Machine Learning Homework 5

Gaussian Process & SVM

I. Gaussian Process

- a. code with detailed explanations (20%)
  - Part1 (10%)

(1) The formula of Rational Quadratic Kernel is

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

Put it into Python code:

```
def RationalQuadraticKernel(x, x_prime, sigma, alpha, lengthscale):
    dist = np.sum(x ** 2, axis=1).reshape(-1, 1) + np.sum(x_prime ** 2, axis=1) - 2 * x.dot(x_prime.transpose())
    kernel = (sigma ** 2) * ((1 + dist / (2 * alpha * (lengthscale ** 2))) ** (-1 * alpha))
    return kernel
```

Fig (1).

(2) Implementation of Gaussian Process

```
47      x, y = getData('./data/input.data')
48      kernel = RationalQuadraticKernel(x, x, sigma, alpha, lengthscale)
49      C = kernel + np.identity(len(x)) * (1 / beta)
50      C_inv = inv(C)
51      x_t = np.linspace(-60, 60, num = 100).reshape(-1, 1) # 100 * 1
52
53      kernel1 = RationalQuadraticKernel(x, x_t, sigma, alpha, lengthscale)# 34*100
54      kernel2 = RationalQuadraticKernel(x_t, x_t, sigma, alpha, lengthscale) # 100* 100
55      mu = kernel1.transpose().dot(C_inv).dot(y)  # mean function
56      var = kernel2 + np.identity(len(x_t), dtype=np.float64) * (1 / beta)
57      var -= kernel1.transpose().dot(C_inv).dot(kernel1)
```

Fig (2).

Using formula in handout p.48:

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

$$\mathbf{C}_{N+1} = \begin{bmatrix} \mathbf{C} & k(\mathbf{x}, \mathbf{x}^*) \\ k(\mathbf{x}, \mathbf{x}^*)^\top & k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1} \end{bmatrix}$$

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

Line 48 is calculating rational quadratic kernel using (training samples, training samples), then calculating C matrix in line 49 by
kernel (training samples, training samples) + I * (1/beta).

Line 53 is calculating rational quadratic kernel (training samples, testing samples), and line 54 is calculating rational quadratic kernel (testing samples, testing samples)

Line 55 is calculating mean function of Gaussian using
rational quadratic kernel (training samples, testing samples)* C(inverse) * y
Line 56, 57 calculates variance function of Gaussian by
rational quadratic kernel (testing samples, testing samples) + I * (1/beta) – rational quadratic kernel (training samples, testing samples) (transpose) * C(inverse) * rational quadratic kernel (training samples, testing samples)

○ Part2 (10%)

Do optimization on parameters (sigma, alpha, lengthscale):

Objective function

```
def ObjectFunction(theta, x, y, beta):
    kernel = RationalQuadraticKernel(x, x, theta[0], theta[1], theta[2])
    C = kernel + np.identity(len(x)) * (1 / beta)
    loglikelihood = 0.5 * np.sum(np.log(det(C))) +0.5 * y.transpose().dot(inv(C)).dot(y)+0.5 * len(x) * np.log(2 * math.pi)

    return loglikelihood
```

Fig (3).

Use marginal likelihood of function of parameters to optimize parameters. The formula is referred to handout p.52

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

```
theta = [sigma, alpha, lengthscale]
param = minimize(ObjectFunction, theta, args=(x, y, beta),
                 bounds=((1e-6, 1e6), (1e-6, 1e6), (1e-6, 1e6)))
sigma_opt = param.x[0]
alpha_opt = param.x[1]
lengthscale_opt = param.x[2]
GaussianProcess(x, x_t, y, sigma_opt, alpha_opt, lengthscale_opt)
```

Fig (4).

To do optimization of parameters (sigma, alpha, lengthscale), I used scipy.optimize.minimize library to achieve that. By passing object function (loglikelihood function), parameters that are hoped to do optimization and additional parameters (x, y, 5), and bounds (which I set (1e-6, 1e6) for all one) to the minimize function, I can get the best ones in return. Then use these returned ones as input to do Gaussian process again.

■ b. experiments settings and results (20%)

○ Part1 (10%)

Initial parameters with sigma = 1, alpha = 1, lengthscale = 1

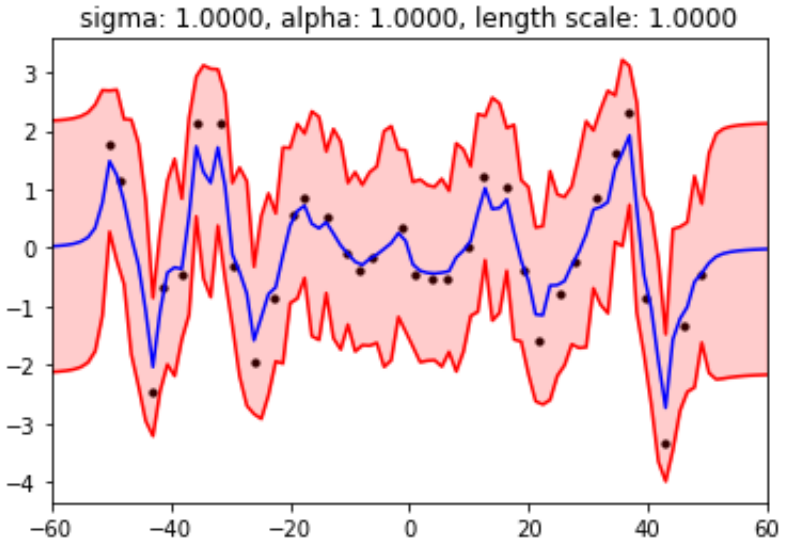

Fig (5).

Using initial parameters setting (sigma, alpha, lengthscale) = (1, 1, 1) to get the figure using 95% confident interval in [-60, 60].

○ Part2 (10%)

Updated parameters after minimizing loglikelihood: sigma = 1.3136, alpha = 379.6947, lengthscale = 3.3176

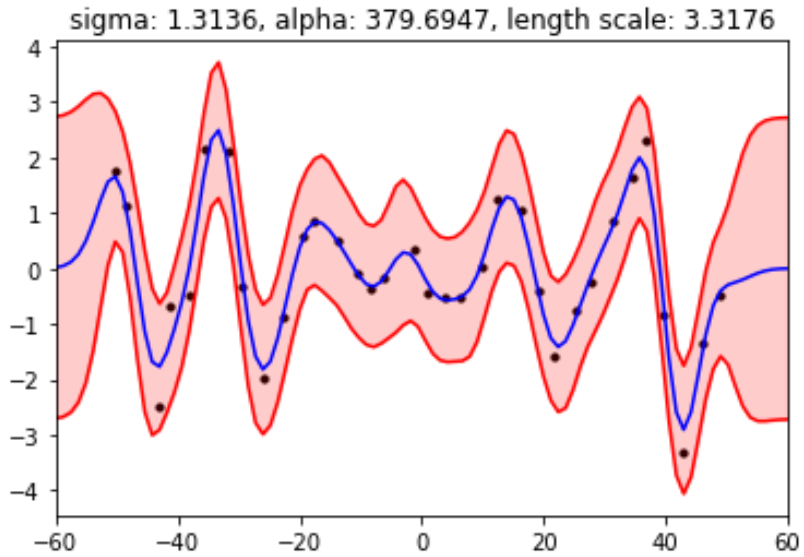

Fig (6).

From the figure can see that after optimization, the predicted range is much

smaller and the margin become more smooth.

■ c. observations and discussion (10%)

Original settings, parameters = (1,1,1)

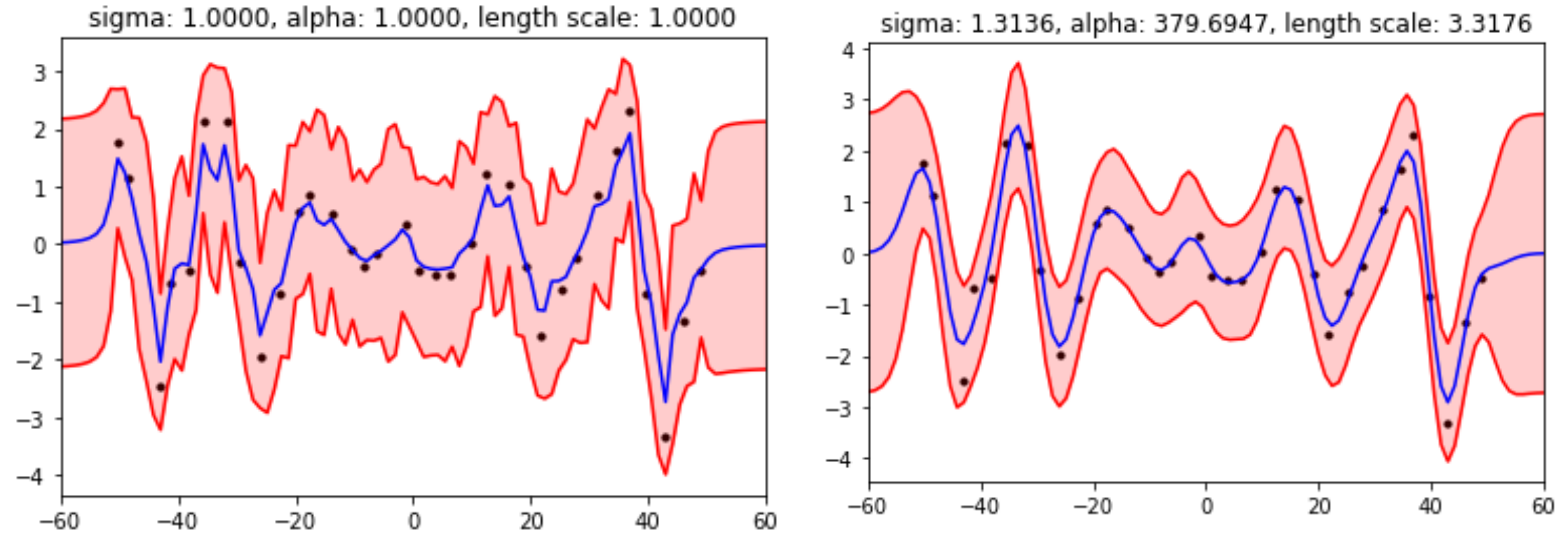

Fig (7).

Set initial parameters = (1.5, 1.5, 1.5)

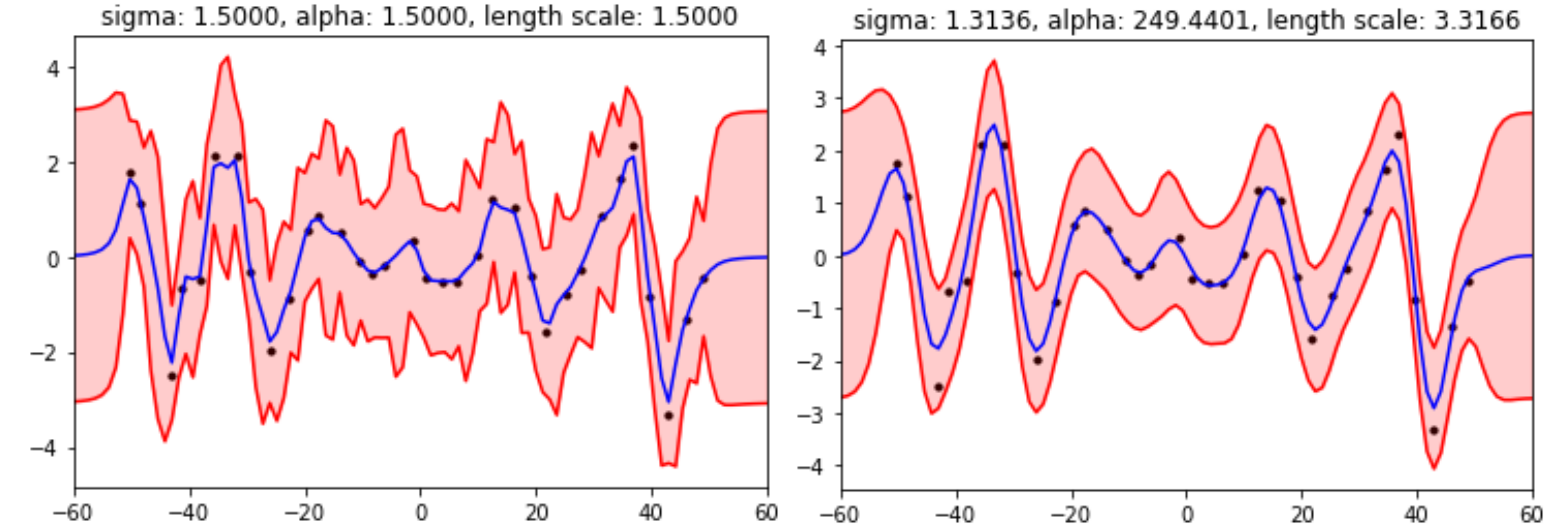

Fig (8).

Set initial parameters = (2, 2, 2)

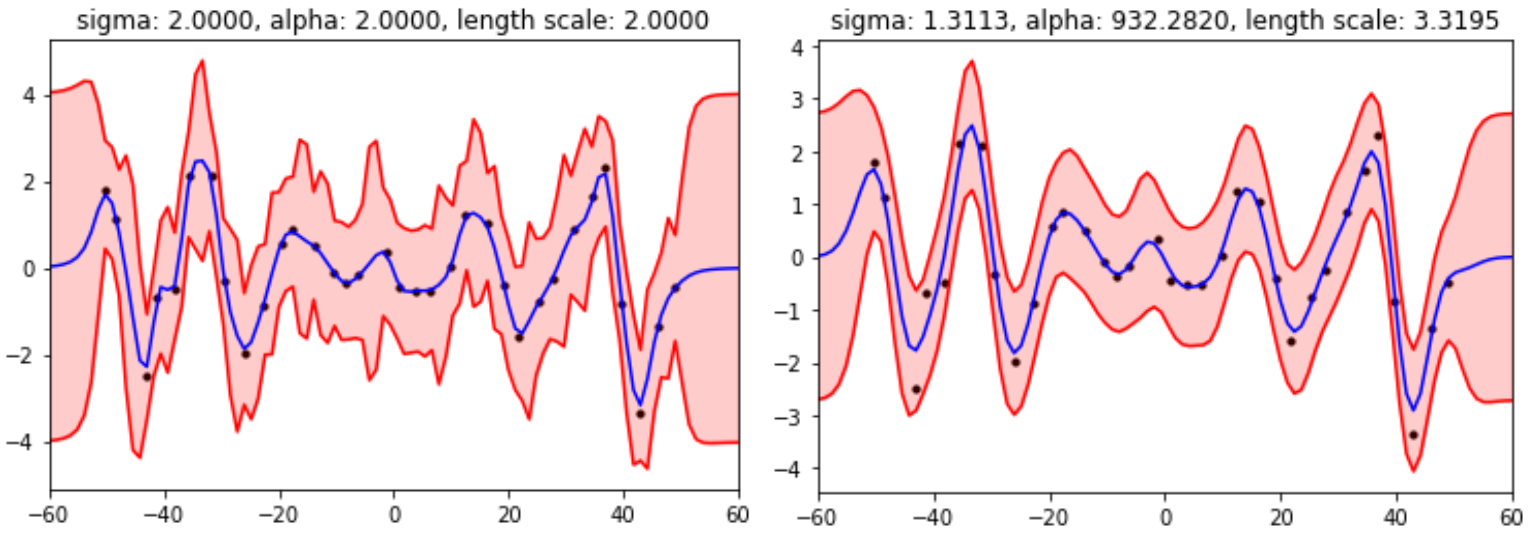

Fig (9).

One can easily tell that when the initial parameters are larger, the original predicted range becomes larger. After the optimization step, we can see that whether the initial settings = (1, 1, 1) or (1.5, 1.5, 1.5) or (2, 2, 2), the optimized sigma is nearly 1.31, same truth in the optimized lengthscale parameters that three settings got almost same value (3.31).

When initial settings = (1.5, 1.5, 1.5), the optimized alpha is the smallest (= 249.44) out of three experiments, and initial settings = (2, 2, 2) get the biggest alpha = (932.28).

II. SVM

■ a. code with detailed explanations (20%)

○ Part1 (5%)

```
def SVM(part, x_train, y_train, x_test, y_test, kernel_type, best_param):
    if part == 1:
        print(kernel_type)
        p  = svm_problem(y_train, x_train)
        param = svm_parameter('-t {} -q'.format(kernel[kernel_type]))
    elif part == 2:
        p  = svm_problem(y_train, x_train)
        param = svm_parameter('-s 0 -t {} -c {} -g {} -q'.format(kernel[kernel_type], best_para
    else:
        p  = svm_problem(y_train, x_train, isKernel=True)
        param = svm_parameter('-s 0 -t {} -c {} -g {} -q'.format(kernel[kernel_type], best_para

    model = svm_train(p, param)
    prediction = svm_predict(y_test, x_test, model)
```

Fig (10).

```
# Part.1
x_train, y_train, x_test, y_test = getData()
SVM(1, x_train, y_train, x_test, y_test, 'linear', None)
SVM(1, x_train, y_train, x_test, y_test, 'polynomial', None)
SVM(1, x_train, y_train, x_test, y_test, 'RBF', None)
```

Fig (11).

0 is for linear kernel, 1 is for polynomial kernel , 2 is for RBF kernel by usage definition in https://www.csie.ntu.edu.tw/~cjlin/libsvm/

```
-t kernel_type : set type of kernel function (default 2)
        0 -- linear: u'*v
        1 -- polynomial: (gamma*u'*v + coef0)^degree
        2 -- radial basis function: exp(-gamma*|u-v|^2)
```

The formula is listed as bellow:

$$\text{Linear kernel: } k(x,y) = < x, y >$$

$$\text{Polynomial kernel: } k(x,y) = (< x, y > + c)^d$$

$$\text{Gaussian Radial Basis Function kernel (RBF): } k(x,y) = e^{-\frac{\|x-y\|^2}{2\sigma^2}}$$

In part1, the requirements can be meet by using three different kernel function: linear kernel, polynomial kernel, RBF kernel in LIBSVM library.

○ Part2 (10%)

```
def SVM(part, x_train, y_train, x_test, y_test, kernel_type, best_param):
    if part == 1:
        print(kernel_type)
        p  = svm_problem(y_train, x_train)
        param = svm_parameter('-t {} -q'.format(kernel[kernel_type]))
    elif part == 2:
        p  = svm_problem(y_train, x_train)
        param = svm_parameter('-s 0 -t {} -c {} -g {} -q'.format(kernel[kernel_type], best_param[0], best_param[1]))
    else:
        p  = svm_problem(y_train, x_train, isKernel=True)
        param = svm_parameter('-s 0 -t {} -c {} -g {} -q'.format(kernel[kernel_type], best_param[0], best_param[1]))

    model = svm_train(p, param)
    prediction = svm_predict(y_test, x_test, model)
```

Fig (12).

```
def GridSearch(x_train, y_train):
    C = [1e-3, 1e-2, 1e-1, 1, 10]
    gamma = [1e-3, 1e-2, 1e-1, 1]
    best_param = (0,0)
    best_acc = 0
    for c in C:
        for g in gamma:
            p  = svm_problem(y_train, x_train)
            param = svm_parameter('-s 0 -t 2 -v 5 -c {} -g {} -q'.format(c,g))
            acc = svm_train(p, param)
            if acc > best_acc:
                best_acc = acc
                best_param = (c,g)

    return best_param, best_acc
```

Fig (13).

```
# Part.2 C-SVC + RBF
best_param , best_acc = GridSearch(x_train, y_train)
print('Best paramters:', best_param)
print('Best Acc:', best_acc)
SVM(2, x_train, y_train, x_test, y_test, 'RBF', best_param)
```

Fig (14).

In part 2, in order to find best parameters (C, gamma) of C-SVC with RBF kernel, I write a GridSearch function to do this.

By setting C in [1e-3, 1e-2, 1e-1, 1, 10] and gamma in [1e-3, 1e-2, 1e-1, 1], and used two for loops to fit the model. Finally, I got best parameters combination of (10, 0.01) (shown in part b (2).) to reach the highest accuracy, then I used theses parameters (C = 10, gamma = 0.01) to fit the model.

○ Part3 (5%)

```
def SVM(part, x_train, y_train, x_test, y_test, kernel_type, best_param):
    if part == 1:
        print(kernel_type)
        p  = svm_problem(y_train, x_train)
        param = svm_parameter('-t {} -q'.format(kernel[kernel_type]))
    elif part == 2:
        p  = svm_problem(y_train, x_train)
        param = svm_parameter('-s 0 -t {} -c {} -g {} -q'.format(kernel[kernel_type], best_param[0], best_param[1]))
    else:
        p  = svm_problem(y_train, x_train, isKernel=True)
        param = svm_parameter('-s 0 -t {} -c {} -g {} -q'.format(kernel[kernel_type], best_param[0], best_param[1]))

    model = svm_train(p, param)
    prediction = svm_predict(y_test, x_test, model)
```

Fig (15).

```
def RBFKernel(x1, x2, gamma):
    dist = np.sum(x1 ** 2, axis=1).reshape(-1, 1) + np.sum(x2 ** 2, axis=1) - 2 * x1.dot(x2.transpose())
    kernel = np.exp((-1 * gamma * dist))

    return kernel
```

Fig (16).

Calculated RBF kernel by formula:

$$k(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

```
# Part.3 User-defined
gamma = best_param[1]
linear_kernel = x_train.dot(x_train.transpose())
RBF_kernel = RBFKernel(x_train, x_train, gamma)
x_kernel = np.hstack((np.arange(1, 5001).reshape((-1, 1)), linear_kernel + RBF_kernel.transpose()))

linear_kernel1 = x_train.dot(x_test.transpose()).transpose()
RBF_kernel1 = RBFKernel(x_train, x_test, gamma).transpose()
x_kernel1 = np.hstack((np.arange(1, 2501).reshape((-1, 1)), linear_kernel1 + RBF_kernel1))
SVM(3, x_kernel, y_train, x_kernel1, y_test, 'user-defined', best_param)
```

Fig (17).

In order to use user-defined kernel, first compute linear kernel (train examples, train examples) by formula, then calculate rbf kernel (train examples, train examples) followed rbf kernel formula. Add this two kernel together to form a new kernel. Do the same operation on kernel (train examples, test examples).

Finally, use this new kernel to train SVM model.

■ b. experiments settings and results (20%)

○ Part1 (5%)

Result is shown as bellow:

```
<libsvm.svm_problem object at
0x00000151AC6D7E40>
Accuracy = 95.08% (2377/2500) (classification)
polynomial kernel
Accuracy = 34.68% (867/2500) (classification)
rbf kernel
Accuracy = 95.32% (2383/2500) (classification)
```

Fig (18).

From the results we can see that linear kernel and RBF kernel performed better than polynomial kernel, and RBF kernel performed a bit better than linear kernel.

○ Part2 (10%)

Grid searching for parameters (C, gamma) using cross validation with C in [1e-3, 1e-2, 1e-1, 1, 10], gamma in [1e-3, 1e-2, 1e-1, 1], then use the best parameters to train C-SVC with RBF kernel SVM. Results is shown bellow:

```
Cross Validation Accuracy = 80.76%
Cross Validation Accuracy = 89.8%
Cross Validation Accuracy = 48.96%
Cross Validation Accuracy = 20.92%
Cross Validation Accuracy = 80.82%
Cross Validation Accuracy = 92.46%
Cross Validation Accuracy = 48.86%
Cross Validation Accuracy = 20.58%
Cross Validation Accuracy = 92.44%
Cross Validation Accuracy = 96.44%
Cross Validation Accuracy = 53.62%
Cross Validation Accuracy = 20.5%
Cross Validation Accuracy = 96.26%
Cross Validation Accuracy = 97.88%
Cross Validation Accuracy = 91.84%
Cross Validation Accuracy = 29.74%
Cross Validation Accuracy = 97.08%
Cross Validation Accuracy = 98.2%
Cross Validation Accuracy = 92.32%
Cross Validation Accuracy = 31.86%
Best paramters: (10, 0.01)
Best Acc: 98.2
```

Fig (19).

The best parameters combination of (C, gamma) is (10, 0.01), and best acc reach to 98.2%

○ Part3 (5%)

```
Accuracy = 95.32% (2383/2500) (classification)
```

Fig (20).

From the result can see that, accuracy of combined kernel is about 95.32%.

Compare above 3 parts of performance, we can find out that the gridsearched parameters used in C-SVC+ RBF kernel performed the best, which achieve 98.2% accuracy. (part2) Besides, using linear kernel, RBF kernel to train SVM model respectively, we can also observe that their performances are almost the same as using combined kernel of both of them.

■ c. observations and discussion (10%)

(1). In Part.1 experiment, I try sigmoid kernel as well, the results are shown as bellow:

```
print('sigmoid kernel')
p  = svm_problem(y_train, x_train)
param = svm_parameter('-t 3 -q')
model = svm_train(p, param)
RBF = svm_predict(y_test, x_test, model)
```

Fig (21).

Output:

```
sigmoid kernel
Accuracy = 94.8% (2370/2500) (classification)
```

Fig (22).

We can see that sigmoid kernel perform approximately the same as linear kernel and rbf kernel, and also better than polynomial kernel.

(2). In part3, I tried another kernel combination of linear kernel and exponential kernel

```
def ExponentialKernel(x1, x2, gamma):
    dist = np.sum(x1, axis=1).reshape(-1, 1) - np.sum(x2 , axis=1)
    kernel = np.exp((-1 * gamma * dist))

    return kernel
```

Fig (23).

Exponential kernel function followed by formula:

$$k(x, y) = \exp\left(-\frac{\|x - y\|}{2\sigma^2}\right)$$

```
# combination of linear kernel and exp kernel
exp_kernel = ExponentialKernel(x_train, x_train, gamma)
x_kernel = np.hstack((np.arange(1, 5001).reshape((-1, 1)), linear_kernel + exp_kernel.transpose()))

exp_kernel1 = ExponentialKernel(x_train, x_test, gamma).transpose()
x_kernel1 = np.hstack((np.arange(1, 2501).reshape((-1, 1)), linear_kernel1 + exp_kernel1))
SVM(3, x_kernel, y_train, x_kernel1, y_test, 'user-defined', best_param)
```

Fig (24).

Combination of linear kernel and exponential kernel
And get result bellow:

```
Accuracy = 95.16% (2379/2500) (classification)
```

Fig (25).

Which almost performed the same as kernel combination of linear kernel and exponential kernel.