Machine Learning HW5

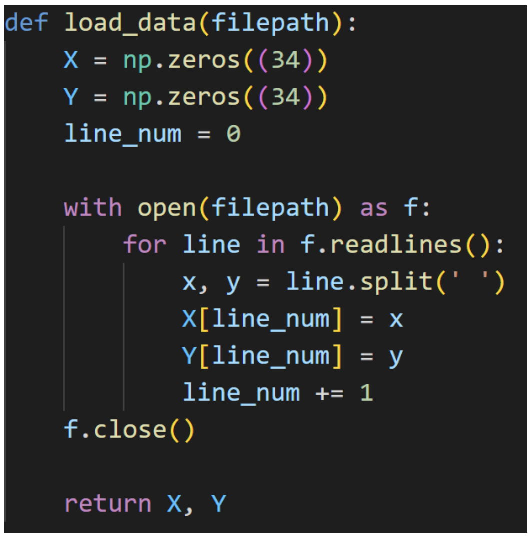
311551094 資科工碩一 廖昱瑋

I. Gaussian Process

a. code with detailed explanations

i. Part 1

1. Load data to array X and Y.

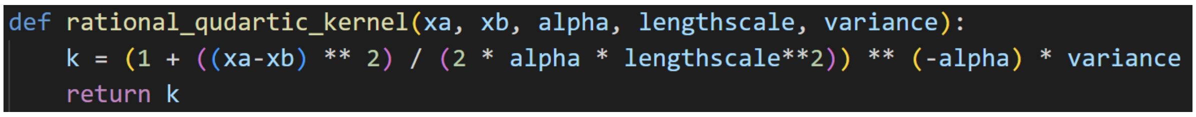


```python
def load_data(filepath):
    X = np.zeros((34))
    Y = np.zeros((34))
    line_num = 0

    with open(filepath) as f:
        for line in f.readlines():
            x, y = line.split(' ')
            X[line_num] = x
            Y[line_num] = y
            line_num += 1
    f.close()

    return X, Y
```

2. Kernel: The rational quadratic kernel is defined as bellow. I set the initial value $\alpha = 1,\ \ell = 1,\ \sigma^2 = 1$ in part 1.

$$k(x_a, x_b) = \sigma^2 \left(1 + \frac{||x_a - x_b||^2}{2\alpha\ell^2}\right)^{-\alpha}$$

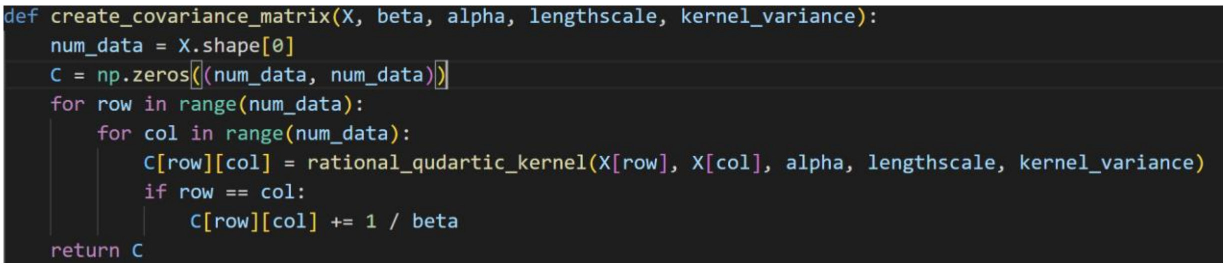
where σ: overall standard deviation(amplitude), ℓ: lengthscale, α: scale-mixture



```python
def rational_qudartic_kernel(xa, xb, alpha, lengthscale, variance):
    k = (1 + ((xa-xb) ** 2) / (2 * alpha * lengthscale**2)) ** (-alpha) * variance
    return k
```

3. Create covariance matrix for data x using below formula. I set the initial value $\beta = 5$.

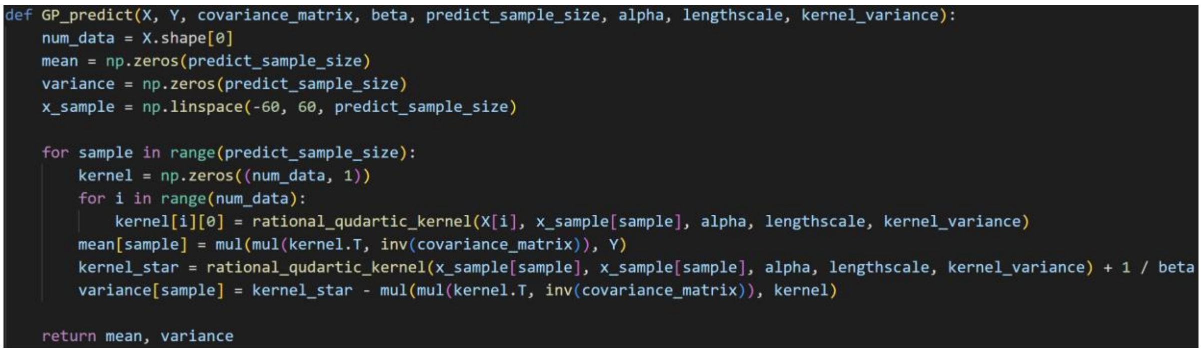
$$\mathbf{C}(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm}$$



```python
def create_covariance_matrix(X, beta, alpha, lengthscale, kernel_variance):
    num_data = X.shape[0]
    C = np.zeros((num_data, num_data))
    for row in range(num_data):
        for col in range(num_data):
            C[row][col] = rational_qudartic_kernel(X[row], X[col], alpha, lengthscale, kernel_variance)
            if row == col:
                C[row][col] += 1 / beta
    return C
```

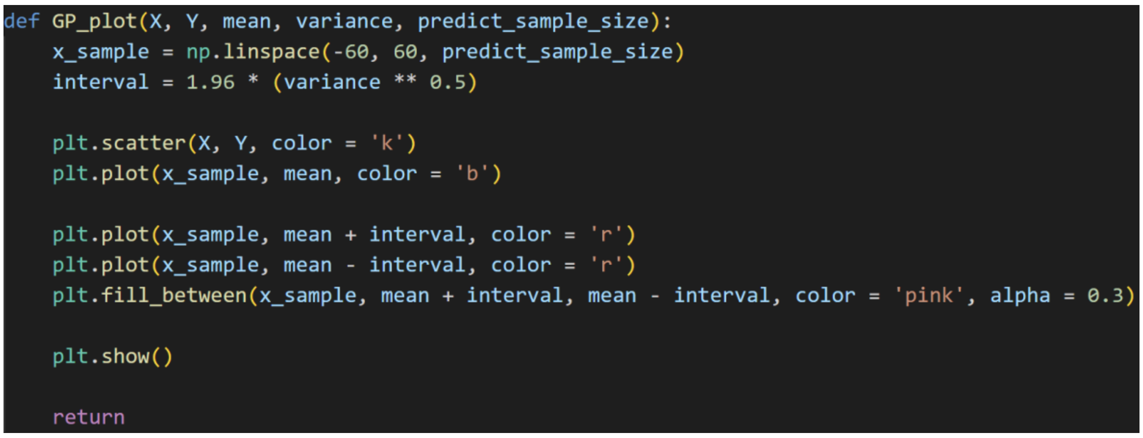
4. The prediction in Gaussian process can be calculated by the below formula. Given a new data point x^*, we can calculate mean and variance of its y distribution. C is the covariance matrix created in step 2. It means relationship between training data. $k(x, x^*)$ means the relationship between test and training data. k^* is relationship between test data and plus a random noise.
   Then, do the prediction for new data point x range from -60 to 60 with 1000 sampling points.

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



```
def GP_predict(X, Y, covariance_matrix, beta, predict_sample_size, alpha, lengthscale, kernel_variance):
    num_data = X.shape[0]
    mean = np.zeros(predict_sample_size)
    variance = np.zeros(predict_sample_size)
    x_sample = np.linspace(-60, 60, predict_sample_size)

    for sample in range(predict_sample_size):
        kernel = np.zeros((num_data, 1))
        for i in range(num_data):
            kernel[i][0] = rational_qudartic_kernel(X[i], x_sample[sample], alpha, lengthscale, kernel_variance)
        mean[sample] = mul(mul(kernel.T, inv(covariance_matrix)), Y)
        kernel_star = rational_qudartic_kernel(x_sample[sample], x_sample[sample], alpha, lengthscale, kernel_variance) + 1 / beta
        variance[sample] = kernel_star - mul(mul(kernel.T, inv(covariance_matrix)), kernel)

    return mean, variance
```

5. This function plots all training data points (black points) and the predictions (red line). 95% confidence interval (pink area) means ±1.96 standard deviation from the mean.



```
def GP_plot(X, Y, mean, variance, predict_sample_size):
    x_sample = np.linspace(-60, 60, predict_sample_size)
    interval = 1.96 * (variance ** 0.5)

    plt.scatter(X, Y, color = 'k')
    plt.plot(x_sample, mean, color = 'b')

    plt.plot(x_sample, mean + interval, color = 'r')
    plt.plot(x_sample, mean - interval, color = 'r')
    plt.fill_between(x_sample, mean + interval, mean - interval, color = 'pink', alpha = 0.3)

    plt.show()

    return
```

ii. Part 2

1. The marginal log likelihood is defined as below formula. In my program, I minimize negative marginal log likelihood to get optimized kernel parameters.

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

```
opt = minimize(negative_marginal_log_likelihood, [ALPHA, LENGTHSCALE, KERNEL_VARIANCE], args = (X, Y, BETA))
alpha_opt, lengthscale_opt, kernel_variance_opt = opt.x
```

```
def negative_marginal_log_likelihood(theta, X, Y, beta):
    alpha = theta[0]
    lengthscale = theta[1]
    kernel_variance = theta[2]
    n = X.shape[0]

    covariance_matrix_theta = create_covariance_matrix(X, beta, alpha, lengthscale, kernel_variance)
    likelihood = np.log(np.linalg.det(covariance_matrix_theta)) / 2
    likelihood += mul(mul(Y.T, inv(covariance_matrix_theta)), Y) / 2
    likelihood += n / 2 * np.log(2 * np.pi)

    return likelihood
```

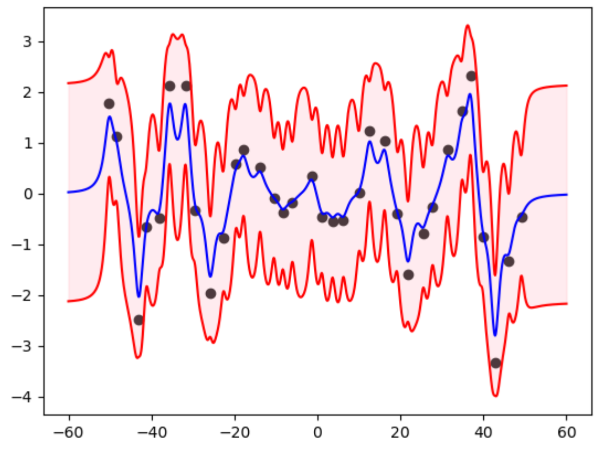
2. After getting the optimized kernel parameters, apply Gaussian process to predict the distribution as part 1.

```
covariance_matrix = create_covariance_matrix(X, BETA, alpha_opt, lengthscale_opt, kernel_variance_opt)
mean , variance = GP_predict(X, Y, covariance_matrix, BETA, PREDICT_SAMPLE_SIZE, alpha_opt, lengthscale_opt, kernel_variance_opt)
GP_plot(X, Y, mean, variance, PREDICT_SAMPLE_SIZE)
```

b. experiments settings and results

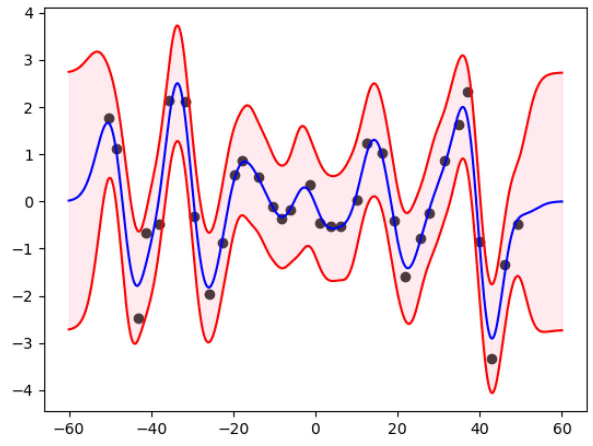
i. Part 1

$$\alpha = 1,\ \ell = 1,\ \sigma^2 = 1,\ \beta = 5$$



ii. Part 2

$$\alpha = 517.95,\ \ell = 3.33,\ \sigma^2 = 1.74,\ \beta = 5$$

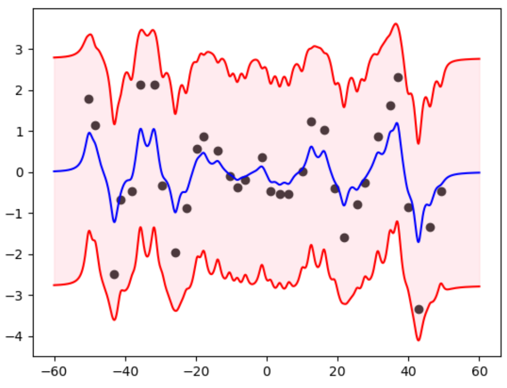
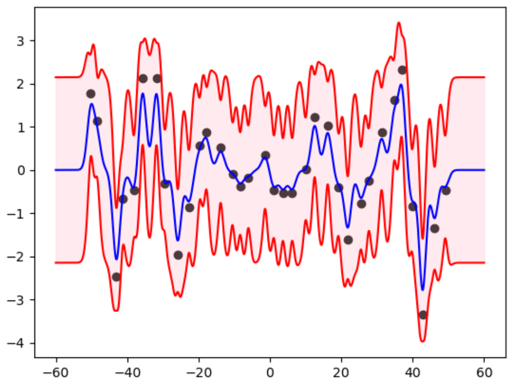


c. observations and discussion

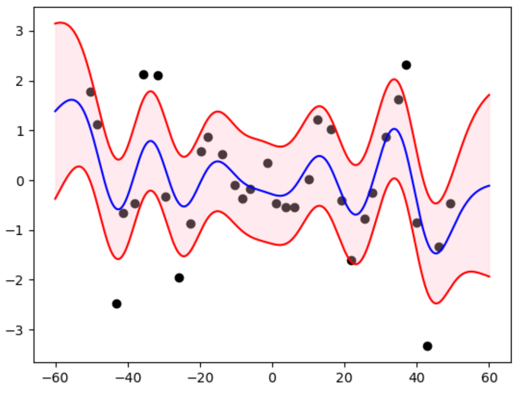
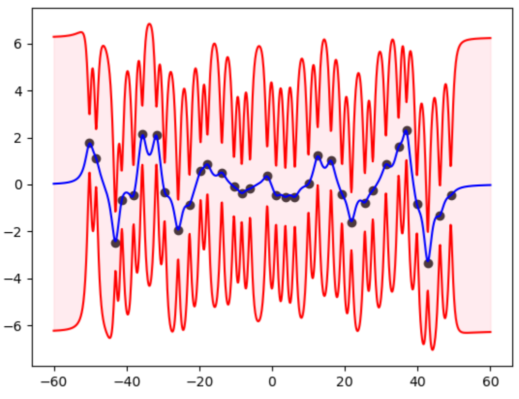
   i. After the parameters of rational quadratic kernel function is optimized, the 95% confidence interval becomes much smoother which leads to better result. However, the mean of prediction (blue line) doesn't change that much.

   ii. In the range of x that is smaller than minimum of training data or larger than maximum of training data, there is no training data points. Therefore, the 95% confidence interval in that region is much larger. The model has less confidence about the prediction in that region.

   iii. $\alpha = 1,\ \ell = 1,\ \sigma^2 = 1,\ \beta = 1$ $\alpha = 100,\ \ell = 1,\ \sigma^2 = 1,\ \beta = 5$

$\alpha = 1,\ \ell = 10,\ \sigma^2 = 1,\ \beta = 5$ $\alpha = 1,\ \ell = 1,\ \sigma^2 = 10,\ \beta = 5$

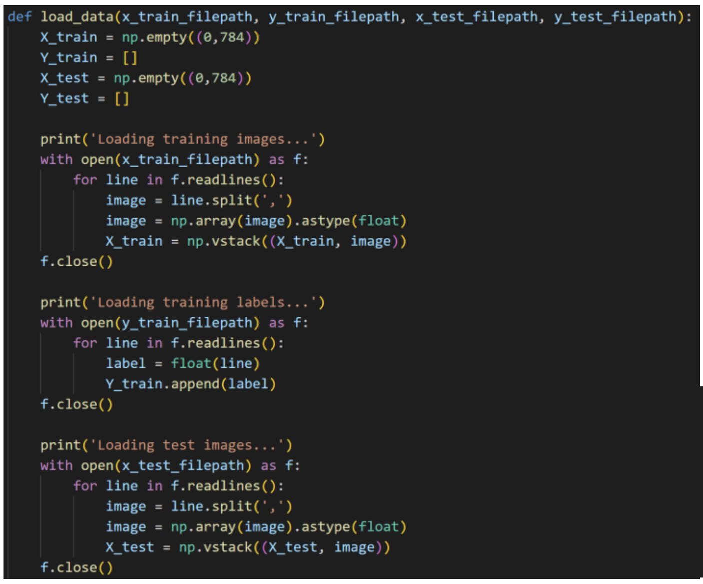
In first figure, we can see when β is small, we get more random noise in data. Therefore, the confidence interval is larger.
In second figure, we can see no matter α is large or small, it hardly affects model. It's because α is only a scale factor. When $\alpha \to \infty$ the rational quadratic kernel converges into the exponentiated quadratic kernel.
In third figure, we can see when ℓ is too large, model tends to be under-fitting.
In forth figure, we can see when σ^2 is too large, the model will be more confidence on the x that has training point but less confidence on the x that doesn't have training data.

II. SVM

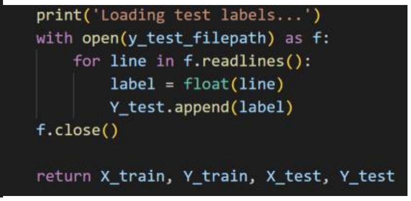
a. code with detailed explanations

i. Part 1

1. Load data to X_train, Y_train, X_test, Y_test

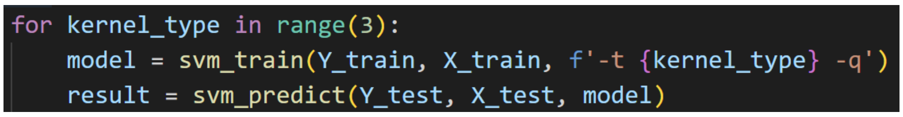


```python
def load_data(x_train_filepath, y_train_filepath, x_test_filepath, y_test_filepath):
    X_train = np.empty((0,784))
    Y_train = []
    X_test = np.empty((0,784))
    Y_test = []

    print('Loading training images...')
    with open(x_train_filepath) as f:
        for line in f.readlines():
            image = line.split(',')
            image = np.array(image).astype(float)
            X_train = np.vstack((X_train, image))
    f.close()

    print('Loading training labels...')
    with open(y_train_filepath) as f:
        for line in f.readlines():
            label = float(line)
            Y_train.append(label)
    f.close()

    print('Loading test images...')
    with open(x_test_filepath) as f:
        for line in f.readlines():
            image = line.split(',')
            image = np.array(image).astype(float)
            X_test = np.vstack((X_test, image))
    f.close()
```



```python
    print('Loading test labels...')
    with open(y_test_filepath) as f:
        for line in f.readlines():
            label = float(line)
            Y_test.append(label)
    f.close()

    return X_train, Y_train, X_test, Y_test
```

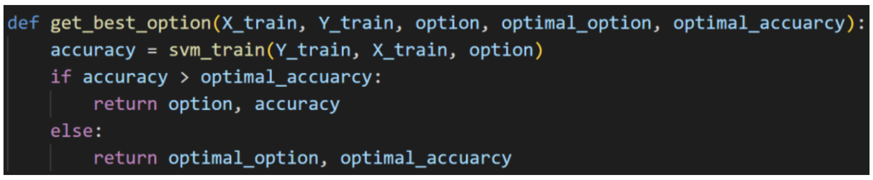
2. Call LIBSVM built-in function *svm_train*, *svm_predict*.
   -t: type of kernel, 0(linear), 1(polynomial), 2(RBF)
   -q: quiet mode(no outputs)



```python
for kernel_type in range(3):
    model = svm_train(Y_train, X_train, f'-t {kernel_type} -q')
    result = svm_predict(Y_test, X_test, model)
```

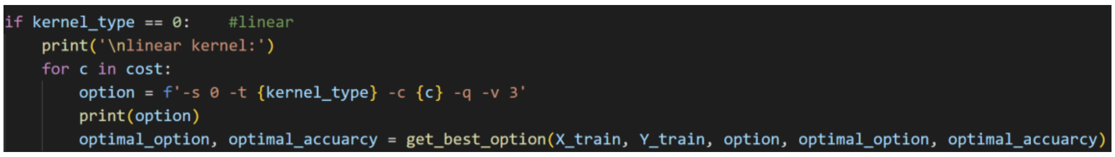
ii. Part 2

1. Use self-defined function *grid_search* to do grid search for each of linear, polynomial, RBF, sigmoid kernel. In every type of kernel, the cost for C-SVC search in [0.001, 0.01, 0.1, 1, 10, 100] and do 3-fold cross validation.
   The parameters feed in *svm_train*:
   -s: type of SVM, 0(C-SVC)
   -c: the parameter C of C-SVC
   -v: n-fold cross validation
2. Define *get_best_option* function to compare current option with previous optimal option. It returns the optimal option and accuracy so far. This function will be used in step 3~6 to find best option while doing grid search.



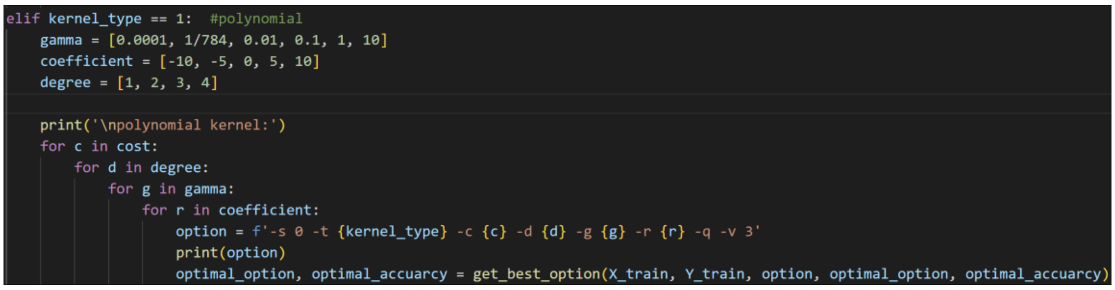
```python
def get_best_option(X_train, Y_train, option, optimal_option, optimal_accuracy):
    accuracy = svm_train(Y_train, X_train, option)
    if accuracy > optimal_accuracy:
        return option, accuracy
    else:
        return optimal_option, optimal_accuarcy
```

3. For linear kernel, search C of C-SVC in [0.001, 0.01, 0.1, 1, 10, 100].



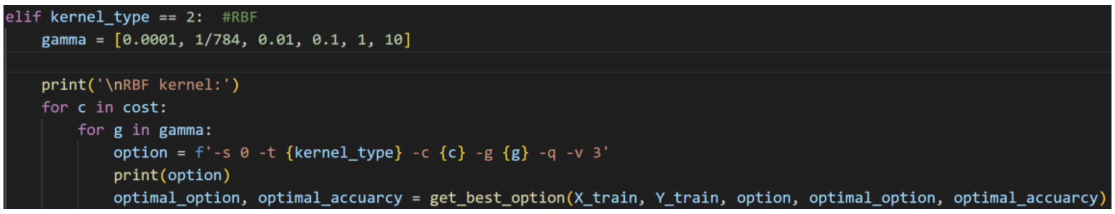
```python
if kernel_type == 0:    #linear
    print('\nlinear kernel:')
    for c in cost:
        option = f'-s 0 -t {kernel_type} -c {c} -q -v 3'
        print(option)
        optimal_option, optimal_accuarcy = get_best_option(X_train, Y_train, option, optimal_option, optimal_accuarcy)
```

4. For polynomial kernel, The parameters feed in *svm_train*:
   -d: degree in kernel function
   -g: gamma in kernel function
   -r: coefficient0 in kernel function
   Search C in [0.001, 0.01, 0.1, 1, 10, 100], gamma in [0.0001, 1/784, 0.01, 0.1, 1, 10], coefficient in [-10, -5, 0, 5, 10], and degree in [1, 2, 3, 4].



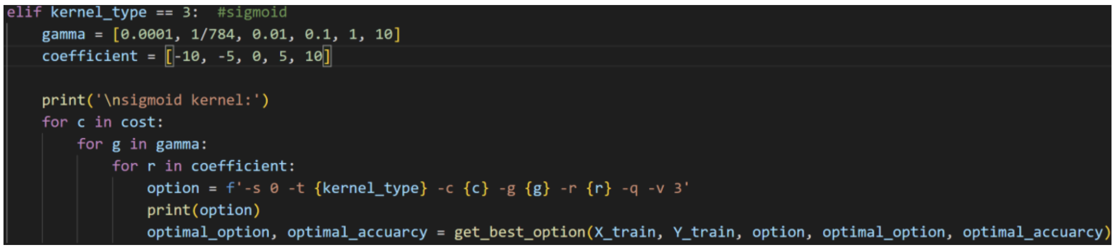
```python
elif kernel_type == 1:  #polynomial
    gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10]
    coefficient = [-10, -5, 0, 5, 10]
    degree = [1, 2, 3, 4]

    print('\npolynomial kernel:')
    for c in cost:
        for d in degree:
            for g in gamma:
                for r in coefficient:
                    option = f'-s 0 -t {kernel_type} -c {c} -d {d} -g {g} -r {r} -q -v 3'
                    print(option)
                    optimal_option, optimal_accuarcy = get_best_option(X_train, Y_train, option, optimal_option, optimal_accuarcy)
```

5. For RBF kernel, search C in [0.001, 0.01, 0.1, 1, 10, 100], gamma in [0.0001, 1/784, 0.01, 0.1, 1, 10].



```python
elif kernel_type == 2:  #RBF
    gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10]

    print('\nRBF kernel:')
    for c in cost:
        for g in gamma:
            option = f'-s 0 -t {kernel_type} -c {c} -g {g} -q -v 3'
            print(option)
            optimal_option, optimal_accuarcy = get_best_option(X_train, Y_train, option, optimal_option, optimal_accuarcy)
```

6. For sigmoid kernel, search C in [0.001, 0.01, 0.1, 1, 10, 100], gamma in [0.0001, 1/784, 0.01, 0.1, 1, 10], coefficient in [-10, -5, 0, 5, 10].



```python
elif kernel_type == 3:  #sigmoid
    gamma = [0.0001, 1/784, 0.01, 0.1, 1, 10]
    coefficient = [-10, -5, 0, 5, 10]

    print('\nsigmoid kernel:')
    for c in cost:
        for g in gamma:
            for r in coefficient:
                option = f'-s 0 -t {kernel_type} -c {c} -g {g} -r {r} -q -v 3'
                print(option)
                optimal_option, optimal_accuarcy = get_best_option(X_train, Y_train, option, optimal_option, optimal_accuarcy)
```

iii. Part 3

1. Self-defined linear kernel and RBF kernel.
   Linear kernel: $u^T * v$
   RBF kernel: $e^{-\gamma||u-v||^2}$

```
def linear_kernel(u, v):
    return u @ v.T

def RBFkernel(u, v):
    gamma = 1 / 784
    dist = np.sum(u ** 2, axis=1).reshape(-1, 1) + np.sum(v ** 2, axis=1) - 2 * u @ v.T

    return np.exp(-gamma * dist)
```

2. Calculate train_kernel and test_kernel separately. They are both combined with linear kernel and RBF kernel. train_kernel is the relationship between training data, and test_kernel is the relationship between training and test data.

```
train_kernel = linear_kernel(X_train, X_train) + RBFkernel(X_train, X_train)
test_kernel = linear_kernel(X_test, X_train) + RBFkernel(X_test, X_train)

# Add index in front of kernel
train_kernel = np.hstack((np.arange(1,train_num + 1).reshape(-1, 1), train_kernel))
test_kernel = np.hstack((np.arange(1,test_num + 1).reshape(-1, 1), test_kernel))
```

3. Call LIBSVM built-in function *svm_train*, *svm_predict*.
   -t: 4 (precomputed kernel)

```
model = svm_train(Y_train, train_kernel, '-t 4 -q')
result = svm_predict(Y_test, test_kernel, model)
```

b. experiments settings and results

i. Part 1

Kernel type	Default Hyperparameters	Test Accuracy
Linear	N.A.	95.08%
Polynomial	-d 3 -g 1/784 -r 0	34.68%
RBF	-g 1/784	95.32%

ii. Part 2

Kernel type	Optimal Hyperparameters	Cross-validation Accuracy	Test Accuracy
Linear	-c 0.01	96.84%	95.96%
Polynomial	-c 0.1 -d 3 -g 1 -r 10	98.18%	97.92%
RBF	-c 10 -g 0.01	98.18%	98.2%
Sigmoid	-c 10 -g 1/784 -r 0	97.04%	95.92%

iii. Part 3

Kernel type	Default Hyperparameters	Test Accuracy
Linear + RBF	-g 1/784	95.96%

c. observations and discussion

   i. The model may have poor accuracy if we have bad hyperparameters. For example, in part 1, I use default parameters for polynomial kernel. The test accuracy is only 34.68%. However, after doing grid search for optimal parameters, the test accuracy raises to 97.92%.
   ii. The cross-validation and test accuracy does not vary a lot for different kernel type in part 2. The accuracies are quite high. It means we truly find a good parameter for the model by doing grid search.
   iii. In part 2, we can observe that test accuracy tends to be lower than cross-validation accuracy. It's because test data definitely not has same distribution as training data. Our optimal hyperparameters are obtained from training data. The optimal hyperparameters in training set may not be optimal parameters in test set. However, the slightly drop of accuracy is normal and acceptable.
   iv. Linear kernel is much faster than other kernels because it is the simplest one.
   v. Polynomial kernel has 4 hyperparameters to tune (if including C in C-SVC) which makes grid search be time-consuming.
   vi. Part 3 shows that we can define our own kernel based on the combination of well-known kernels.