibration.
        threshold_width (int): Nombre maximum de classes dans l'ensemble␣
↪prédictif avant de signaler une incertitude.
        threshold_confidence (float): Confiance cumulative minimum avant de␣
↪signaler une incertitude.

    Returns:
        dict: Résultat de la prédiction conforme avec une alerte d'incertitude.
    """
    probs = model.predict(x)[0]
    labels = label_encoder.classes_
    idx = np.argsort(probs)[::-1]

    # Construction de l'ensemble prédictif
    p = 0
    i = 0
    ens = {}
```

```
    while p < q_hat:
        ens[labels[idx[i]]] = probs[idx[i]]
        p += probs[idx[i]]
        i += 1

    # Détection d'incertitude
    uncertainty_flag = False
    if len(ens) > threshold_width or p < threshold_confidence:
        uncertainty_flag = True

    return {
        "predictions": ens,
        "uncertainty": uncertainty_flag
    }
```

On génère 10 à 15 exemples pour tester la sortie avec nos critère d'incertitude.

[29]:
```
import random

def generate_examples_with_uncertainty_limited(X_test, y_test, q_hat,␣
↪label_encoder, threshold_width=2, threshold_confidence=0.8, max_examples=15):
    """
    Génère un nombre limité d'exemples de prédictions avec détection␣
↪d'incertitude.

    Args:
        X_test (ndarray): Données de test.
        y_test (ndarray): Étiquettes de test (encodées).
        q_hat (float): Seuil de calibration pour la prédiction conforme.
        label_encoder (LabelEncoder): Encodeur d'étiquettes.
        threshold_width (int): Critère d'incertitude pour le nombre de classes.
        threshold_confidence (float): Critère d'incertitude pour la confiance␣
↪cumulative.
        max_examples (int): Nombre maximum d'exemples à afficher.

    Returns:
        None (Affiche les exemples et leurs prédictions).
    """
    # Décoder les étiquettes
    decoded_labels = label_encoder.inverse_transform(np.argmax(y_test, axis=1))

    # Sélectionner aléatoirement des indices pour limiter les exemples
    selected_indices = random.sample(range(len(X_test)), min(max_examples,␣
↪len(X_test)))

    for count, idx in enumerate(selected_indices, start=1):
        sample = X_test[idx].reshape(1, -1)  # Exemple à prédire
```

```
        truth = decoded_labels[idx]  # Classe réelle
        result = ens_pred_with_uncertainty(sample, q_hat, threshold_width,␣
 ↪threshold_confidence)

        print(f"Exemple {count} - Classe réelle : {truth}")
        print(f"Ensemble prédit : {result['predictions']}")
        if result['uncertainty']:
            print("  Incertitude détectée : intervention humaine recommandée.")
        else:
            print("  Prédiction confiante : tri automatique possible.")
        print("------")

# Appel de la fonction pour générer jusqu'à 15 exemples
generate_examples_with_uncertainty_limited(X_test, y_test, q_hat,␣
 ↪label_encoder, max_examples=15)
```

```
1/1             0s 35ms/step
Exemple 1 - Classe réelle : DERMASON
Ensemble prédit : {'DERMASON': 0.99995077, 'SIRA': 4.8174665e-05}
  Prédiction confiante : tri automatique possible.
------
1/1             0s 56ms/step
Exemple 2 - Classe réelle : SEKER
Ensemble prédit : {'SEKER': 0.99999344, 'DERMASON': 6.226328e-06}
  Prédiction confiante : tri automatique possible.
------
1/1             0s 32ms/step
Exemple 3 - Classe réelle : HOROZ
Ensemble prédit : {'HOROZ': 0.97241277, 'SIRA': 0.027318733, 'CALI':
0.00016732626, 'DERMASON': 9.261057e-05, 'BARBUNYA': 7.561736e-06}
  Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 31ms/step
Exemple 4 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.94465476, 'HOROZ': 0.052384146, 'DERMASON':
0.0027381883, 'CALI': 0.00010419344, 'BARBUNYA': 7.533915e-05, 'SEKER':
4.3410517e-05}
  Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 61ms/step
Exemple 5 - Classe réelle : CALI
Ensemble prédit : {'CALI': 0.99592495, 'BARBUNYA': 0.003361405, 'BOMBAY':
0.0007093498}
  Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 107ms/step
Exemple 6 - Classe réelle : SEKER
```

```
Ensemble prédit : {'SEKER': 0.87860924, 'DERMASON': 0.09783493, 'SIRA':
0.023265101, 'BARBUNYA': 0.00021284973, 'CALI': 6.460639e-05, 'HOROZ':
1.3161509e-05}
 Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 43ms/step
Exemple 7 - Classe réelle : SEKER
Ensemble prédit : {'SEKER': 0.98867583, 'DERMASON': 0.011173747, 'SIRA':
0.00014925055}
 Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 33ms/step
Exemple 8 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.9444622, 'HOROZ': 0.050638095, 'DERMASON':
0.0045866617, 'BARBUNYA': 0.00013514339, 'CALI': 0.00012962907, 'SEKER':
4.831911e-05}
 Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 36ms/step
Exemple 9 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.9933588, 'HOROZ': 0.002287087, 'BARBUNYA':
0.0014500702, 'SEKER': 0.0012870718, 'CALI': 0.0011006924, 'DERMASON':
0.00051619846}
 Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 33ms/step
Exemple 10 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.95761347, 'SEKER': 0.032817587, 'BARBUNYA':
0.004729843, 'CALI': 0.0027503883, 'HOROZ': 0.0014214457, 'DERMASON':
0.00066654297}
 Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 36ms/step
Exemple 11 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.9889414, 'HOROZ': 0.0055630123, 'DERMASON':
0.0050231684, 'CALI': 0.0002705886, 'SEKER': 0.00011562325, 'BARBUNYA':
8.619548e-05}
 Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 32ms/step
Exemple 12 - Classe réelle : DERMASON
Ensemble prédit : {'DERMASON': 0.99998116, 'SIRA': 1.4799389e-05}
 Prédiction confiante : tri automatique possible.
------
1/1             0s 35ms/step
Exemple 13 - Classe réelle : HOROZ
Ensemble prédit : {'HOROZ': 0.99905163, 'SIRA': 0.0009407327, 'DERMASON':
5.5776472e-06}
```

```
  Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 32ms/step
Exemple 14 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.9680935, 'CALI': 0.012633612, 'BARBUNYA':
0.0094177425, 'HOROZ': 0.0069066305, 'SEKER': 0.0028479612, 'DERMASON':
9.963134e-05}
  Incertitude détectée : intervention humaine recommandée.
------
1/1             0s 42ms/step
Exemple 15 - Classe réelle : SIRA
Ensemble prédit : {'SIRA': 0.99438184, 'BARBUNYA': 0.0014742776, 'HOROZ':
0.0013084284, 'CALI': 0.0012778786, 'SEKER': 0.0011315563, 'DERMASON':
0.0004259646}
  Incertitude détectée : intervention humaine recommandée.
------
```

Super ! On voit bien ici que les observations claires ont été automatiquement classées, et celles ambigues ont été signalée pour intervention humaine.

Cette approche garantit un tri plus fiable. La fléxibilité du choix des critères d'incertitude la rend adaptable, et donc peut être étendue à d'autres applications industrielles.