Machine Learning Homework 5 Report

Yuan-Ming Chang
Student ID: A132525

1 Gaussian Process

1.1 Task1

Kernel Function: Rational Quadratic Kernel

$$k(x, x') = \sigma_f^2 \left(1 + \frac{(x - x')^2}{2\alpha l^2}\right)^{-\alpha} \tag{1}$$

Hyperparameters:

- **Alpha (α):** Scale mixture parameter. As $\alpha \to \infty$, the Rational Quadratic kernel approaches the RBF kernel.
- **Length parameter (l):** Controls the smoothness of the function
- **Sigma f (σ_f):** Controls the vertical variation.

Code:

```
def rational_quadratic_kernel(x1, x2, alpha, length_scale, sigma_f):
    d = np.sum((x1[:, None] - x2[None, :])**2, axis=-1)
    return (sigma_f ** 2) * (1 + d / (2 * alpha * length_scale ** 2)) ** (-alpha)
```

Prediction of the Regression

$$\mu(\mathbf{x}^*) = k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} \mathbf{y} \tag{2}$$

$$\sigma^2(\mathbf{x}^*) = k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*) \tag{3}$$

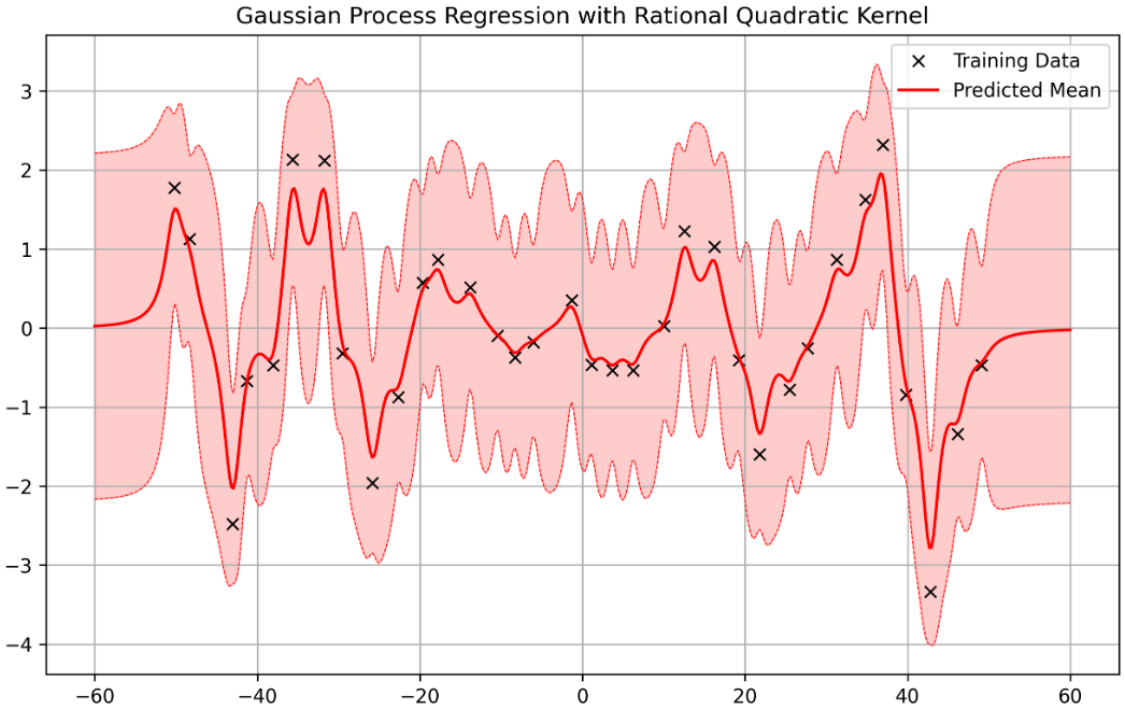
$$k^* = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1} \tag{4}$$

Following the equations, we return the mean and standard deviation of the covariance matrix of the data distributions. During the process, we need to calculate three pairs of kernel functions: (x, x), (x, x*) and (x*, x*). β^{-1} represents the variance in the noise of the observation in the Gaussian process regression, controlling how much noise we assume is present in the data. We place β^{-1} only on the diagonal to model independent observation noise for each point.

Code:

```python
def predict_kernel(X_train, y_train, X_test, alpha, length_scale, sigma_f):
    beta = 5
    k = rational_quadratic_kernel(X_train, X_train, alpha, length_scale, sigma_f)
    k_star = rational_quadratic_kernel(X_train, X_test, alpha, length_scale, sigma_f)
    k_star_star = rational_quadratic_kernel(X_test, X_test, alpha, length_scale, sigma_f)
    beta_I = np.eye(len(X_train)) * (1/beta)
    k += beta_I
    beta_I = np.eye(len(X_test)) * (1/beta)
    k_star_star += beta_I
    k_inv = np.linalg.inv(k)
    mu = k_star.T @ k_inv @ y_train
    cov = k_star_star - k_star.T @ k_inv @ k_star
    return mu, np.sqrt(np.diag(cov))
```

Experiment: Default hyperparameters: $\alpha = \mathrm{l} = \sigma_f = 1.0$



1.2 Task2

Optimize Kernel Parameters by Minimizing Negative Log-Likelihood

To improve the prediction, we optimize the kernel hyperparameters by minimizing the negative log marginal likelihood.maximizing the marginal log-likelihood of the data. We use the **scipy.optimize.minimize** function to find the optimal values of the kernel hyperparameters.

Notice that the purpose of the minimize function is to minimize the objective value, so we need to take the negative of the marginal log-likelihood. By minimizing the negative log-likelihood, we are effectively maximizing the original marginal log-likelihood, which helps us find the hyperparameters that best fit the observed data.

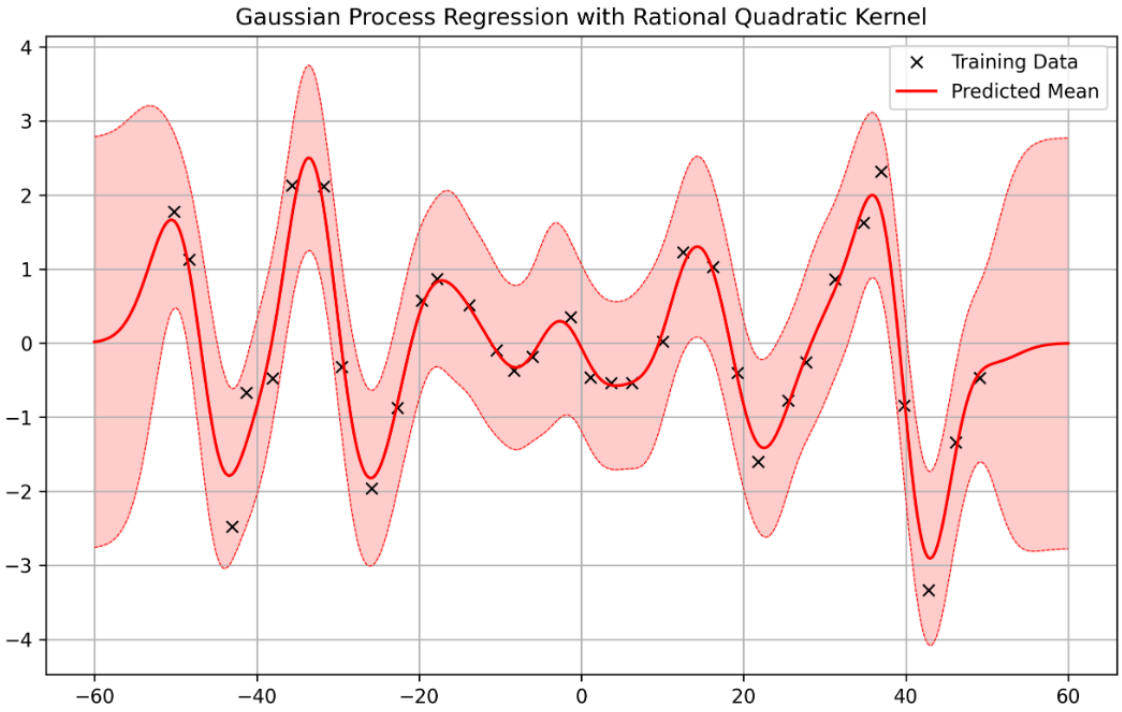
Code:

```python
def marginal_likelihood(parameters, X_train, y_train):
    beta = 5
    log_alpha, log_length_scale, log_sigma_f = parameters
    alpha = np.exp(log_alpha)
    length_scale = np.exp(log_length_scale)
    sigma_f = np.exp(log_sigma_f)
    k = rational_quadratic_kernel(X_train, X_train, alpha, length_scale, sigma_f)
    beta_I = np.eye(len(X_train)) * (1/beta)
    k += beta_I
    k_inv = np.linalg.inv(k)
    return -(- len(X_train) / 2 * np.log(2 * np.pi) - 0.5 * np.log(np.linalg.det(k)) - 0.5 * y_train.T @ k_inv @ y_train)
```

```python
parameters = np.log([1.0, 1.0, 1.0])
res = minimize(marginal_likelihood, parameters, args=(X_train, y_train), method='L-BFGS-B')
final_alpha, final_length_scale, final_sigma_f = np.exp(res.x)
mean, std = predict_kernel(X_train, y_train, X_test, final_alpha, final_length_scale, final_sigma_f)
```

We take the logarithm of the hyperparameters for the initial because they must remain positive during optimization. By optimizing in log-space, we allow the optimizer to search freely over the entire real line, while ensuring that the actual hyperparameters (after applying the exponential function) are always positive. This approach improves numerical stability and prevents the optimizer from proposing invalid (e.g., negative) values during the search.

Experiment: The optimized hypermarameters: $\alpha = 8.4556$, $\mathrm{l} = 1.1994$, $\sigma_f = 0.2724$



1.3 Observations and Discussion

After the two experiments above (the figures of Task 1 and Task 2), we observe that after hyperparameter optimization, the predicted mean function of the y distribution becomes smoother and better captures the underlying trend compared to before optimization. This improvement occurs because the optimization process adjusts key kernel hyperparameters, which helps avoid overfitting and improves generalization to unseen data. In addition, we observe that the confidence intervals become more stable, reflecting a more reliable estimation of uncertainty after optimization.

2 SVM on MNIST dataset

2.1 Task 1

I try three types of kernel functions:

- **Linear kernel**
- **Polynomial kernel**
- **RBF kernel**

Code: (-s : svm type, -t : kernel type)

```
kernel_types = ['linear', 'polynomial', 'rbf']
kernel_params = ['-s 0 -t 0', '-s 0 -t 1', '-s 0 -t 2']

for i, kernel in enumerate(kernel_types):
    print(f"Training SVM with {kernel} kernel...")
    start = time.time()
    model = svm_train(y_train.tolist(), X_train.tolist(), kernel_params[i])
    end = time.time()
    print(f"Training time: {end - start:.2f} seconds")
    print("Training:")
    _, train_acc, _ = svm_predict(y_train.tolist(), X_train.tolist(), model)
    print("Testing:")
    _, test_acc, _ = svm_predict(y_test.tolist(), X_test.tolist(), model)
    print()
```

I implement the SVM model using the functions **svm_train** and **svm_predict** from the **LIBSVM** library.

- **svm_train:** Input the training sets (X, y) and the option of the kernel parameters.
- **svm_predict:** Input the training/testing sets (X, y) and the trained model. Output the prediction result, we focus on **training/testing accuracy** and **time consuming**.

Experiment: The training and testing accuracy of three types of kernel function.

Linear kernel:

```
Training time: 3.48 seconds
Training accuracy of linear kernel:
Accuracy = 100% (5000/5000) (classification)
Testing accuracy of linear kernel:
Accuracy = 95% (2375/2500) (classification)
```

Polynomial kernel:

```
Training time: 17.10 seconds
Training:
Accuracy = 95.16% (4758/5000) (classification)
Testing:
Accuracy = 92.44% (2311/2500) (classification)
```

RBF kernel:

```
Training time: 5.52 seconds
Training:
Accuracy = 99.12% (4956/5000) (classification)
Testing:
Accuracy = 95.96% (2399/2500) (classification)
```

We compare the performance of different kernel functions and observe that, although the linear kernel achieves the highest training accuracy (100%), its testing accuracy (95%) is lower than that of the RBF kernel (95.96%), suggesting that it may suffer from slight overfitting. I also observe that the polynomial kernel takes much longer to train and yields the lowest training and testing accuracy.

2.2 Task 2

Various values were set for the parameters C (cost) ,γ (gamma), and d (degree) respectively.

- C: [0.1, 1, 10]
- γ: [0.001, 0.01, 0.1]
- d: [0, 1, 2]

I calculate the average cross-validation accuracy for each pair of parameters, setting the number of folds in cross-validation to 5. During training phase of each parameter pair, I save the validation accuracy to **acc_matrix**, which is used to save and visualize the results of the grid search. After the grid search, I calculate the testing accuracy for the best model.

Code:

Linear kernel: Perform grid search with C as the tunable parameter.

```python
def linear(X_train, y_train, X_test, y_test, C_list):
    final_C = 0
    best_acc = -1
    acc_matrix = np.zeros((len(C_list), 1))
    for i, C in enumerate(C_list):
        print(f"Training SVM with linear kernel, C={C}...")
        start = time.time()
        acc = svm_train(y_train.tolist(), X_train.tolist(), f'-s 0 -t 0 -c {C} -v 5 -q')
        end = time.time()
        acc_matrix[i, 0] = acc
        print(f"CV Accuracy = {acc:.2f}%, Time: {end - start:.2f} sec")

        if acc > best_acc:
            best_acc = acc
            final_C = C
            print("Update best parameters\n")

    print(f"Best parameters: C={final_C}, CV acc={best_acc:.2f}%")
    # Final model training
    model = svm_train(y_train.tolist(), X_train.tolist(), f'-s 0 -t 0 -c {final_C}')
    _, train_acc, _ = svm_predict(y_train.tolist(), X_train.tolist(), model)
    _, test_acc, _ = svm_predict(y_test.tolist(), X_test.tolist(), model)

    print(f"Train acc = {train_acc[0]:.2f}%, Test acc = {test_acc[0]:.2f}%")
    svm_save_model('result/task2_linear.txt', model)

    return acc_matrix
```

Polynomial kernel: Perform grid search with C, γ and d as the tunable parameter.

```python
def poly(X_test, y_test, X_train, y_train, C_list, gamma_list, d_list):
    final_C = final_gamma = final_d = 0
    best_acc = -1
    acc_matrix = np.zeros((len(C_list), len(gamma_list), len(d_list)))
    for i, C in enumerate(C_list):
        for j, gamma in enumerate(gamma_list):
            for k, d in enumerate(d_list):
                print(f"Training SVM with polynomial kernel, C={C}, gemma={gamma}, d={d}...")
                start = time.time()
                acc = svm_train(y_train.tolist(), X_train.tolist(), f'-s 0 -t 1 -c {C} -d {d} -g {gamma} -v 5 -q')
                end = time.time()
                acc_matrix[i, j, k] = acc
                print(f"CV Accuracy = {acc:.2f}%, Time: {end - start:.2f} sec")

                if acc > best_acc:
                    best_acc = acc
                    final_C = C
                    final_d = d
                    final_gamma = gamma
                    print("Update best parameters\n")

    print(f"Best parameters: C={final_C}, gamma={final_gamma}, d={final_d}, CV acc={best_acc:.2f}%")
    # Final model training
    model = svm_train(y_train.tolist(), X_train.tolist(), f'-s 0 -t 1 -c {final_C} -g {gamma} -d {final_d}')
    _, train_acc, _ = svm_predict(y_train.tolist(), X_train.tolist(), model)
    _, test_acc, _ = svm_predict(y_test.tolist(), X_test.tolist(), model)

    print(f"Train acc = {train_acc[0]:.2f}%, Test acc = {test_acc[0]:.2f}%")
    svm_save_model('result/task2_poly.txt', model)

    return acc_matrix
```

RBF kernel: Perform grid search with C and γ as the tunable parameter.

```python
def rbf(X_test, y_test, X_train, y_train, C_list, gamma_list):
    acc_matrix = np.zeros((len(C_list), len(gamma_list)))
    for i, C in enumerate(C_list):
        for j, gamma in enumerate(gamma_list):
            print(f"Training SVM with RBF kernel, C={C}, gamma={gamma}...")
            start = time.time()
            acc = svm_train(y_train.tolist(), X_train.tolist(), f'-s 0 -t 2 -c {C} -g {gamma} -v 5 -q')
            end = time.time()
            acc_matrix[i, j] = acc
            print(f"CV Accuracy = {acc:.2f}%, Time: {end - start:.2f} sec")

            if acc > best_acc:
                best_acc = acc
                final_C = C
                final_gamma = gamma
                print("Update best parameters\n")

    print(f"Best parameters: C={final_C}, gamma={final_gamma}, CV acc={best_acc:.2f}%")
    # Final model training
    model = svm_train(y_train.tolist(), X_train.tolist(), f'-s 0 -t 2 -c {final_C} -g {final_gamma}')
    _, train_acc, _ = svm_predict(y_train.tolist(), X_train.tolist(), model)
    _, test_acc, _ = svm_predict(y_test.tolist(), X_test.tolist(), model)

    print(f"Train acc = {train_acc[0]:.2f}%, Test acc = {test_acc[0]:.2f}%")
    svm_save_model('result/task2_rbf.txt', model)
```

Experiment:

Linear kernel function: $c = 1$, the model gets the highest performance (96.22%), it's testing accuracy is **95%**.

-c	Accuracy (cross-validation)
0.1	95.82%
1	**96.22%**
10	96.18%

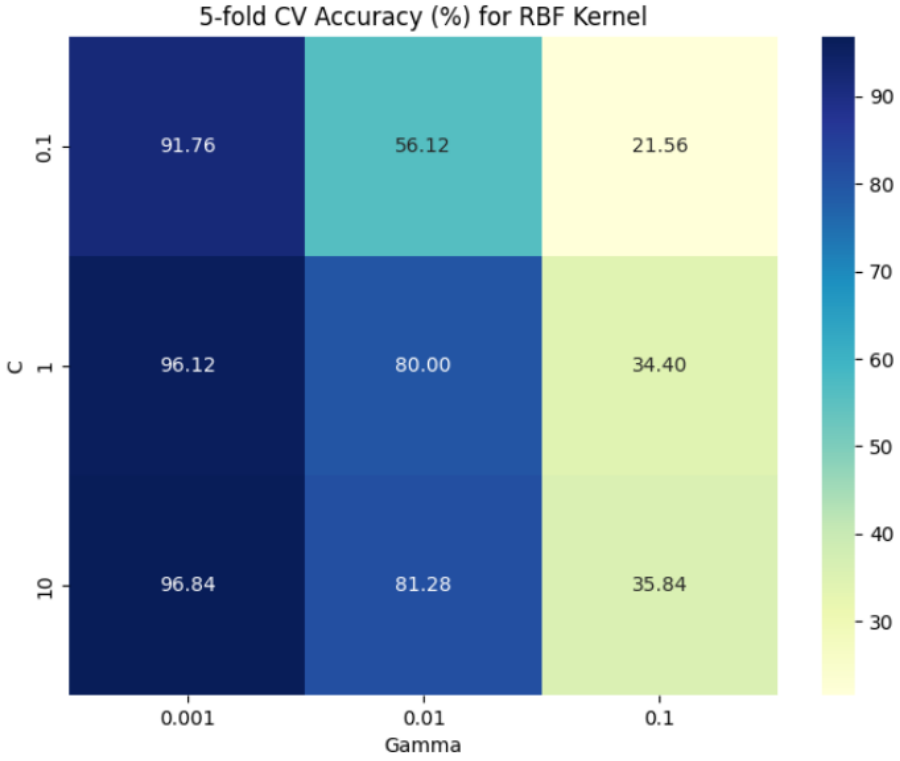
Polynomial kernel function: $c = 0.1$, $\gamma = 0.1$ and $d = 2$, the model gets the highest performance, it's testing accuracy is **97.4%**

-c	-g	-d	Accuracy (cross-validation)
0.1	0.001	0	20.00%
0.1	0.001	1	93.04%
0.1	0.001	2	54.48%
0.1	0.01	0	20.00%
0.1	0.01	1	95.80%
0.1	0.01	2	97.28%
0.1	0.1	0	20.00%
0.1	0.1	1	95.96%
0.1	0.1	2	97.08%
1	0.001	0	20.00%
1	0.001	1	95.96%
1	0.001	2	95.92%
1	0.01	0	20.00%
1	0.01	1	96.44%
1	0.01	2	97.32%
1	0.1	0	20.00%
1	0.1	1	95.84%
1	0.1	2	97.24%
10	0.001	0	20.00%
10	0.001	1	96.28%
10	0.001	2	**97.40%**
10	0.01	0	20.00%
10	0.01	1	95.32%
10	0.01	2	97.32%
10	0.1	0	20.00%
10	0.1	1	95.76%
10	0.1	2	97.28%

RBF kernel function: $c = 10$ and $\gamma = 0.001$, the model gets the highest performance, it's testing accuracy is **96.66%**

-c	-g	Accuracy (cross-validation)
0.1	0.001	91.76%
0.1	0.01	56.12%
0.1	0.1	21.56%
1	0.001	96.12%
1	0.01	80.00%
1	0.1	34.40%
10	0.001	**96.84%**
10	0.01	81.28%
10	0.1	35.84%

Heatmap of RBF kernel function:



Look at the heatmap, we observe that when $C = 10$, $\gamma = 0.001$, the average validation accuracy is the highest.

2.3 Task 3

I combine linear kernel and RBF kernel and get a new kernel function, the equation is below:

$$K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}' + \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right) \tag{5}$$

Code: (-t : kernel type, -g γ parameter)

```python
def linear_rbf_kernel(X1, X2, gamma):
    linear = np.dot(X1, X2.T)
    rbf = np.sum(X1**2, axis=1)[:, None] + np.sum(X2**2, axis=1) - 2 * np.dot(X1, X2.T)
    rbf = np.exp(-gamma * rbf)
    return linear + rbf
```

```python
# Task 3
final_C = final_gamma = 0
best_acc = -1
acc_matrix = np.zeros((len(C_list), len(gamma_list)))
for i, gamma in enumerate(gamma_list):
    K_train = linear_rbf_kernel(X_train, X_train, gamma)
    K_train = np.hstack([np.arange(1, K_train.shape[0]+1).reshape(-1, 1), K_train])

    K_test = linear_rbf_kernel(X_test, X_train, gamma)
    K_test = np.hstack([np.arange(1, K_test.shape[0]+1).reshape(-1, 1), K_test])
    for j, C in enumerate(C_list):
        print(f"Training SVM with linear + RBF kernel, C={C}, gamma={gamma}...")
        start = time.time()
        acc = svm_train(y_train.tolist(), K_train.tolist(), f'-s 0 -t 4 -c {C} -g {gamma} -v 5 -q')
        acc_matrix[j, i] = acc
        end = time.time()
        print(f"Training time: {end - start:.2f} seconds")
        print(f"CV Accuracy = {acc:.2f}%, Time: {end - start:.2f} sec")
        if acc > best_acc:
            best_acc = acc
            final_C = C
            final_gamma = gamma
            print("Update best parameters\n")
print(f"Best parameters: C={final_C}, gamma={final_gamma}, CV acc={best_acc:.2f}%")
# Final model training
model = svm_train(y_train.tolist(), K_train.tolist(), f'-s 0 -t 4 -c {final_C} -g {final_gamma}')

print("Training:")
_, acc_train, _ = svm_predict(y_train.tolist(), K_train.tolist(), model)
print("Testing:")
_, acc_test, _ = svm_predict(y_test.tolist(), K_test.tolist(), model)
```

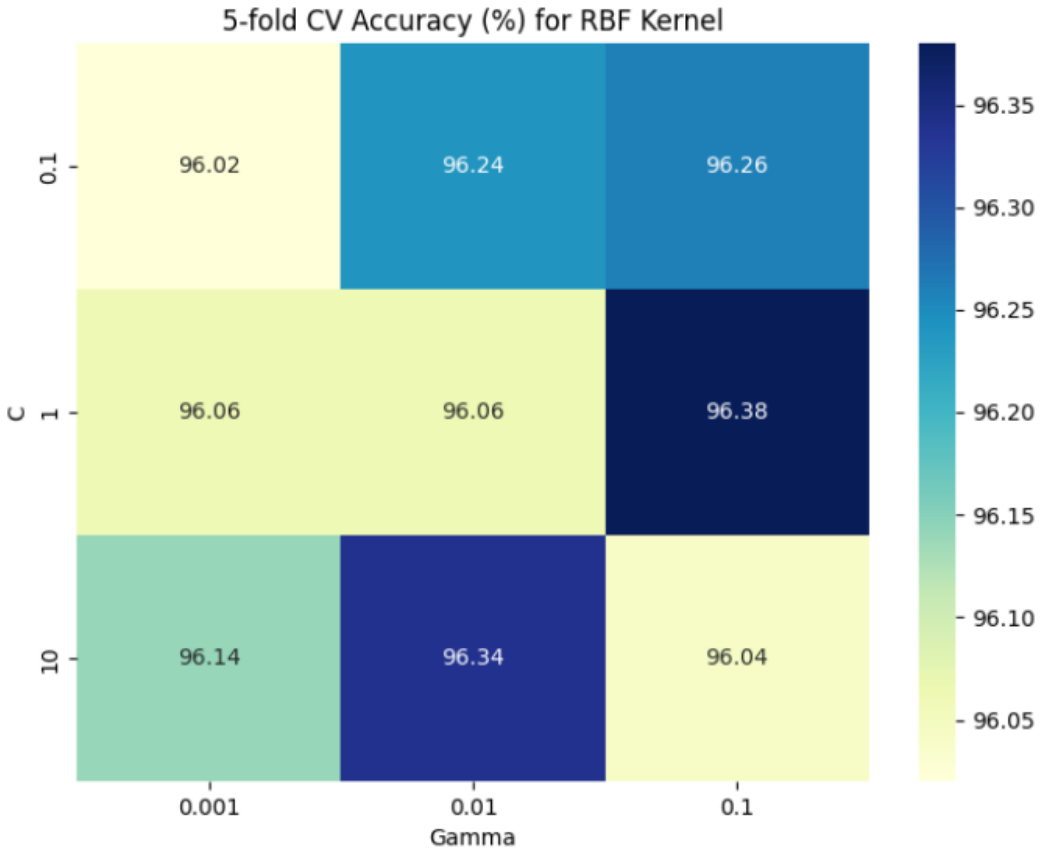
In this task, I also implement the grid search to seek for the best parameter. I set the kernel type (-t) to 4, which is required when precomputing all the kernel values manually. In other words, -t 4 is the way to use a custom kernel in LIBSVM.

Experiment:

-c	-g	Accuracy (cross-validation)
0.1	0.001	96.02%
0.1	0.01	96.24%
0.1	0.1	96.26%
1	0.001	96.06%
1	0.01	96.06%
1	0.1	**96.38%**
10	0.001	96.14%
10	0.01	96.34%
10	0.1	96.04%

The best model gets the testing accuracy in **94.96%**. In addition, we observe that the results across different parameter pairs are stable (around 96–96.5%).

Heatmap:



2.4 Observations and Discussion

Task 1

In the experiment section, with default parameters, we observe that the RBF kernel achieves the highest testing accuracy. Although the linear kernel reaches 100% training accuracy, its testing accuracy is the lowest, which suggests it may suffer from slight overfitting. Additionally, the training time and accuracy of the polynomial kernel are not satisfactory. This may be because, while it is capable of modeling complex patterns, it can be computationally intensive and sensitive to the choice of degree and other parameters. In this case, it may either underfit or overfit the data, leading to poorer performance and longer training times.

Task 2

I use grid search to find the best parameters for each kernel function in order to maximize their performance. As a result, in the experiment section, the performance of each kernel function improves compared with Task 1.

- **Linear kernel:** remain 95%
- **Polynomial kernel:** 92.44% → **97.40%**
- **RBF kernel:** 95.96% → **96.66%**

After observing the results, I guess that because the polynomial and RBF kernels can perform non-linear mapping, they achieve higher performance in most cases, especially the polynomial kernel, which showed poor performance in Task 1. Therefore, choosing the right parameters is important for it.

Task 3

I designed the linear + RBF kernel and performed grid search. In the experiment section, although its performance isn't the highest, the results across different parameter pairs appear more stable compared to other kernel functions, consistently achieving around 96–96.5%.

This result suggests that combining the linear and RBF kernels allows the model to capture both linear and non-linear patterns in the data simultaneously. Because the hybrid kernel has both flexibility (from RBF) and simplicity (from linear), it may be less sensitive to the exact parameter choices, leading to more consistent performance across different (C, γ) combinations.