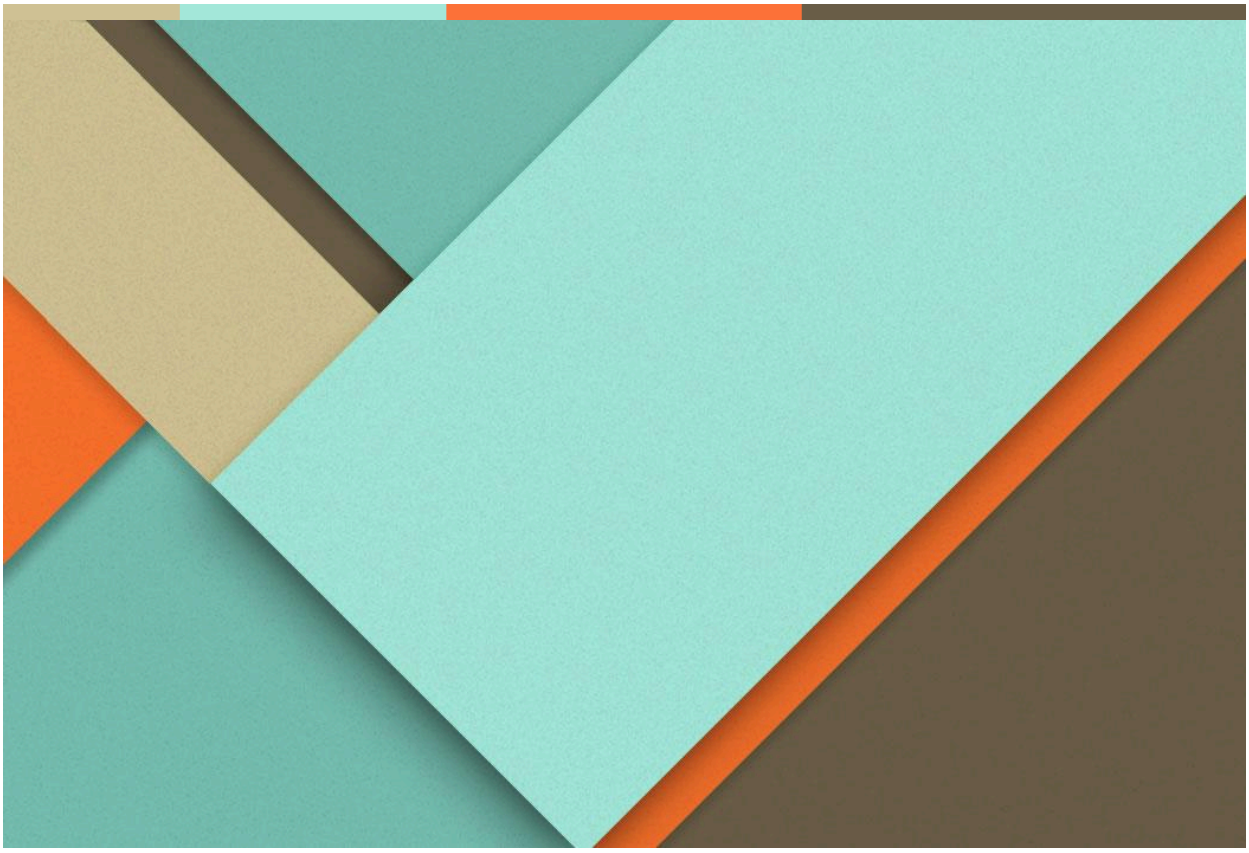


Automated Hyperparameter Optimization using AutoML Techniques

From 25 may 2024 To 20 june 2024

Chaman Singhal
Enrollment No. -> 22118019
Branch -> Metallurgy
IIT Roorkee

1. Introduction

Hyperparameter optimization (HPO) plays a crucial role in machine learning by significantly impacting the performance, efficiency, and generalizability of ML models. Manual hyperparameter optimization (HPO) faces several significant challenges, including the complexity and high dimensionality of the hyperparameter space, which involves numerous settings and their intricate interactions. This makes manual tuning both difficult and inefficient. Additionally, the computational cost is substantial, as training and validating models for each hyperparameter configuration can be extremely time-consuming and resource-intensive. Manual methods, such as grid search or random search, often fail to explore the hyperparameter space thoroughly, leading to suboptimal model performance.

2. Methodology

2.1 Overview of AutoML Techniques

AutoML techniques aim to automate the end-to-end process of applying machine learning to real-world problems.Some Key techniques include:

- **Bayesian Optimization:** Uses Bayes' theorem to predict the probability distribution of model performance based on past evaluations, iteratively refining the search for optimal hyperparameters.
- **Random Forests for Optimization:** Uses an ensemble of decision trees to model the relationship between hyperparameters and model performance, guiding the search process.
- **Tree-Parzen Estimator (TPE):** A sequential model-based optimization algorithm that builds probabilistic models to distinguish between good and bad hyperparameter configuration

Among these techniques I have used Bayesian Optimization for this project.

2.2 What is Bayesian Optimization

Bayesian optimization leverages **Bayesian technique** to set a prior over the **objective function**, then adding some new information to get a posterior function. A prior represents what we know before the new information is available and a posterior represents what we know about the objective function, given the new information. More specifically, a sample of the search space (i.e. a set of hyperparameters in this context) is collected and then objective function is calculated for the given sample (i.e. model is trained and evaluated). Since an objective **function** is not readily accessible, a **"surrogate function"** is used as a Bayesian approximation of the objective function.

Surrogate function is then updated with the information from the previous sample to get us from prior to posterior. Posterior, representing our best knowledge of the objective function at that point in time, is then used to guide an "acquisition function". **Acquisition function** (e.g. Expected Improvement) optimizes the conditional probability of the locations within the search space to acquire new samples from the search space that are more likely to optimize the original cost function.

To continue with the **Expected Improvement** example, the acquisition function computes the Expected Improvement for each point in the grid of hyperparameters and returns the one with the largest value. Then the newly-collected sample will run through the cost function, posterior is updated and the process repeats until an acceptable optimized point of the objective function is reached, a good-enough result is produced, or resources are exhausted.

2.3 Why use Bayesian Optimization

Conventional hyperparameter optimization methodologies, such as grid and random searches, require calculating a given model's cost function multiple times to find the most optimized combination of hyperparameters. Since many modern machine learning architectures include a large number of

hyperparameters (e.g. deep neural networks), calculating cost function becomes computationally expensive, decreasing the appeal of conventional methodologies such as grid search. In such scenarios, Bayesian optimization has become one of the common hyperparameter optimization methodologies, since it can find an optimized solution with significantly lower number of iterations compared to traditional approaches such as Grid and Random Search .

Bayesian optimization, a class of global optimization methods where we want to find the minimum (or maximum) of a **black-box function f(x)** on some bounded set χ. It is a type of sequential model-based optimization (SMBO) algorithm, and unlike grid search or random search, we use the results of our previous iteration to improve our sampling method of the next experiment.

2.4 Gaussian Process

According to the definition in Wikipedia, a Gaussian process is a stochastic process, in which a collection of random variables drawn from a multivariate normal distribution indexed by time or space. The distribution over functions has a continuous domain, e.g. time or space, as Gaussian process is a joint distribution of all the random variables. Define some space: Input space: χ
model scalar function: $f : \chi \to \mathbb{R}$
Positive definite covariance function: $C : \chi \times \chi \to \mathbb{R}$
mean function: $m : \chi \to \mathbb{R}$ From what we learned on class, assume we have this function vector $f = (f(x_1), \dots f(x_n))$, for any choice of input points, $(x_1, \dots, x_n)$, the marginal distribution over f:

$P(F) = \int f_{f \in F} P(f) df$, is multi-variate Gaussian. Then the distribution $P(f)$ over the function f is said to be a Gaussian Process.

We write a Gaussian Process thus:

$f(x) \sim GP(m(x), k(x, x\prime))$ where the mean and covariance functions can be thought of as the infinite dimensional mean vector and covariance matrix respectively.

The figures below illustrates how we fit a GP.

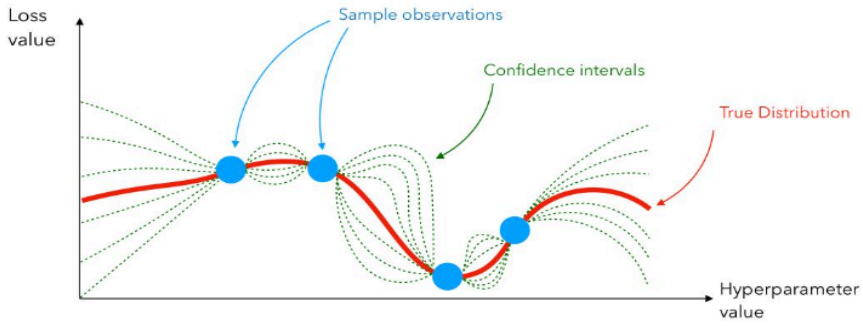
2.5 Surrogate Function

Surrogate function is a technique used to best approximate the mapping of input examples to an output score. Probabilistically, it summarizes the conditional probability of an objective function (*f*), given the available data (*D*) or *P(f|D)*.

We suppose that the function f,which represents the distribution of a loss in terms of the value of a hyperparameter, has a meanμ and a covariance K, and is a realization of a Gaussian Process. The Gaussian Process is a tool used to infer the value of a function. Predictions follow a normal distribution. Therefore :

$$p(y|x,D)=N(y|\hat{}\mu,\hat{}\sigma 2)$$

We use that set of predictions and pick new points where we should evaluate next. We can plot a Gaussian Process between 4 samples this way :



Even though the true distribution is unknown (the red line), we can infer its value using the Gaussian Process (confidence interval lines in green).

Once we identify a new point using the acquisition function, we add it to the samples and re-build the Gaussian Process with that new information... We

keep doing this until we reach the maximal number of iterations or the limit time for example. This is an iterative process.

Uncertainty in Optimization 😣:

During the optimization process, the goal of each update is to find the minimum:

$$: x^* = argmin_{x \in X} f(x).$$

GP gives us an easy closed-form of marginal means and variance. For searching the next minimum, we can do more exploration on places with high variance, but also can do more exploitation by seeking places with low means.

The acquisition functions provide a way to balance the exploration and exploitation to determine the next point to evaluate.

2.6 Acquisition Function

We can solve the aforementioned tradeoff of exploitation (the mean is low) and exploration (the variance is high) with the acquisition function .The prior and existing data points induce a posterior over functions: the acquisition function, and the goal is to evaluate the points and move to the point which maximizes the acquisition function. Several different functions have been proposed. We evaluated all three acquisition function proposed by author and recommend Expected Improvement.

How do we know which point we should evaluate next? There are two guidelines :

- Pick points that yield, on the approximated curve, a low value.
- Pick points in areas we have less explored.

There is an exploration/exploitation trade-off to make. This tradeoff is taken into account in an acquisition function.

The acquisition function is defined as :

$$A(x)=\sigma(x)(\gamma(x)\Phi(\gamma(x))+N(\gamma(x)))$$

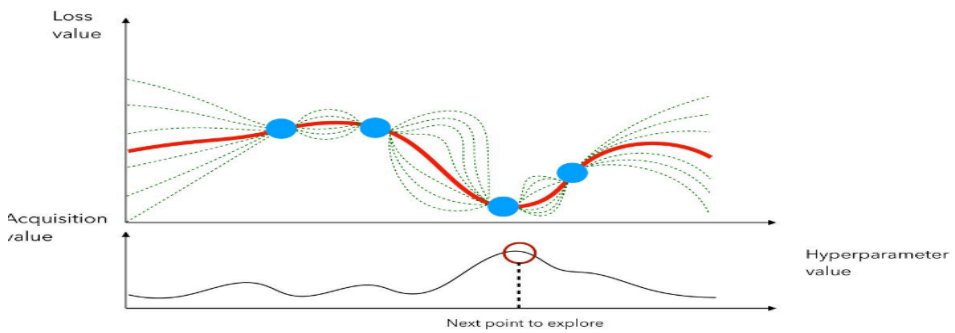
where :

$$\gamma(x) = \frac{f(x^c) - \mu(x)}{\sigma(x)}$$

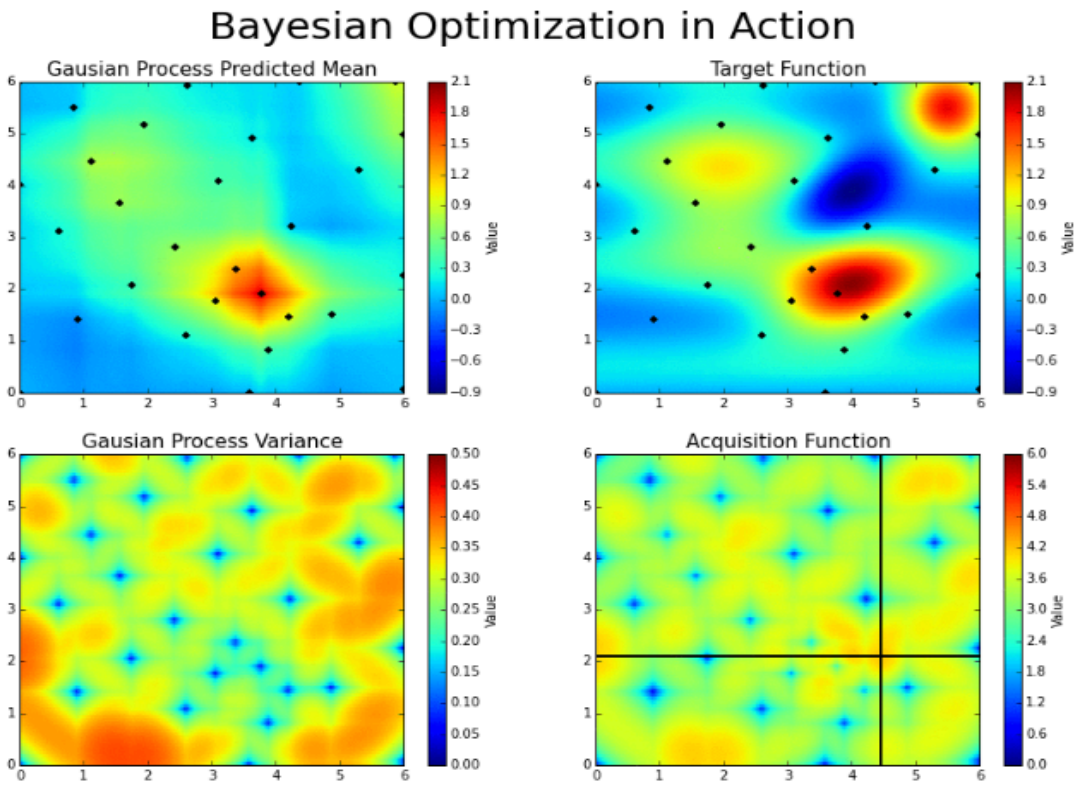
- f(x^c) the current guessed arg min, μ(x) the guessed value of the function at x, and σ(x) the standard deviation of output at x.

- Φ(x)and N(x) are the CDF and the PDF of a standard normal

This acquisition function is the most common and is called the *expected improvement*. We then compute the acquisition score of each point, pick the point that has the highest activation, and evaluate $f(x)$ at that point, and so on…



We identified a previous point that gave a small loss. We can also see the loss minimization problem as a maximization problem on the chosen metric.The gaussian hyperparameter optimization process can be illustrated in the following way 😀:



- I have used **Expected Improvement** as an acquisition function in this project because researchers claim that this could be better-behaved than **probability of improvemen**t, and not require one more tuning parameter κ in LCB.

- Till now, we have discussed most of the theory of **Bayesian optimization ,** now deep dive into its implementation. And I have not discussed the full mathematics behind this because it's very vast.if you want you can read from references given at last.

3. Implementation

In our implementation, I applied a Gaussian process with the ARD 5/2 kernel and the Expected Improvement acquisition function. For the sake of simplicity, I did not use parallelization.

I have done only for numerical hyperparameters of different models.

I optimized two hyperparameters for two Machine Learning algorithms from the scikit learn library:

SVM classifier (SVC) and Stochastic Gradient Descent classifier (SGDClassifier with loss='hinge' which corresponds to a linear SVM algorithm).

1. For the SGDclassifier the two hyperparameters are:

- L1_ratio an elasticity parameter
- Alpha, a regularization constant

2. For the SVC the two hyperparameters are:

- C, a penalty term of the error function
- Gamma coefficient specific to the kernel used

You can also do this for more than 2 numerical hyperparameters but it will take a long time to compute that's why I am doing only for 2 important parameters.

In addition to the iris dataset, I also used a sklearn function "make_classification" which randomly generates a new dataset.

We start by defining the functions useful for calculating Bayesian optimization (kernel, expected improvement, determination of the next point of application) then functions specific to the display of the figures.

3.1 “Expected Improvement” Acquisition Function

- x: array type, point to calculate
- gaussian_process: The Gaussian process previously trained on the hyperparameters
- evaluated_loss: Numpy array containing previously evaluated loss function values
- greater_is_better: Boolean indicating whether to maximize or minimize the loss function
- n_params: int indicating the dimension of the hyperparameters

```
def expected_improvement(x, gaussian_process, evaluated_loss,
greater_is_better=False, n_params=1):
    x_to_predict = x.reshape(-1, n_params)  # Reshape x to match the
number of parameters

    # Predict the mean and standard deviation of the loss at points x
    mu, sigma = gaussian_process.predict(x_to_predict, return_std=True)

    # Determine the best loss value based on the greater_is_better flag
    if greater_is_better:
        loss_optimum = np.max(evaluated_loss)
    else:
        loss_optimum = np.min(evaluated_loss)
```

```
    # Scaling factor for the calculation based on whether greater loss
is better
    scaling_factor = (-1) ** (not greater_is_better)

    # Calculate the Expected Improvement (EI)
    with np.errstate(divide='ignore'):
        Z = scaling_factor * (mu - loss_optimum) / sigma
        expected_improvement = (scaling_factor * (mu - loss_optimum) *
norm.cdf(Z) +
                                sigma * norm.pdf(Z))
        expected_improvement[sigma == 0.0] = 0.0  # Handle the case
where sigma is zero

    return expected_improvement
```

1. The Gaussian Process model is used to predict the mean (`mu`) and standard deviation (`sigma`) of the objective function at the points in `x_to_predict`.

2. The function determines the best observed value of the objective function. If higher values are better, it finds the maximum; otherwise, it finds the minimum.
3. The variable `Z` standardizes the difference between the predicted mean and the optimal loss, scaled by the standard deviation.

The Expected Improvement is computed using the formula:

$$\mathrm{EI}(x) = (y_{\text{best}} - \mu(x))\Phi\left(\frac{y_{\text{best}} - \mu(x)}{\sigma(x)}\right) + \sigma(x)\phi\left(\frac{y_{\text{best}} - \mu(x)}{\sigma(x)}\right)$$

4. The function returns the negative of the expected improvement. This is often done because optimization routines typically minimize objective functions, so negating EI allows these routines to work correctly.

3.2 Function determining the next Acquisition point:

The `sample_next_hyperparameter` function is designed to find the next set of hyperparameters to evaluate in the Bayesian optimization process. It uses an acquisition function to guide the search and ensures that the chosen hyperparameters are those expected to provide the greatest improvement to the model.

- acquisition_func: acquisition function to optimize
- gaussian_process: The Gaussian process previously trained on the hyperparameters
- evaluated_loss: Numpy array containing previously evaluated loss function values
- greater_is_better: Boolean indicating whether to maximize or minimize the loss function
- bounds: Tuple indicating the definition limits for the L-BFGS "optimizer"
- n_restarts: int indicating the number of iterations performed on the minimizer with different values

```
def sample_next_hyperparameter(acquisition_func, gaussian_process,
evaluated_loss, greater_is_better=False,
                               bounds=(0, 10), n_restarts=25):


    # Initialisation :
```

```
    best_x = None
    best_acquisition_value = 1
    n_params = bounds.shape[0]

    for starting_point in np.random.uniform(bounds[:, 0], bounds[:, 1],
size=(n_restarts, n_params)):

        res = minimize(fun=acquisition_func,
                       x0=starting_point.reshape(1, -1),
                       bounds=bounds,
                       method='L-BFGS-B',
                       args=(gaussian_process, evaluated_loss,
greater_is_better, n_params))
        # # Hyperparameter change condition :
        if res.fun < best_acquisition_value:
            best_acquisition_value = res.fun
            best_x = res.x

    return best_x
```

1. The loop generates `n_restarts` starting points uniformly at random within the specified bounds for each hyperparameter.
2. If the acquisition value found (`res.fun`) is better (lower) than the current `best_acquisition_value`, update `best_acquisition_value` and `best_x` with the results of this optimization run.
3. After all restarts, the function returns the best set of hyperparameters found.

3.3 Function optimizing the loss function using Gaussian processes

The bayesian_optimisation function implements Bayesian Optimization to find the optimal set of hyperparameters for a given model by iteratively sampling hyperparameters and fitting a Gaussian Process (GP) to the observed data.

- n_iters: int indicating the number of iterations to use
- sample_loss: function to optimize
- bounds: Array indicating the bounds of the loss function
- x0: array defining initial points where to evaluate the loss function. If x0=None these points are chosen randomly
- n_pre_samples: int indicating the number of initial points if x0=None
- gp_params: dictionary indicating the hyperparameters to pass to the Gaussian process.
- random_search: Boolean indicating whether we use a random search or an L-BFGS optimization on the acquisition function
- alpha: double indicating the variance associated with the Gaussian Process error term
- epsilon: double indicating the tolerance level for float evaluation

```
# Bayesian Optimization function (as defined previously)
def bayesian_optimisation(n_iters, sample_loss, bounds, x0=None,
n_pre_samples=5,
                          gp_params=None, random_search=False, alpha=1e-5,
epsilon=1e-7):
    x_list = []
    y_list = []

    n_params = bounds.shape[0]

    if x0 is None:
```

```
        for params in np.random.uniform(bounds[:, 0], bounds[:, 1],
(n_pre_samples, bounds.shape[0])):
            x_list.append(params)
            y_list.append(sample_loss(params))
    else:
        for params in x0:
            x_list.append(params)
            y_list.append(sample_loss(params))

    xp = np.array(x_list)
    yp = np.array(y_list)

    if gp_params is not None:
        model = gp.GaussianProcessRegressor(**gp_params)
    else:
        kernel = gp.kernels.Matern(nu=5/2)
        model = gp.GaussianProcessRegressor(kernel=kernel,
                                            alpha=alpha,
                                            n_restarts_optimizer=10,
                                            normalize_y=True)

    for n in range(n_iters):
        model.fit(xp, yp)

        if random_search:
            x_random = np.random.uniform(bounds[:, 0], bounds[:, 1],
size=(random_search, n_params))
```

```
            ei = -1 * expected_improvement(x_random, model, yp,
greater_is_better=True, n_params=n_params)
            next_sample = x_random[np.argmax(ei), :]
        else:
            next_sample = sample_next_hyperparameter(expected_improvement,
model, yp, greater_is_better=True, bounds=bounds, n_restarts=100)

        if np.any(np.abs(next_sample - xp) <= epsilon):
            next_sample = np.random.uniform(bounds[:, 0], bounds[:, 1],
bounds.shape[0])

        cv_score = sample_loss(next_sample)

        x_list.append(next_sample)
        y_list.append(cv_score)

        xp = np.array(x_list)
        yp = np.array(y_list)

    return xp, yp
```

1. If no initial samples (`x0`) are provided, generate `n_pre_samples` random samples within the specified bounds and evaluate the loss for each.
2. If `x0` is provided, use these as the initial samples.
3. Initialize the Gaussian Process model with provided parameters (`gp_params`) or default settings.
4. Fit the Gaussian Process model to the observed data.
5. If `random_search` is enabled, generate random points within the bounds, compute the Expected Improvement (EI) for each, and select the one with the highest EI.

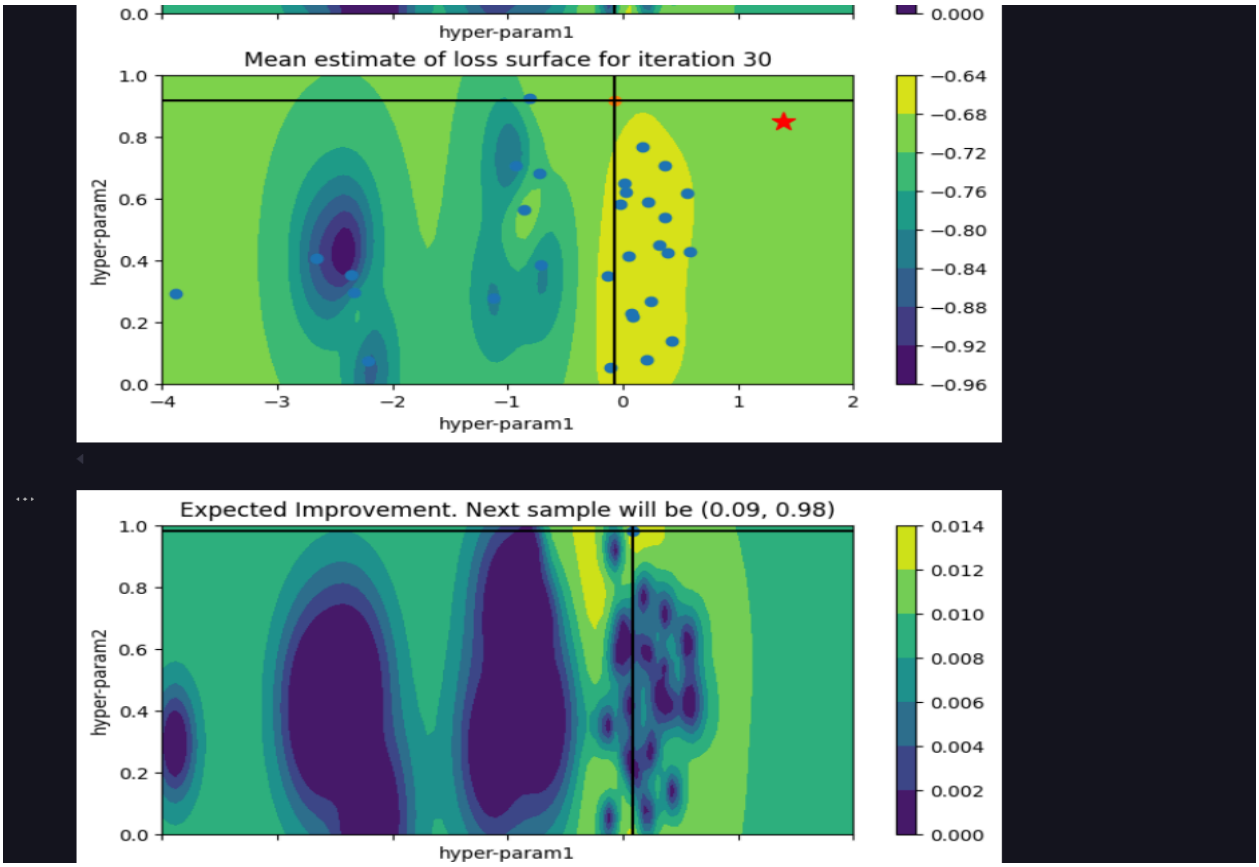
6. If `random_search` is not enabled, use the `sample_next_hyperparameter` function to find the next sample.

3.4 Functions for displaying results at each Iteration

The following functions will be used to create graphs to illustrate the implementation of the optimization.The graphs obtained will represent on the abscissa and ordinate our different hyperparameters with a color code corresponding to isovalues of the loss function and the Expected Improvement respectively.

Here , I am not pasting the code because it is just a matplot library implementation , if you want you can see the code from my github repo.So,i am showing just some images from the result of plots.

This is only one sample image of an iteration on SGDClassifier on iris dataset.



If you take a look at all the images in that folder you will notice that the **algorithm does not tend towards the optimal point (red star) determined previously. However, the optimum point obtained during the last iterations is in a rather weak iso-value zone, which reassures us in the convergence of the algorithm towards optimal values.**

3.5 Implementation on SVM Classifier(SVC)

The `sample_loss2` function is designed to evaluate the performance of an SVM (Support Vector Machine) classifier with given hyperparameters using cross-validation. It returns the mean cross-validation score, which serves as the metric to optimize in the Bayesian optimization process.

```
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()
```

- **`params`**: An array of hyperparameters where `params[0]` is the logarithm (base 10) of `C` and `params[1]` is the logarithm (base 10) of `gamma`.
- `10 ** params[0]`: Converts the logarithmic scale parameter back to the original scale for `C`.
- `10 ** params[1]`: Converts the logarithmic scale parameter back to the original scale for `gamma`.

```
Cs = np.linspace(5, -1, 100)

gammas = np.linspace(-1, -7, 100)

param_grid = np.array([[C, gamma] for gamma in gammas for C in Cs])

real_loss = [sample_loss2(params) for params in param_grid]

maximum=param_grid[np.array(real_loss).argmax(), :]

maximum
```

The above code performs a grid search over specified ranges of hyperparameters `C` and `gamma` for an SVM classifier. It evaluates the performance of each hyperparameter combination using the `sample_loss2` function and identifies the combination that yields the highest cross-validation score.

- A grid of all possible combinations of `C` and `gamma` values is created using a nested list comprehension.
- The `sample_loss2` function is applied to each set of parameters in `param_grid` to compute the cross-validation score.
- The index of the maximum value in `real_loss` is found using `argmax()`.
- The corresponding hyperparameter combination in `param_grid` is identified as `maximum`.

```
bounds = np.array([[-1, 5], [-7, -1]])

xp, yp = bayesian_optimisation(n_iters=30,
                               sample_loss=sample_loss2,
                               bounds=bounds,
                               n_pre_samples=3,
```

```
                                random_search=100000)

print("Evaluated points:")

print(xp)

print("Corresponding losses:")

print(yp)
```

This code runs Bayesian optimization for 30 iterations to optimize the hyperparameters `C` and `gamma` for an SVM classifier. It uses the `sample_loss2` function to evaluate the performance of each hyperparameter combination through cross-validation.

4. Comparison for Bayesian Optimization and Hyperopt