Statistique Bayésienne
Practical Bayesian Optimization of Machine Learning Algorithm

Ecrit par

Vincent LE MEUR
&
Timothée Watrigant
&
Thomas Levy

Professeurs référents :

Remi Bardenet
Arnak Dalalyan
Nicolas Chopin

2017

Références

L'article étudié est le suivant : Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical Bayesian optimization of machine learning algorithms". In Advances in neural information processing systems, 2012.

Il est disponible ici : https://arxiv.org/pdf/1206.2944.pdf

Introduction

Le choix des hyper-paramètres d'un modèle de Machine Learning peut être fastidieux et relève plus souvent de l'expérience empirique que d'une méthode exacte.

Cette optimisation peut être vue comme celle d'une fonction inconnue $f(x)$ qui évalue l'efficacité d'un algorithme avec des hyperparamètres x fixés.

L'article propose un algorithme basé sur l'optimisation bayésienne avec un prior Processus Gaussien (GP) pour automatiser le choix optimal des hyper-paramètres. Cet algorithme a été appliqué et validé sur 3 problèmes de Machine Learning.

Nous allons décrire le principe de l'optimisation Bayésienne, puis l'algorithme utilisé par les auteurs de l'étude. Enfin, nous l'avons appliqué à de nouveaux problèmes de machine learning.

1. L'optimisation Bayésienne

L'idée est de considérer que la fonction inconnue $f(x)$ est issue d'un Prior et dont la distribution a posteriori reste cohérente avec ce Prior au fur et à mesure des observations. L'objectif est de trouver le minimum de cette fonction $f : \mathscr{X} \to \mathscr{R}$ où $\mathscr{X}$ désigne l'ensemble dans lequel évolue les hyperparamètres.

Dans notre cas précis, une observation correspond à faire tourner un algorithme de Machine Learning avec un ensemble d'hyperparamètres fixés et d'en évaluer l'efficacité.

L'optimisation bayésienne permet de choisir de manière "optimale" le prochain set d'hyperparamètres à tester. En effet, cette approche intégre l'incertitude sur la fonction f de façon à la réduire lors de la prochaine évaluation. Ainsi comme dans beaucoup d'approches bayésiennes, l'ensemble de l'information des données est utilisée.

Pour pratiquer de l'optimisation bayésienne, il y a deux choix principaux à réaliser. La premier choix est celui du **Prior** concernant la fonction f. Le second choix est celui de la **fonction d'acquisition**. Cette dernière va servir à évaluer le prochain point à tester.

Les auteurs de l'article ont choisi comme prior un **Processus Gaussien** qui offre une excellente flexibilité sur la fonction f. Ainsi par définition, nos observations obtenues seront de la forme : $\{x_n, y_n\}$ avec $y_n \sim \mathcal{N}(f(x_n), v)$ avec v la variance du bruit de mesure des observations. Le support d'un tel processus peut se résumer avec deux fonctions : $m : \mathscr{X} \to \mathscr{R}$ la fonction moyenne, et $K : \mathscr{X} \times \mathscr{X} \to \mathscr{R}$ la fonction de covariance

La fonction d'acquisition quand à elle désigne une fonction $a : \mathscr{X} \to \mathscr{R}^+$ qui nous permet de choisir le prochain point d'évaluation : $x_{next} = argmax_{\mathscr{X}} a(x)$. Ces fonctions dépendent des paramètres du Processus Gaussien ainsi que des précédentes observations. Plusieurs fonctions sont alors possibles comme "Probability of Improvment" qui maximise la probabilité d'avoir une "meilleure observation" par rapport à toutes les observations précédentes ou encore le "GP Upper Confidence Bound". Le choix des auteurs de l'article s'est porté sur l' **Expected Improvement** a_{EI}

Soit θ les hyperparamètres de notre GP, μ la fonction moyenne de prédiction et σ la fonction de variance sous le prior GP (à ne pas confondre avec les fonctions m et K précédentes),$x_{best} = argmin_{x_1,..,x_n} f(x)$ la meilleure observation actuelle, $\gamma(x) = \dfrac{f(x_{best}) - \mu(x; \{x_n, y_n\}, \theta)}{\sigma(x; \{x_n, y_n\}, \theta)}$ et Φ la fonction de répartition de la loi Normale alors l'Expected Improvment est définie par :

$$a_{EI}(x; \{x_n, y_n\}, \theta) = \sigma(x; \{x_n, y_n\}, \theta)(\gamma(x)\Phi(x) + \mathcal{N}(\gamma(x); 1))$$

Ce choix est justifié par de bonnes performances en minimisation et l'absence d'hyperparamètres supplémentaires.

2. Points clés de l'étude

Le travail réalisé dans ce contexte s'attaque à trois difficultés précises :

— La première est celle du choix de la fonction de covariance K du GP qui peut être déterminant d'un problème à l'autre.

— La deuxième est la volonté de prendre en compte le temps d'évaluation de la fonction f qui peut changer radicalement d'un problème à l'autre. En effet, on rappelle que l'évaluation de cette fonction implique la mise en place d'un algorithme de Machine Learning complet.

— Enfin, malgré la nature séquentielle de cette optimisation bayésienne, les auteurs ont cherché à tirer profit de la parallélisation des calculs propres aux environnements distribués.

a) Le choix de la fonction de covariance

Le choix de cette fonction covariance peut induire d'importantes hypothèses sur la fonction f.

Un choix courant est celui du "squared exponential kernel" (ou ARD). Néanmoins, les fonctions obtenues pour ce choix ont une régularité importante. Bien que cela soit positif d'un point de vue de l'optimisation, cela reste très peu réaliste pour l'optimisation complexe de notre fonction f.

C'est pourquoi les auteurs ont plutôt choisi comme fonction de covariance un choix plus "exotique", l'ARD Matérn 5/2 kernel :

$$K_{M52}(x,x') = \theta_0(1+\sqrt{5r^2(x,x')}+\tfrac{5}{3}r^2(x,x'))e^{-\sqrt{5r^2(x,x')}} \text{ avec } r^2(x,x') = \sum_{d=1}^{D}(x_d - x'_d)^2/\theta_d^2$$

On constate alors la présence de D+3 hyperparamètres issue de notre processus Gaussien (D est la dimension des vecteurs $x \in \mathcal{X}$) :

— D paramètres de longeurs d'échelles $\theta_{1:D}$

— θ_0 l'amplitude de la covariance

— ν le bruit de mesure et m la moyenne

On concatène alors tous ces paramètres en un unique vecteur θ

Il y a donc D+3 hyperparamètres à sélectionner. Pour avoir un traitement bayésien complet, l'idée des auteurs a été de modifier la fonction d'acquisition a précédente en intégrant selon ce vecteur θ. On obtient alors la fonction d'acquisition intégrée suivante :

$$\hat{a}(x;\{x_n,y_n\}) = \int a(x:\{x_n,y_n\},\theta)p(\theta|\{x_n,y_n\}_{n=1}^N)d\theta$$

avec $p(\theta|\{x_n,y_n\}_{n=1}^N)$ la distribution marginale issue des données et du processus Gaussien.

Il s'agit d'une généralisation permettant de prendre en compte l'incertitude sur le choix des hyperparamètres du GP.

b) La prise en compte du temps d'évaluation

Bien que l'optimisation bayésienne précédente nous permette d'effectuer un choix optimal pour la prochaîne évaluation de f, cela peut se solder en pratique par un temps d'exécution très long (cela dépend fortement de la nature de l'espace des hyperparamètres $\mathscr{X}$). Cela rendrait donc l'étude non applicable en pratique avec des configurations machines classiques.

La fonction f nous est inconnue tout comme la fonction $c : \mathscr{X} \rightarrow \mathscr{R}^+$ qui évalue le temps d'évaluation de la fonction f au point x.

Le point clé ici est d'utiliser toute la "machinerie" de notre optimisation bayésienne pour évaluer $ln\ c(x)$ en plus de $f(x)$ en supposant que ces deux fonctions sont indépendantes l'une de l'autre.

Ainsi il est possible de tracer l'"expected improvement per second" en divisant par cette évaluation.

L'algorithme nous permet non seulement de choisir des points qui donneront une bonne optimisation de f mais également des points pour lesquels cette évaluation n'est pas trop couteuse en temps d'exécution.

c) La parallélisation de l'optimisation Bayésienne par des acquisitions de Monte Carlo

Dans le cas où on veut paralléliser, il faut changer la fonction d'acquisition qui pourrait utiliser des roll-outs optimaux mais ils sont difficiles à mettre en place.

Ici, il est décidé d'utiliser une stratégie séquentielle, profitant des propriétés d'inférence des processus gaussiens. Cela permettra de calculer les estimations de Monte-Carlo de la fonction d'acquisition pour différents résultats possibles dépendants des évaluations en cours.

En ayant déjà N réalisations effectués (xn,yn) et J évaluations en cours (xi), on cherche à trouver un nouveau point basé sur les N couples ainsi que toutes les sorties possibles d'évaluations en cours (yi). D'où l'intégration sur $\mathscr{R}$ pour les différents y. On parallélise sur le calcul de l'intégrale de la fonction d'acquisition sur tout l'espace de définition $\mathscr{R}^J$ des points à évaluer en cours. Cela est mise en place avec la création d'échantillons dans le cadre d'une méthode de Monte Carlo.

3. Implémentation

Contexte

Nous avons initialement étudié l'implémentation de l'auteur de l'article appelée Spearmint (disponible ici : https://github.com/JasperSnoek/spearmint).

Spearmint a alors été exécuté dans le cadre de l'optimisation de l'algorithme branin hoo (algorithme étudié par les auteurs de l'article) en choisissant (fenêtre de commande ci-dessous) un processus gaussien avec Expected Improvment ("GPEIOptChooser"). L'argument noiseless = 1 permet de ne pas estimer le bruit de mesure. Dans le cas d'évaluation de fonction f déterministe comme celle de cet exemple, cela permet d'améliorer considérablement le temps d'évaluation. Pour des problèmes non déterministes, cet argument peut en revanche se montrer indispensable.

```
lemeur@lemeur-VirtualBox: ~/Bureau/spearmint-master/spearmint/bin
lemeur@lemeur-VirtualBox:~/Bureau/spearmint-master/spearmint/bin$ ./spearmint ..
/examples/braninpy/config.pb --driver=local --method=GPEIOptChooser --method-arg
s=noiseless=1
setting up signal handler...
Using experiment configuration: ../examples/braninpy/config.pb
experiment dir: /home/lemeur/Bureau/spearmint-master/spearmint/examples/braninpy


----------------------------------------
Current best: 0.397904 (job 20039)
19988 candidates   0 pending   52 complete
Choosing next candidate...
1/10] mean: 113.94  amp: 60.46  noise: 0.0010 min_ls: 0.4189  max_ls: 0.7685
2/10] mean: 96.87  amp: 60.64  noise: 0.0010 min_ls: 0.4165  max_ls: 0.8695
3/10] mean: 51.32  amp: 60.22  noise: 0.0010 min_ls: 0.4261  max_ls: 0.7292
4/10] mean: 132.59  amp: 60.69  noise: 0.0010 min_ls: 0.4561  max_ls: 0.8219
5/10] mean: 114.12  amp: 60.61  noise: 0.0010 min_ls: 0.4121  max_ls: 0.8476
6/10] mean: 117.41  amp: 60.62  noise: 0.0010 min_ls: 0.4726  max_ls: 0.8241
7/10] mean: 99.61  amp: 60.87  noise: 0.0010 min_ls: 0.4365  max_ls: 0.9126
8/10] mean: 91.17  amp: 60.35  noise: 0.0010 min_ls: 0.4564  max_ls: 0.8882
9/10] mean: 105.65  amp: 57.18  noise: 0.0010 min_ls: 0.3930  max_ls: 0.7170
10/10] mean: 111.23  amp: 54.35  noise: 0.0010 min_ls: 0.3777  max_ls: 0.7423
selected job 20040 from the grid.
Submitted job as process: 2403
```

Nous voyons à l'écran le résultat d'une itération avec la valeur actuelle de la fonction d'acquisition (0,397904) ainsi que les paramètres du processus Gaussien lors de cette exécution (moyenne, amplitude) en conservant un bruit de mesure fixe (à 0,001). Nous pouvons voir ci-dessous le résultat de cette itération :

```
Running in wrapper mode for '20039'

Running python job.

Anything printed here will end up in the output directory for job #: 20039
{u'X': array([ 0.62826736,  0.49853927])}
0.397904146102
Got result 0.397904

Job file reloaded.
Completed successfully in 0.01 seconds. [0.397904]
setting job 20039 complete
set...
```

Nous retrouvons la valeur de la fonction d'acquisition pour cette évaluation ainsi que la valeur actuelle des deux hyperparamètres ([0,62826736, 0,49853927]). Le processus s'exécutant indéfiniment, il est nécessaire de l'arrêter dès lors qu'on constate une convergence satisfaisante de la fonction d'acquisition. En fonction de la méthode spécifiée nous pouvons avoir des informations différentes à l'écran comme par exemple le temps d'exécution en choisissant la méthode GPEI per second.

Nous avons néanmoins, décidé de changer d'approche à cause des difficultés suivantes :

- Implémentation ancienne, plus mise à jour qui repose sur Python2 : Python3 est devenu la version Python principale actuelle.
- Librairies difficilement integrables sous Windows et Mac : Nous avons du faire tourner spearmint sous une machine virtuelle Linux (Distribution Ubuntu 16.04).

Au final, cette implementation devenait complexe à utiliser et à étendre en particulier dans un Notebook Python (qui était notre choix d'environnement d'implémentation).

Nous avons donc étudié une autre implémentation (disponible ici : https://github.com/thuijskens/bayesian-optimization) que nous avons adapté dans le cadre de notre étude.

Notre implémentation

Dans notre implémentation, nous avons appliqué un processus Gaussien avec le kernel ARD 5/2 et la fonction d'acquisition Expected Improvment. Dans un soucis de simplicité, nous n'avons pas utilisé de parallélisation.

Nous avons mis en application cet algorithme d'optimisation sur un problème de classification multi-classe. Notre choix s'est porté vers le célèbre dataset iris.

Nous avons optimisé deux hyperparamètres pour deux algorithmes de Machine Learning de la librairie scikit learn :
SVM classifier (SVC) et Stochastic Gradient Descent classifier (SGDClassifier avec loss='hinge' qui correspond à un algorithme SVM linéaire).

Pour le SGDclassifier les deux hyperparamètres sont :

- l1_ratio un paramètre d'élasticité
- alpha, une constante de régularisation

Pour le SVC les deux hyperparamètres sont :

- C, un terme de pénalité de la fonction d'erreur
- γ coefficient propre au kernel utilisé

En plus du dataset iris, nous avons également utilisé une fonction sklearn "make_classification" qui génère aléatoirement un nouveau dataset.

En outre, nous avons utilisé les fonctions de sklearn pour la définition du kernel Mattern 5/2, et pour le processus gaussien.

Nous avons alors défini les fonctions utiles pour calculer l'optimisation bayésienne (kernel, expected improvment, détermination du prochain point d'application) ainsi que des fonctions propres à l'affichage des figures (voir notebook). Nous allons alors ici simplement détailler les appels de ces fonctions et les commentaires associés.

1. SGDClassifier

Nous allons ici tenter d'optimiser les hyperparamètres de l'algorithme de classification de sklearn SGDClassifier avec loss='hinge' sur le dataset iris :

```
# Récupération des données
data,target = load_iris(True)
# On définit ici notre classifieur :
def sample_loss(params):
    return cross_val_score(SGDClassifier(alpha=10 **params[0], l1_ratio=params[1],loss='hinge'),
                           X=data, y=target, cv=3).mean()
```

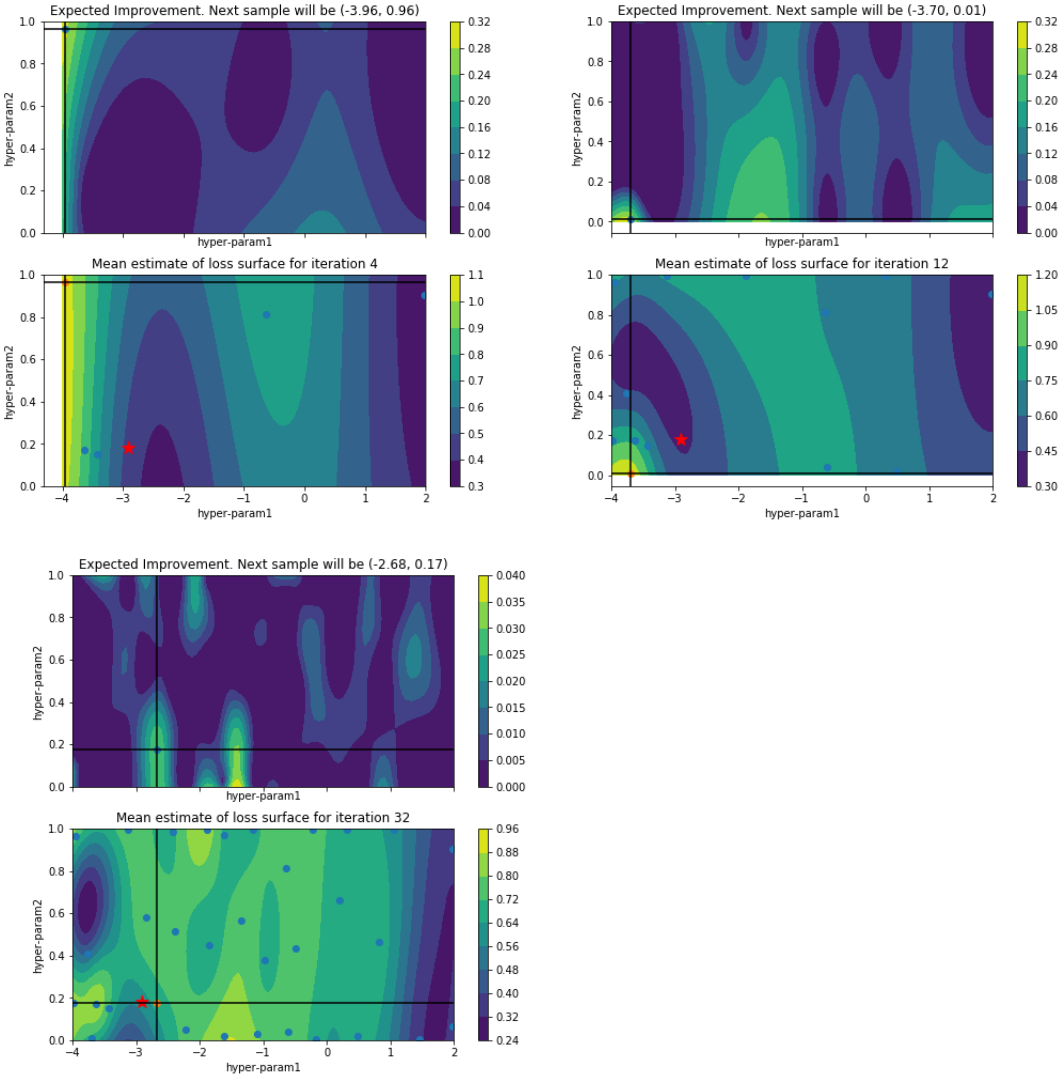
Nous cherchons une valeur "optimale" (à un nombre de combinaisons des deux hyperparamètres fixés) des hyperparamètres indiquée par une étoile rouge sur nos futurs graphes :

```
# Espace dans lequel va évoluer nos hyperparamètres
alphas = np.linspace(-4,2,100)
l1_ratios = np.linspace(0,1,100)

# On définit une matrice regroupant toutes les combinaisons des deux variables précédentes
param_grid = np.array([[alpha, l1_ratio] for alpha in alphas for l1_ratio in l1_ratios])

# On calcule la perte associée à ces hyperparamètres
real_loss = [sample_loss(params) for params in param_grid]

# On détermine les hyperparamètres "optimaux"
maximum=param_grid[np.array(real_loss).argmax(), :]
maximum
### OUTPUT :
array([-2.90909091,  0.18181818])
```

On met en place ci-dessous notre optimisation bayésienne :

```
# On détermine l'espace dans lequel évolue nos paramètres
bounds = np.array([[-4,2], [0,1]])

# On lance notre algorithme d'optimisation
xp, yp = bayesian_optimisation(n_iters=30,
                               sample_loss=sample_loss,
                               bounds=bounds,
                               n_pre_samples=3,
                               random_search=1000)
```

Nous allons alors observer les résultats de notre optimisation en affichant les résultats graphiques de trois itérations (au début, milieu et fin du processus). Les abscisses et ordonnées correspondent à nos deux hyperparamètres à optimiser et le code couleur aux zones d'isovaleurs pour notre fonction de perte. Pour chaque itération , nous avons deux graphes : le premier indique l'Expected improvment et le prochain point à évaluer, le second la valeur de la fonction de perte en ce point.



Nous constatons bien une convergence de notre algorithme dans une zone proche de la valeur calculée (étoile rouge). Avant cela l'algorithme a exploré l'espace de nos hyperparamètres en optimisant à chaque étape la prochaine exploration.

2. SVM Classifier (SVC)

Nous appliquons à présent notre algorithme avec le modèle SVC

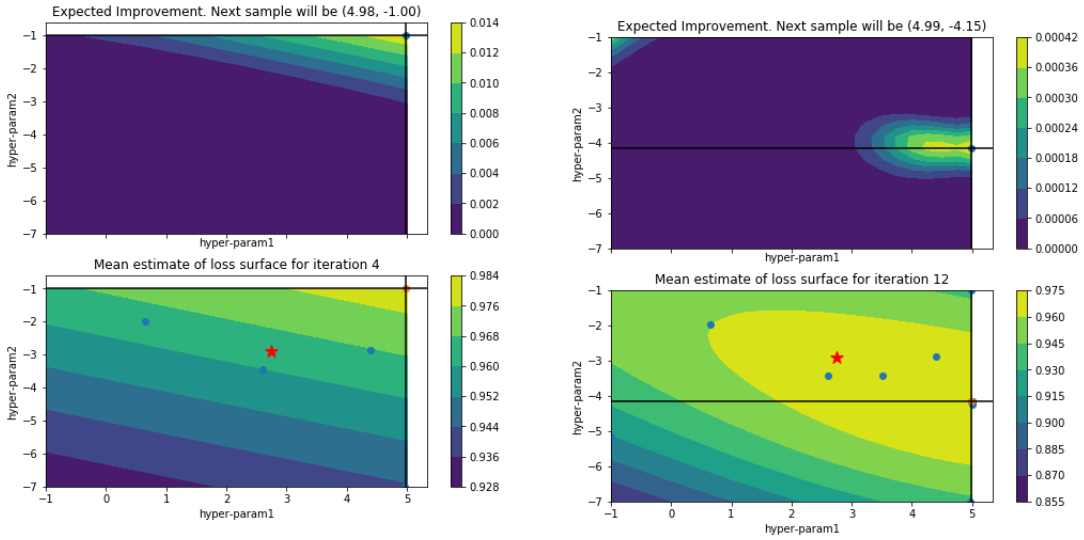
```
def sample_loss2(params):
    return cross_val_score(SVC(C=10 ** params[0], gamma=10 ** params[1], random_state=12345),
                           X=data, y=target, cv=3).mean()
```

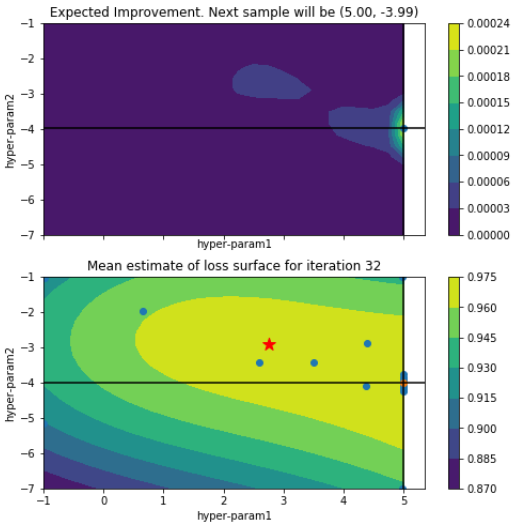
On cherche à nouveau une valeur "optimale" des hyperparamètres :

```
# Espace dans lequel va évoluer nos hyperparamètres
Cs = np.linspace(5, -1, 25)
gammas = np.linspace(-1, -7, 20)

# On définit une matrice regroupant toutes les combinaisons des deux variables précédentes
param_grid = np.array([[C, gamma] for gamma in gammas for C in Cs])

# On calcule la perte associée à ces hyperparamètres
real_loss = [sample_loss2(params) for params in param_grid]

# On détermine les hyperparamètres "optimaux"
maximum=param_grid[np.array(real_loss).argmax(), :]
maximum
## Output :
array([ 2.75       , -2.89473684])
```

Nous mettons alors en place cette nouvelle optimisation en affichant les résultats :

```
# On détermine l'espace dans lequel évolue nos paramètres
bounds = np.array([[-1, 5], [-7, -1]])

# On lance notre algorithme d'optimisation
xp, yp = bayesian_optimisation(n_iters=30,
                               sample_loss=sample_loss2,
                               bounds=bounds,
                               n_pre_samples=3,
                               random_search=100000)
```





Nous avons à nouveau affiché trois itérations. Nous constatons qu'entre la première et la deuxième itération affichée, la connaissance sur l'espace de nos hyperparamètres s'est affinée. En revanche la dernière itération nous indique qu'il semble que l'algorithme reste dans une zone précise de nos hyperparamètres située à droite. Cela peut signifier que la valeur optimale du premier hyperparamètre (hyperparam1 sur le graph) doit être encore plus importante que les bornes que nous avons fixé. On constate en revanche que notre dernière valeur et la "valeur optimale" appartiennent toutes les deux à une même zone d'isovaleurs.

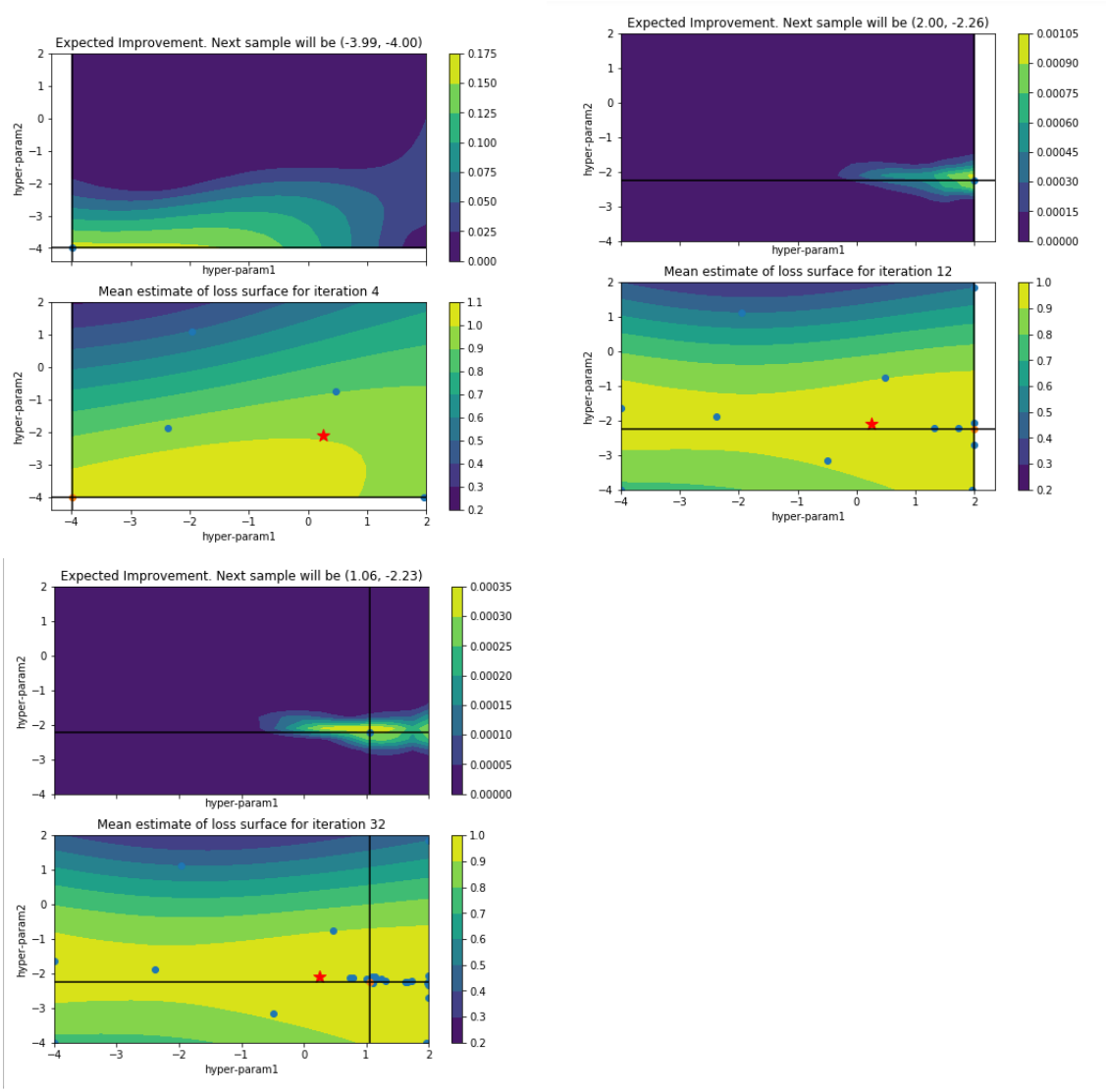
3. Utilisation d'un nouveau dataset de classification binaire (avec make_classification)

Nous allons dans cette partie changer de dataset en générant un nouveau dataset à l'aide de la fonction make_classification pour évaluer un algorithme SVC et nous appliquons toujours notre optimisation sur celui-ci :

```
def sample_loss3(params):
    return cross_val_score(SVC(C=10 ** params[0], gamma=10 ** params[1], random_state=12345),
                           X=data, y=target, cv=3, scoring='roc_auc').mean()
```

```
# Espace dans lequel va évoluer nos hyperparamètres
#Cs = np.linspace(5, -1, 25)
Cs = np.linspace(2, -4, 25)

#gammas = np.linspace(-1, -7, 20)
gammas = np.linspace(2, -4, 20)

# On définit une matrice regroupant toutes les combinaisons des deux variables précédentes
param_grid = np.array([[C, gamma] for gamma in gammas for C in Cs])

# On calcule la perte associée à ces hyperparamètres
real_loss = [sample_loss3(params) for params in param_grid]

# On détermine les hyperparamètres "optimaux"
maximum=param_grid[np.array(real_loss).argmax(), :]
maximum
## Output :
array([ 0.25       , -2.10526316])
```

```
# On détermine l'espace dans lequel évolue nos paramètres
bounds = np.array([[-4, 2], [-4, 2]])

# On lance notre algorithme d'optimisation
xp, yp = bayesian_optimisation(n_iters=30,
                               sample_loss=sample_loss3,
                               bounds=bounds,
                               n_pre_samples=3,
                               random_search=100000)
```



Dans ce dernier exemple nous avons une convergence claire vers notre valeur "optimale" en une trentaine d'itérations ce qui valide l'efficacité de la méthode.

Conclusion

Nous avons étudié et présenté une méthode d'optimisation Bayésienne. Cette dernière nous permet à partir de processus Gaussien d'optimiser intelligement nos hyperparamètres d'algorithmes de machine learning en plusieurs itérations. Cela permet d'adapter automatiquement les hyperparamètres optimaux en fonction des données étudiées.

Il est évidemment possible d'étendre cette approche à plus de 2 hyper-paramètres (par exemple pour du deep-learning à multi couches). Nous nous sommes ici limité à 2 hyper-paramètres pour permettre un affichage (en 2 dimensions) ainsi que pour limiter le temps d'exécution.

Par soucis de simplicité, nous avons par ailleurs choisi une fonction d'acquisition qui ne contient pas une intégration selon les hyperparamètres du processus Gaussien (Expected Improvment "amélioré"). Nous n'avons également pas considéré le temps d'exécution ni la possibilité d'une parallélisation comme décris dans l'article. Ces notions permettraient d'adapter cette optimisation à des situations d'apprentissage bien plus complexe que celles étudiées (où temps d'exécution et dimension des features deviennent problématiques).

Nous avons évalué deux algorithmes sur un même dataset d'une part et l'un des algorithmes sur un autre dataset d'autre part pour tester la reproductivité de cette méthode. Avec nos exemples, nous avons pu constater l'efficacité de cette approche dans la détermination d'hyperparamètres optimaux. Nous relevons un écart (plus ou moins variable en fonction des cas) avec la "valeur optimale théorique" (étoile rouge) car celle-ci n'est pas une valeur exacte mais choisie parmi un nombre limité de valeurs. Il s'agit alors là d'une démarche élégante et adaptative pour s'attaquer au problème complexe d'optimisation d'hyperparamètres dans un algorithme de Machine Learning.