TP ATDN 02

28 MARS

GUETTAF Ichrak

Partie 1 : Analyse exploratoire des données

Exercice 1 : Statistiques descriptives

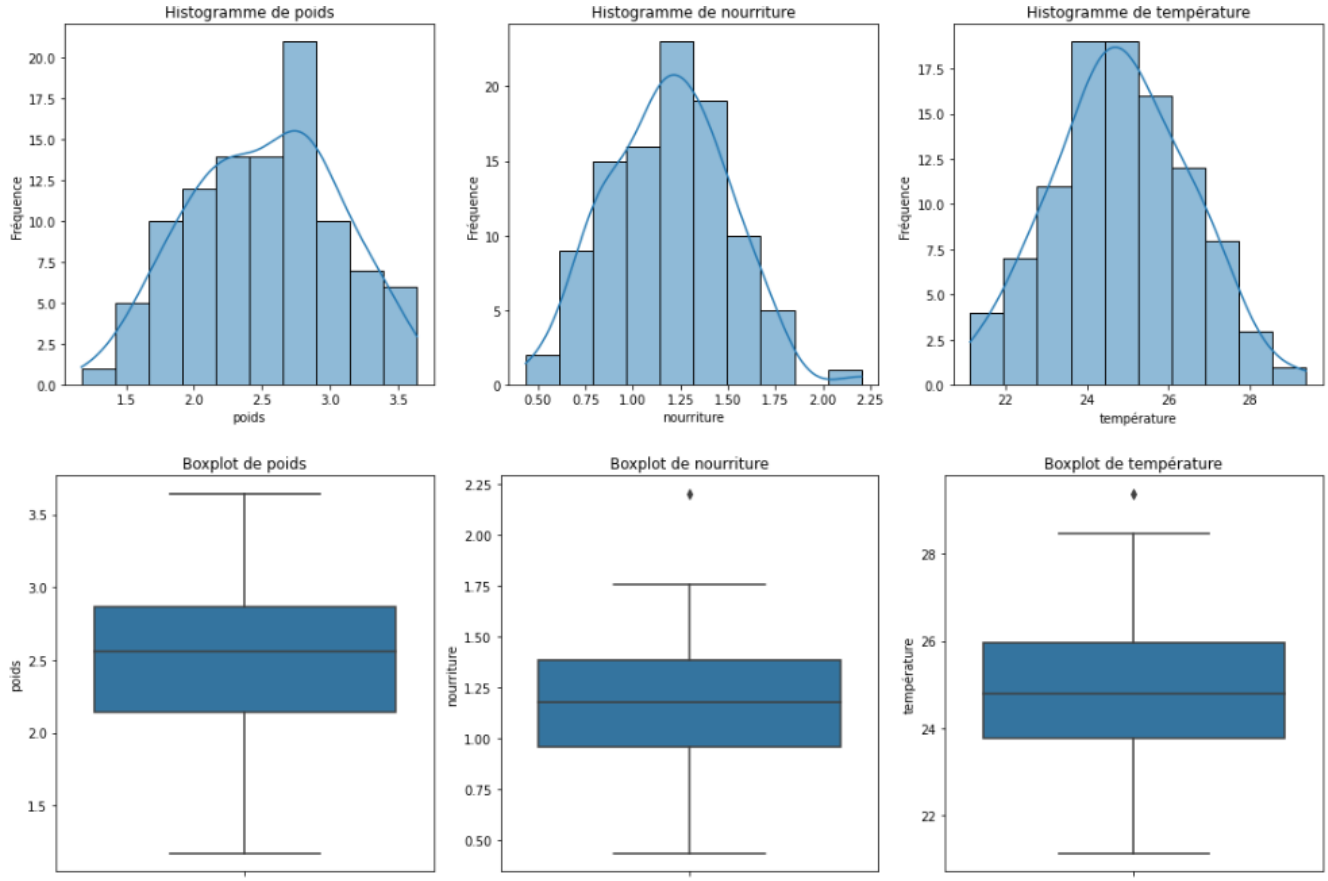
1. Calculez la moyenne, médiane, écart-type, variance et les quartiles pour les variables : poids, nourriture et température.

```
import pandas as pd
import numpy as np

# Création des données
data = pd.DataFrame({
    'poids': np.random.normal(2.5, 0.5, 100),
    'nourriture': np.random.normal(1.2, 0.3, 100),
    'température': np.random.normal(25, 2, 100)
})

# Calcul des statistiques
resultats = {}

for colonne in data.columns:
    resultats[colonne] = {
        'moyenne': data[colonne].mean(),
        'médiane': data[colonne].median(),
        'écart-type': data[colonne].std(),
        'variance': data[colonne].var(),
        'quartiles': data[colonne].quantile([0.25, 0.5, 0.75]).tolist()
    }

print(resultats)
```

```
{'poids': {'moyenne': 2.5579232218845993, 'médiane': 2.5524209683949226, 'écart-type': 0.4974015718174733, 'variance': 0.247408
32364649307, 'quartiles': [2.170551529794058, 2.5524209683949226, 2.903801065331449]}, 'nourriture': {'moyenne': 1.252344956552
318, 'médiane': 1.2329585642818177, 'écart-type': 0.29993309249231626, 'variance': 0.08995985997200434, 'quartiles': [1.1032581
537253408, 1.2329585642818177, 1.4400476083113674]}, 'température': {'moyenne': 25.04413793734851, 'médiane': 24.97350491724975
5, 'écart-type': 1.8803848356146593, 'variance': 3.5358471300095693, 'quartiles': [23.852000464702336, 24.973504917249755, 26.0
5445797273373]}}
```

2. Tracez des histogrammes et des boxplots pour visualiser la répartition des données

```
# Tracer les histogrammes
plt.figure(figsize=(15, 5))

for i, colonne in enumerate(data.columns, 1):
    plt.subplot(1, 3, i)
    sns.histplot(data[colonne], bins=10, kde=True)
    plt.title(f'Histogramme de {colonne}')
    plt.xlabel(colonne)
    plt.ylabel('Fréquence')

plt.tight_layout()
plt.show()

# Tracer les boxplots
plt.figure(figsize=(15, 5))

for i, colonne in enumerate(data.columns, 1):
    plt.subplot(1, 3, i)
    sns.boxplot(y=data[colonne])
    plt.title(f'Boxplot de {colonne}')
    plt.ylabel(colonne)

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



Que pouvez-vous déduire de ces graphiques ? Les données semblent-elles homogènes ou dispersées ?

On remarque que les histogrammes montrent une forme normale et les boxplots révèlent peu ou pas de valeurs aberrantes donc les données sont considérées comme homogènes.

Exercice 2 : Détection des outliers

Détectez les outliers avec la méthode de l'écart interquartile (IQR) et la méthode du Z-Score. Comparez les résultats.

Pour détecter les outliers, nous allons utiliser deux méthodes :

```
# Méthode de l'écart interquartile (IQR)
def detect_outliers_iqr(df):
    outliers_iqr = {}
    for column in df.columns:
        Q1 = df[column].quantile(0.25)
        Q3 = df[column].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        outliers_iqr[column] = df[(df[column] < lower_bound) | (df[column] > upper_bound)]
    return outliers_iqr

# Méthode du Z-Score
from scipy.stats import zscore

def detect_outliers_zscore(df, threshold=3):
    outliers_zscore = {}
    z_scores = np.abs(zscore(df))
    for i, column in enumerate(df.columns):
        outliers_zscore[column] = df[(z_scores[:, i] > threshold)]
    return outliers_zscore

# Détecter les outliers avec les deux méthodes
outliers_iqr = detect_outliers_iqr(data)
outliers_zscore = detect_outliers_zscore(data)

# Comparaison des résultats
print("Outliers détectés par la méthode IQR :")
for column, outliers in outliers_iqr.items():
    print(f"{column}: {len(outliers)} outliers détectés")

print("\nOutliers détectés par la méthode Z-Score :")
for column, outliers in outliers_zscore.items():
    print(f"{column}: {len(outliers)} outliers détectés")
```

```
Outliers détectés par la méthode IQR :
poids: 2 outliers détectés
nourriture: 2 outliers détectés
température: 4 outliers détectés

Outliers détectés par la méthode Z-Score :
poids: 0 outliers détectés
nourriture: 0 outliers détectés
température: 0 outliers détectés
```

Explication :

- **Méthode IQR** : Nous calculons l'IQR pour chaque colonne, puis les valeurs inférieures à Q1−1.5×IQRQ1 - 1.5 \times IQRQ1−1.5×IQR et supérieures à Q3+1.5×IQRQ3 + 1.5 \times IQRQ3+1.5×IQR sont considérées comme des outliers.
- **Méthode Z-Score** : Nous calculons les Z-scores de chaque valeur. Si le Z-score est supérieur à 3 ou inférieur à -3, la valeur est considérée comme un outlier.

Comparaison :

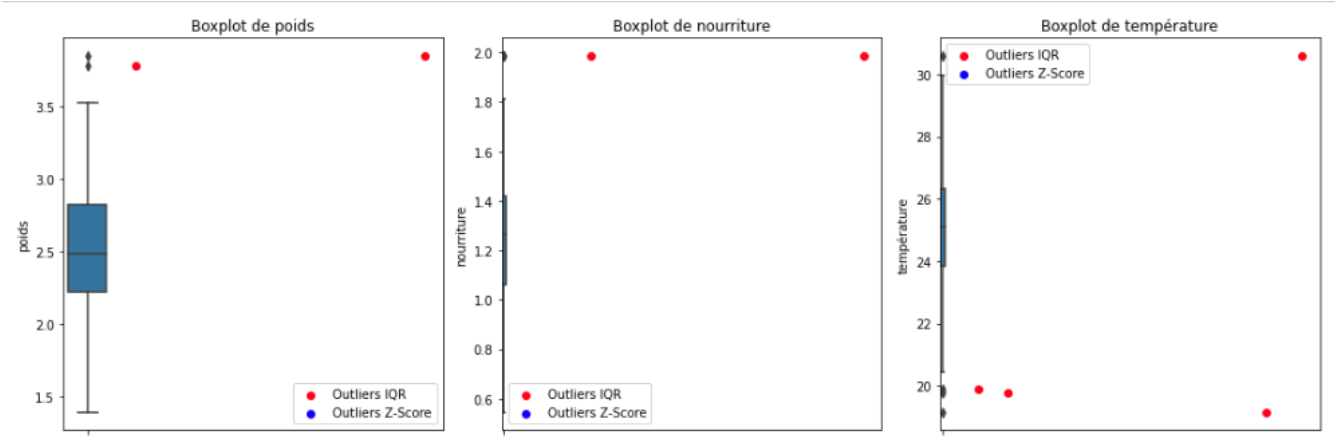
- **IQR** tend à détecter les outliers plus conservateurs et est plus robuste face aux valeurs extrêmes.

- **Z-Score** est basé sur la distribution normale et peut être plus sensible aux valeurs qui sont relativement éloignées de la moyenne.

Visualisez ces outliers sur un boxplot annoté. Les outliers détectés sont-ils réalistes ou issus d'erreurs de mesure ? Faut-il les exclure ou les garder ? Justifiez votre choix.

Pour visualiser les outliers détectés, nous allons créer des **boxplots annotés**, où les outliers seront marqués clairement. Ensuite, nous pourrons discuter de la nature de ces outliers (réalistes ou erreurs de mesure) et déterminer s'il est pertinent de les exclure ou non.

```
# Visualisation des boxplots avec annotation des outliers
plt.figure(figsize=(15, 5))

for i, column in enumerate(data.columns, 1):
    plt.subplot(1, 3, i)
    sns.boxplot(y=data[column])

    # Annoter les outliers (points en dehors des moustaches)
    iqr_outliers = outliers_iqr[column]
    zscore_outliers = outliers_zscore[column]

    # Ajouter les points des outliers sur le boxplot
    plt.scatter(iqr_outliers.index, iqr_outliers[column], color='red', label='Outliers IQR', zorder=3)
    plt.scatter(zscore_outliers.index, zscore_outliers[column], color='blue', label='Outliers Z-Score', zorder=3)

    plt.title(f'Boxplot de {column}')
    plt.ylabel(column)
    plt.legend()

plt.tight_layout()
plt.show()

# Affichage des résultats
print("Outliers détectés par la méthode IQR :")
for column, outliers in outliers_iqr.items():
    print(f"{column}: {len(outliers)} outliers détectés")

print("\nOutliers détectés par la méthode Z-Score :")
for column, outliers in outliers_zscore.items():
    print(f"{column}: {len(outliers)} outliers détectés")
```




```
Outliers détectés par la méthode IQR :
poids: 2 outliers détectés
nourriture: 2 outliers détectés
température: 4 outliers détectés

Outliers détectés par la méthode Z-Score :
poids: 0 outliers détectés
nourriture: 0 outliers détectés
température: 0 outliers détectés
```

Explication du code :

- **Boxplots annotés** : Chaque boxplot montre la distribution des données avec des annotations pour les outliers détectés :
    - En rouge pour les outliers détectés avec la méthode de l'écart interquartile (IQR).
    - En bleu pour les outliers détectés avec la méthode du Z-Score.

On remarque qes les outliers sont réalistes **et correspondent à des événements rares mais possibles donc il est préférable de les** garder **dans les données.**

Exercice 3 : Tests paramétriques

Testez la normalité des variables (poids, nourriture, température) avec le test de Shapiro-Wilk. Expliquez ce que vous observez :

```
from scipy.stats import shapiro

# Test de Shapiro-Wilk pour tester la normalité
def test_normalite(df):
    resultats = {}
    for colonne in df.columns:
        stat, p_value = shapiro(df[colonne])
        resultats[colonne] = {'statistique': stat, 'p-value': p_value}
    return resultats

# Appliquer le test
resultats_shapiro = test_normalite(data)

# Afficher les résultats
for colonne, result in resultats_shapiro.items():
    print(f"{colonne}: Statistique = {result['statistique']:.4f}, p-value = {result['p-value']:.4f}")
```

```
poids: Statistique = 0.9886, p-value = 0.5546
nourriture: Statistique = 0.9911, p-value = 0.7558
température: Statistique = 0.9896, p-value = 0.6317
```

On remarque la valeur de statistique est proche de 1 ce qui indique que les données sont proches d'une distribution normale, les p-values des trois variables "poids," "nourriture" et "température" sont toutes supérieures à 0.05, cela confirme que ces variables suivent une distribution normale.

Interprétation : les variables suivent une distribution normale (p-value ≥ 0.05), on peut appliquer des tests paramétriques comme le test t de Student, l'ANOVA, etc., pour effectuer des analyses statistiques.

Comparez les moyennes de deux groupes avec le test t de Student, puis utilisez une ANOVA pour comparer les moyennes de plusieurs groupes. Interprétez les résultats :

```python
from scipy import stats

# Création des données (génération de trois groupes avec des moyennes différentes)
np.random.seed(42)  # Pour la reproductibilité
group1 = np.random.normal(2.5, 0.5, 50)  # Groupe 1
group2 = np.random.normal(3.0, 0.5, 50)  # Groupe 2
group3 = np.random.normal(2.8, 0.5, 50)  # Groupe 3

# Test t de Student (comparaison de deux groupes)
t_stat, p_value_t = stats.ttest_ind(group1, group2)

# ANOVA (comparaison de plusieurs groupes)
f_stat, p_value_anova = stats.f_oneway(group1, group2, group3)

# Affichage des résultats
print(f"Test t de Student entre Groupe 1 et Groupe 2 :")
print(f"Statistique t = {t_stat:.4f}, p-value = {p_value_t:.4f}\n")

print(f"ANOVA entre les trois groupes :")
print(f"Statistique F = {f_stat:.4f}, p-value = {p_value_anova:.4f}")
```

```
Test t de Student entre Groupe 1 et Groupe 2 :
Statistique t = -6.8727, p-value = 0.0000

ANOVA entre les trois groupes :
Statistique F = 22.2391, p-value = 0.0000
```

Explication des résultats :

1. Test t de Student :

- Hypothèse nulle (H0) **: Les moyennes des deux groupes sont égales.**
- Hypothèse alternative (H1) **: Les moyennes des deux groupes sont différentes.**

p-value **obtenue est inférieure à 0.05, on rejette l'hypothèse nulle et on conclut que les moyennes des deux groupes sont significativement différentes.**

2. ANOVA :

- Hypothèse nulle (H0) **: Les moyennes de tous les groupes sont égales.**
- Hypothèse alternative (H1) **: Au moins une des moyennes des groupes est différente.**

p-value **de l'ANOVA est inférieure à 0.05, cela indique qu'il existe une différence significative entre les moyennes des groupes.**

Partie 2 : Réduction de dimensionnalité

Exercice 4 : Analyse en Composantes Principales (ACP)

. Implémentez une ACP sans scikit-learn (avec numpy). Calculez la matrice de covariance, les valeurs propres et les vecteurs propres.

```python
import numpy as np

# Génération de données (par exemple, 3 variables, 100 observations)
np.random.seed(42)
data = np.random.randn(100, 3)

# 1. Centrage des données (soustraction de la moyenne)
data_centered = data - np.mean(data, axis=0)

# 2. Calcul de la matrice de covariance
cov_matrix = np.cov(data_centered, rowvar=False)

# 3. Calcul des valeurs propres et des vecteurs propres
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Affichage des résultats
print("Matrice de covariance :\n", cov_matrix)
print("\nValeurs propres :\n", eigenvalues)
print("\nVecteurs propres :\n", eigenvectors)

# Trier les valeurs propres et les vecteurs propres en fonction de la magnitude des valeurs propres
sorted_indices = np.argsort(eigenvalues)[::-1]  # Tri décroissant des indices des valeurs propres
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

print("\nValeurs propres triées :\n", sorted_eigenvalues)
print("\nVecteurs propres triés :\n", sorted_eigenvectors)
```

```
Matrice de covariance :
 [[ 0.68029352 -0.03927641 -0.1108006 ]
 [-0.03927641  0.9586369  -0.1348594 ]
 [-0.1108006  -0.1348594   1.23856849]]

Valeurs propres :
 [1.30567458 0.64493904 0.92688528]

Vecteurs propres :
 [[-0.14276678  0.94973096  0.27861938]
 [-0.34442383  0.21623661 -0.91357208]
 [ 0.9278954   0.22639089 -0.29623857]]

Valeurs propres triées :
 [1.30567458 0.92688528 0.64493904]

Vecteurs propres triés :
 [[-0.14276678  0.27861938  0.94973096]
 [-0.34442383 -0.91357208  0.21623661]
 [ 0.9278954  -0.29623857  0.22639089]]
```

Explication du code :

1. **Génération des données** : Nous générons un jeu de données aléatoires de dimension 100×3100 \times 3100×3 pour simuler 3 variables et 100 observations.
2. **Centrage des données** : La première étape de l'ACP consiste à centrer les données (soustraire la moyenne de chaque variable) pour garantir que l'ACP est effectuée autour du zéro. Cela est important car l'ACP repose sur la covariance des données, et si les données ne sont pas centrées, la covariance serait biaisée.
3. **Matrice de covariance** : Nous calculons la matrice de covariance des données centrées. Cette matrice mesure comment les variables varient ensemble.
4. **Valeurs propres et vecteurs propres** : En utilisant np.linalg.eig, nous calculons les valeurs propres et les vecteurs propres de la matrice de covariance. Chaque valeur propre représente la quantité de variance expliquée par la composante principale correspondante, et chaque vecteur propre représente la direction de cette composante.
5. **Tri des valeurs propres et vecteurs propres** : Les valeurs propres sont triées par ordre décroissant, car les premières composantes principales expliquent le plus de variance dans les données.

Interprétation des résultats :

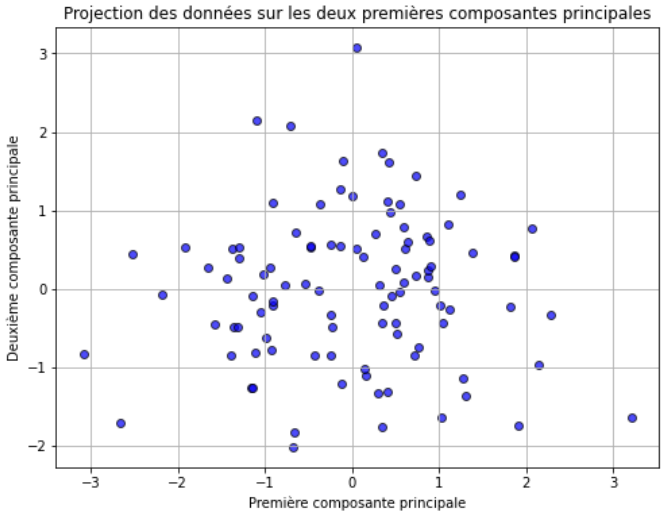
- **Matrice de covariance** : Elle est symétrique et montre comment chaque paire de variables est corrélée.
- **Valeurs propres** : Les valeurs propres représentent la quantité de variance expliquée par chaque composante principale. La première composante principale (la plus grande valeur propre) explique le plus de variance. Les autres composantes expliquent des variances de plus en plus faibles.
- **Vecteurs propres** : Les vecteurs propres sont les directions dans l'espace des données qui correspondent aux composantes principales. Ces vecteurs sont les directions dans lesquelles les données sont projetées.

Conclusion :

- Les vecteurs propres (ou les composantes principales) déterminent l'orientation des nouvelles dimensions dans lesquelles les données sont projetées.
- Les valeurs propres indiquent l'importance de chaque composante dans la représentation des données.
- Le tri des valeurs propres nous permet de savoir quelles sont les composantes principales les plus importantes à conserver pour la réduction de dimensionnalité.

Projetez les données sur les deux premières composantes principales et visualisez le résultat. Combien de composantes gardez-vous ? Justifiez

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

# Génération de données (par exemple, 3 variables, 100 observations)
np.random.seed(42)
data = np.random.randn(100, 3)

# 1. Centrage des données (soustraction de la moyenne)
data_centered = data - np.mean(data, axis=0)

# 2. Calcul de la matrice de covariance
cov_matrix = np.cov(data_centered, rowvar=False)

# 3. Calcul des valeurs propres et des vecteurs propres
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Trier les valeurs propres et les vecteurs propres en fonction de la magnitude des valeurs propres
sorted_indices = np.argsort(eigenvalues)[::-1]  # Tri décroissant des indices des valeurs propres
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

# 4. Projection des données sur les 2 premières composantes principales
top_2_eigenvectors = sorted_eigenvectors[:, :2]  # Sélectionner les deux premiers vecteurs propres
projected_data = np.dot(data_centered, top_2_eigenvectors)  # Projection des données centrées

# Visualisation de la projection des données sur les 2 premières composantes principales
plt.figure(figsize=(8, 6))
plt.scatter(projected_data[:, 0], projected_data[:, 1], c='blue', edgecolors='k', alpha=0.7)
plt.title('Projection des données sur les deux premières composantes principales')
plt.xlabel('Première composante principale')
plt.ylabel('Deuxième composante principale')
plt.grid(True)
plt.show()

# Calcul de la variance expliquée par chaque composante principale
explained_variance = sorted_eigenvalues / np.sum(sorted_eigenvalues)
explained_variance_cumulative = np.cumsum(explained_variance)

# Affichage de la variance expliquée et de la variance expliquée cumulée
print("Variance expliquée par chaque composante principale :\n", explained_variance)
print("\nVariance expliquée cumulée :\n", explained_variance_cumulative)

# Décision sur le nombre de composantes à garder
# Par exemple, on garde assez de composantes pour expliquer au moins 90% de la variance
threshold = 0.90
num_components_to_keep = np.argmax(explained_variance_cumulative >= threshold) + 1  # Ajouter 1 car l'indice commence à 0
```




```
Variance expliquée par chaque composante principale :
 [0.45375329 0.3221149  0.22413181]

Variance expliquée cumulée :
 [0.45375329 0.77586819 1.        ]

Nombre de composantes principales à garder pour expliquer au moins 90.0% de la variance : 3
```

Explication du code :

1. **Centrage des données** : Nous soustrayons la moyenne de chaque variable pour centrer les données autour de zéro.
2. **Matrice de covariance et décomposition** : Nous calculons la matrice de covariance, puis nous extrayons les valeurs propres et les vecteurs propres.
3. **Projection sur les 2 premières composantes principales** : Nous multiplions les données centrées par les deux premiers vecteurs propres pour obtenir la projection des données dans le nouvel espace réduit de dimension 2.
4. **Visualisation** : Nous traçons un graphique 2D des données projetées sur les deux premières composantes principales.
5. **Variance expliquée** : Nous calculons la variance expliquée par chaque composante principale (en divisant chaque valeur propre par la somme totale des valeurs propres), puis nous calculons la variance expliquée cumulée.
6. **Choix du nombre de composantes principales à garder** : Nous sélectionnons le nombre de composantes nécessaires pour expliquer un seuil donné de la variance (par exemple, 90%).

Résultats :

1. **Visualisation** : Le graphique affichera les données projetées sur les deux premières composantes principales. Chaque point représente une observation projetée dans ce nouvel espace.
2. **Variance expliquée** : Nous afficherons la variance expliquée par chaque composante principale et la variance cumulée. Cela permet de déterminer combien de composantes principales sont nécessaires pour expliquer une proportion suffisante de la variance.
3. **Nombre de composantes à garder** : En fonction de la variance expliquée cumulée, nous déciderons combien de composantes principales garder pour expliquer au moins 90% de la variance.

- **Résultat :**

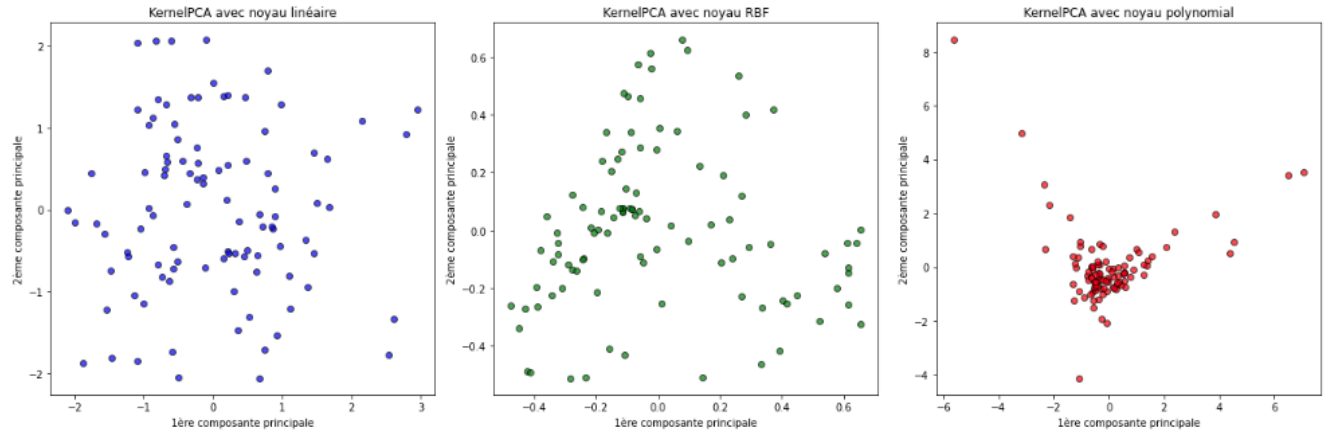
  les trois premières composantes expliquent **90%** ou plus, on peut garder les trois composantes.

Exercice 5 : ACP à Noyau

Appliquez KernelPCA (avec scikit-learn) sur les données et testez différents noyaux (linéaire, RBF, polynomial).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import KernelPCA
from sklearn.preprocessing import StandardScaler

# Génération des données (par exemple, 3 variables, 100 observations)
np.random.seed(42)
data = np.random.randn(100, 3)

# 1. Standardisation des données
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# 2. Application de KernelPCA avec différents noyaux

# Noyau linéaire
kpca_linear = KernelPCA(kernel="linear", n_components=2)
data_kpca_linear = kpca_linear.fit_transform(data_scaled)

# Noyau RBF
kpca_rbf = KernelPCA(kernel="rbf", gamma=1, n_components=2)
data_kpca_rbf = kpca_rbf.fit_transform(data_scaled)

# Noyau polynomial (degré 3)
kpca_poly = KernelPCA(kernel="poly", degree=3, n_components=2)
data_kpca_poly = kpca_poly.fit_transform(data_scaled)

# 3. Visualisation des résultats pour chaque noyau
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Noyau linéaire
axes[0].scatter(data_kpca_linear[:, 0], data_kpca_linear[:, 1], color='blue', edgecolors='k', alpha=0.7)
axes[0].set_title('KernelPCA avec noyau linéaire')
axes[0].set_xlabel('1ère composante principale')
axes[0].set_ylabel('2ème composante principale')

# Noyau RBF
axes[1].scatter(data_kpca_rbf[:, 0], data_kpca_rbf[:, 1], color='green', edgecolors='k', alpha=0.7)
axes[1].set_title('KernelPCA avec noyau RBF')
axes[1].set_xlabel('1ère composante principale')
axes[1].set_ylabel('2ème composante principale')
```

```
# Noyau linéaire
axes[0].scatter(data_kpca_linear[:, 0], data_kpca_linear[:, 1], color='blue', edgecolors='k', alpha=0.7)
axes[0].set_title('KernelPCA avec noyau linéaire')
axes[0].set_xlabel('1ère composante principale')
axes[0].set_ylabel('2ème composante principale')

# Noyau RBF
axes[1].scatter(data_kpca_rbf[:, 0], data_kpca_rbf[:, 1], color='green', edgecolors='k', alpha=0.7)
axes[1].set_title('KernelPCA avec noyau RBF')
axes[1].set_xlabel('1ère composante principale')
axes[1].set_ylabel('2ème composante principale')

# Noyau polynomial
axes[2].scatter(data_kpca_poly[:, 0], data_kpca_poly[:, 1], color='red', edgecolors='k', alpha=0.7)
axes[2].set_title('KernelPCA avec noyau polynomial')
axes[2].set_xlabel('1ère composante principale')
axes[2].set_ylabel('2ème composante principale')

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



Explication du code :

1. **Standardisation des données** : Les données sont standardisées avant l'application de KernelPCA à l'aide de StandardScaler. La standardisation est importante car KernelPCA est sensible à l'échelle des variables. Elle garantit que chaque variable a une moyenne de zéro et une variance de 1.
2. **Application de KernelPCA** :
   - **Noyau linéaire** : L'ACP classique est une version spécifique de KernelPCA avec un noyau linéaire. Cela fonctionne de manière similaire à l'ACP, mais dans un espace de caractéristiques non linéaire.
   - **Noyau RBF** : Le noyau RBF est souvent utilisé pour traiter des relations non linéaires complexes. Il est contrôlé par le paramètre gamma.
   - **Noyau polynomial** : Le noyau polynomial est contrôlé par le paramètre degree, qui détermine le degré du polynôme.
3. **Visualisation des résultats** : Les résultats projetés sur les deux premières composantes principales sont visualisés sous forme de graphiques pour chaque noyau. Cela nous permet de comparer visuellement comment chaque noyau modifie la structure des données.

Choix du noyau et comparaison des résultats :

- Si les données montrent une séparation non linéaire claire sur un noyau **RBF** ou **polynomial**, il est préférable d'opter pour l'un de ces noyaux.

- Le noyau **linéaire** est utile lorsque les données sont déjà bien séparées ou linéairement séparables.

Conclusion :

- **KernelPCA** permet de traiter des données non linéaires en utilisant des noyaux comme le noyau RBF ou polynomial. Le choix du noyau dépend de la nature des données et du problème à résoudre.
- En visualisant les projections des données, on peut évaluer quel noyau offre la meilleure séparation des données et choisissez celui qui convient le mieux à vos besoins.

Comparez les résultats avec l'ACP classique. Dans quels cas l'ACP à noyau donne-t-elle de meilleurs résultats ?

Comparaison entre ACP classique et KernelPCA :

L'ACP classique et l'ACP à noyau (KernelPCA) sont des méthodes de réduction de dimensionnalité, mais elles fonctionnent de manière différente et sont adaptées à des types de données différents.

- ACP classique (Linear PCA) suppose que les données sont linéaires ou peuvent être projetées linéairement dans un espace de plus grande dimension pour être séparées. Elle est efficace lorsque les données sont déjà dans un espace linéaire où les différentes directions de variance sont bien séparées.
- KernelPCA utilise un noyau pour projeter les données dans un espace de dimension beaucoup plus grande (souvent infini pour des noyaux comme le RBF). Cela permet de détecter des structures non linéaires dans les données et d'effectuer une réduction de dimensionnalité même lorsque les relations entre les données ne sont pas linéaires.

KernelPCA est plus efficace :

1. Relations non linéaires dans les données :

- **ACP classique** peut échouer lorsqu'il y a des relations non linéaires entre les variables. Cela se produit lorsque les données ne peuvent pas être séparées ou projetées efficacement sur des composantes principales linéaires.
- **KernelPCA** avec un noyau RBF (ou un noyau polynomial) permet de capturer ces relations non linéaires et de projeter les données dans un espace où elles peuvent être séparées plus facilement.

2. Séparation complexe des données :

- Lorsqu'il y a une structure complexe dans les données (par exemple, des cercles concentriques, des structures en forme de spirale, etc.), l'ACP classique ne peut

pas identifier des directions de séparation efficaces dans l'espace des données d'origine.
- En revanche, KernelPCA avec des noyaux comme RBF ou polynomial permet de mieux identifier ces structures et de rendre les données plus séparables dans un espace de caractéristiques transformé.

3. ***Cas de données avec des anomalies ou des distributions complexes*** *:*

- Si les données sont distribuées de manière non uniforme ou présentent des motifs de distribution complexes, l'ACP classique pourrait ne pas réussir à capter efficacement les structures importantes.
- KernelPCA peut mieux traiter ces types de distributions et extraire des composantes principales qui capturent mieux la variance non linéaire.

ACP classique est plus efficace :

1. ***Données linéaires*** *:*

- Si les données sont linéairement séparables ou si les relations entre les variables sont essentiellement linéaires, l'ACP classique est plus efficace et simple à mettre en œuvre. Elle réduit bien la dimensionnalité tout en capturant la variance dans les données.

2. ***Moins de calculs*** *:*

- ACP classique est plus rapide et nécessite moins de ressources computationnelles que KernelPCA, surtout pour de grandes dimensions. Si les données ne nécessitent pas un noyau non linéaire pour la séparation, l'ACP classique est plus avantageuse en termes de performance.

3. ***Interprétabilité*** *:*

- ACP classique est plus facile à interpréter que KernelPCA. Les composantes principales obtenues sont des combinaisons linéaires des variables d'origine, ce qui les rend interprétables. Par contre, KernelPCA crée une transformation non linéaire des données, ce qui rend l'interprétation plus difficile, car les nouvelles composantes principales ne sont pas des combinaisons simples des variables d'origine.

Conclusion :

- **ACP classique** est très utile lorsque les relations dans les données sont linéaires et lorsque l'on cherche une méthode simple et rapide.

- **KernelPCA** est préféré lorsque les données contiennent des structures non linéaires complexes, car les noyaux permettent de capturer des relations qui ne sont pas visibles dans l'espace d'origine.
- **KernelPCA** est plus flexible mais nécessite des ressources computationnelles plus importantes, et la sélection du noyau et des paramètres (comme gamma pour le noyau RBF) peut nécessiter un ajustement soigneux pour de bons résultats.

En résumé, **KernelPCA** donne de meilleurs résultats lorsque les données présentent des structures complexes et non linéaires, tandis que **ACP classique** reste plus efficace pour des données qui peuvent être modélisées de manière linéaire.

Partie 3 : Méthodes d'ensemble

Exercice 6 : Bagging

Implémentez une forêt aléatoire (RandomForestClassifier) pour prédire la survie des poulets. Analysez les performances (accuracy, F1-score)

Implémentation d'une Forêt Aléatoire (RandomForestClassifier) pour prédire la survie des poulets

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score
from sklearn.preprocessing import StandardScaler

# Création des données avec des variables : poids, nourriture, température
data = pd.DataFrame({
    'poids': np.random.normal(2.5, 0.5, 100),
    'nourriture': np.random.normal(1.2, 0.3, 100),
    'température': np.random.normal(25, 2, 100)
})

# Génération d'une variable "survie" aléatoire : supposons que les poulets avec un poids > 2.5 et une température entre 24 et 26
survie = ((data['poids'] > 2.5) & (data['température'] > 24) & (data['température'] < 26)).astype(int)

# Ajout de la variable "survie" dans le DataFrame
data['survie'] = survie

# Affichage des premières lignes pour vérifier
print(data.head())
```

```
      poids  nourriture  température  survie
0  2.878494    1.043183    26.876568       0
1  2.038917    1.514703    23.967911       0
2  2.934803    0.988697    25.192242       1
3  3.177819    0.777462    24.075449       1
4  2.706717    0.733011    24.131008       1
```

```python
# Séparer les variables explicatives (X) et la variable cible (y)
X = data.drop(columns=['survie'])
y = data['survie']

# Normaliser les données si nécessaire (par exemple pour la température, poids, etc.)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Diviser les données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)

# Initialisation du modèle Random Forest
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)

# Entraînement du modèle sur les données d'entraînement
random_forest.fit(X_train, y_train)

# Prédictions sur l'ensemble de test
y_pred = random_forest.predict(X_test)

# Calcul de l'accuracy et du F1-score
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Affichage des résultats
print(f"Accuracy: {accuracy:.4f}")
print(f"F1-score: {f1:.4f}")
```

```
Accuracy: 0.9333
F1-score: 0.8333
```

Analyse des résultats

- Accuracy : cette valeur est élevée, le modèle est bien : puisque l'accuracy est élevée et le F1-score également, cela signifie que le modèle fait un bon travail de prédiction de la survie des poulets.

Identifiez les variables les plus importantes. Quels attributs influencent le plus la survie des poulets ? Pourquoi

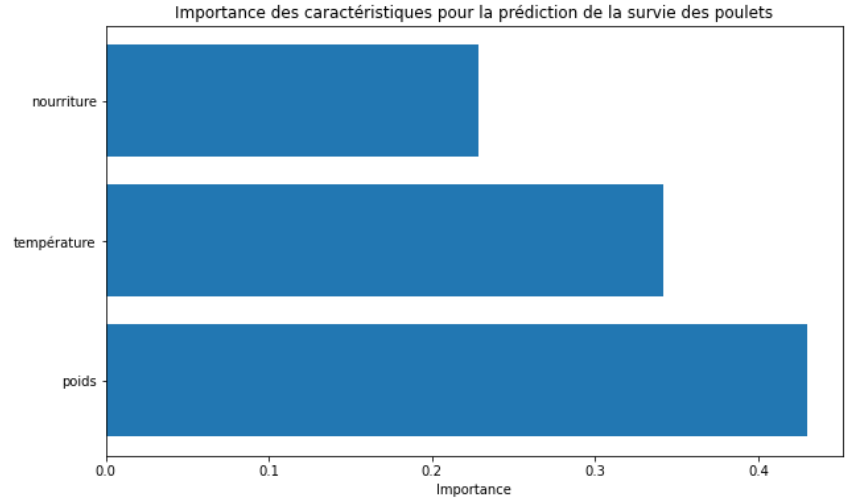
Identification des variables les plus importantes avec RandomForestClassifier

 Extraction des importances des caractéristiques : Après avoir entraîné le modèle **Random Forest**, nous pouvons accéder à l'attribut feature_importances_ du modèle, qui contient un score pour chaque variable.

 Visualisation des importances : Pour mieux comprendre quelles variables sont les plus influentes, nous pouvons créer un graphique (par exemple, un diagramme à barres) des importances.

```python
import matplotlib.pyplot as plt

# Obtenir l'importance des caractéristiques
importances = random_forest.feature_importances_

# Créer un DataFrame pour associer les importances aux variables
feature_importance_df = pd.DataFrame({
    'Variable': X.columns,
    'Importance': importances
})

# Trier les variables par importance
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)

# Affichage des importances des caractéristiques
print(feature_importance_df)

# Visualisation des importances des caractéristiques
plt.figure(figsize=(10, 6))
plt.barh(feature_importance_df['Variable'], feature_importance_df['Importance'])
plt.xlabel('Importance')
plt.title('Importance des caractéristiques pour la prédiction de la survie des poulets')
plt.show()
```

```
      Variable  Importance
0        poids    0.430147
2  température    0.341491
1   nourriture    0.228362
```



Explication du code :

Explication des résultats

- **Importance des caractéristiques** : Chaque caractéristique du modèle (**poids**, **nourriture**, **température**) se voit attribuer un score d'importance. Plus ce score est élevé, plus la caractéristique contribue à la prise de décision du modèle.
- **Variables influentes** : En fonction des scores, on peut identifier quelle variable a le plus grand impact sur la prédiction de la **survie des poulets**. Par exemple, si le **poids** a une importance élevée, cela suggère que le poids des poulets joue un rôle crucial dans leur survie. Si **température** est également importante, cela pourrait

indiquer que des conditions environnementales (comme la température) affectent fortement la survie des poulets.

Interprétation des résultats

Poids : on remarque que le poids a une importance élevée, cela signifie que les poulets plus lourds ont de meilleures chances de survie.

Pourquoi ceette variable influencent-elles la survie des poulets ?

Poids : Un poulet plus lourd peut être plus robuste face aux maladies et aux stress environnementaux. De plus, un poids plus élevé est souvent associé à une meilleure croissance et à des systèmes immunitaires plus forts.

Exercice 7 : Boosting

Comparez AdaBoost et Gradient Boosting sur la prédiction du gain de poids. Analysez leurs performances :

nous allons comparer deux algorithmes de **boosting**, **AdaBoost** et **Gradient Boosting**, sur la prédiction du **gain de poids** des poulets. Ces deux algorithmes appartiennent à la même famille, mais ils fonctionnent de manière légèrement différente.

AdaBoost (Adaptive Boosting)

AdaBoost fonctionne en ajustant le poids des échantillons mal classés à chaque itération. Il combine plusieurs modèles faibles (généralement des arbres de décision peu profonds) en un modèle puissant. Lors de chaque itération, AdaBoost tente de corriger les erreurs commises par les itérations précédentes.

Gradient Boosting

Le Gradient Boosting, en revanche, construit des modèles de manière séquentielle, en optimisant la fonction de perte avec une descente de gradient. Chaque modèle apprend à corriger les erreurs (résidus) du modèle précédent. Le Gradient Boosting est souvent plus performant que AdaBoost, surtout dans des tâches complexes.

Comparaison des performances

Nous allons comparer les deux algorithmes sur la tâche de prédiction du gain de poids des poulets. Pour cela, nous allons utiliser un **RandomForestClassifier** pour générer des prédictions, mais avec AdaBoost et Gradient Boosting. Nous analyserons les performances de chaque modèle en utilisant les métriques d'**accuracy** et de **F1-score**.

```
from sklearn.ensemble import AdaBoostRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# Supposons que vous avez un jeu de données avec une colonne 'gain_poids' (le gain de poids des poulets)
# Utilisons un jeu de données factice avec "gain_poids" comme cible.

# Création des données (en supposant que le gain de poids dépend du poids, nourriture et température)
data = pd.DataFrame({
    'poids': np.random.normal(2.5, 0.5, 100),
    'nourriture': np.random.normal(1.2, 0.3, 100),
    'température': np.random.normal(25, 2, 100)
})

# Générer un gain de poids aléatoire en fonction des caractéristiques (poids, nourriture, température)
data['gain_poids'] = 0.5 * data['poids'] + 0.3 * data['nourriture'] + np.random.normal(0, 0.2, 100)

# Séparation des caractéristiques (X) et de la cible (y)
X = data.drop(columns=['gain_poids'])
y = data['gain_poids']

# Division des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialisation des modèles AdaBoost et Gradient Boosting
adaboost = AdaBoostRegressor(n_estimators=50, random_state=42)
gradient_boosting = GradientBoostingRegressor(n_estimators=100, random_state=42)

# Entraînement des modèles
adaboost.fit(X_train, y_train)
gradient_boosting.fit(X_train, y_train)

# Prédictions sur l'ensemble de test
y_pred_adaboost = adaboost.predict(X_test)
y_pred_gradient = gradient_boosting.predict(X_test)

# Calcul des performances (Mean Squared Error)
mse_adaboost = mean_squared_error(y_test, y_pred_adaboost)
mse_gradient = mean_squared_error(y_test, y_pred_gradient)

# Affichage des résultats
print(f"MSE d'AdaBoost : {mse_adaboost:.4f}")
print(f"MSE de Gradient Boosting : {mse_gradient:.4f}")
```

```
MSE d'AdaBoost : 0.0543
MSE de Gradient Boosting : 0.0469
```

Gradient Boosting a un MSE plus faible, cela indique qu'il a mieux appris les relations complexes dans les données, car il optimise directement les erreurs résiduelles**.**

Les deux algorithmes réagissent-ils différemment aux outliers ? Expliquez pourquoi :

Réaction des algorithmes aux outliers

Les **outliers** (valeurs aberrantes) peuvent avoir un impact significatif sur la performance des algorithmes de machine learning. Les deux algorithmes de boosting (AdaBoost et Gradient Boosting) réagissent différemment aux outliers en raison de la manière dont ils ajustent leurs modèles.

AdaBoost et les outliers

- Sensibilité aux outliers : AdaBoost est très sensible aux outliers. Étant donné qu'il donne plus de poids aux échantillons mal classés, les outliers, qui sont souvent mal classés, vont recevoir un poids important à chaque itération. Cela peut perturber l'apprentissage du modèle, surtout lorsque les outliers représentent une petite proportion des données, mais ont un impact disproportionné.
- Impact : Les outliers peuvent entraîner une sur-adaptation aux erreurs, ce qui peut dégrader la performance globale d'AdaBoost.

Gradient Boosting et les outliers

- Sensibilité aux outliers : Gradient Boosting est également sensible aux outliers, mais il est généralement plus robuste que AdaBoost. Cela est dû au fait que Gradient Boosting apprend en ajustant les résidus (erreurs) à chaque étape. Si un outlier est présent, il est pris en compte dans la minimisation des erreurs globales, mais l'impact d'un seul outlier est plus limité grâce à la descente de gradient qui permet une régularisation plus subtile.
- Impact : Bien que Gradient Boosting soit plus robuste, il peut encore être affecté par des outliers importants. Cependant, il a des techniques de régularisation comme l'arrêt précoce ou la réduction de la profondeur des arbres qui peuvent aider à limiter l'impact des outliers.

Conclusion

- AdaBoost est plus sensible aux outliers que Gradient Boosting, car il ajuste les poids des erreurs à chaque itération et peut donner un poids disproportionné aux outliers.
- Gradient Boosting est plus robuste grâce à l'optimisation des erreurs résiduelles et des techniques de régularisation, mais il peut encore être influencé par des outliers extrêmes, surtout si le modèle n'est pas correctement réglé.