Linear Algebra and Optimization for Machine Learning: Project 2

Zhaonan Meng (5841003) Alvedian Mauditra Aulia Matin (5689252) Zuhair (5523435)

January 6, 2023

In this project, we perform hyperparameter tuning of a Radial Basis Function (RBF)-kernelized SVM (support vector machine) using grid search and Bayesian Optimization (BO). The objective of this project is to find the best combination of C (the regularization parameter) and γ (the kernel scaling parameter) that maximizes the accuracy of the five-fold cross-validation for a support vector classification model with RBF kernel built by Python function SVC. The model is trained and validated on the given dataset *Heart.csv*. The grid search and BO are presented in Section 1 and 2 respectively. In section 3 we make a few comments on comparing both methods.

1 Grid Search

As given in the instruction of this project, we implement a grid search for $(\log_{10} C, \log_{10} \gamma) \in [0, 9] \times [-10, 0]$. We use two different mesh grids: uniform and nonuniform grid (the grid is denser on small values of the hyperparameters). Two different scaling methods for the data are also considered, a standard scaler (scales the data to mean zero and variance around one) and a min-max scaler (scales the data to min zero and maximum one). To make the computational time faster, multiprocessing package is also implemented and compared. Table 1.1 shows the five-fold cross-validation accuracy, computation time, and the best (C, γ) of different grid search settings with 20 pools of multiprocessors. The code for this part of project is in `GridSearch.py`.

1.1 Scaling The Data

After dropping the last column of *Heart.csv*, the distribution of the data on each feature can be seen as in Fig. 1.1. It seems the data is distributed (skewed) normally on some features. Thus, our hypothesis is the standard scaler will give a better performance than the min-max scaler. Nevertheless, we try both methods (`prepare_input()`). First, we implement a uniform grid for the specified interval for C and γ with the size 100×100. As we have predicted, from Table 1.1, we can see that the standard scaler gives slightly better accuracy (85.19%) than the min-max scaler (84.81%). Hence, we will use standard scaler to try different grid size (Subsection 1.3).

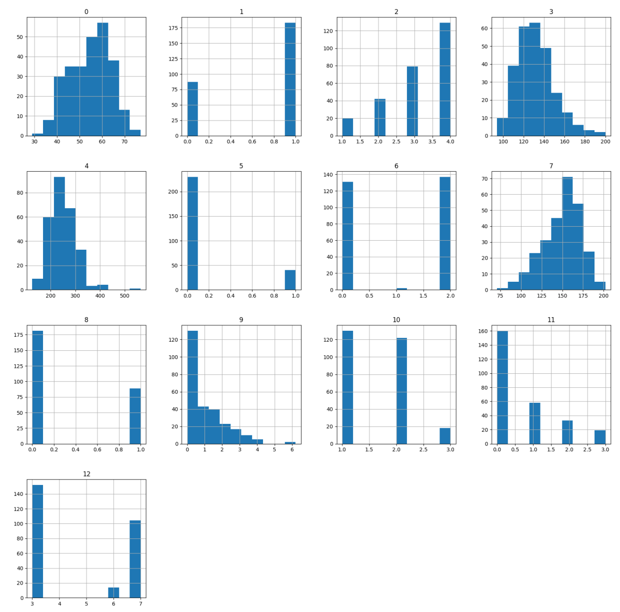

Figure 1.1: The distribution of Heart.csv on each feature

1.2 Type of Grid

Fig. 1.2 shows the classification accuracy of the five-fold cross-validation described by (C, γ) on 100×100 uniform grid where the data is scaled using standard and min-max scaler. From the plots, we observe that the higher accuracy happens mostly on smaller values of (C, γ). Thus, we make C and γ denser on the smaller values, as depicted in Fig. 1.3 (`ununiform_grid()`).

Table 1.1: The comparison of different grid search settings with 20 pools of multiprocessors

Grid Size	Grid Type	Scaler	The Best Performance	Compt. Time (hh:mm:ss)	The Best $(C,\ \gamma)$
10×10	Nonuniform	Standard	84.81%	00:00:09	$(5.62, 4.64 \times 10^{-4})$
100×100	Uniform	MinMax	84.81%	00:01:08	$(4.65 \times 10^{8}, 1.00 \times 10^{-10})$
		Standard	85.19%	00:00:58	$(1.00, 0.20)$
	Nonuniform	MinMax	85.56%	00:02:25	$(2.51 \times 10^{6}, 7.15 \times 10^{-7})$
		Standard	**87.04%**	**00:03:33***	$(3.98 \times 10^{8}, 4 \times 10^{-9})$
1000×1000	Nonuniform	Standard	87.41%	06:37:24	$(1.72 \times 10^{7}, 2.57 \times 10^{-9})$

*without multiprocessing package the computation time takes 00:19:14

By using the nonuniform grid, we obtain the accuracy on 100×100 grid in Fig. 1.4. From Table 1.1, we obtain higher best performances than using the uniform grid for both scalers, min-max (85.56%) and standard (87.04%). Moreover, standard scaled data still performs better than the min-max scaled data on the nonuniform grid. Hence, we only consider standard scaled data on the nonuniform grid for the next subsection.

1.3 Grid Size

After determining the scaler and the grid type, which are standard scaler and nonuniform grid, now we try a smaller and a larger grid size. We use 10×10 grid and 1000×1000 grid, giving the accuracy in Fig. 1.5. From Table 1.1, we can see that the 10×10 grid gives significantly lower performance (84.81%) than the 100×100 grid (87.04%). On the other hand, the 1000×1000 only performs slightly better (87.41%) than the 100×100 grid (87.04%), but with significantly longer computing time (about 6.5 hours, compared to 3.5 minutes). Therefore, the grid size of 100×100 is a better choice.

1.4 Number of Pools for Multiprocessor

Using parallel computing in a grid search algorithm can significantly improve the speed of the algorithm. In traditional, non-parallel algorithms, only one process is used to search through the grid and evaluate the parameters. This can take a significant amount of time, especially if the grid is large and the parameters are numerous.On the other hand, parallel computing allows for multiple processes to be used concurrently, allowing for a faster search through the grid. This can greatly reduce the overall time it takes to complete the algorithm, making it a much more efficient and effective solution. For instance, performing the grid search using parallel computing of standard scaled data on 100×100 nonuniform grid takes only about 3.5 minutes, otherwise it takes about 20 minutes (Table 1.1). The next thing to do is determining the ideal number of pools.

Pools refer to a group of processes that are used concurrently to perform a task. In Python, pools can be created using the multiprocessing library, which allows for the creation of a specified number of processes to be used in parallel. In our case, using a pool of 20 for multiprocessing in the grid search algorithm is a good choice because it allows for a balance between speed and feasibility for our computer's memory. In general, a larger pool size would allow for faster computation through parallel processing, but it could also potentially strain our computer's memory and potentially cause it to crash. On the other hand, a smaller pool size would result in slower computation but would be less demanding on our computer's memory. By choosing a pool size of 20, we are able to take advantage of the speed benefits of parallel processing without pushing our computer's memory to its limits, ensuring that our grid search algorithm runs efficiently without causing any issues with our computer's memory.

Therefore, based on our experiments and analysis, the optimal setting for the grid search algorithm of our case is the 100×100 nonuniform grid on standard scaled data. It gives a relatively high best performance (87.04%) with only 3.5 minutes of computing time, where 20 pools are used for multiprocessing. The best (C, γ) is $(398107170.5534969; 4.291934260128778 \times 10^{-9}) \approx (3.98 \times 10^{8}, 4 \times 10^{-9})$.

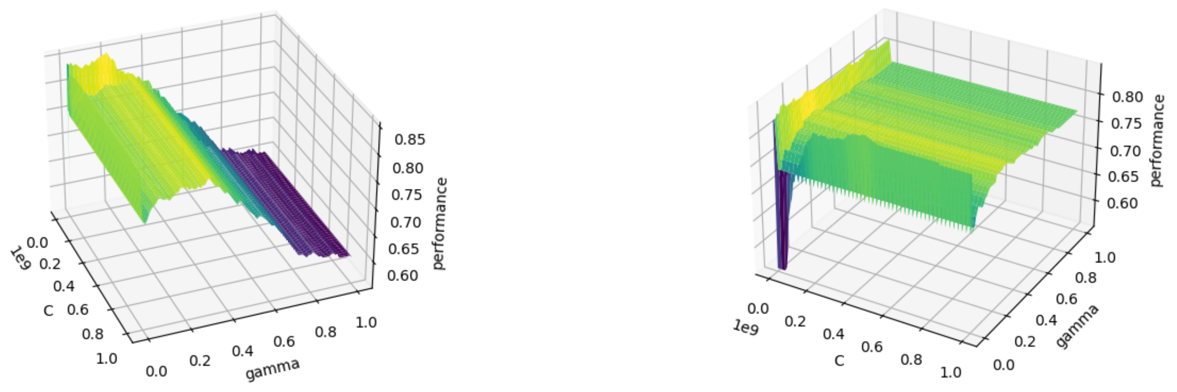

Figure 1.2: The classification accuracy of 100 × 100 uniform grid on standard (left) and min-max (right) scaled data.

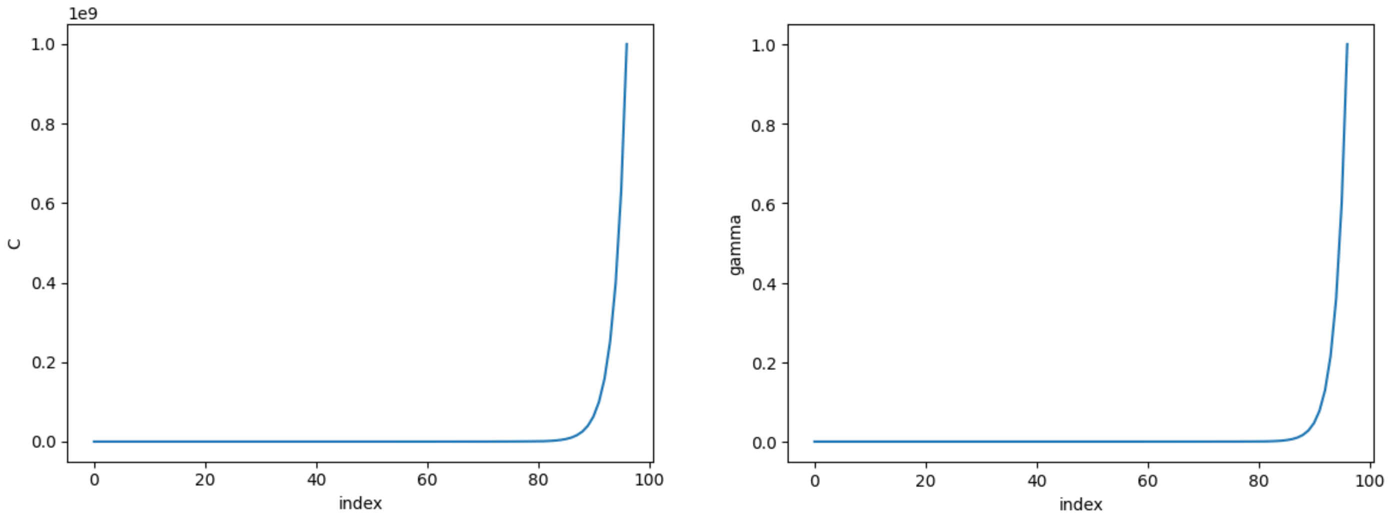

Figure 1.3: Nonuniform grid of C (left) and γ (right), which are denser on the smaller values.

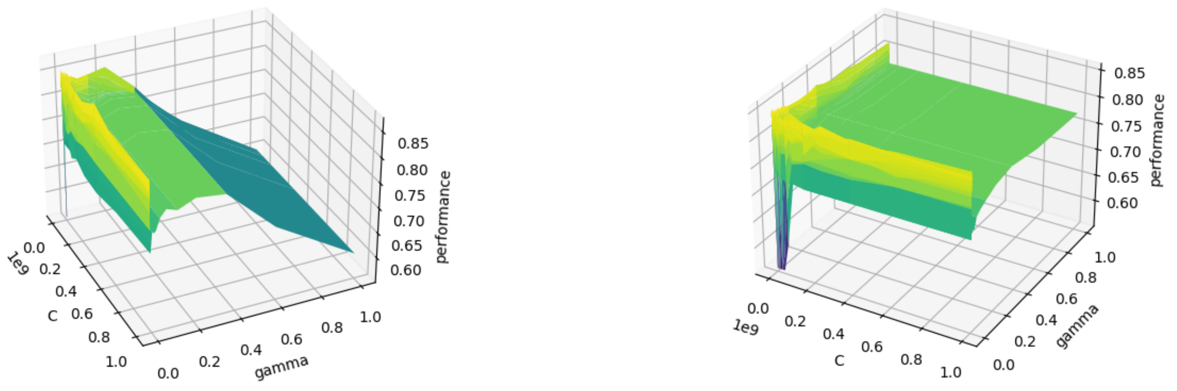

Figure 1.4: The classification accuracy of 100 × 100 nonuniform grid on standard (left) and min-max (right) scaled data.

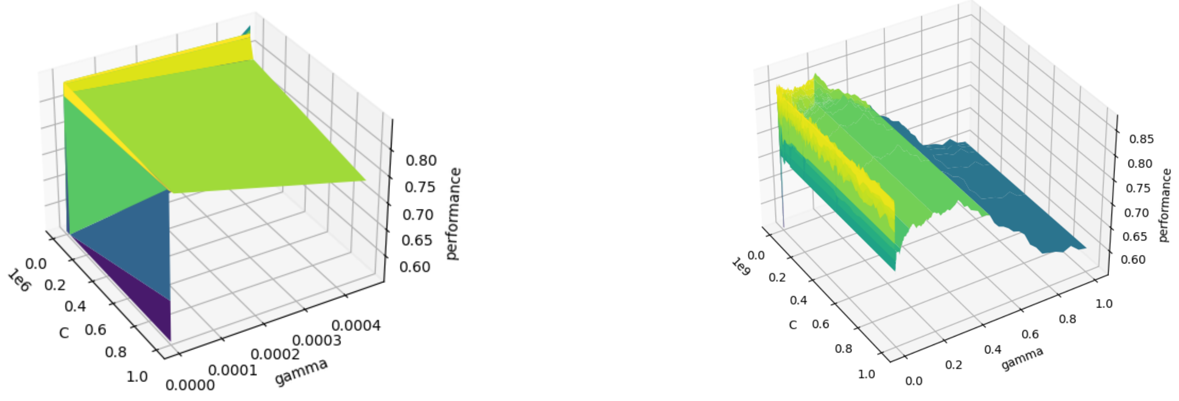

Figure 1.5: The classification accuracy on nonuniform grid of standard scaled data using 10×10 grid (left) and 1000×1000 grid (right).

2 Bayesian Optimization

As discussed in section 1, the serial computation of grid search takes a huge amount of time. What's more, once the function $G(h)$ (performance) is evaluated at a given point h (hyperparameters), this information is not used anymore. Bayesian optimization is designed to solve such information waste. In the following sections we will introduce some fundamental theories of BO (subsection 2.1), present and compare the results of BO utilizing two different optimization techniques to minimize acquisition functions in subsection 2.2 and 2.3. As we are a team of three, we mainly work on the first acquisition function. nonetheless, we will also have a brief discussion about the second type. Python implementation of Bayesian optimization consists of four `.py` files: `GD_BO.py`, `MBFGS_BO.py`, `optimization.py` and `start_bayesian_opt.py`. For details of running programs, please read `readme.txt`.

2.1 Basic Idea of BO

The core idea of Bayesian optimization is to construct a 'confidence region' following the principle that we have more confidence when we try a point h close to the already evaluated ones, compared to trying a value far away from them. From the lecture note, we know that Bayesian optimization implements this idea with two most important ingredients being:

· the assumed prior distribution used to construct the 'guess' about the shape of G,

· the acquisition function used to quantify where the gain of evaluating the next sample is the largest.

In this project, we assume the prior is normally distributed. The function per each point h is a normal random variable with expectation $\mathbb{E}(G(h)) = 0$ and the covariance of the function between points h and h' is $\mathbb{E}(G(h)G(h')) = K(h, h')$, where $K(\cdot, \cdot)$ is some selected kernel function. In our implementation of BO, we conventionally choose the Gaussian kernel:

$$\exp(-\alpha \|h - h'\|_2^2), \ \alpha > 0.$$

Please note that here by using the Gaussian kernel, we introduce a new hyperparameter α, which can influence the size of the trust region and play an important role in searching the best performance in the scenario of BO. In the following sections you will find that we use different α when we implement BO with two different optimization algorithms, lying on the fact that the performance of BO is quite sensitive to the kernel size α.

With the assumptions above and some pairs $(h_t, y_t = G(h_t))$ which have been already evaluated, we now can compute the conditional distribution of a multivariate normal distribution of $G(h)$ given the values $y = (y_1, \cdots, y_t)$. To derive the mean and the standard deviation, keys to computing the probability (or in other words, values of the acquisition function), we assume that the distribution of $g = (G(h_1), \cdots, G(h_t), G(h))$ follows a multivariate normal distribution with

$$\begin{aligned}
\mathbb{E}(g) &= 0, \\
\mathbb{E}(gg^\top) &= \begin{bmatrix} \Sigma & k \\ k^\top & K(h,h) \end{bmatrix}, \\
\text{where } \Sigma &= \begin{bmatrix} K(h_1,h_1) & \cdots & K(h_1,h_t) \\ \vdots & \ddots & \vdots \\ K(h_t,h_1) & \cdots & K(h_t,h_t) \end{bmatrix}, \\
\text{and } k &= \begin{bmatrix} K(h_1,h), \cdots, K(h_t,h) \end{bmatrix}^\top .
\end{aligned}$$

By means of the ingredients above, we can compute the mean $\mu(h)$ and the standard deviation $\sigma(h)$ of the conditional distribution of $G(h)$ given $h_1, \cdots, h_t$:

$$\begin{aligned}
\mu(h) &= k^\top \Sigma^{-1} y, \\
\sigma(h) &= \sqrt{K(h,h) - k^\top \Sigma^{-1} k}.
\end{aligned}$$

Given the probability distribution of unknown $G(h)$, acquisition functions now are available. Acquisition functions, as the objects to minimize, are constructed to search the next best point. There are basically three types of commonly used acquisition functions to minimize. In our project, we will mainly focus on the first type:
· (negative of) the probability of improving upon the so-far best value y_{best}:

$$a(h, y_{best}) = \mathbb{P}(G(h) < y_{best})$$

Now we are so close to the next best point h, and the only thing left to do is to minimize the acquisition function. Apparently, there are a lot of options for the optimization algorithms, such as gradient descent method or Quasi-Newton method. Here we will use two methods to minimize the first acquisition function separately: the most classical choice, gradient descent method with line search, and the most frequently used algorithm for this problem, BFGS algorithm. Two methods can lead to totally different performance of BO.

2.2 BO with Gradient Descent Method

When it comes to optimization in machine learning, gradient descent method is probably the first idea coming to one's mind. The pseudo code of the simplest gradient descent method is as follows:

```
Require: initial $x_0 \in \mathbb{R}^n$
  for $k = 0, 1, 2, \cdots,$ do
    Compute $\nabla f(x_k)$
    Use line search for determining $t_k$
    $x_{k+1} \leftarrow x_k - t_k \nabla f(x_k)$
    if $\|\nabla f(x_k)\|_2 \leq \epsilon$ then
      Stop and return $x_k$
    end if
  end for
```

$\nabla f(x_k)$ is computed by forward difference method: $(f(x_k + \delta) - f(x_k))/\delta$, where δ is set to some small value such as 10^{-6} in our project. The stopping criterion evaluates the norm of the gradient. If $\|\nabla f(x_k)\|_2 \leq \epsilon$, where ϵ is set to 10^{-5}, the program finds the minimum. Usually line search is used in order to avoid too long or too short step size t_k. Here we employ the bisection algorithm for line search, given by the following pseudo code:

```
Require: $a = 0, b \in \mathbb{R}_+$ such that $g'(a) < 0, g'(b) > 0$
  while $|a - b| > \epsilon$ do
    Set $c = (a+b)/2$ Evaluate $g'(c)$
    if $g'(c) > 0$ then
      Set $b \leftarrow c$
    else
      Set $a \leftarrow c$
    end if
  end while
```

Gradient descent method is easy to implement and doesn't take much computational resource for every iteration. However, sometimes it could take too many iterations (between 50 and 70 for most cases) to find the local minimum, which is quite inefficient. What's worse, searching step size by bisection is also a big overhead. Hence, compared by BO with BFGS method, BO using gradient descent algorithm takes much more time, about 227 seconds, to finish 100 iterations. And we have to limit our maximum number of iterations to 50 for our gradient descent algorithm to shorten the running time.

By means of minimizing the first acquisition function through gradient descent method, the highest accuracy found by Bayesian optimization within 100 iterations is 85.9259%. The corresponding pair of $(\log_{10} C, \log_{10} \gamma)$ is $(6.37958714, -8.68603352)$ ($(C, \gamma) \approx (2.3966 \times 10^6, 2.0605 \times 10^{-9})$). Given the previous discussion in the first section, to achieve such a high accuracy, we scale the training data by standard scaler. Besides this, as mentioned in section 2.1, one also needs to adjust the value of the kernel hyperparameter α to improve the performance. In this section, we choose $\alpha = 0.5$. Fig. 2.1 illustrates how the best-so-far accuracy varies on the iteration number.

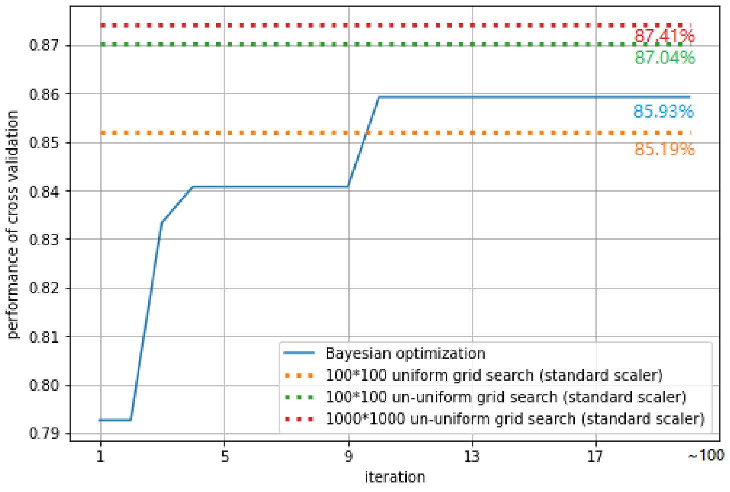

Figure 2.1: best-so-far accuracy vs iteration number.

From Fig. 2.1, we observe that the gradient descent based Bayesian optimization can easily find a pair of $(\log_{10} C, \log_{10} \gamma)$ performing better than the best pair found by uniform grid search, within no more than 20 iterations. This is quite impressive, especially given that BO takes much less time than grid search method does. However, BO fails to break through 85.9259% after the 10th iteration. As a result, the best accuracy found by BO doesn't exceed 87.04% found by 100×100 un-uniform grid search.

It's worth noting that if we switch from the first acquisition function to the second type, $a(h, y_{best}) = \mathbb{E}(\min\{G(h) - y_{best}, 0\})$, the procedure of optimizing the acquisition function could spend a great deal of more time, and the best performance found is 85.5556%. Apparently, the classical gradient descent method is not a good choice for the second acquisition function. As this report mainly focuses on the first type, detailed discussion will not be given here.

The distribution of the points found using BO (inside the optimization domain) is not as concentrated as we expected. Instead, 100 points form several small clusters (Fig. 2.2). We suppose it is because the gradient descent algorithm can easily get stuck in the local minimums around the random initial positions which are distributed in various locations inside the domain.

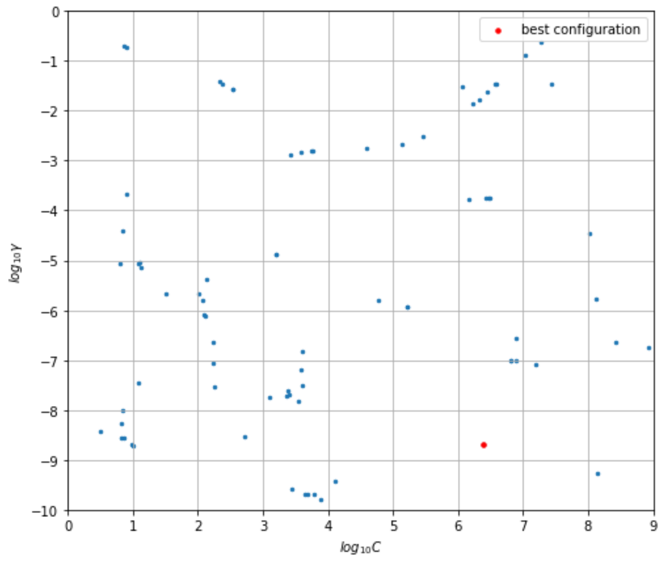

Figure 2.2: Distribution of 100 points found by gradient descent based Bayesian optimization. The red point indicates the configuration performing best, $(6.37958714, -8.68603352)$.

At the end of this section, let's take a glimpse of the visualization of the first acquisition function along with the gradient descent optimization. Fig. 2.3 shows the evolution of the shape of the first acquisition function by sampling the function of 4 random iterations from 100 BO iterations. The red line represents the path that the gradient descent method takes to find the new best point. From Fig. 2.3, we observe that the acquisition function becomes more and more 'tortuous' during Bayesian optimization. This may not be a good thing for the gradient descent method since such 'tortuous' shape could lead to numerical instability to some extent. Additionally, we also find that the performance of the gradient descent method is highly sensitive to the starting position and the algorithm cannot always find an appropriate local minimum (e.g. the 13-th iteration in Fig. 2.3).

A major drawback of the gradient descent method, as what we have discussed earlier, is the big computational cost. Although computing and updating the gradient itself is not so that expensive within a single iteration, a large number of iterations can make the optimization process quite inefficient. Bisection line search can help reduce the total amount of the iterations, but it introduces a new loop which again brings new computational overhead. Usually the number of the iterations is more than 50. To balance the trade-off between the performance and the time cost, we limit the maximum number of iterations to 50 for the gradient descent algorithm. In the next section, we will introduce another algorithm to optimize the acquisition function, BFGS method. You will find that the various performance of BO equipped with BFGS method is quite different with what we got in this section.

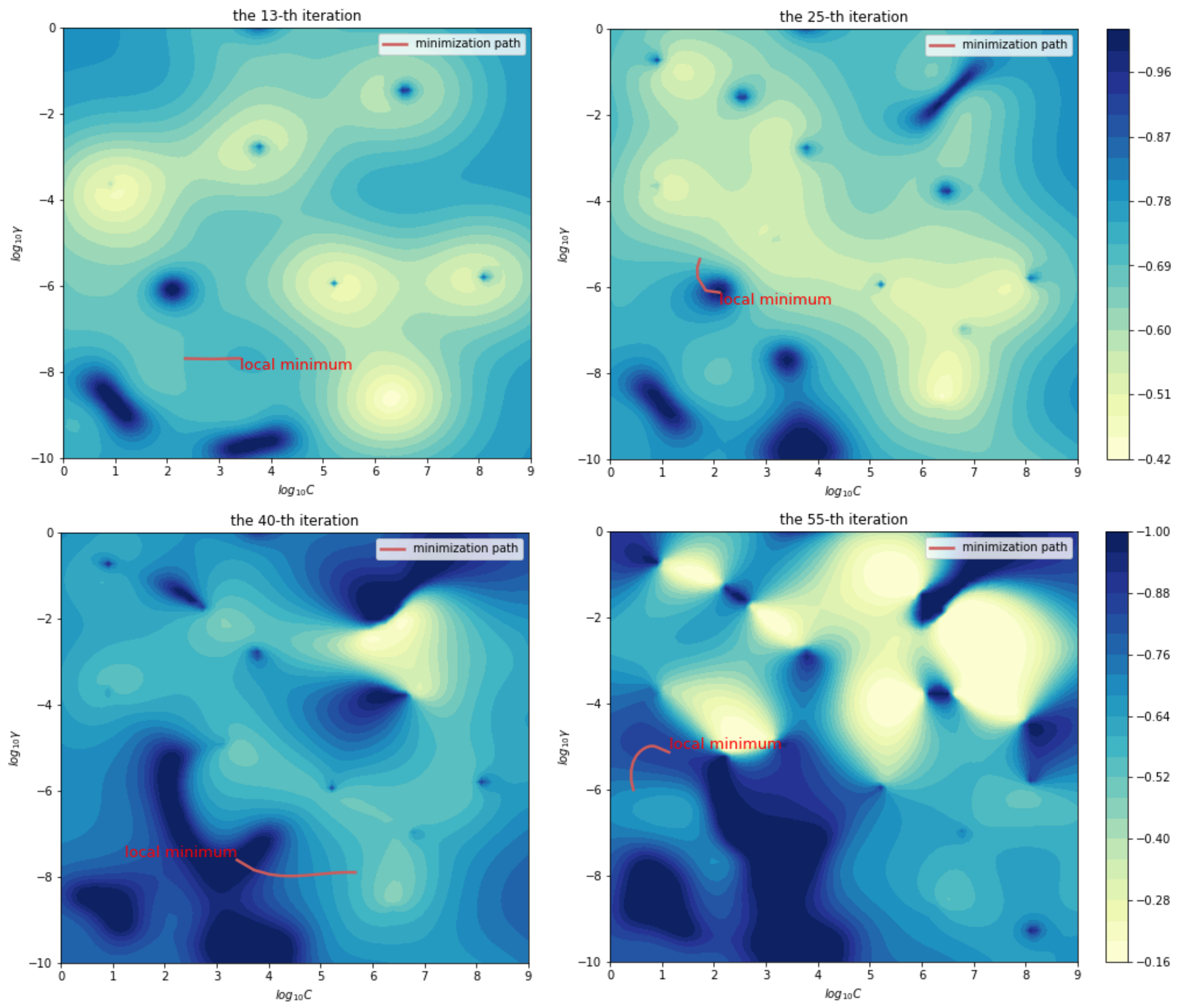

Figure 2.3: Shape of the first acquisition function for four random BO iterations along with the path that the gradient descent method takes to find the new best point.

2.3 BO with BFGS Method

Besides the classical gradient descent algorithm, We also implemented a Quasi-Newton method to minimize the acquisition function. Using exact line search, Broyden's class of quasi-Newton methods, which includes Broyden–Fletcher–Goldfarb–Shanno (BFGS) methods, achieves global and superlinear convergence in finding minimizers of convex, continuously-differentiable functions [1]. In this project, we implemented a modified BFGS algorithm designed by Li and Fukushima [2], which converges globally for convex and non-convex functions. This modified algorithm was chosen as the acquisition functions are highly non-convex, as shown in Fig. 2.3.

For each iteration k, define $\gamma_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, $t_k = 1 + \max\left\{-\frac{\gamma_k^T s_k}{\|s_k\|_2^2}\right\}$, $s_k = x_{k+1} - x_k$, and $y_k = \gamma_k + t_k\|\nabla f(x_k)\|_2 s_k$. The algorithm is then given by the following pseudo-code:

```
**Require:** initial $x_0 \in \mathbb{R}^n$, initialize $B_0$ as the $n \times n$ identity matrix, $\sigma \in \left(0, \frac{1}{2}\right)$, $\rho \in (0,1)$
for $k = 0, 1, 2, \cdots$, do
    Solve for $p_k$ in $B_k p_k = \nabla f(x_k)$
    Find smallest non-negative integer $j_k$ fulfilling $f(x_k + \rho^{j_k}) \leq f(x_k) + \sigma\rho^{j_k}\nabla f(x_k)^T p_k$
    $\lambda_k \leftarrow \rho^{j_k}$
    $x_{k+1} \leftarrow x_k + \lambda_k p_k$
    Update the Hessian approximation matrix by setting $B_{k+t} \leftarrow B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$
    if $\|\nabla f(x_k)\|_2 \leq \epsilon$ then
        Stop and return $x_k$
    end if
end for
```

The above algorithm generates a sequence of points $\{x_k\}$ which converges superlinearly to a point x^*, with $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite, where f is the acquisition function being minimized.

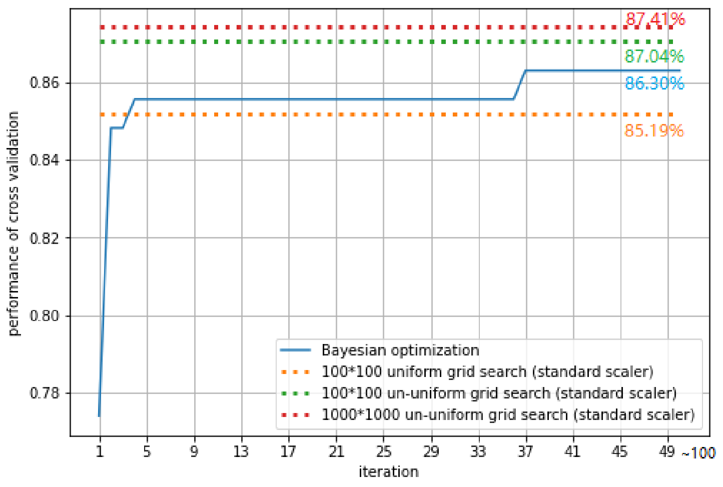

Figure 2.4: best-so-far accuracy vs iteration number.

Now let's see by applying BFGS algorithm above if we can improve the performance of BO. We again standardize our training data and then switch the kernel size α from 0.5 to 1.0, since with the former α BFGS-BO cannot find a better accuracy efficiently. It's worth noting that a suitable kernel size can depend on many settings of BO such as the model $G(h)$ itself and the choice of the optimization algorithm. Sometimes an inappropriate α could make the problem ill-conditioned and it impossible to compute $\mu(h)$ and $\sigma(h)$ that are necessary for constructing the acquisition function.

Figure 2.4 shows that BFGS-based Bayesian optimization takes 3 iterations to find a configuration of $(\log_{10} C, \log_{10}\gamma)$ which performs better than the best pair found by 100×100 uniform grid search. The final best accuracy found within 100 iterations is 86.2963% (found in the 36-th iteration), and the corresponding pair is $(8.8213433, -7.56759872)$. In addition, what's much better than using gradient descent method is that BO equipped with BFGS takes much less time to finish 100 iterations. The total running time is around 60 seconds. However, BFGS-BO also fails to find an accuracy higher than one found by 100×100 un-uniform grid search, 87.04%.

BFGS-based Bayesian optimization is also faster for the second acquisition function. It took 5705.4335 seconds to run 100 iterations of the GD-based Bayesian optimization method, achieving a peak performance score of 85.5556%. Meanwhile, it took 414.8515 seconds to run 100 iterations of the BFGS-based Bayesian optimization method with the same peak performance score.

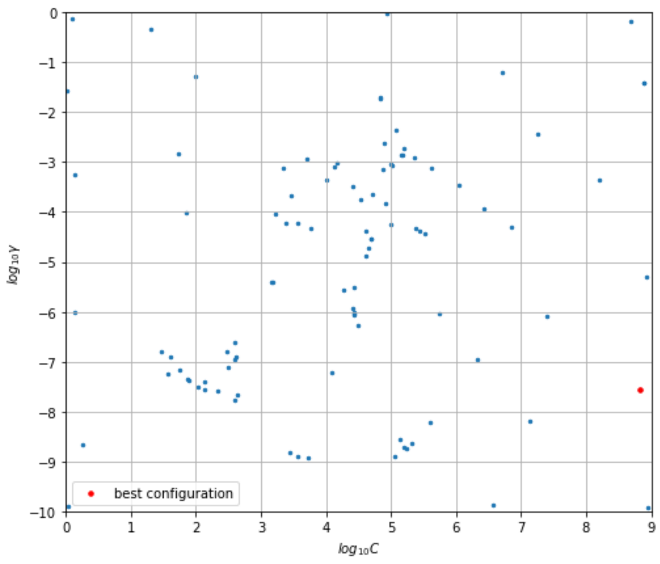

Figure 2.5: Distribution of 100 points found by BFGS-based Bayesian optimization. The red point indicates the configuration performing best, $(8.8213433, -7.56759872)$.

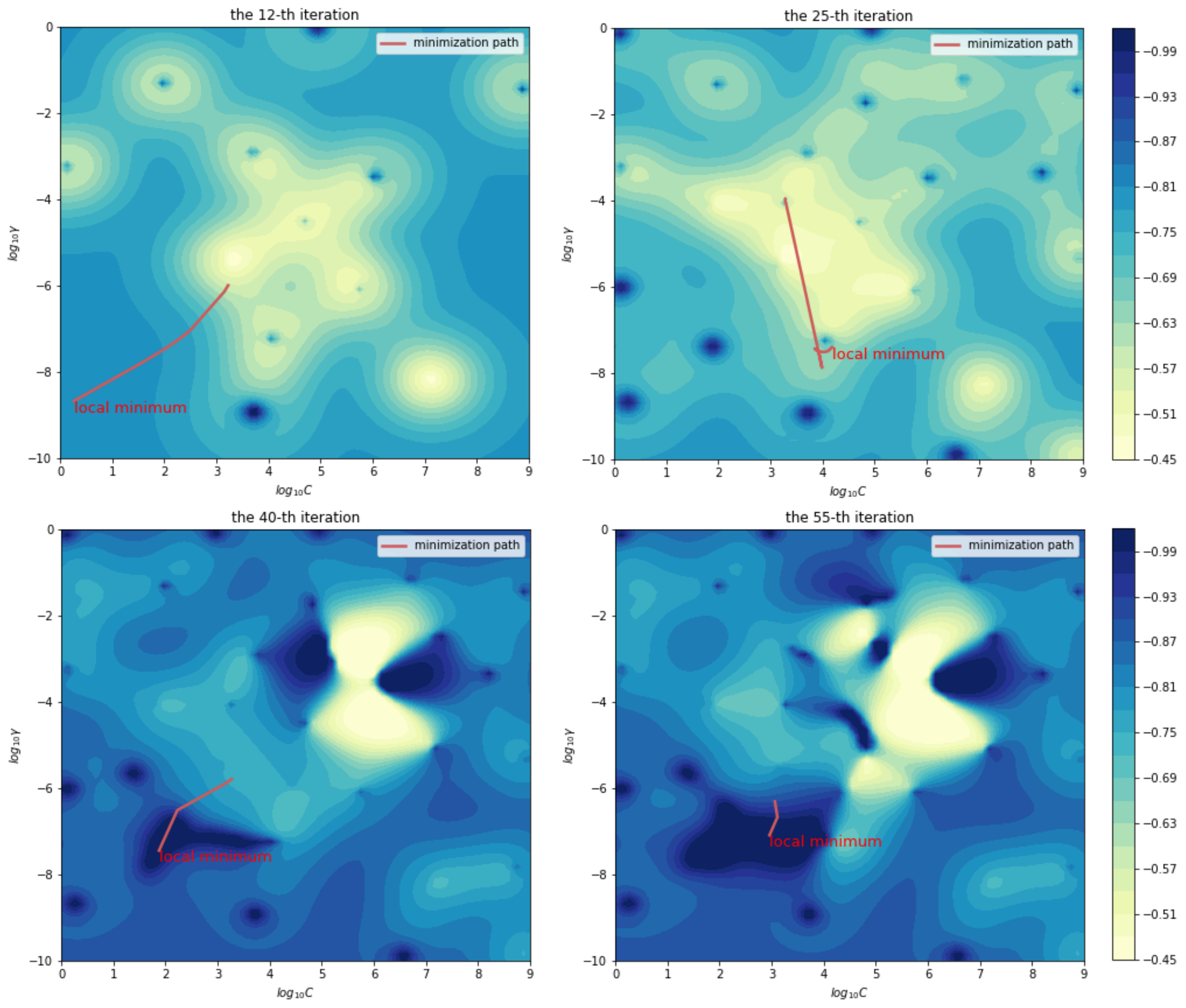

Figure 2.6: Shape of the first acquisition function for four random BO iterations along with the path that BFGS method takes to find the new best point.

From the distribution of 100 points found by BFGS-BO illustrated by Fig. 2.5, we can observe that BFGS gives a very different searching process, compared with the gradient descent method. The distribution of points is more concentrated. The best pair of $(\log_{10} C, \log_{10} \gamma)$ found is very close to the upper boundary of $\log_{10} C$.

Lastly, figure 2.6 depicts in detail how BFGS works on searching local minimums of the first acquisition function. From the red path shown in the figure we can directly feel the efficiency of BFGS method. BFGS usually can travel a long distance within a few iterations, which means it often takes fewer iterations to find the local minimum. Like what we did before, we again restrict the maximum iterations of BFGS to 50. However, compared with gradient descent, it is much less frequent for BFGS to consume more than 50 iterations to find a minimum.

3 Conclusion

Our implementation shows that Bayesian optimization is indeed an extremely powerful tool for searching optimal hyperparameters, yet its performance highly depends on other additional 'hyperparameters', such as the kernel size α, the choice of optimization algorithms, and even the random seeds for initialization. Such sensitivity to different settings forces us to adjust those 'hyperparameters' to find optimal hyperparameters for our model. To improve the robustness and stability of BO, more investigation on Gaussian model and optimization of acquisition functions is needed. By evaluating much fewer points (10-30), BO easily finds a higher accuracy than coarse uniform grid search (such as 100×100) does. However, our BO fails to beat un-uniform grid search. To excess the best accuracy of 87% found by un-uniform GS, we suppose that it's essential to increase the number of maximum iterations, and narrow the range of initial points before optimizing acquisition functions, as we've already known in which part of the domain most combinations of (C, γ) performing well gathers.

References

[1] Richard H. Byrd, J. Nocedal, and Ya-Xiang Yuan. "Global Convergence of a Class of Quasi-Newton Methods on Convex Problems". In: *SIAM Journal on Numerical Analysis* 24.5 (1987). DOI: 10.1137/0724077.

[2] Dong-Hui Li and Masao Fukushima. "A modified BFGS method and its global convergence in nonconvex minimization". In: *Journal of Computational and Applied Mathematics* 129.1-2 (2001), pp. 15–35. DOI: https://doi.org/10.1016/S0377-0427(00)00540-9.