HW5

I. Gaussian Process

1. Code (20%)

Task1:

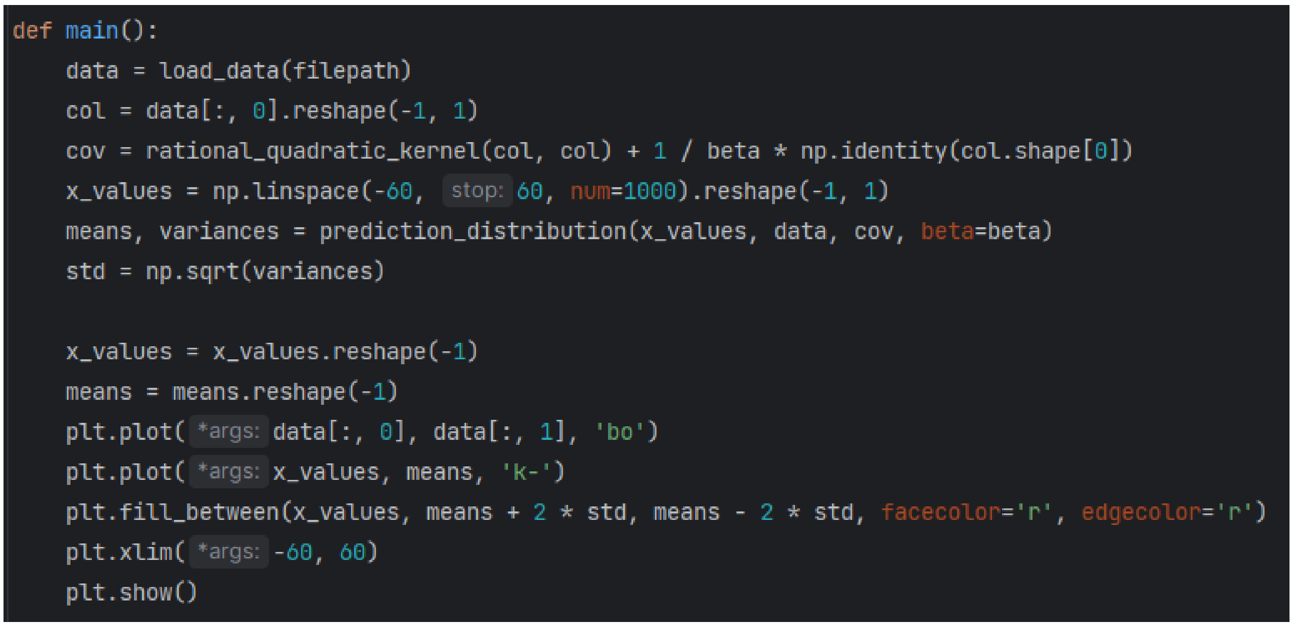
To calculate the covariance of the marginal likelihood, I utilize the rational quadratic kernel, with all its parameters set to 1. Subsequently, I compute the means and variances of the sample points generated by np.linspace(-60, 60, num=1000) using the prediction_distribution() function. Finally, visualize the results by plotting the computed means and variances.

The formula for rational quadratic kernel and mean and variance of prediction distribution are shown below.

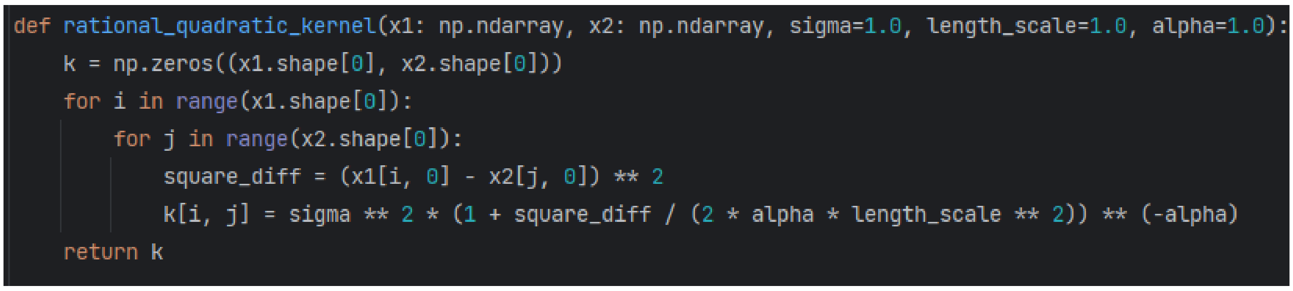
rational quadratic kernel: $k(x_i, x_j) = \sigma^2 \left(1 + \frac{\|x_i - x_j\|^2}{2\alpha\ell^2}\right)^{-\alpha}$

mean and variance of prediction distribution:

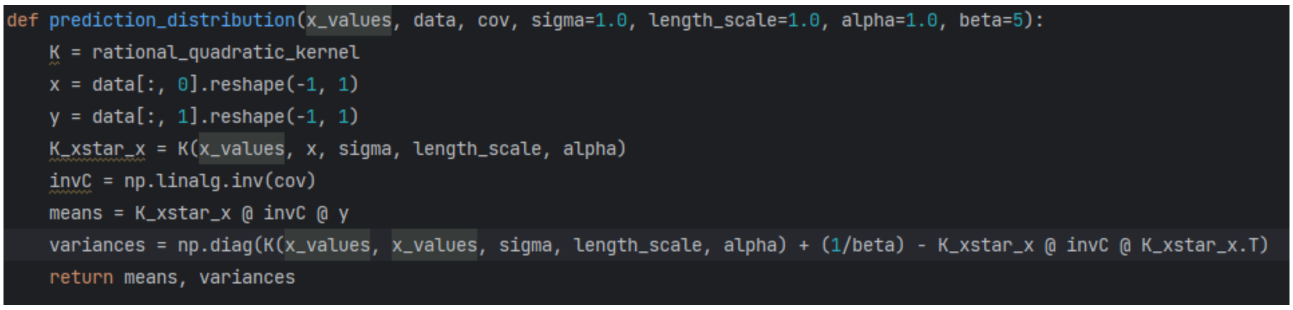
$$\mu(\mathbf{x}^*) = k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1}\mathbf{y}$$
$$\sigma^2(\mathbf{x}^*) = k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(\mathbf{x}, \mathbf{x}^*)$$
$$k^* = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1}$$



```python
def main():
    data = load_data(filepath)
    col = data[:, 0].reshape(-1, 1)
    cov = rational_quadratic_kernel(col, col) + 1 / beta * np.identity(col.shape[0])
    x_values = np.linspace(-60, stop: 60, num=1000).reshape(-1, 1)
    means, variances = prediction_distribution(x_values, data, cov, beta=beta)
    std = np.sqrt(variances)

    x_values = x_values.reshape(-1)
    means = means.reshape(-1)
    plt.plot( *args: data[:, 0], data[:, 1], 'bo')
    plt.plot( *args: x_values, means, 'k-')
    plt.fill_between(x_values, means + 2 * std, means - 2 * std, facecolor='r', edgecolor='r')
    plt.xlim( *args: -60, 60)
    plt.show()
```



```python
def rational_quadratic_kernel(x1: np.ndarray, x2: np.ndarray, sigma=1.0, length_scale=1.0, alpha=1.0):
    k = np.zeros((x1.shape[0], x2.shape[0]))
    for i in range(x1.shape[0]):
        for j in range(x2.shape[0]):
            square_diff = (x1[i, 0] - x2[j, 0]) ** 2
            k[i, j] = sigma ** 2 * (1 + square_diff / (2 * alpha * length_scale ** 2)) ** (-alpha)
    return k
```

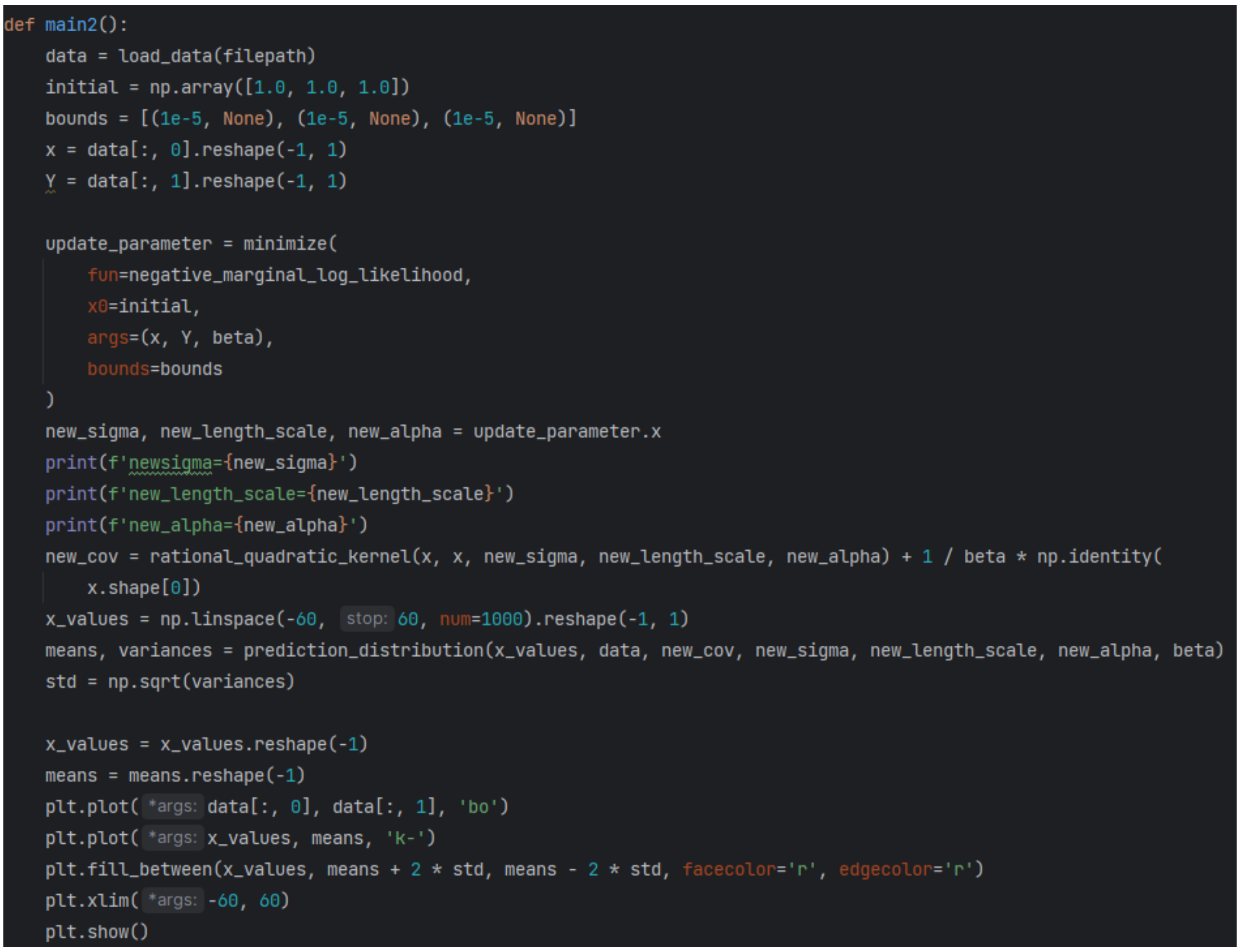


```
def prediction_distribution(x_values, data, cov, sigma=1.0, length_scale=1.0, alpha=1.0, beta=5):
    K = rational_quadratic_kernel
    x = data[:, 0].reshape(-1, 1)
    y = data[:, 1].reshape(-1, 1)
    K_xstar_x = K(x_values, x, sigma, length_scale, alpha)
    invC = np.linalg.inv(cov)
    means = K_xstar_x @ invC @ y
    variances = np.diag(K(x_values, x_values, sigma, length_scale, alpha) + (1/beta) - K_xstar_x @ invC @ K_xstar_x.T)
    return means, variances
```

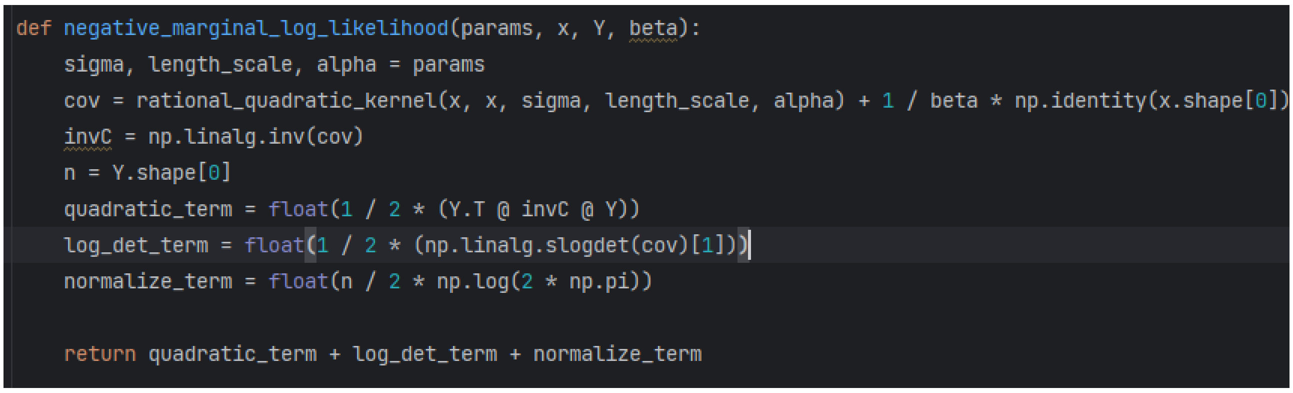
Task2:

I used scipy.optimize.minimize to minimize the negative marginal log likelihood. The marginal log likelihood shows in below. The initial parametric of kernel are all set to 1. After I got the new parameters of kernel , I do the same as task1 but using new parameters for kernel. The rational_quadratic_kernel() and prediction_distribution() are same as in task1.

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



```
def main2():
    data = load_data(filepath)
    initial = np.array([1.0, 1.0, 1.0])
    bounds = [(1e-5, None), (1e-5, None), (1e-5, None)]
    x = data[:, 0].reshape(-1, 1)
    Y = data[:, 1].reshape(-1, 1)

    update_parameter = minimize(
        fun=negative_marginal_log_likelihood,
        x0=initial,
        args=(x, Y, beta),
        bounds=bounds
    )
    new_sigma, new_length_scale, new_alpha = update_parameter.x
    print(f'newsigma={new_sigma}')
    print(f'new_length_scale={new_length_scale}')
    print(f'new_alpha={new_alpha}')
    new_cov = rational_quadratic_kernel(x, x, new_sigma, new_length_scale, new_alpha) + 1 / beta * np.identity(
        x.shape[0])
    x_values = np.linspace(-60,  stop: 60, num=1000).reshape(-1, 1)
    means, variances = prediction_distribution(x_values, data, new_cov, new_sigma, new_length_scale, new_alpha, beta)
    std = np.sqrt(variances)

    x_values = x_values.reshape(-1)
    means = means.reshape(-1)
    plt.plot( *args: data[:, 0], data[:, 1], 'bo')
    plt.plot( *args: x_values, means, 'k-')
    plt.fill_between(x_values, means + 2 * std, means - 2 * std, facecolor='r', edgecolor='r')
    plt.xlim( *args: -60, 60)
    plt.show()
```



```python
def negative_marginal_log_likelihood(params, x, Y, beta):
    sigma, length_scale, alpha = params
    cov = rational_quadratic_kernel(x, x, sigma, length_scale, alpha) + 1 / beta * np.identity(x.shape[0])
    invC = np.linalg.inv(cov)
    n = Y.shape[0]
    quadratic_term = float(1 / 2 * (Y.T @ invC @ Y))
    log_det_term = float(1 / 2 * (np.linalg.slogdet(cov)[1]))
    normalize_term = float(n / 2 * np.log(2 * np.pi))

    return quadratic_term + log_det_term + normalize_term
```

2. Experiments (20%)

Task1:

Initial hyperparameters: sigma=1.0, length_scale=1.0, alpha=1.0

Result:

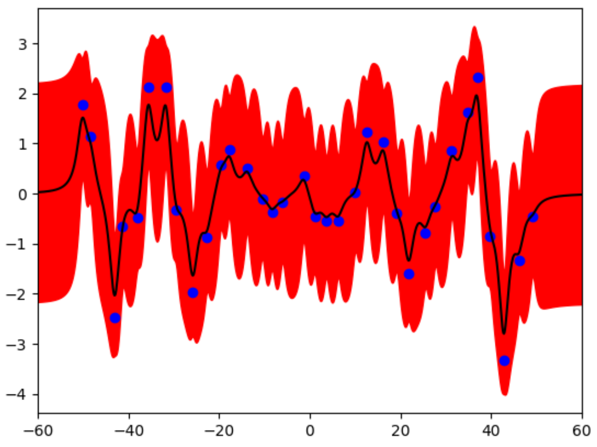


Image1:

Task2:

Initial hyperparameters: sigma=1.0, length_scale=1.0, alpha=1.0

Optimal hyperparameters:

sigma=1.3109717036753885

length_scale=3.3188614346512195

alpha=468.45411131705134

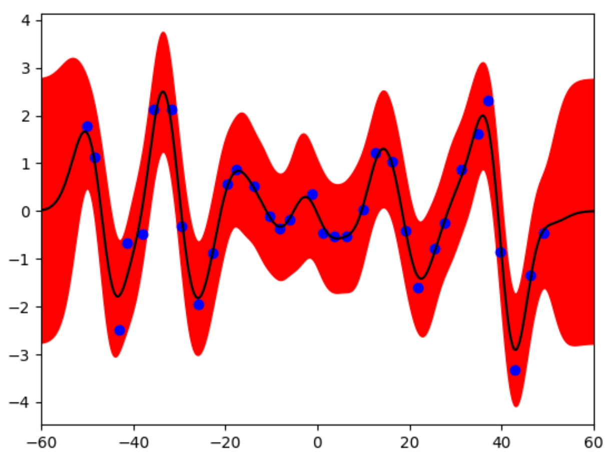


Image2:

3. Observations and Discussion (10%)

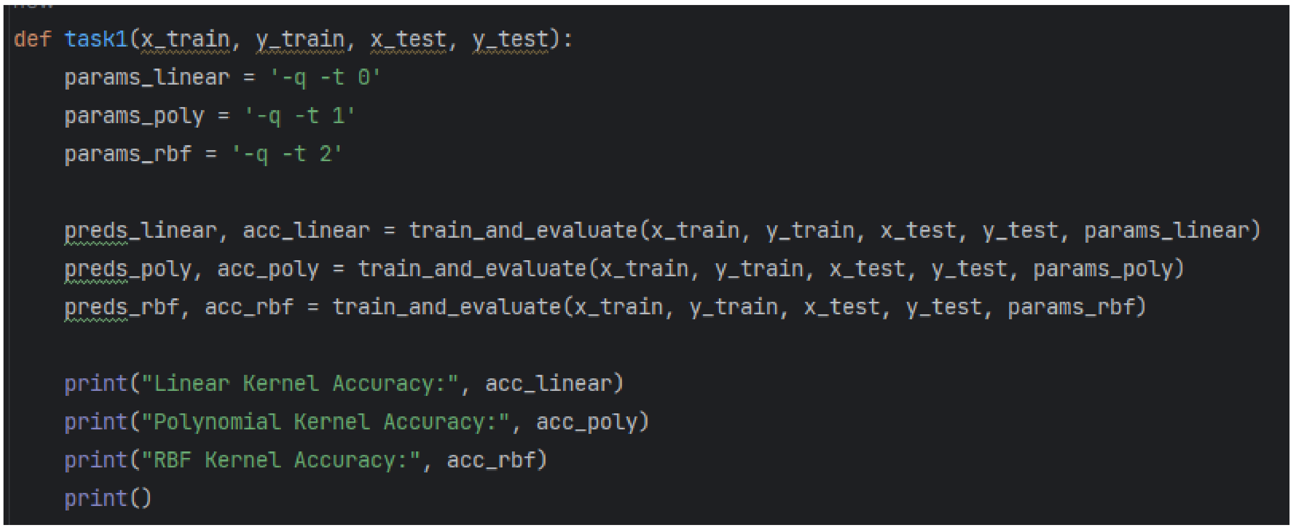
After optimizing the hyperparameters, the variance of the predictive distribution appears smaller within the range of sample points compared to before optimization. This phenomenon can be observed in Image1 and Image2. The reason is that the optimized hyperparameters adjust the kernel to better fit the training data, thereby reducing uncertainty in regions with sample points. In contrast, regions far from the observed points still exhibit higher uncertainty since the model has less information.

II. SVM on MNIST

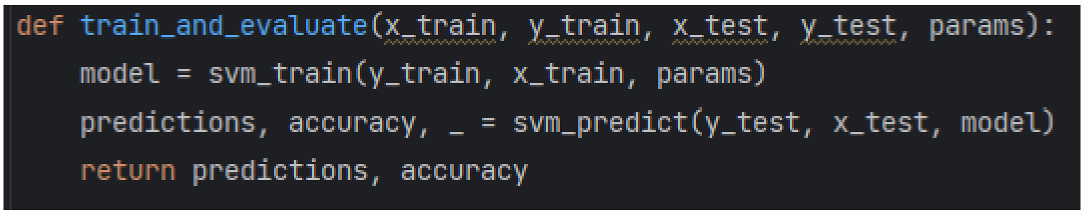
1. Code (20%)

Task1:

Since LIBSVM already provides linear, polynomial, and RBF kernels, I select the corresponding parameters for each kernel and pass them to the train_and_evaluate() function. This function trains the model using the specified kernel and evaluates its performance by predicting the results, ultimately calculating the accuracy of the predictions.



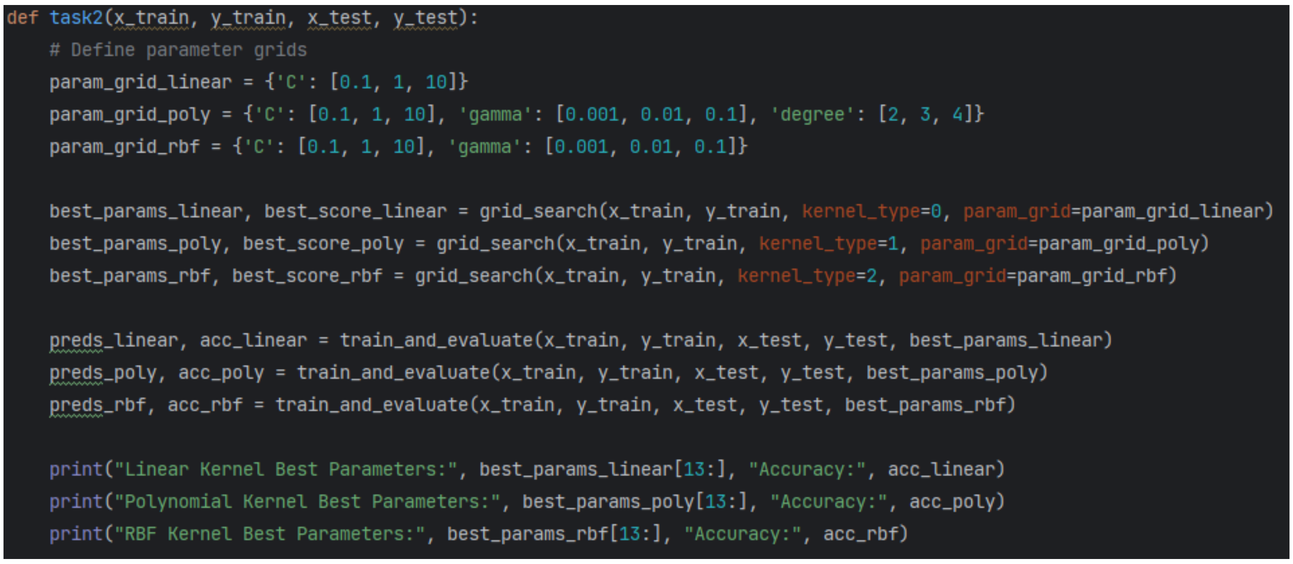
```python
def task1(x_train, y_train, x_test, y_test):
    params_linear = '-q -t 0'
    params_poly = '-q -t 1'
    params_rbf = '-q -t 2'

    preds_linear, acc_linear = train_and_evaluate(x_train, y_train, x_test, y_test, params_linear)
    preds_poly, acc_poly = train_and_evaluate(x_train, y_train, x_test, y_test, params_poly)
    preds_rbf, acc_rbf = train_and_evaluate(x_train, y_train, x_test, y_test, params_rbf)

    print("Linear Kernel Accuracy:", acc_linear)
    print("Polynomial Kernel Accuracy:", acc_poly)
    print("RBF Kernel Accuracy:", acc_rbf)
    print()
```



```python
def train_and_evaluate(x_train, y_train, x_test, y_test, params):
    model = svm_train(y_train, x_train, params)
    predictions, accuracy, _ = svm_predict(y_test, x_test, model)
    return predictions, accuracy
```

Task2:

First, I define the parameters of the corresponding kernel for the grid search. Within the grid_search() function, I retrieve the corresponding parameters from the param_grid and perform cross-validation for each combination of parameters. The cross-validation results are then compared, and the parameters yielding the highest accuracy are selected and output as the best parameters for each kernel. Finally, I use these parameters to train the model and calculate the accuracy of the predictions. The train_and_evaluate function is the same as in task1.



```python
def task2(x_train, y_train, x_test, y_test):
    # Define parameter grids
    param_grid_linear = {'C': [0.1, 1, 10]}
    param_grid_poly = {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1], 'degree': [2, 3, 4]}
    param_grid_rbf = {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1]}

    best_params_linear, best_score_linear = grid_search(x_train, y_train, kernel_type=0, param_grid=param_grid_linear)
    best_params_poly, best_score_poly = grid_search(x_train, y_train, kernel_type=1, param_grid=param_grid_poly)
    best_params_rbf, best_score_rbf = grid_search(x_train, y_train, kernel_type=2, param_grid=param_grid_rbf)

    preds_linear, acc_linear = train_and_evaluate(x_train, y_train, x_test, y_test, best_params_linear)
    preds_poly, acc_poly = train_and_evaluate(x_train, y_train, x_test, y_test, best_params_poly)
    preds_rbf, acc_rbf = train_and_evaluate(x_train, y_train, x_test, y_test, best_params_rbf)

    print("Linear Kernel Best Parameters:", best_params_linear[13:], "Accuracy:", acc_linear)
    print("Polynomial Kernel Best Parameters:", best_params_poly[13:], "Accuracy:", acc_poly)
    print("RBF Kernel Best Parameters:", best_params_rbf[13:], "Accuracy:", acc_rbf)
```

```python
def grid_search(x_train, y_train, kernel_type, param_grid):
    best_score = -np.inf
    best_params = None
    results = []
    for C in param_grid['C']:
        for gamma in param_grid.get('gamma', [None]):  # gamma only for RBF and poly
            for degree in param_grid.get('degree', [None]):  # degree only for poly
                params = f"-q -s 0 -t {kernel_type} -c {C}"
                if gamma is not None:
                    params += f" -g {gamma}"
                if degree is not None:
                    params += f" -d {degree}"
                score = svm_train(y_train, x_train, params + " -v 5")  # Cross-validation
                if score > best_score:
                    best_score = score
                    best_params = params
                if kernel_type == 0:
                    results.append([C, score])
                elif kernel_type == 1:
                    results.append([C, gamma, degree, score])
                elif kernel_type == 2:
                    results.append([C, gamma, score])
    if kernel_type == 0:...
    elif kernel_type == 1:...
    elif kernel_type == 2:...
    return best_params, best_score
```

Task3:

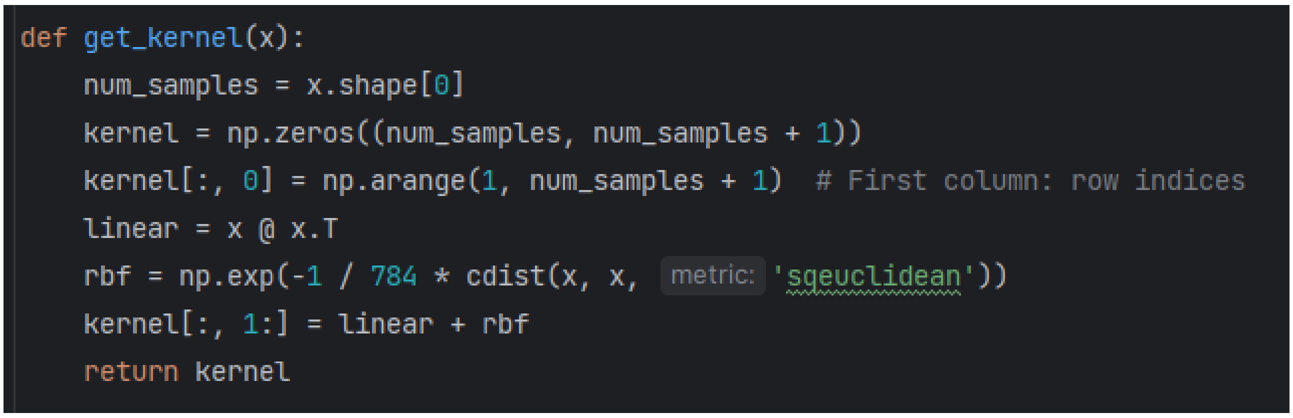
Since the kernel specified in the specification is not included in LIBSVM, I use the get_kernel() function to construct a compatible kernel for training. Within get_kernel(), I construct the linear and RBF kernels based on the formulas provided below. The parameter γ is chosen as 1/features, which equals 1/784. To calculate the squared Euclidean distance, I utilize cdist.

Once the kernel is constructed, I use the train_and_evaluate() function (as defined in Task 1) to train the model and compute the accuracy of the predictions.

- Linear: $K(x, x') = x \cdot x'$

- RBF: $K(x, x') = \exp(-\gamma \|x - x'\|^2)$

```python
def task3(x_train, y_train, x_test, y_test):
    params_linear_rbf = '-q -t 4'
    kernel_train = get_kernel(x_train)
    kernel_test = get_kernel(x_test)
    preds_linear_rbf, acc_linear_kbf = train_and_evaluate(kernel_train, y_train, kernel_test, y_test, params_linear_rbf)
    print(f'Linear Kernel + RBF Kernel Accuracy:{acc_linear_kbf}')
```



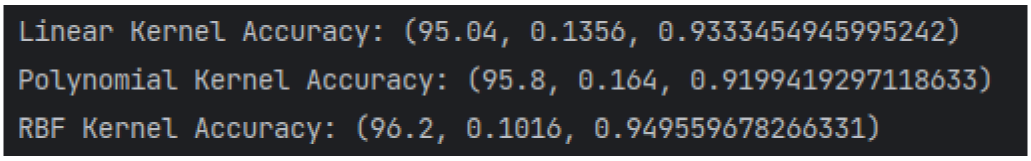
```python
def get_kernel(x):
    num_samples = x.shape[0]
    kernel = np.zeros((num_samples, num_samples + 1))
    kernel[:, 0] = np.arange(1, num_samples + 1)  # First column: row indices
    linear = x @ x.T
    rbf = np.exp(-1 / 784 * cdist(x, x, metric: 'sqeuclidean'))
    kernel[:, 1:] = linear + rbf
    return kernel
```

2. Experiments (20%)

Task1:

Parameters: C:1, Degree:3, Gamma:1/ num_features = 1/784

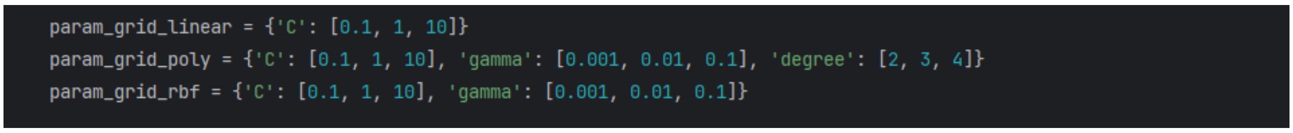
Accuracy format: (accuracy, MSE, SCC)



```
Linear Kernel Accuracy: (95.04, 0.1356, 0.9333454945995242)
Polynomial Kernel Accuracy: (95.8, 0.164, 0.9199419297118633)
RBF Kernel Accuracy: (96.2, 0.1016, 0.949559678266331)
```

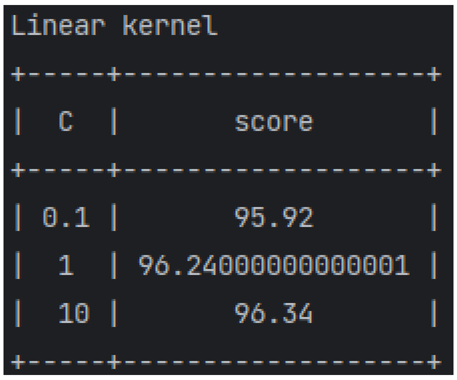
Task2:

Initial parameters: (poly = polynomial)

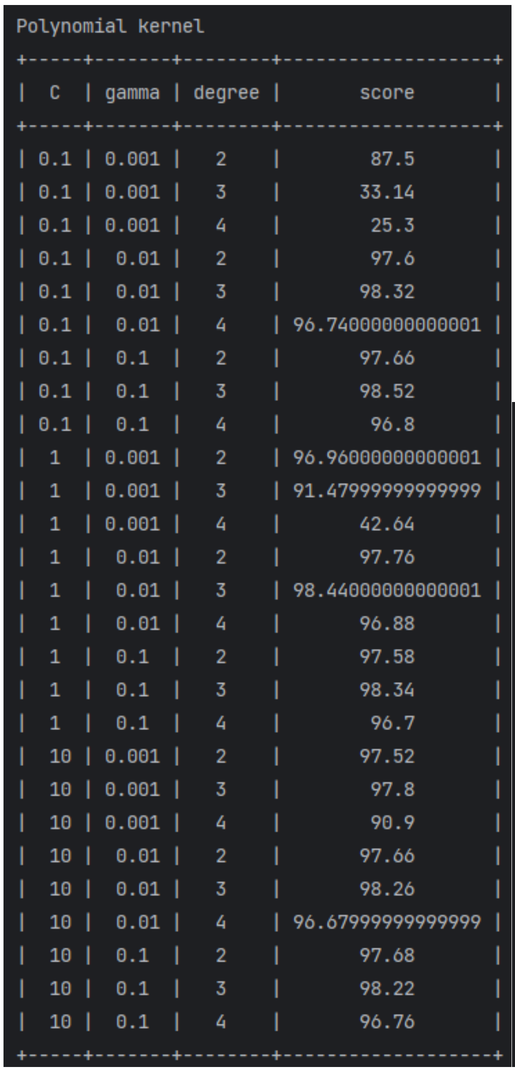


```python
    param_grid_linear = {'C': [0.1, 1, 10]}
    param_grid_poly = {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1], 'degree': [2, 3, 4]}
    param_grid_rbf = {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1]}
```

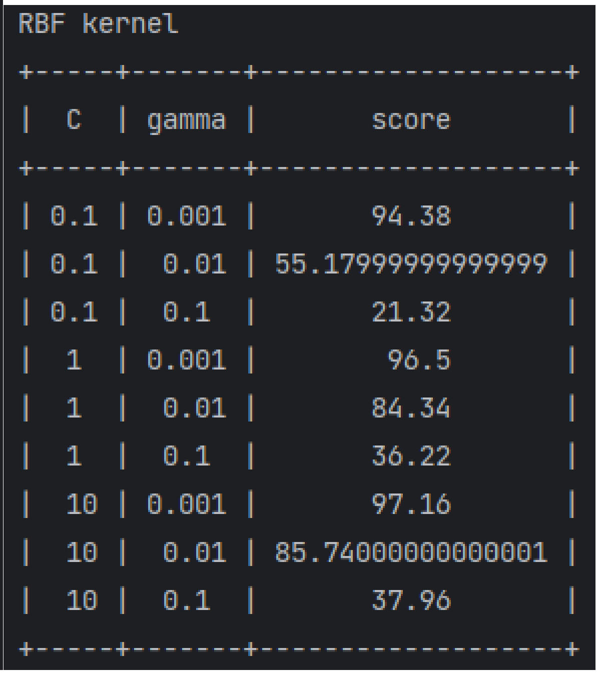
In grid search:



```
Linear kernel
+-----+-------------------+
|  C  |       score       |
+-----+-------------------+
| 0.1 |       95.92       |
|  1  | 96.24000000000001 |
|  10 |       96.34       |
+-----+-------------------+
```



```
Polynomial kernel
+-----+-------+--------+-------------------+
|  C  | gamma | degree |       score       |
+-----+-------+--------+-------------------+
| 0.1 | 0.001 |   2    |        87.5       |
| 0.1 | 0.001 |   3    |       33.14       |
| 0.1 | 0.001 |   4    |        25.3       |
| 0.1 |  0.01 |   2    |        97.6       |
| 0.1 |  0.01 |   3    |       98.32       |
| 0.1 |  0.01 |   4    | 96.74000000000001 |
| 0.1 |  0.1  |   2    |       97.66       |
| 0.1 |  0.1  |   3    |       98.52       |
| 0.1 |  0.1  |   4    |        96.8       |
|  1  | 0.001 |   2    | 96.96000000000001 |
|  1  | 0.001 |   3    | 91.47999999999999 |
|  1  | 0.001 |   4    |       42.64       |
|  1  |  0.01 |   2    |       97.76       |
|  1  |  0.01 |   3    | 98.44000000000001 |
|  1  |  0.01 |   4    |       96.88       |
|  1  |  0.1  |   2    |       97.58       |
|  1  |  0.1  |   3    |       98.34       |
|  1  |  0.1  |   4    |        96.7       |
|  10 | 0.001 |   2    |       97.52       |
|  10 | 0.001 |   3    |        97.8       |
|  10 | 0.001 |   4    |        90.9       |
|  10 |  0.01 |   2    |       97.66       |
|  10 |  0.01 |   3    |       98.26       |
|  10 |  0.01 |   4    | 96.67999999999999 |
|  10 |  0.1  |   2    |       97.68       |
|  10 |  0.1  |   3    |       98.22       |
|  10 |  0.1  |   4    |       96.76       |
+-----+-------+--------+-------------------+
```

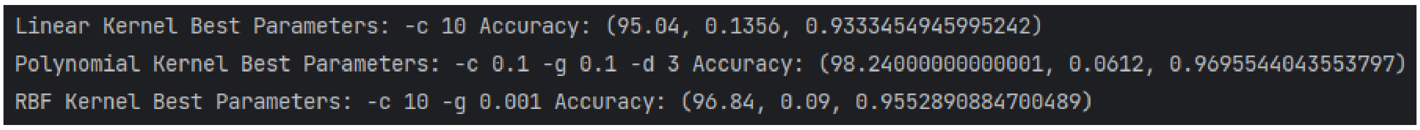


```
RBF kernel
+-----+-------+-------------------+
|  C  | gamma |       score       |
+-----+-------+-------------------+
| 0.1 | 0.001 |       94.38       |
| 0.1 |  0.01 | 55.17999999999999 |
| 0.1 |  0.1  |       21.32       |
|  1  | 0.001 |        96.5       |
|  1  |  0.01 |       84.34       |
|  1  |  0.1  |       36.22       |
|  10 | 0.001 |       97.16       |
|  10 |  0.01 | 85.74000000000001 |
|  10 |  0.1  |       37.96       |
+-----+-------+-------------------+
```

After grid search, the result and its corresponding accuracy:

-c = C, -g = gamma, -d = degree

Accuracy format: (accuracy, MSE, SCC)

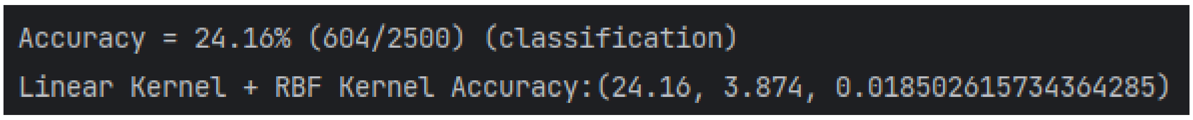


```
Linear Kernel Best Parameters: -c 10 Accuracy: (95.04, 0.1356, 0.9333454945995242)
Polynomial Kernel Best Parameters: -c 0.1 -g 0.1 -d 3 Accuracy: (98.24000000000001, 0.0612, 0.9695544043553797)
RBF Kernel Best Parameters: -c 10 -g 0.001 Accuracy: (96.84, 0.09, 0.9552890884700489)
```

Task3:

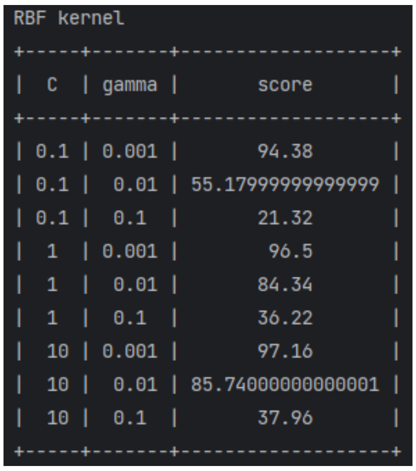
Parameters: C=1, gamma = 1/784

Accuracy format: (accuracy, MSE, SCC)



```
Accuracy = 24.16% (604/2500) (classification)
Linear Kernel + RBF Kernel Accuracy:(24.16, 3.874, 0.018502615734364285)
```

3. Observations and Discussion (10%)



```
RBF kernel
+-----+-------+-------------------+
|  C  | gamma |       score       |
+-----+-------+-------------------+
| 0.1 | 0.001 |       94.38       |
| 0.1 |  0.01 | 55.17999999999999 |
| 0.1 |  0.1  |       21.32       |
|  1  | 0.001 |        96.5       |
|  1  |  0.01 |       84.34       |
|  1  |  0.1  |       36.22       |
|  10 | 0.001 |       97.16       |
|  10 |  0.01 | 85.74000000000001 |
|  10 |  0.1  |       37.96       |
+-----+-------+-------------------+
```

RBF kernel can have better performance when the gamma is close to 1/num_feature, in this homework the number of features is equal to 784 which is close to 0.001, so the result that use gamma =0.001 will significant outperform than other conditions. I think the reason is that When γ is set close to 1/num_features, The kernel scales its influence according to the dimensionality of the feature space. It balances the trade-off between capturing the overall structure of the data and being responsive to local variations, preventing them from overfitting and underfitting.

In task1, linear and RBF kernel have the accuracy of testing data for 95.04% and 96.2% respectively. However, when I sum up these two kernels to be the new kernel, the performance significantly decreases. I think the combination of non-linear and linear components may not align with the data's structure in this homework. Using grid search to tune the hyperparameter may be the solution.