I. Gaussian Process

1. Code:

First, we need to load the data. ↓

```
# Load the data from the file
data = np.loadtxt('./data/input.data')
X = data[:, 0].reshape(-1, 1)
Y = data[:, 1].reshape(-1, 1)
# print(X.shape, Y.shape)  # (34, 1) (34, 1)
```

In Task 1, we must apply the Rational Quadratic Kernel to compute similarities between data points. The formula for the Rational Quadratic Kernel is as follows:

$$k(x, x') = \sigma^2(1 + \frac{\|x - x'\|^2}{2\alpha l^2})^{-\alpha}$$

```
def rational_quadratic_kernel(x1, x2, sigma, alpha, length_scale):
    """
    Computes the Rational Quadratic kernel between two sets of points.

    Parameters:
    1. x1, x2: Input data, can be of different lengths
    2. sigma: The amplitude parameter
    3. alpha: The shape parameter
    4. length_scale: Length scale controls how far apart inputs must be to
       be considered correlated
    """
    distance = cdist(x1, x2, 'euclidean')
    kernel = (sigma**2) * (1 + distance**2 / (2 * alpha *
             length_scale**2))**(-alpha)

    return kernel
```

↑ In addition to the input data x1 and x2, three parameters—sigma (σ), alpha (α), and length scale (l)—must be set to define a Rational Quadratic Kernel. The parameter σ controls the overall amplitude of the kernel, α adjusts the kernel's flexibility, and l defines the correlation distance between input points. It is important to note that x1 and x2 in the formula can have different shapes. The function **scipy.spatial.distance.cdist**

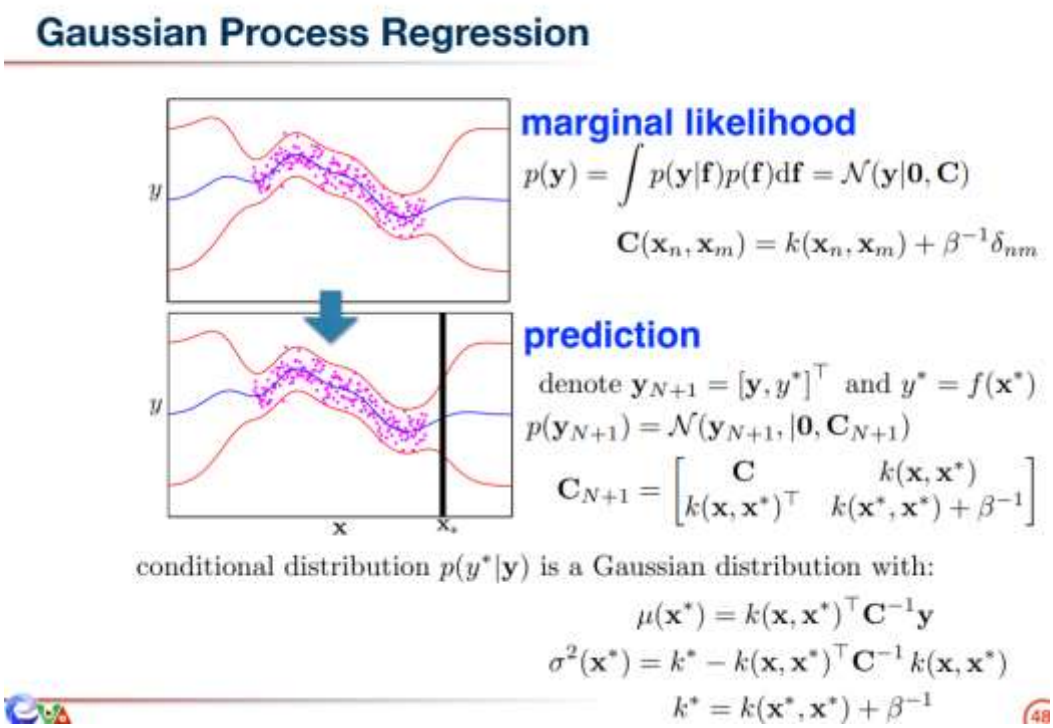
is used to compute the pairwise distances between the two input collections.

Next, the Rational Quadratic Kernel is applied to perform Gaussian Process Regression. The following formula can be found on page 48 of Kernel_GP_SVM.pdf:

$$C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$$

$$\mu(x^*) = k(x, x^*)^T C^{-1} y$$

$$\sigma^2(x^*) = k(x^*, x^*) + \beta^{-1} - k(x, x^*)^T C^{-1} k(x, x^*)$$




```
def gaussian_process_regression(X_train, Y_train, X_pred, sigma=1.0,
                                alpha=1.0, length_scale=1.0, beta=5):
    """
    Perform Gaussian Process Regression to predict the distribution of f at
    X_pred.

    Parameters:
    1. X_train: Training input data with shape (n, 1)
    2. Y_train: Training output data with shape (n, 1)
    3. X_pred: Prediction points with shape (m, 1)
    4. sigma: Amplitude parameter for kernel
    5. alpha: Shape parameter for kernel
```

```
    6. length_scale: Length scale parameter for kernel
    7. beta: Noise variance (inverse of observation noise)
    """
    # Compute covariance matrices
    C = rational_quadratic_kernel(X_train, X_train, sigma, alpha,
        length_scale) + np.eye(len(X_train)) / beta  # K(X, X) + (β^-1)I
    K_s = rational_quadratic_kernel(X_train, X_pred, sigma, alpha,
        length_scale)  # K(X, X*)
    K_ss = rational_quadratic_kernel(X_pred, X_pred, sigma, alpha,
        length_scale) + np.eye(len(X_pred)) / beta  # K(X*, X*) + (β^-1)I

    C_inv = np.linalg.inv(C)

    # Compute the mean and covariance of the posterior distribution
    mu_s = (K_s.T).dot(C_inv).dot(Y_train)  # μ(x*)
    cov_s = K_ss - (K_s.T).dot(C_inv).dot(K_s)  # σ^2(x*)

    return mu_s, cov_s
```

A function is then defined to compute the negative marginal log-likelihood, which is used to optimize the kernel parameters. The following formula can be found on page 52 of Kernel_GP_SVM.pdf:

$$\ln p(y|\theta) = -\frac{1}{2} ln\|C_\theta\| - \frac{1}{2} y^T C_\theta^{-1} y - \frac{N}{2} \ln (2\pi)$$

- Consider covariance function C with hyper-parameters θ

$$k_\theta(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 \exp\{-\theta_1 \frac{\|\mathbf{x}_n - \mathbf{x}_m\|^2}{2}\} + \theta_2 + \theta_3 \mathbf{x}_n^\top \mathbf{x}_m$$

  - Given $\mathcal{D} = \{(\mathbf{x}_i, y_i)_{n=1}^N\} = (\mathbf{X}, \mathbf{y})$, the marginal likelihood is function of θ

$$p(\mathbf{y}|\theta) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\theta)$$

$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2} \ln |\mathbf{C}_\theta| - \frac{1}{2} \mathbf{y}^\top \mathbf{C}_\theta^{-1} \mathbf{y} - \frac{N}{2} \ln (2\pi) \Rightarrow \frac{\partial \ln p(\mathbf{y}|\theta)}{\partial \theta}$$

To maximize the likelihood function while utilizing **scipy.optimize.minimize** for optimization, we multiply the function by -1. This transforms the problem into minimizing the negative log-likelihood, enabling the direct application of **scipy.optimize.minimize**. ↓

```
def negative_log_likelihood(params, X_train, Y_train, beta=5):
    """
    Compute the negative log marginal likelihood for the GP with Rational
    Quadratic kernel.

    Parameters:
    1. params: List of kernel parameters [sigma, alpha, length_scale]
    2. X_train: Training input data
    3. Y_train: Training output data
    4. beta: Noise variance (inverse of observation noise)
    """
    sigma, alpha, length_scale = params
    K = rational_quadratic_kernel(X_train, X_train, sigma, alpha,
                                  length_scale)
    C = K + np.eye(len(X_train)) / beta
    C_inv = np.linalg.inv(C)

    NLL = 0.5 * (np.log(np.linalg.det(C)) +
                 (Y_train.T).dot(C_inv).dot(Y_train) + len(C) * np.log(2 *
                 np.pi))
    # print(NLL.shape) # (1, 1)

    return NLL[0, 0]
```

↑ Note that matrix operations are being performed so that the result will be a (1,1) matrix. The value should be extracted using NLL[0, 0] to convert to a scalar.

Finally, an initial set of parameters (σ, α, and l, which are arbitrarily chosen in Task 1) must be set. After performing Gaussian Process Regression, the results are visualized. The 95% confidence interval is calculated using 1.96 × std. ↓

```
# Task 1
X_pred = np.linspace(-60, 60, 1000).reshape(-1, 1)
initial_params = [1.0, 1.0, 1.0]  # [sigma, alpha, length_scale]
beta = 5
mu_s, cov_s = gaussian_process_regression(X, Y, X_pred,
             sigma=initial_params[0], alpha=initial_params[1],
             length_scale=initial_params[2], beta=beta)
# print(mu_s.shape, cov_s.shape)  # (1000, 1) (1000, 1000)
```

```
std = np.sqrt(np.diag(cov_s))
print(f"Initial Parameters: σ={initial_params[0]}, α={initial_params[1]},
     length_scale={initial_params[2]}")
nll_initial = negative_log_likelihood(initial_params, X, Y, beta)
print(f"Initial Negative Log-Likelihood: {nll_initial:.3f}")

plt.figure(figsize=(10, 6))
plt.scatter(X, Y, color='red', label='Training Data')
plt.plot(X_pred, mu_s, color='blue', label='Mean Prediction')
plt.fill_between(X_pred.flatten(),
                 mu_s.flatten() - 1.96 * std,
                 mu_s.flatten() + 1.96 * std,
                 color='blue', alpha=0.2, label='95% Confidence Interval')
plt.title('Gaussian Process Regression with Rational Quadratic Kernel')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(-60, 60)
plt.legend()
plt.show()
```

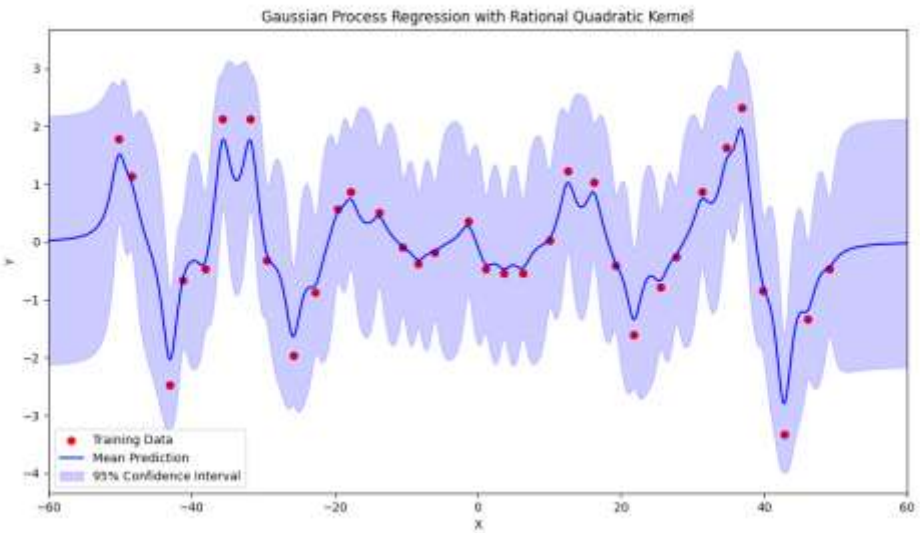
In Task 2, **scipy.optimize.minimize** was used for optimization. The required inputs include the function to be optimized (**negative_log_likelihood**), the parameters (initial_params), the necessary input data, and any additional constraints. For this assignment, the constraints I set are kernel parameters >= 0.01 and a maximum iteration limit of 1000. ↓

```
# Task 2
result = minimize(negative_log_likelihood, initial_params, args=(X, Y,
         beta), bounds=[(1e-2, None), (1e-2, None), (1e-2, None)],
         options={'maxiter': 1000})
optimized_params = result.x
print(f"Optimized Parameters: σ={optimized_params[0]:.3f},
α={optimized_params[1]:.3f}, length_scale={optimized_params[2]:.3f}")
nll_optimized = negative_log_likelihood(optimized_params, X, Y, beta)
print(f"Optimized Negative Log Likelihood: {nll_optimized:.3f}")
mu_s_optimized, cov_s_optimized = gaussian_process_regression(X, Y, X_pred,
                                  sigma=optimized_params[0],
                                  alpha=optimized_params[1],
```

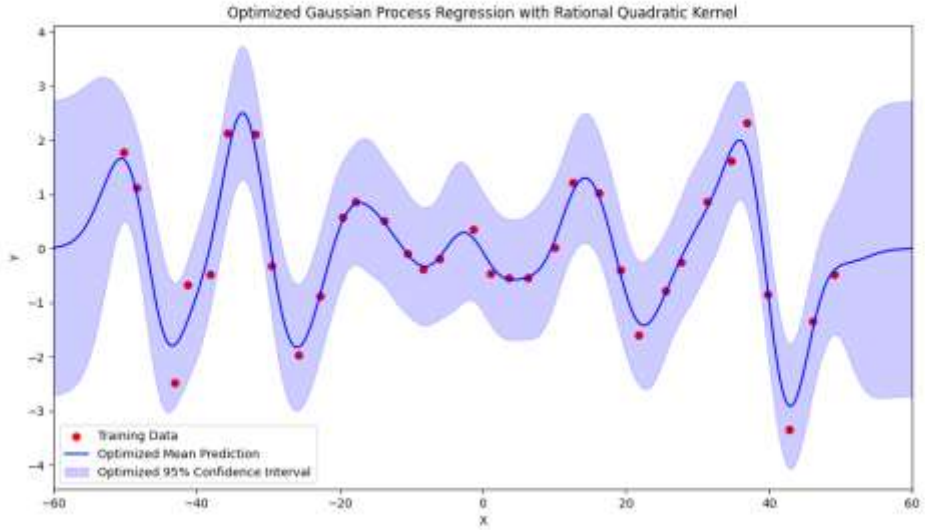
```
                                    length_scale=optimized_params[2],
                                    beta=beta)
std_optimized = np.sqrt(np.diag(cov_s_optimized))

plt.figure(figsize=(10, 6))
plt.scatter(X, Y, color='red', label='Training Data')
plt.plot(X_pred, mu_s_optimized, color='blue', label='Optimized Mean
         Prediction')
plt.fill_between(X_pred.flatten(),
                 mu_s_optimized.flatten() - 1.96 * std_optimized,
                 mu_s_optimized.flatten() + 1.96 * std_optimized,
                 color='blue', alpha=0.2, label='Optimized 95% Confidence
                 Interval')
plt.title('Optimized Gaussian Process Regression with Rational Quadratic
           Kernel')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(-60, 60)
plt.legend()
plt.show()
```

2. Experiment:

Task 1: σ =1.0, α =1.0, l =1.0 ↓

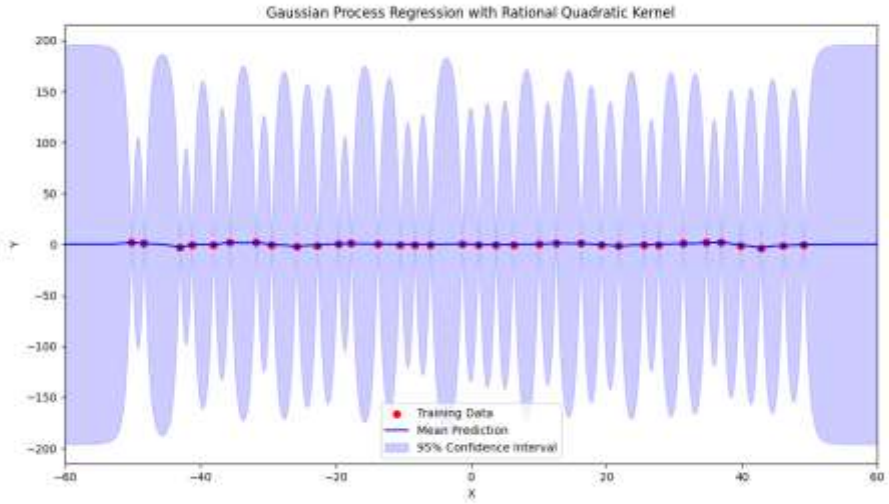


Task 2: σ =1.314, α =221.236, l =3.317 ↓



3. Observations and Discussion:

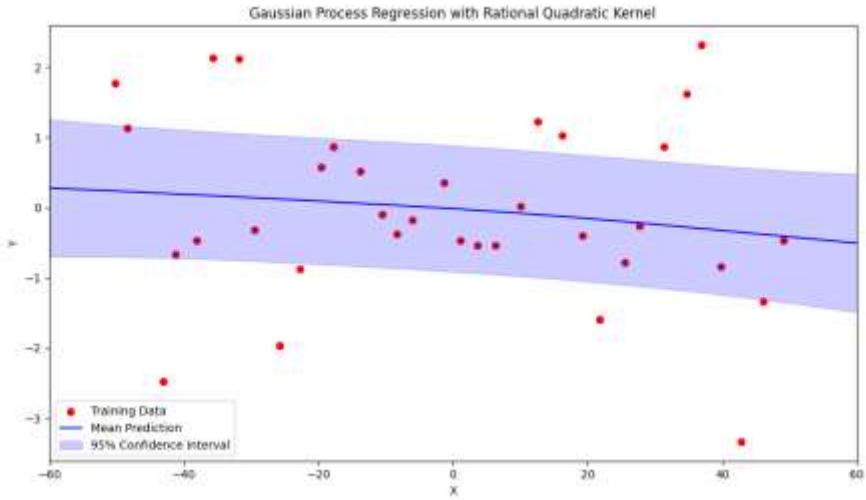
As discussed earlier, the three parameters—sigma, alpha, and length scale—are significant. Sigma controls the overall amplitude of the kernel function's output. One can observe from the output's y-axis values that as the sigma increases, the confidence interval (related to the standard deviation) estimated by the Gaussian process regression becomes wider. ↓



(↑ σ =100.0, α =1.0, l =1.0)

Length scale measures the degree to which the distance between input points affects the output. It is squared and corresponds to the Euclidean distance. As the length

scale increases, the influence of distance is "compressed," meaning the kernel function's decay with distance becomes slower. This allows points that are farther apart to have higher similarity still. ↓



(↑ σ =1.0, α =1.0, l =100.0)

Alpha's role is less intuitive. It controls the smoothness of the function's output, which is harder to describe visually. I will not provide a diagram to explain it.

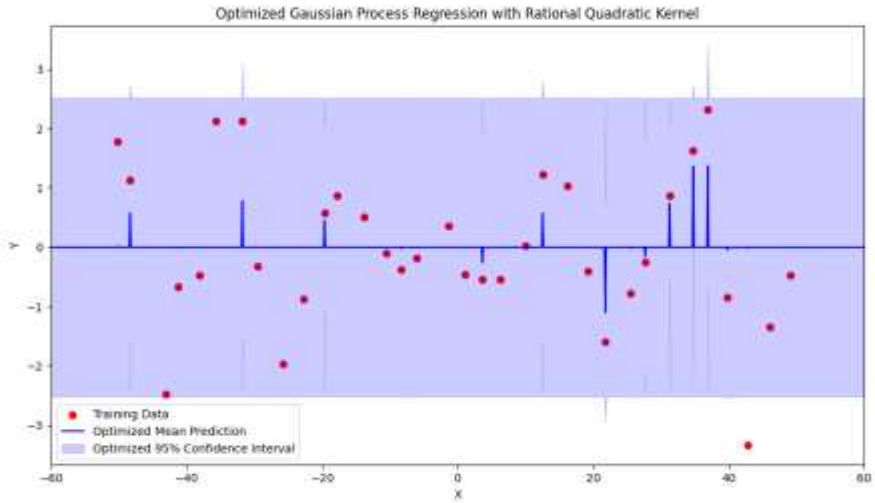
On the other hand, from the experiment results of Task 1 and Task 2, it is clear that the optimization has made the predictions finer and more accurate. The Negative Log Likelihood (NLL) values for the two tasks are as follows:

- Initial Negative Log-Likelihood: 55.923 (Task 1: σ =1.0, α =1.0, l =1.0)
- Optimized Negative Log-Likelihood: 50.685 (Task 2: σ =1.314, α =221.236, l =3.317)

Finally, I tried setting the initial values as follows:

1. σ =100.0, α =1.0, l =1.0
2. σ =1.0, α =100.0, l =1.0
3. σ =1.0, α =1.0, l =100.0

After optimization, only the final set of initial parameters, σ =1.0, α =1.0, l =100.0, showed a significant deviation from the results of Task 2. ↓



This highlights the importance of the length scale in measuring the similarity between two points. It also aligns with the intuition that the professor emphasized during the lecture: The kernel K should be chosen to express the property that for similar points x_n and x_m, the corresponding values y(x_n) and y(x_m) will be more strongly correlated than for dissimilar points.

II. SVM on MNIST

1. Code:

In this assignment (SVM on MNIST), I encountered issues related to specific package versions, which will be discussed in the Observations and Discussion section. I provide the package versions here to ensure the TA can run my code correctly. ↓

```
numpy==1.19.0
libsvm==3.23.0.4
scipy==1.5.1
pandas==1.1.5
```

After resolving the package version issues, we need to load the data. ↓

```
# Load the data
X_train = pd.read_csv('./data/X_train.csv', header=None).values
Y_train = pd.read_csv('./data/Y_train.csv', header=None).values.ravel()
X_test = pd.read_csv('./data/X_test.csv', header=None).values
```

```
Y_test = pd.read_csv('./data/Y_test.csv', header=None).values.ravel()
```

Before executing each task, I will reset the environment variables to ensure the experimental results are reproducible. ↓

```
def reset_environment():
    gc.collect()
    np.random.seed(42)
```

The methods already implemented in **libsvm** will be used in this task. For Task 1, linear, polynomial, and RBF kernels are applied. The **svm_train** function is utilized with the -t parameter to specify the corresponding kernel ('linear': 0, 'polynomial': 1, 'RBF': 2). The -c parameter refers to the coefficient of the penalty term, and -q is used to suppress the output messages. After training and obtaining the model's weights, **svm_predict** is used to generate predictions and calculate accuracy on the test set. It is important to note that these functions' input parameters should first have the label, followed by the data. ↓

```
# Task 1
reset_environment()
print("Task 1: Training and evaluating with different kernels")
kernels = {'linear': 0, 'polynomial': 1, 'RBF': 2}

for kernel_name, kernel_type in kernels.items():
    print(f"Training with {kernel_name} kernel...")
    model = svm_train(Y_train, X_train, f'-t {kernel_type} -c 1 -q')
    print(f"Evaluating with {kernel_name} kernel...")
    p_label, p_acc, p_val = svm_predict(Y_test, X_test, model)
    # print(f"{kernel_name} kernel accuracy: {p_acc[0]}%")
    print("------------------------------------------------------------")
```

In Task 2, we use grid search to determine suitable hyperparameters through cross-validation. For the linear kernel, the only tunable parameter is the penalty coefficient (-c). For the polynomial kernel, the adjustable parameters include the penalty coefficient (-c), the kernel coefficient (-g), and the polynomial degree (-d). Although the offset (-r) can also be tuned, I omitted it in this task to reduce runtime. The primary parameters to

adjust for the RBF kernel are the penalty coefficient (-c) and the kernel coefficient (-g). It is worth mentioning that, unlike in Task 1, 5-fold cross-validation is employed in Task 2 to optimize the hyperparameters. This is configured using the -v parameter in the options for **svm_train**. ↓

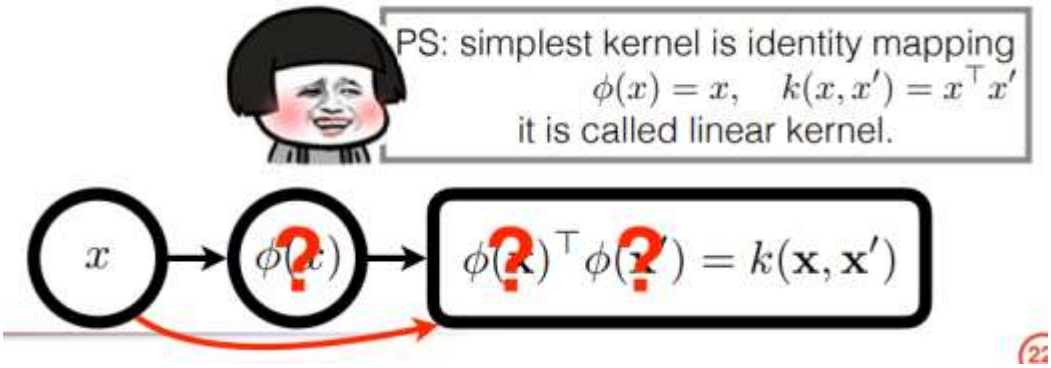
```
def grid_search(X_train, Y_train, kernel_type, param_grid):
    best_acc = 0
    best_params = None

    for C in param_grid['C']:
        for gamma in param_grid.get('gamma', [None]):
            for degree in param_grid.get('degree', [None]):
                options = f'-t {kernel_type} -c {C}'

                if gamma is not None:
                    options += f' -g {gamma}'
                if degree is not None:
                    options += f' -d {degree}'

                options += ' -v 5 -q'
                print(f"Training with options: {options}")

                if kernel_type == 4:
                    formatted_X_train = np.hstack((np.arange(1,
                     X_train.shape[0] + 1).reshape(-1, 1),
                     custom_kernel(X_train, X_train, gamma)))
                    acc = svm_train(Y_train, formatted_X_train.tolist(),
                                    options)
                else:
                    acc = svm_train(Y_train, X_train, options)

                if acc > best_acc:
                    best_acc = acc
                    best_params = {'C': C}

                    if gamma is not None:
                        best_params['gamma'] = gamma
```

```
                    if degree is not None:
                        best_params['degree'] = degree

    return best_params, best_acc


# Task 2
reset_environment()
print("\nTask 2: Grid search for best parameters")
param_grid_task2 = {
    'linear': {'C': [0.1, 1, 10]},
    'polynomial': {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1],
                   'degree': [2, 3]},
    'RBF': {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1]}
    }

for kernel_name, kernel_type in kernels.items():
    print(f"Performing grid search for {kernel_name} kernel...")
    best_params, best_acc = grid_search(X_train, Y_train, kernel_type,
                                        param_grid_task2[kernel_name])
    print(f"Best parameters for {kernel_name} kernel: {best_params}")
    print(f"Best 5-fold cross-validation accuracy: {best_acc:.2f}%")
    print("-------------------------------------------------------")

model_task2 = svm_train(Y_train, X_train, f'-t 2 -c 10 -g 0.01 -q')
print(f"Evaluating with RBF kernel...")
p_label, p_acc, p_val = svm_predict(Y_test, X_test, model_task2)
# print(f"RBF kernel accuracy: {p_acc[0]}%")
```

↑ Finally, based on the experimental results of 5-fold cross-validation on the training set, the RBF kernel with {'C': 10, 'gamma': 0.01} is used for inference on the test set.

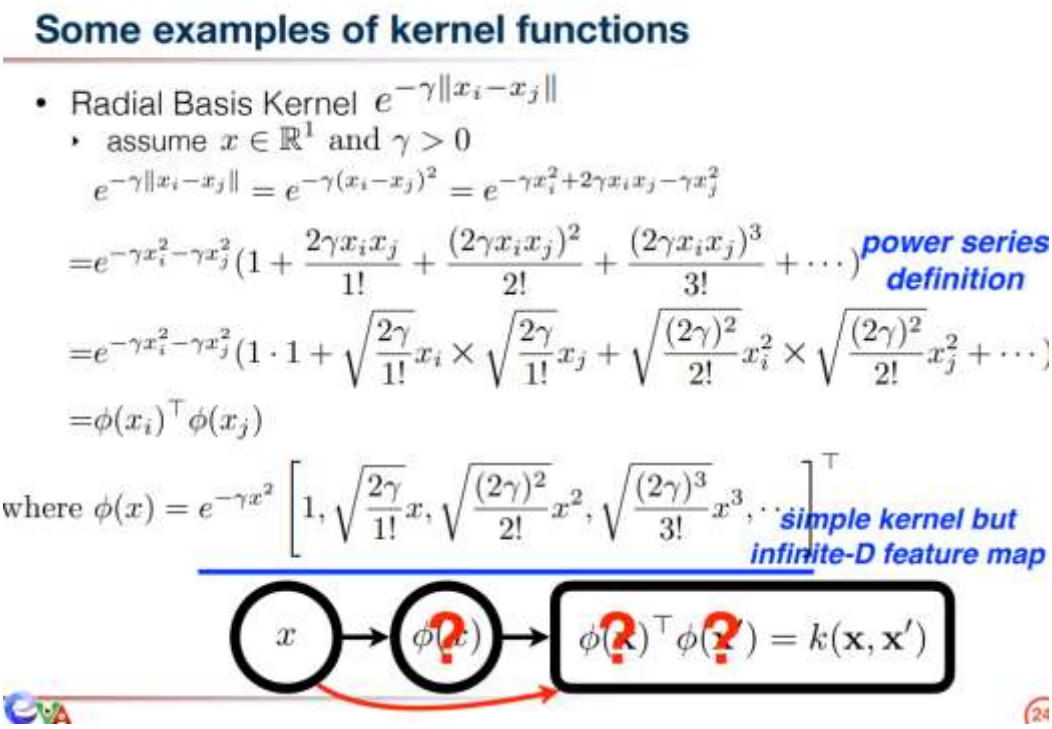
In task 3, a custom kernel is defined by combining the linear kernel and the RBF kernel. The formula for the linear kernel can be found on page 22 of the Kernel_GP_SVM.pdf:

$$k(x, x') = x^T x'$$



On the other hand, the formula for the RBF kernel can be found on page 24 of the Kernel_GP_SVM.pdf:

$$k(x, x') = e^{-\gamma(x-x')^2}$$



One can build a simple custom kernel based on this. ↓

```
def custom_kernel(x1, x2, gamma=0.01):
    # print(x1.shape, x2.shape)
    linear_kernel = np.dot(x1, x2.T)
    rbf_kernel = np.exp(-gamma * cdist(x1, x2, 'sqeuclidean'))

    return linear_kernel + rbf_kernel
```

To use the custom kernel, the option "-t 4 " is set when calling **svm_train**. However,

the input must first be manually transformed to match the custom kernel's processing and formatted to comply with **libsvm**'s requirements, which include adding sample indices starting from 1 in the first column. The following code snippet demonstrates the necessary steps: ↓

```
if kernel_type == 4:
    formatted_X_train = np.hstack((np.arange(1, X_train.shape[0] +
                        1).reshape(-1, 1), custom_kernel(X_train,
                        X_train, gamma)))
```

After preparing the data in the **libsvm** format, one can follow the same approach to use grid search to evaluate the performance of the custom kernel: ↓

```
# Task 3
reset_environment()
print("\nTask 3: Using custom kernel (linear + RBF) with Grid Search")
param_grid_task3 = {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1]}

best_params_task3, best_acc_task3 = grid_search(X_train, Y_train, 4,
                                                param_grid_task3)
print(f"Best parameters for custom kernel (linear + RBF):
     {best_params_task3}")
print(f"Best 5-fold cross-validation accuracy: {best_acc_task3:.2f}%")

formatted_X_train = np.hstack((np.arange(1, X_train.shape[0] + 1).reshape(-
                            1, 1), custom_kernel(X_train, X_train,
                            best_params_task3['gamma'])))
model_task3 = svm_train(Y_train, formatted_X_train.tolist(), f'-t 4 -c
                        {best_params_task3["C"]} -g
                        {best_params_task3["gamma"]} -q')
print(f"Evaluating with custom kernel (linear + RBF)...")
formatted_X_test = np.hstack((np.arange(1, X_test.shape[0] + 1).reshape(-1,
                            1), custom_kernel(X_test, X_train,
                            best_params_task3['gamma'])))
p_label, p_acc, p_val = svm_predict(Y_test, formatted_X_test.tolist(),
                                    model_task3)
# print(f"Custom kernel (linear + RBF) accuracy: {p_acc[0]}%")
```

2. Experiment:

I will present the experimental results in tables here to keep the layout organized.

Task 1 (default hyperparameters, no cross-validation):

Kernel Type	Accuracy (on test set)
Linear	95.08%
Polynomial	34.68%
RBF	95.32%

Task 2 (grid search for best hyperparameters with 5-fold cross-validation):

(1) Linear kernel:

-c	Accuracy (cross-validation)
0.1	96.74%
1	96.02%
10	96.24%

(2) Polynomial kernel:

-c	-g	-d	Accuracy (cross-validation)
0.1	0.001	2	45.62%
0.1	0.001	3	28.24%
0.1	0.01	2	94.7%
0.1	0.01	3	90.34%
0.1	0.1	2	98.16%
0.1	0.1	3	97.74%
1	0.001	2	80.42%
1	0.001	3	28.24%
1	0.01	2	97.62%

1	0.01	3	96.64%
1	0.1	2	98%
1	0.1	3	97.6%
10	0.001	2	94.7%
10	0.001	3	62.9%
10	0.01	2	98.14%
10	0.01	3	97.68%
10	0.1	2	98.14%
10	0.1	3	97.56%

(3) RBF kernel:

-c	-g	Accuracy (cross-validation)
0.1	0.001	92.64%
0.1	0.01	96.42%
0.1	0.1	54.42%
1	0.001	96.14%
1	0.01	97.8%
1	0.1	91.76%
10	0.001	97.12%
10	0.01	98.3%
10	0.1	92.48%

Finally, use the RBF kernel with the parameters "-c 10 -g 0.01", which achieved the highest accuracy on the training set, to test on the test set: ↓

-c	-g	Accuracy (on test set)
10	0.01	98.2%

Task 3 (linear kernel + RBF kernel together, grid search for best hyperparameters with 5-fold cross-validation):

-c	-g	Accuracy (cross-validation)
0.1	0.001	96.94%
0.1	0.01	96.94%
0.1	0.1	96.76%
1	0.001	96.2%
1	0.01	96.38%
1	0.1	96.58%
10	0.001	96.14%
10	0.01	96.32%
10	0.1	96.2%

The results of the test set: ↓

-c	-g	Accuracy (on test set)
0.1	0.001	95.8%

3. Observations and Discussion:

As I mentioned earlier, when the package is installed in a newer version, one may encounter the following error:

```
Traceback (most recent call last):
  File "D:\課程\機器學習\作業\HW5\svm_mnist.py", line 65, in <module>
    model = svm_train(Y_train, X_train, f'-t {kernel_type} -c 1 -q')
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\shihh\AppData\Local\Programs\Python\Python312\Lib\site-packages\libsvm\svmutil.py", line 85, in svm_train
    if isinstance(arg1, (list, tuple)) or (scipy and isinstance(arg1, scipy.ndarray)):
                                                                      ^^^^^^^^^^^^^
  File "C:\Users\shihh\AppData\Local\Programs\Python\Python312\Lib\site-packages\scipy\__init__.py", line 152, in __getattr__
    raise AttributeError(
AttributeError: Module 'scipy' has no attribute 'ndarray'
```

I found a similar issue online (the link will direct to a CSDN blog). In short, this problem only arises after updating the version of **scipy.** One can install an older library version to avoid encountering the error.

Based on the results of Task 1, one might assume that the polynomial kernel performs worse in this task. However, this is not entirely reasonable, as the linear kernel can be seen as a particular case of the polynomial kernel. Therefore, the polynomial kernel should not perform worse than the linear kernel. It turns out that the issue lies in the tuning of hyperparameters. In Task 2, after using grid search to find the appropriate hyperparameters, it becomes clear that the polynomial kernel can also achieve an accuracy of over 95% in this task.

The best predictive performance of the custom kernel (linear + RBF) after hyperparameter tuning did not surpass that of using the RBF kernel alone. This is because I naively combined it as linear + RBF when setting up the custom kernel without assigning adjustable weights. In theory, setting appropriate weights (a * linear + b * RBF) could allow the custom kernel to combine the strengths of both kernels, leading to a more robust model with better predictive performance.