TP2

April 2, 2025

1 TP2 : Optimisation Bayésienne et Modèles Bayésiens à Noyau

[12]:
```
# Chargement des bibliothèques
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from scipy.stats import zscore
from skopt import gp_minimize
from skopt.space import Real, Integer
from sklearn.ensemble import RandomForestRegressor
from skopt.plots import plot_convergence
from sklearn.model_selection import train_test_split, GridSearchCV,␣
↪RandomizedSearchCV
from skopt import BayesSearchCV
from sklearn.gaussian_process import GaussianProcessRegressor,␣
↪GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, DotProduct
from sklearn.metrics import classification_report, accuracy_score
from sklearn.svm import SVC
from sklearn.preprocessing import PolynomialFeatures
```

1.1 Partie 1 : Optimisation Bayésienne

1.1.1 Fondements théoriques

Le principe de l'optimisation bayésienne : Principalement utilisée pour l'optimisation des hyperparamètres des modèles de l'apprentissage automatique. En effet, lorsqu'on cherche à trouver le meilleur modèle, il faut tester plusieurs hyperparamètres, effectuer plusieurs entraînements et validations, ce qui est coûteux en temps et en ressources. L'optimisation bayésienne a été donc mise en place pour éviter ces contraintes. Elle découle du théorème de Bayes, elle utilise un modèle probabiliste (processus gaussien) pour estimer la relation entre les hyperparamètres et les performances du modèle. Pour ce faire, il faudrait identifier la valeur la plus probable et la dispersion probable autour de cette moyenne tout en séléction les observations à évaluer (optimisation) en utilisant une fonction d'acquisition qui choisit la prochains hyperparamètres à tester. Le modèle est entraîné par la suite avec ces hyperparamètres et le processus est répété jusqu'à ce que le modèle converge et donc atteint la meilleure performance.

L'optimisation bayésienne permet de gérer les fonctions coûteuses en minimisant le nombre d'évaluations qui nécessitent beaucoup de temps et de ressources. Plutôt que tester toutes les observations, on construit un modèle probabiliste qui calcule la moyenne estimée de la fonction et une incertitude sur cette estimation, la fonction d'acquisition décide par la suite les potentielles observations à tester, le modèle probabiliste est mis à jour au fur et à mesure pour trouver les observations les plus prometteuses sans tester la totalité et éviter d'épuiser les ressouces.

Le processus gaussien : Il repose sur l'hypothèse que les données suivent une distribution gaussienne. Dans le cadre de la prédiction, la formule de Bayes sur laquelle repose ce processus permet de calculer la probabilité à posteriori d'un évènement en combinant les informations à priori et celles apportées par les données observées et cela en combinant la moyenne et la covariance à priori, le processus gaussien génère une distribution de probabilité sur les possible fonctions.

En apprentissage automatique, la fonction objectif étant de trouver les meilleurs paramètres pour un modèle performant, étant couteux d'évaluer toutes les observations avec plusieurs paramètres, on utilise l'optimisation bayésienne avec le processus gaussien qui optimise cette fonction objectif (test des hyperparamètres par exemple).
En comparaison avec les autres technique de recherche d'optimisation de paramètres comme Grid-Search qui effectue des tests exhaustifs sur toutes les solution afin de trouver les meilleurs paramètres qui s'avère très long et couteux, le processus bayésien sélectionne les plus prometteuses à tester et donc effectue le minimum de tests possible.

Les fonctions d'acquisition : Dans l'optimisation bayésienne, la fonction d'acquisition décide de la prochaine combinaison d'hyperparamètres à tester.
Expected Improvement : choisit les points avec la plus grande amélioration attendue.
Upper Confidence Bound : favorise les zones avec une haute incertitude.
Probability of Improvement : sélectionne les points qui ont une forte probabilité d'amélioration.
Cette fonction d'acquisition équilibre entre exploration et exploitation :
Exploitation : tester des valeurs proches de celles qui ont déjà donné de bons résultats.
Exploration : tester de nouvelles zones avec un fort potentiel (grande incertitude).

1.1.2 Implémentation et applications

[3]:
```
# Chargement des données
data = pd.read_csv(r'tp2_atdn_donnees.csv')
data.head()
```

[3]:

	Humidité (%)	Température (°C)	pH du sol	Précipitations (mm)	Type de sol	Rendement agricole (t/ha)
0	52.472407	27.454043	6.055399	179.770446	Limoneux	7.038885
1	87.042858	23.402409	7.125703	169.795469	Limoneux	7.712547
2	73.919637	17.738190	8.118838	56.410516	Limoneux	6.587578
3	65.919509	30.344875	7.696675	135.311957	Sableux	
4	39.361118	27.118279	7.919683	145.048905	Sableux	

```
3                    7.907268
4                    6.889830
```

Analyse exploratoire :

[16]:
```
print("Statistiques :\n", data.describe())
```

```
Statistiques :
        Humidité (%)  Température (°C)   pH du sol  Précipitations (mm)  \
count     500.000000        500.000000  500.000000           500.000000
mean       59.913703         22.048785    7.052674           174.119122
std        17.921305          7.137336    0.891579            71.752464
min        30.303695         10.115801    5.514820            50.804566
25%        44.476781         15.727481    6.223684           110.268568
50%        60.789825         21.795539    7.119215           177.222834
75%        75.367493         28.158421    7.832031           234.344063
max        89.577888         34.992942    8.498241           299.586878

       Rendement agricole (t/ha)
count                 500.000000
mean                    6.758773
std                     1.207358
min                     3.032377
25%                     5.969749
50%                     6.781866
75%                     7.635114
max                    10.025551
```

Analyse statistique : Les moyennes des facteurs sont globalement cohérentes et semblent centrales par rapport aux médianes. Cela peut signifier que les distributions sont symétriques ##### Interprétation gloable : Les données montrent une forte dépendance du rendement agricole aux facteurs environnementaux : les zones avec des températures ou des précipitations extrêmes peuvent avoir des rendements plus faibles, le pH du sol et l'humidité semblent globalement favorables.

[14]:
```
# Vérification des valeurs manquantes
print("Valeurs manquantes : \n", data.isnull().sum())
```

```
Valeurs manquantes :
 Humidité (%)                 0
Température (°C)              0
pH du sol                     0
Précipitations (mm)           0
Type de sol                   0
Rendement agricole (t/ha)     0
dtype: int64
```

[20]:
```
# Vérification des doublons
print(f"Doublons : {data.duplicated().sum()}")
```

```
Doublons : 0
```

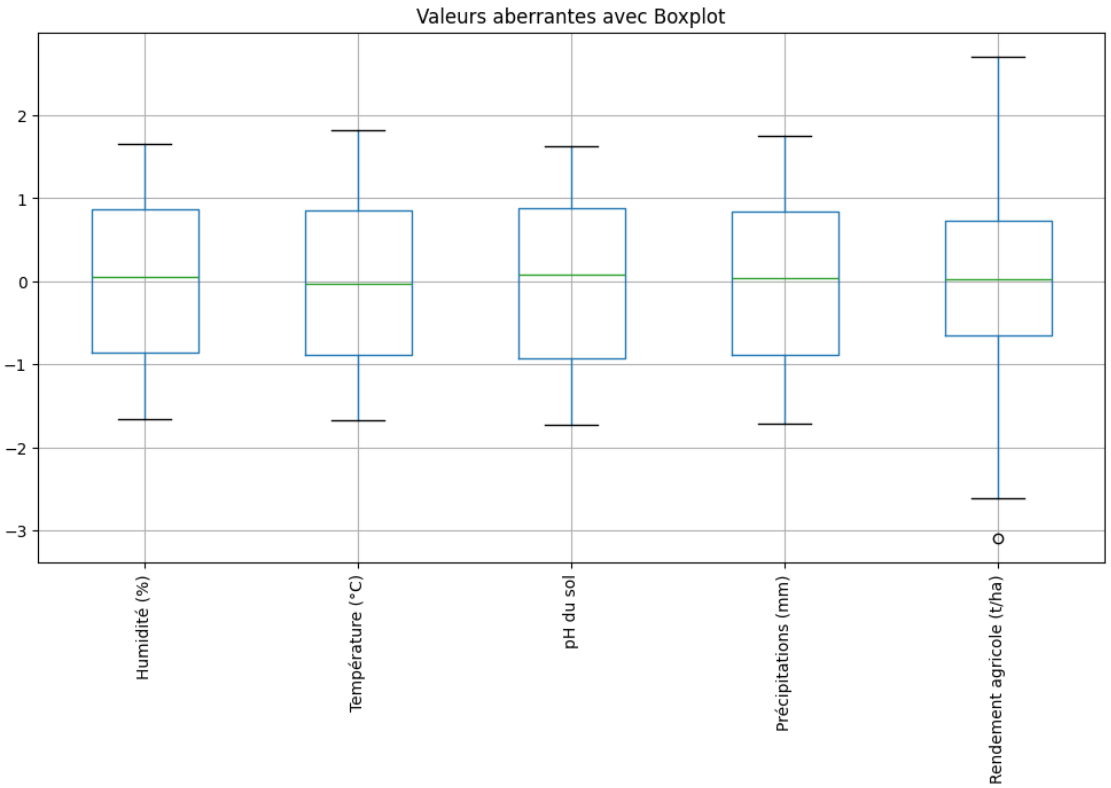
[4]:
```
# Normalisation et standardisation afin de mettre les variables sur la même␣
 ↪échelle
scaler = StandardScaler()
data[['Humidité (%)','Température (°C)','pH du sol','Précipitations␣
 ↪(mm)','Rendement agricole (t/ha)']] = scaler.fit_transform(data[['Humidité␣
 ↪(%)','Température (°C)','pH du sol','Précipitations (mm)','Rendement␣
 ↪agricole (t/ha)']])
data.head()
```

[4]:

	Humidité (%)	Température (°C)	pH du sol	Précipitations (mm)	Type de sol	\
0	-0.415636	0.758080	-1.119670	0.078840	Limoneux	
1	1.515310	0.189844	0.081991	-0.060318	Limoneux	
2	0.782307	-0.604555	1.197012	-1.642125	Limoneux	
3	0.335457	1.163515	0.723038	-0.541390	Sableux	
4	-1.147973	0.710990	0.973416	-0.405552	Sableux	

	Rendement agricole (t/ha)
0	0.232236
1	0.790759
2	-0.141936
3	0.952199
4	0.108657

[52]:
```
# Valeurs aberrantes
# Tracer les boxplots pour chaque variable numérique
plt.figure(figsize=(12, 6))
data.boxplot()
plt.xticks(rotation=90)
plt.title("Valeurs aberrantes avec Boxplot")
plt.show()
```



On remarque l'existence d'un outlier dans les valeurs du rendement.

[74]:
```
# Calcul du Z-score pour rendement
data["z_score_rendement"] = zscore(data["Rendement agricole (t/ha)"])

# Détection (Z-score > 3 ou < -3)
out = data[(data["z_score_rendement"] > 3) | (data["z_score_rendement"] < -3)]

print(out[["Rendement agricole (t/ha)"]])
```

```
     Rendement agricole (t/ha)
294                  -3.089498
```

cet outlier est dans la ligne 294 et a la valeur -3.089498

[80]:
```
print("valeurs rendement \n",data)
print("outlier\n",data.iloc[294])
```

```
valeurs rendement
      Humidité (%)  Température (°C)  pH du sol  Précipitations (mm)  \
0       -0.415636          0.758080  -1.119670             0.078840
1        1.515310          0.189844   0.081991            -0.060318
2        0.782307         -0.604555   1.197012            -1.642125
3        0.335457          1.163515   0.723038            -0.541390
```

```
4       -1.147973          0.710990   0.973416           -0.405552
..            …                …          …                   …
495     -0.486644         -1.368719   0.507433            0.559700
496      0.285179          1.526474   0.343327            1.604819
497     -1.410328         -1.210110  -0.182098           -1.491052
498      1.594671          1.641911  -0.464043           -1.532567
499      1.634270         -0.126033   1.164637           -0.747375

    Type de sol  Rendement agricole (t/ha)  z_score_rendement
0      Limoneux                   0.232236           0.232236
1      Limoneux                   0.790759           0.790759
2      Limoneux                  -0.141936          -0.141936
3       Sableux                   0.952199           0.952199
4       Sableux                   0.108657           0.108657
..          …                          …                  …
495    Limoneux                  -0.411246          -0.411246
496    Argileux                   1.728026           1.728026
497    Argileux                  -2.041613          -2.041613
498     Sableux                   0.681695           0.681695
499    Limoneux                   1.101855           1.101855

[500 rows x 7 columns]
outlier
 Humidité (%)                -1.245021
Température (°C)            -1.626327
pH du sol                    0.953293
Précipitations (mm)         -1.586214
Type de sol                  Argileux
Rendement agricole (t/ha)   -3.089498
z_score_rendement           -3.089498
Name: 294, dtype: object
```

Après vérification et comparaison avec les autres valeurs des facteurs dans le dataset, on a constaté que ette valeur pourrait refléter un rendement exceptionnellement bas dû à des conditions défavorables, précipitations faibles (un manque d'eau peut expliquer ce rendement bas, surtout dans un sol argileux connu pour retenir l'humidité).
On peut garder cette valeur afin de tester la capacité de notre modèle à gérer des valeurs extrêmes et à prédire des rendements dans des conditions variées, ainsi tester la capacité du modèle bayésien à gérer l'incertitude.

[5]:
```
# Encodage des données catégorique
label_encoder = LabelEncoder()
data["Type de sol"] = label_encoder.fit_transform(data["Type de sol"])
# Limoneux=1, Sableux=2, Argileux=0
```

[133]:
```
data.head()
```

[133]:

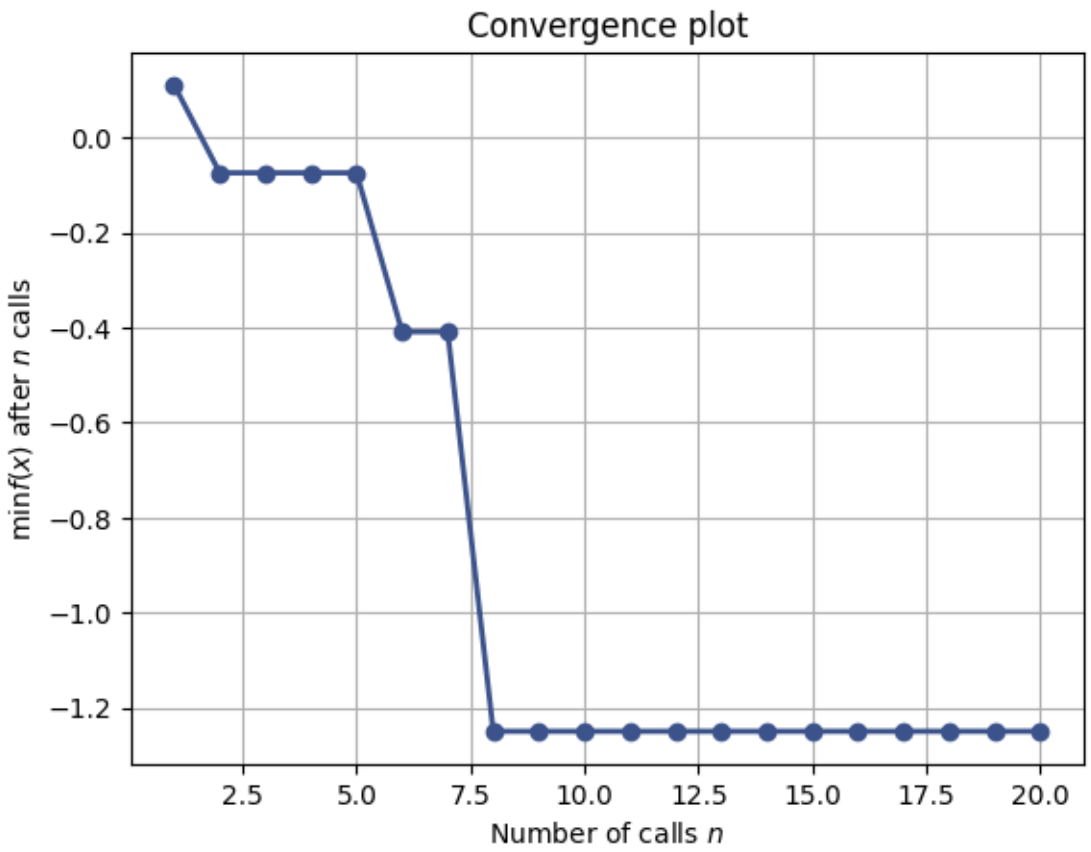
	Humidité (%)	Température (°C)	pH du sol	Précipitations (mm)	\
0	-0.415636	0.758080	-1.119670	0.078840	
1	1.515310	0.189844	0.081991	-0.060318	
2	0.782307	-0.604555	1.197012	-1.642125	
3	0.335457	1.163515	0.723038	-0.541390	
4	-1.147973	0.710990	0.973416	-0.405552	

	Type de sol	Rendement agricole (t/ha)
0	1	0.232236
1	1	0.790759
2	1	-0.141936
3	2	0.952199
4	2	0.108657

Implémentation d'une optimisation bayésienne :

[106]:
```
X = data[['Humidité (%)', 'Température (°C)']].values
y = data['Rendement agricole (t/ha)'].values

#Espace de recherche
space = [Real(data["Humidité (%)"].min(), data["Humidité (%)"].max(),␣
 ↪name='humidité'),
       Real(data["Température (°C)"].min(), data["Température (°C)"].max(),␣
 ↪name='température')]

# Fonction objectif à minimiser (pour maximiser le rendement)
def fct_obj(params):
    h, t = params
    rf = RandomForestRegressor(random_state=42)
    rf.fit(X, y)
    rendement = rf.predict([[h, t]])[0]
    return -rendement


res = gp_minimize(fct_obj, space, n_calls=20, random_state=42)

plot_convergence(res)
plt.show()
```

```
C:\Users\samia\AppData\Local\Programs\Python\Python312\Lib\site-
packages\skopt\optimizer\optimizer.py:517: UserWarning: The objective has been
evaluated at point [1.6569047069613854, 1.8154000051638077] before, using random
point [-1.0662859135218197, 0.7042186965374062]
  warnings.warn(
```



[107]:
```
print(f"Meilleure combinaison : Humidité={res.x[0]:.2f}, Température={res.x[1]:.
↪2f}")
```

```
Meilleure combinaison : Humidité=1.63, Température=0.48
```

Ces valeurs (normalisées) correspondent à la meilleure combinaison pour maximiser le rendement. D'après le graphique, l'algorithme évalue la fonction objectif plus de 7 fois en ajustant les paramètres pour converger et trouver la meilleure combinaison.

Ajuster les hyperparamètres d'un modèle de régression avec l'optimisation bayésienne :

[122]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

param_space = {
    'n_estimators': (10, 200),  # Nombre d'arbres
    'max_depth': (1, 20),       # Profondeur maximale
    'min_samples_split': (2, 10),  # Nombre minimum d'échantillons nécessaires␣
↪pour diviser un nœud
```

```
    'min_samples_leaf': (1, 10)   # Nombre minimum d'échantillons nécessaires à␣
 ↪chaque feuille
}
rf = RandomForestRegressor(random_state=42)

opt = BayesSearchCV(
    estimator=rf,
    search_spaces=param_space,
    n_iter=100,  # Nombre d'itérations
    cv=5,        # Validation croisée
    random_state=42
)

opt.fit(X_train, y_train)

# Meilleurs hyperparamètres trouvés
print("Meilleurs paramètres trouvés :", opt.best_params_)

# Évaluer sur l'ensemble de test
test_score = opt.score(X_test, y_test)
print(f"Score sur le test : {test_score:.4f}")
```

```
Meilleurs paramètres trouvés : OrderedDict({'max_depth': 4, 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 200})
Score sur le test : 0.3068
```

[115]:
```
#Gridsearch
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [5, 10, 15],
}
grid_search = GridSearchCV(
    estimator=RandomForestRegressor(random_state=42),
    param_grid=param_grid,
    cv=5,
    scoring='neg_mean_squared_error'
)
grid_search.fit(X_train, y_train)
print("Meilleurs paramètres :", grid_search.best_params_)
# Évaluer sur l'ensemble de test
test_score_G = grid_search.score(X_test, y_test)
print(f"Score sur le test : {test_score_G:.4f}")
```

```
Meilleurs paramètres : {'max_depth': 5, 'n_estimators': 50}
Score sur le test : -0.6186
```

[116]:
```
#RandomSearch
param_distributions = {
```

```
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [5, 10, 15, None],
    'min_samples_split': [2, 5, 10],
}
random_search = RandomizedSearchCV(
    estimator=RandomForestRegressor(random_state=42),
    param_distributions=param_distributions,
    n_iter=20,  # Nombre d'essais aléatoires
    cv=5,
    scoring='neg_mean_squared_error'
)
random_search.fit(X_train, y_train)
print("Meilleurs paramètres :", random_search.best_params_)
# Évaluer sur l'ensemble de test
test_score_R = grid_search.score(X_test, y_test)
print(f"Score sur le test : {test_score_R:.4f}")
```

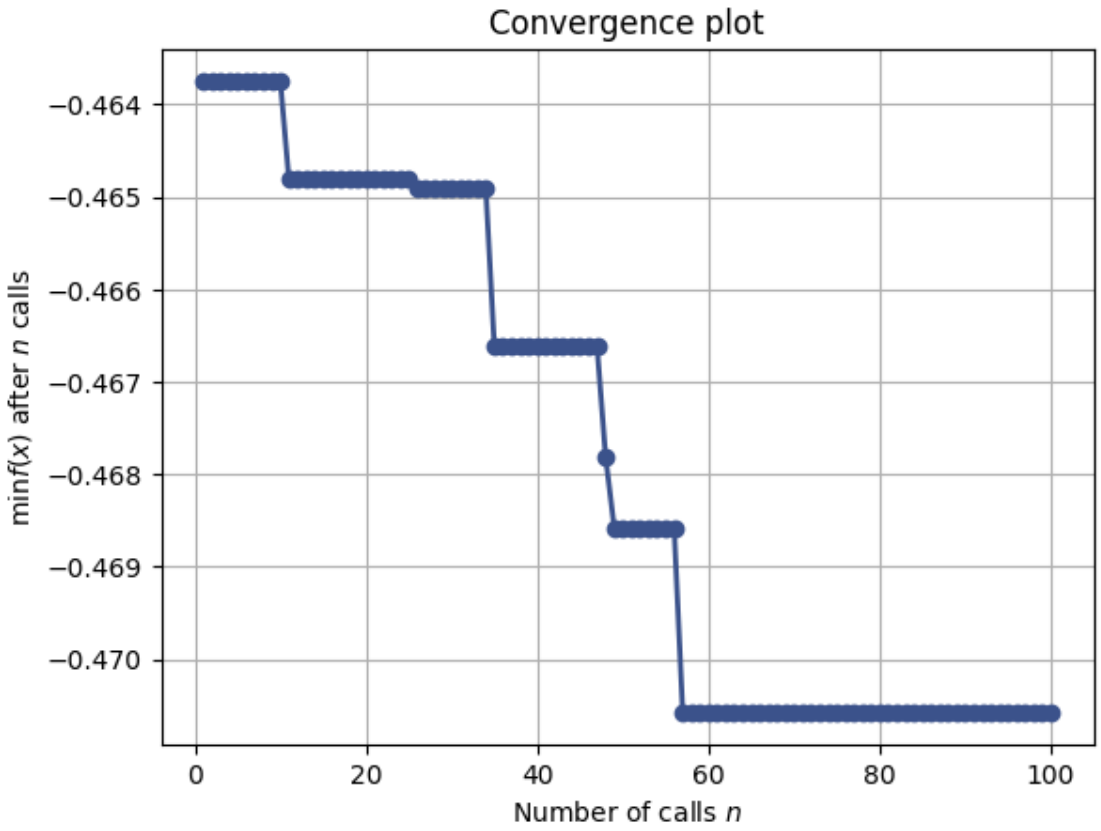
```
Meilleurs paramètres : {'n_estimators': 50, 'min_samples_split': 2, 'max_depth':
5}
Score sur le test : -0.6186
```

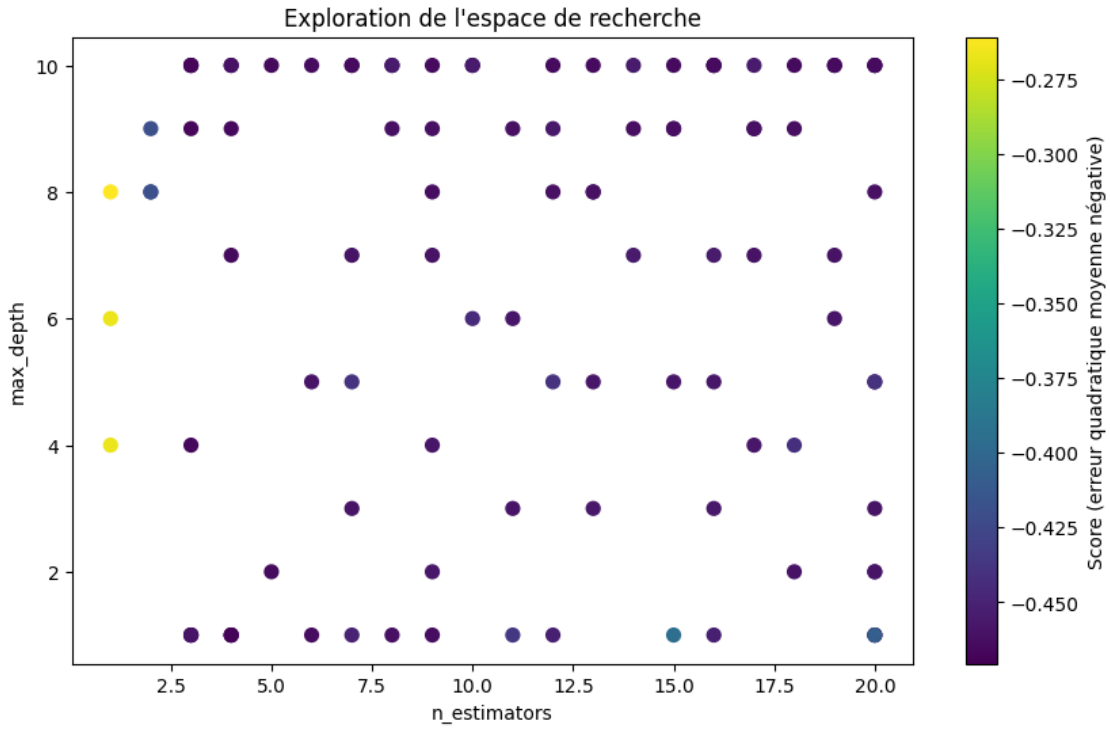
Comparaison entre les 3 méthodes :

- BayesSearchCV a trouvé des paramètres optimaux avec une profondeur de 3 et un grand nombre d'arbres (200), ce qui a conduit à une meilleure performance (score positif). De plus, la validation croisée montre que l'optimisation bayésienne a efficacement équilibré exploration et exploitation pour trouver un ensemble de paramètres performant, en utilisant un processus probabiliste intelligent qui cible des régions prometteuses de l'espace de recherche.
- GridSearch, avec une recherche exhaustive, a trouvé une profondeur plus grande (5) mais un plus petit nombre d'arbres (50), ce qui semble moins adapté, le score négatif montre une moins bonne performance par rapport à Bayes Search.
- RandomSearch a donné des résultats similaires à GridSearch, montrant qu'il a probablement exploré une partie limitée de l'espace de recherche. Cette méthode ne semble pas optimale pour maximiser les performances du modèle.

[123]:
```
#Visualisation du processus d'optimisation
plot_convergence(opt.optimizer_results_[0])
# Extraire les points testés et leurs scores
tested_points = np.array(opt.optimizer_results_[0].x_iters)
scores = np.array(opt.optimizer_results_[0].func_vals)

# Tracer les points testés
plt.figure(figsize=(10, 6))
plt.scatter(tested_points[:, 0], tested_points[:, 1], c=scores, cmap='viridis',␣
 ↪s=50)
plt.colorbar(label="Score (erreur quadratique moyenne négative)")
plt.xlabel("n_estimators")
plt.ylabel("max_depth")
```

```
plt.title("Exploration de l'espace de recherche")
plt.show()
```





En observant la courbe, on constate, après presque 60 appels, le modèle converge et la courbe se stabilise indiquant que le modèle a identifié la région optimale.
Dans le tracé des points, les zones avec de nombreuses tentatives montrent que l'algorithme a exploré en détail ces régions.
Points explorés : Les premiers essais sont répartis aléatoirement dans l'espace de recherche, en s'appuyant sur des estimations probabilistes.
Points exploités : L'algorithme concentre les recherches autour des régions qui montrent de bonnes performances.

Avantages et limites de l'optimisation bayésienne : L'optimisation bayésienne explore l'espace des hyperparamètres de manière intelligente en s'appuyant sur un modèle probabiliste (processus gaussien). Elle ajuste les recherches pour se concentrer sur les régions prometteuses, ce qui réduit le temps total d'évaluation.
Contrairement à Grid Search qui évalue toutes les combinaisons possibles, ou Random Search qui teste un grand nombre de points aléatoires, l'optimisation bayésienne converge plus rapidement vers des solutions optimales avec moins d'essais.
Bien qu'elle réduise les évaluations (appels de la fonction objectif), les calculs pour ajuster le modèle probabiliste (régression) peuvent être coûteux, en particulier avec de nombreux paramètres.

1.2 Partie 2 : Modèles Bayésiens à Noyau

1.2.1 Fondements théoriques

L'inférence bayésienne : c'est une technique d'apprentissage qui utilise les probabilités pour définir et raisonner nos croyances et de mettre à jour ces croyances lorsque de nouvelles observations

sont faites. L'oobjectif étant de trouver un modèle performant qui prédit correctement, nous avons déja une idée sur ce modèle avant même de voir les données, le principe de l'inférence bayésienne consiste à exprimer cette connaissance a priori par une distribution de probabilité sur les modèles possibles, appelée distribution a priori.

On part d'une croyance initiale (à priori) avant de voir les données, on observe les nouvelles données, on met à jour les croyances à priori avec le théorème de Bayes, en multipliant la probabilité à priori par la vraisemblance (la probabilité qu'un modèle obtienne ces données), puis normaliser la distribution à posteriori pour que sa somme soit égale à 1. Ainsi, on obtient une croyance mise à jour à posteriori qui devient l'à priori pour la mise à jour suivante.

La théorie des méthodes à noyau : Afin de modéliser des relations complexe entre les données (relations linéaires) on fait appel à une fonction noyau qui permet de projeter les données dans des dimensions plus élevées sans avoir à les transformer explicitement. Les processus gaussiens utilisent les noyaux pour spécifier la corrélation entre les observations (fonction de covariance).
Il est utile d'utiliser un noyau dans un modèle bayésien afin de capturer les relations complexes entre les observations et donc plus d'informations sur la forme des fonctions à posteriori ainsi qu'en utilisant la covariance définie par le noyau, le modèle peut donner une estimation de l'incertitude associée à chaque prédiction.

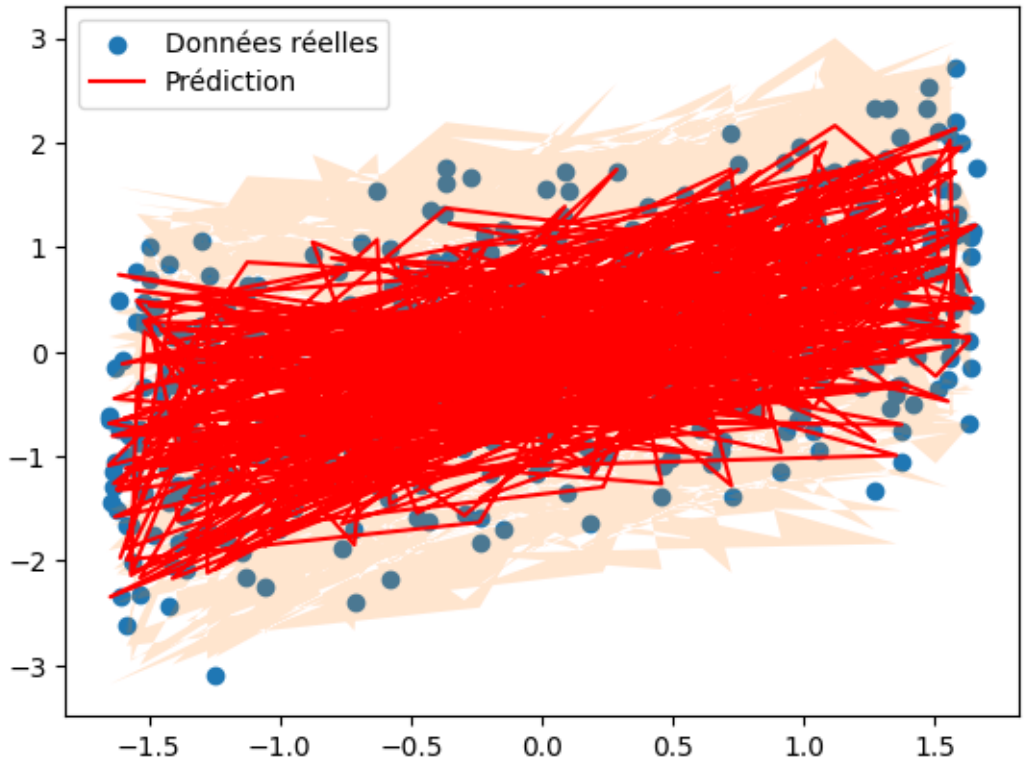
Distibution à priori et à posteriori : Une distribution a priori représente ce que l'on sait ou suppose sur une variable aléatoire avant d'observer les données. Quant à la distribution à posteriori est ce qu'on obtient après la mise à jour des croyances à priori après avoir observé les données et appliqué le théorème de Bayes.
Exemple :
Au départ, on peut supposer que le rendement est normalement distribué autour d'une certaine moyenne. Après la collecte des données réelles, on constate que le rendement est différent, on calcule la vraisemblance (la probabilité d'obtenir ces données en fonction du modèle). Grace au théorème de Bayes, combinant l'à priori et la vraisemblance pour obtenir la distribution à posteriori. Cette nouvelle distribution tient compte à la fois des croyances initiales (à priori) et des données observées.

1.2.2 Implémentation et applications

Implémentation d'une régression bayésienne à noyau :

[152]:
```
X = data[["Humidité (%)", "Température (°C)","pH du sol","Précipitations␣
 ↪(mm)","Type de sol"]].values
y = data["Rendement agricole (t/ha)"].values

kernel = RBF(length_scale=1.0, length_scale_bounds=(1e-4, 10.0)) +␣
 ↪WhiteKernel(noise_level=1e-3)

gpr = GaussianProcessRegressor(kernel=kernel, random_state=42)
gpr.fit(X, y)

y_pred, sigma = gpr.predict(X, return_std=True)
```

[153]:
```
# Visualisation
plt.scatter(X[:, 0], y, label="Données réelles")
plt.plot(X[:, 0], y_pred, color='r', label="Prédiction")
plt.fill_between(X[:, 0], y_pred - 1.96*sigma, y_pred + 1.96*sigma, alpha=0.2)
plt.legend()
plt.show()
```



[16]:
```
X = data[["Humidité (%)", "Température (°C)","pH du sol","Précipitations␣
 ↪(mm)"]].values
y = data["Type de sol"].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

kernel = RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e6)) +␣
 ↪WhiteKernel(noise_level=1e-3)

gpc = GaussianProcessClassifier(kernel=kernel)
gpc.fit(X_train, y_train)

y_pred_gpc = gpc.predict(X_test)
```

```
# Performances
print("Classification bayésienne à noyau :")
print(classification_report(y_test, y_pred_gpc, zero_division=0))
print(f"Précision : {accuracy_score(y_test, y_pred_gpc)}")
```

```
Classification bayésienne à noyau :
              precision    recall  f1-score   support

           0       0.31      1.00      0.47        31
           1       0.00      0.00      0.00        35
           2       0.00      0.00      0.00        34

    accuracy                           0.31       100
   macro avg       0.10      0.33      0.16       100
weighted avg       0.10      0.31      0.15       100

Précision : 0.31
```

[173]:
```
print("Classes réelles :", np.unique(y))
print("Classes prédites :", np.unique(y_pred_gpc))
```

```
Classes réelles : [0 1 2]
Classes prédites : [0]
```

[175]:
```
unique, counts = np.unique(y_train, return_counts=True)
print("Répartition des classes dans l'entraînement :", dict(zip(unique,␣
 ↪counts)))
```

```
Répartition des classes dans l'entraînement : {0: 146, 1: 130, 2: 124}
```

Le résultat montre que le modèle bayésien de classification à noyau n'a prédit qu'une seule classe (0). Cependant les classes ne sont pas déséquilibrées, le modèle n'a pas suffisamment appris à séparer les classes ou les données étant très complexe, le noyau RBF n'est pas adapté, ce qui a entrainé un overfitting de la classe majoritaire (0).

Comparaison avec SVM :

[183]:
```
svm = SVC(kernel="rbf", class_weight="balanced", random_state=42)
svm.fit(X_train, y_train)
y_pred_svm = svm.predict(X_test)

# Performances
print("Classification SVM :")
print(classification_report(y_test, y_pred_svm))
print(f"Précision : {accuracy_score(y_test, y_pred_svm)}")
```

```
Classification SVM :
              precision    recall  f1-score   support
```

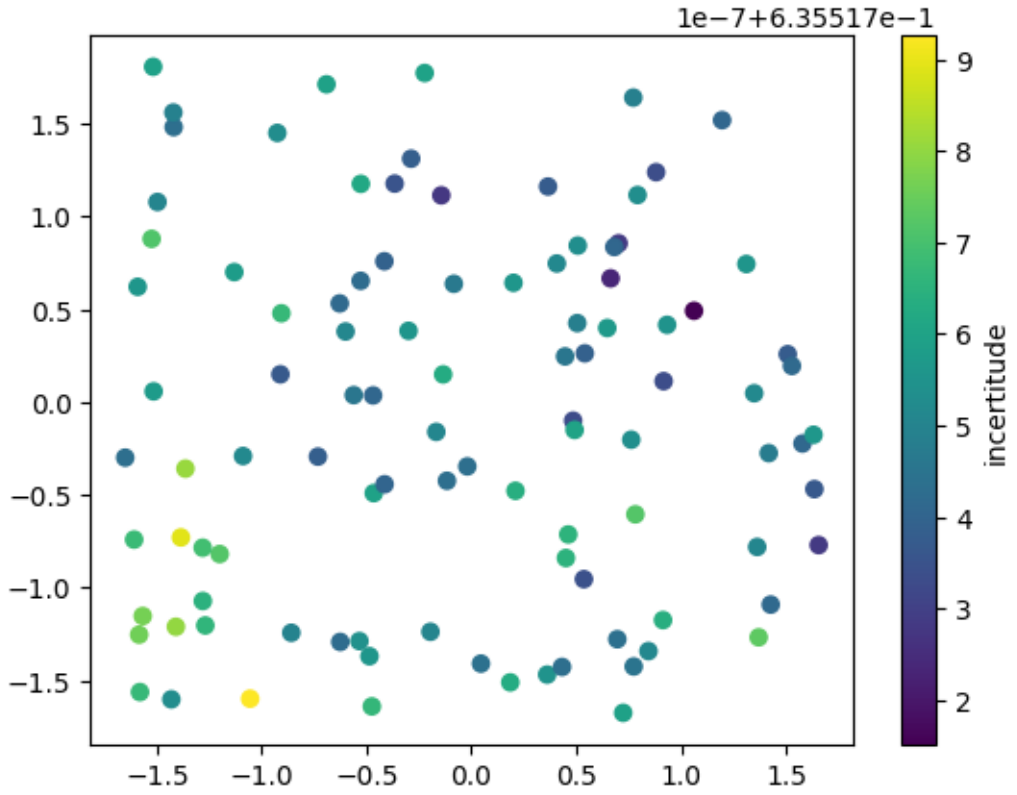
```
              0       0.56      0.45      0.50        31
              1       0.43      0.66      0.52        35
              2       0.38      0.24      0.29        34

    accuracy                           0.45       100
   macro avg       0.46      0.45      0.44       100
weighted avg       0.45      0.45      0.43       100

Précision : 0.45
```

Le SVM est globalement plus équilibré, avec des prédictions sur toutes les classes La classe 1 est relativement mieux prédite avec des limites sur la classe 2. La précision globale (45%) est meilleure que celle de la méthode bayésienne.

Le modèle bayésien souffre d'un biais fort envers la classe 0, le noyau gaussien (RBF) pourrait nécessiter un ajustement supplémentaire pour mieux capturer les caractéristiques des classes.
Le SVM montre une meilleure répartition des prédictions entre les classes, en utilisant class_weight="balanced", le SVM compense automatiquement l'impact des classes majoritaires.

Analyse de l'incertitude dans les prédictions :

[188]:
```
# Probabilités prédictives pour chaque classe
p = gpc.predict_proba(X_test)
# Identifier les classes avec la probabilité la plus basse pour chaque point
incertitude = 1 - np.max(p, axis=1)

plt.scatter(X_test[:, 0], X_test[:, 1], c=incertitude, cmap='viridis',␣
 ↪label="Incertitude")
plt.colorbar(label="incertitude")
plt.show()
```



La couleur des points représente l'incertitude du modèle, allant du bleu (faible incertitude) au jaune (forte incertitude).
Zones à faible incertitude (bleu foncé) :
Ici, le modèle est très confiant dans ses prédictions. Ces points sont probablement proches des données d'entraînement, ce qui permet au modèle d'être sûr des classes prédites.
Zones à forte incertitude (jaune/vert clair) :
Ces régions contiennent les points où le modèle doute, soit parce qu'ils se trouvent à la frontière entre plusieurs classes, soit parce qu'ils sont dans des zones peu représentées par les données d'entraînement.
Ces incertitudes peuvent être réduites en améliorant les hyperparamètres du modèle.

Test de différents noyaux (linéaire, RBF, polynomial) :

[14]:
```
X = data[["Humidité (%)", "Température (°C)", "pH du sol", "Précipitations␣
 ↪(mm)"]].values
y = data["Type de sol"].values

# Séparation 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)

# Définir un noyau linéaire
```

```python
kernel = DotProduct() + WhiteKernel(noise_level=1e-3)

# Création et entraînement du modèle
gpc = GaussianProcessClassifier(kernel=kernel)
gpc.fit(X_train, y_train)

# Prédiction
y_pred_gpc = gpc.predict(X_test)

# Performances
print("Classification bayésienne à noyau linéaire :")
print(classification_report(y_test, y_pred_gpc, zero_division=0))
print(f"Précision : {accuracy_score(y_test, y_pred_gpc):.2f}")
```

```
Classification bayésienne à noyau linéaire :
              precision    recall  f1-score   support

           0       0.32      0.61      0.42        31
           1       0.23      0.17      0.20        35
           2       0.29      0.12      0.17        34

    accuracy                           0.29       100
   macro avg       0.28      0.30      0.26       100
weighted avg       0.28      0.29      0.25       100

Précision : 0.29
```

Différence entre les noyaux : Un noyau linéaire est utilisé lorsqu'on suppose une relation linéaire entre les caractéristiques et la cible. Rapide et efficace pour des données avec des relations linéaires mais moins performant si les données présentent des relations non linéaires complexes. Un noyau RBF modélise des relations non linéaires complexes. Chaque point influence les prédictions localement, en fonction de sa distance aux autres points. Il peut être plus coûteux en termes de calcul, et risque un overfitting avec des petits ensembles de données. Un noyau polynomial permet de modéliser des relations non linéaires dans un cadre polynomiale. Il peut capturer des interactions complexes jusqu'à un degré donné.

Choix du noyau et distribution à priori : Si les relations entre les variables sont complexes et non linéaires, un noyau RBF avec une distribution à priori bien ajustée peut fournir des prédictions robustes. En revanche, un noyau linéaire pourrait suffire si ces relations sont principalement directionnelles et linéaires et les classes sont bien séparables.