2021NCTU Machine Learning-HW5

0610168 沈立淞

I. Gaussian Process

a. Code explanations:

Gaussian process is a nonparametric approach perhaps most powerful regressor before deep learning. Assume that the function that we want to fit is infinite dimension function, how do we get resulted functional distribution from finite training data? Given a finite set S = {x1, x2, …, xn} and its mapping:{f(x1), f(x2), …, f(xn)}, the main idea is that we can view each element in S as a random variable, and assume its mapping is isotropic Gaussian, thus {f(x1), f(x2), …, f(xn)} forming multivariate Gaussian distribution. We expect that if xi is close to xj, then f(xi) should close to f(xj). We check all points x in domain, examining the correlation between x and all the other random variable in set S, then we could approximately get a possible region of its label, based on S. So the problem becomes how do we define a correlation function to check the similarity between two data points. Here we use kernel function as our tool to check similarity, which has lots of nice properties, helping form a legal correlation(Gram) matrix for gaussian process.

In our homework, we have 34x2 matrix as our input data. Each row represents (Xi, Yi) in a 2D space, where Yi = *f*(Xi) + ϵi , ϵi is a noisy observation $\sim$ *N*(·|0, β-1). Function code is shown in figure 1 and 2.

We use Rational Quadratic Kernel as our kernel function:

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

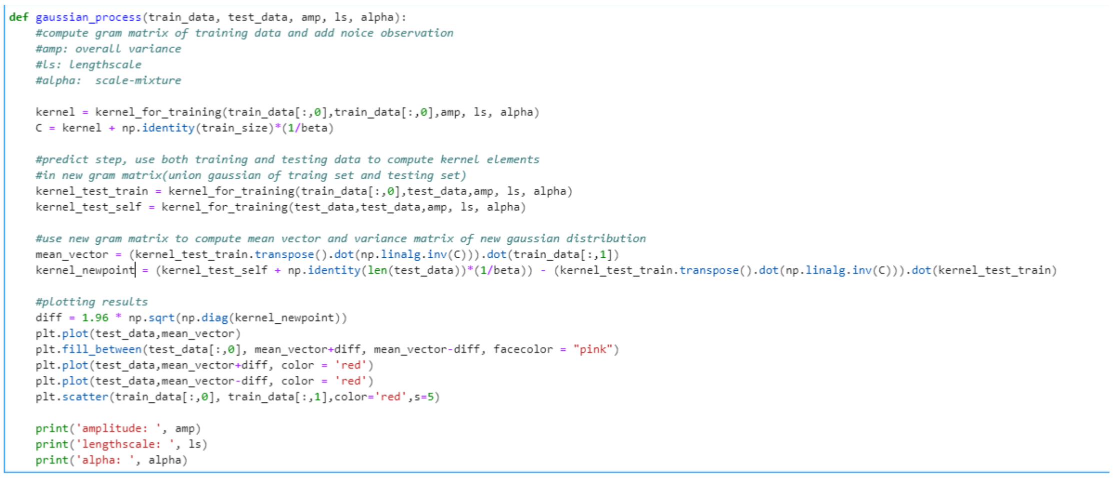
First, use training data, kernel function and noise observation to make our Gram matrix (here turn out to be a 32x32 matrix). Then use testing data (1000 points ranging from -60 to 60 in this case to represent a continuous function) and training data to compute the correlation by kernel function. Using results above, calculate the mean value and variance matrix with following formula:

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

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

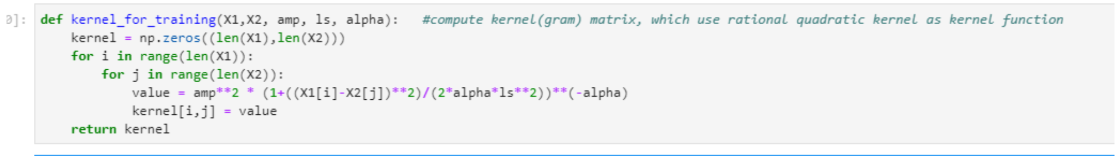
$$k^* = k(x^*, x^*) + \beta^{-1}$$

Finally, we can use the mean vector to represents the mean value of predict function.
As for 95% confidence interval, this is equal to $\mu \pm$ 1.96*standard derivation of each random variable. We can just use the diagonal elements of new variance matrix to get the result.



```python
def gaussian_process(train_data, test_data, amp, ls, alpha):
    #compute gram matrix of training data and add noice observation
    #amp: overall variance
    #ls: lengthscale
    #alpha:  scale-mixture

    kernel = kernel_for_training(train_data[:,0],train_data[:,0],amp, ls, alpha)
    C = kernel + np.identity(train_size)*(1/beta)

    #predict step, use both training and testing data to compute kernel elements
    #in new gram matrix(union gaussian of traing set and testing set)
    kernel_test_train = kernel_for_training(train_data[:,0],test_data,amp, ls, alpha)
    kernel_test_self = kernel_for_training(test_data,test_data,amp, ls, alpha)

    #use new gram matrix to compute mean vector and variance matrix of new gaussian distribution
    mean_vector = (kernel_test_train.transpose().dot(np.linalg.inv(C))).dot(train_data[:,1])
    kernel_newpoint = (kernel_test_self + np.identity(len(test_data))*(1/beta)) - (kernel_test_train.transpose().dot(np.linalg.inv(C))).dot(kernel_test_train)

    #plotting results
    diff = 1.96 * np.sqrt(np.diag(kernel_newpoint))
    plt.plot(test_data,mean_vector)
    plt.fill_between(test_data[:,0], mean_vector+diff, mean_vector-diff, facecolor = "pink")
    plt.plot(test_data,mean_vector+diff, color = 'red')
    plt.plot(test_data,mean_vector-diff, color = 'red')
    plt.scatter(train_data[:,0], train_data[:,1],color='red',s=5)

    print('amplitude: ', amp)
    print('lengthscale: ', ls)
    print('alpha: ', alpha)
```

Fg.1 gaussian process



```python
0]: def kernel_for_training(X1,X2, amp, ls, alpha):   #compute kernel(gram) matrix, which use rational quadratic kernel as kernel function
        kernel = np.zeros((len(X1),len(X2)))
        for i in range(len(X1)):
            for j in range(len(X2)):
                value = amp**2 * (1+((X1[i]-X2[j])**2)/(2*alpha*ls**2))**(-alpha)
                kernel[i,j] = value
        return kernel
```

Fg.2 rational quadratic kernel

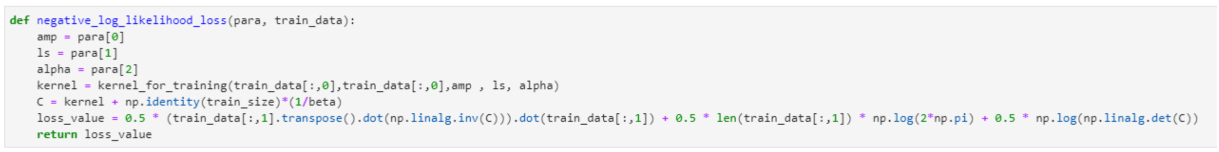
As for tuning hyper parameter, code is shown in figure 3 and 4.
First, we define the loss function as target function to find extreme parameter, here we use the marginal log-likelihood:

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

What's worth noticed is that we need to find the parameter θ such that $\ln P(Y/\theta)$ has maximum value. To form a minimization problem , we add a negative sign:

$$\text{-}\ln P(Y/\theta) = \frac{1}{2} ln|C_\theta| + \frac{1}{2} y^T C_\theta{}^{-1} y + \frac{N}{2} \ln(2\pi)$$

With formula above, now we can use scipy.optimize.minimize() to find the extreme parameter of kernel function.



```python
def negative_log_likelihood_loss(para, train_data):
    amp = para[0]
    ls = para[1]
    alpha = para[2]
    kernel = kernel_for_training(train_data[:,0],train_data[:,0],amp , ls, alpha)
    C = kernel + np.identity(train_size)*(1/beta)
    loss_value = 0.5 * (train_data[:,1].transpose().dot(np.linalg.inv(C))).dot(train_data[:,1]) + 0.5 * len(train_data[:,1]) * np.log(2*np.pi) + 0.5 * np.log(np.linalg.det(C))
    return loss_value
```

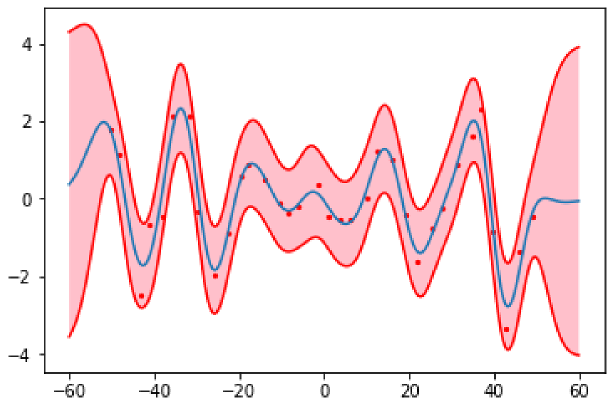
Fg.3 negative marginal log-likelihood

```
#part 2 optimize kernel parameter
init_guess = [amp, ls, alpha]
para = minimize(negative_log_likelihood_loss,init_guess,args=(data))
gaussian_process(data,test_data, para.x[0], para.x[1], para.x[2])
```

Fg.4 tuning hyper parameter

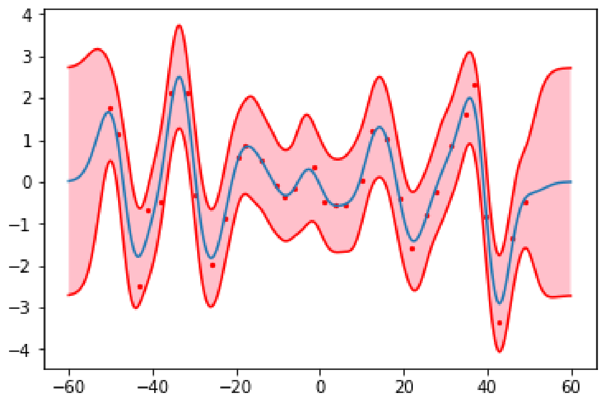
b. Experiments setting and results:

```
amplitude:  2
lengthscale:  5
alpha:  10
```



Fg.5 results of applying gaussian process

```
Initial parameters:
amplitude:  2
lengthscale:  5
alpha:  10

result parameter and results:
amplitude:  1.3135093201102617
lengthscale:  3.318293093010842
alpha:  803.8192713497057
```



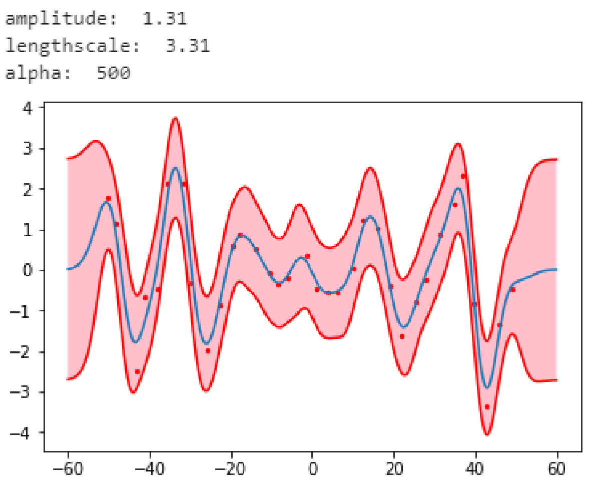
Fg.6 results of optimizing kernel parameters

c. Observations and discussion

As I mentioned above, gaussian process is a nonparametric approach, given that the kernel parameter is determined. That is to say, the model performance is basically based on kernel parameter(also called hyper parameter in gaussian process).

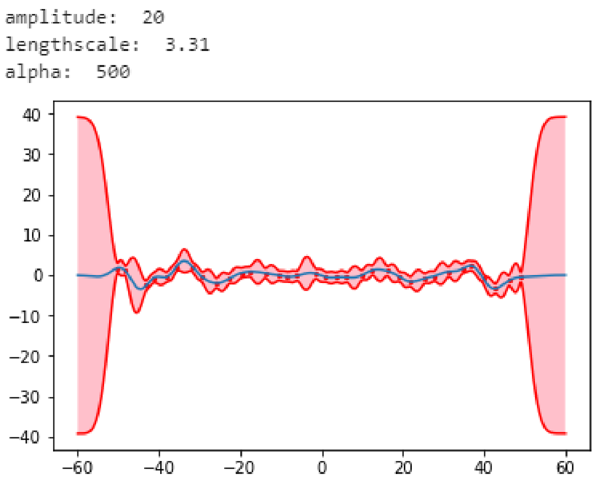
$$k(x_a, x_b) = \sigma^2 (1 + \frac{\|x_a - x_b\|^2}{2\alpha l^2})^{-\alpha}$$

In Ration Rational Quadratic Kernel, as σ^2 grows, we can observe in figure 7 and 8 such that the 95% confidence interval grows fast when there is no training points around the test data, thus σ^2 is also called amplitude, which can control the confidence interval of resulted function distribution.



```
amplitude:  1.31
lengthscale:  3.31
alpha:  500
```

Fg.7 small σ^2

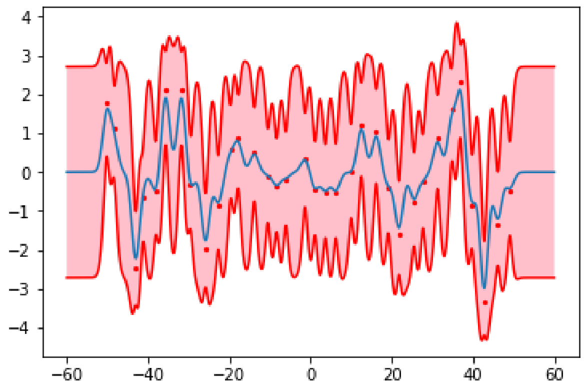


```
amplitude:  20
lengthscale:  3.31
alpha:  500
```

Fg.8 large σ^2

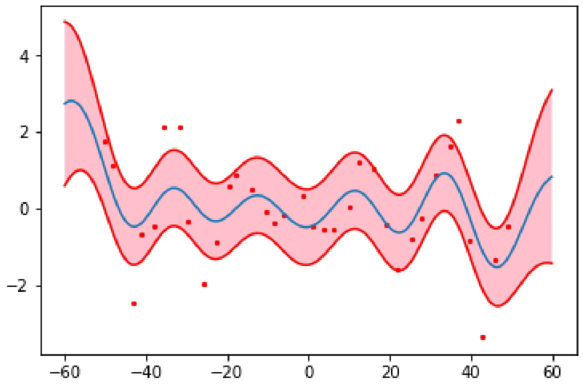
As length scale(l) grows, we can observe in figure 9 and 10 such that the mean value of function becomes smoother. This indicate that length scale could control the degree of smoothy of our function.

```
result parameter and results:
amplitude:  1.31
lengthscale:  1
alpha:  500
```



Fg.9 small lengthscale

```
amplitude:  1.31
lengthscale:  10
alpha:  500
```



Fg.10 larger lengthscale

II. SVM

a. Code explanations:

Part1. Applying different kernels:

The code shown in figure 11 is based on libsvm.

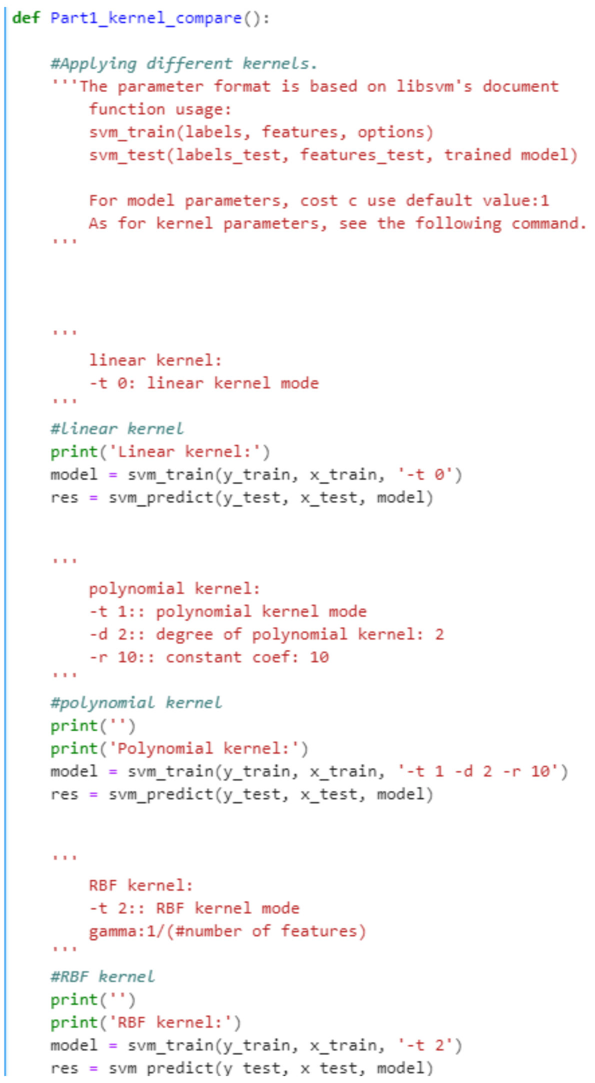
For kernels, we use the following formula:

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

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

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

$$d \in Z^+, \sigma \in \mathcal{R} - \{0\}$$



```
def Part1_kernel_compare():

    #Applying different kernels.
    '''The parameter format is based on libsvm's document
        function usage:
        svm_train(labels, features, options)
        svm_test(labels_test, features_test, trained model)

        For model parameters, cost c use default value:1
        As for kernel parameters, see the following command.
    '''



    '''
        linear kernel:
        -t 0: linear kernel mode
    '''
    #linear kernel
    print('Linear kernel:')
    model = svm_train(y_train, x_train, '-t 0')
    res = svm_predict(y_test, x_test, model)


    '''
        polynomial kernel:
        -t 1:: polynomial kernel mode
        -d 2:: degree of polynomial kernel: 2
        -r 10:: constant coef: 10
    '''
    #polynomial kernel
    print('')
    print('Polynomial kernel:')
    model = svm_train(y_train, x_train, '-t 1 -d 2 -r 10')
    res = svm_predict(y_test, x_test, model)


    '''
        RBF kernel:
        -t 2:: RBF kernel mode
        gamma:1/(#number of features)
    '''
    #RBF kernel
    print('')
    print('RBF kernel:')
    model = svm_train(y_train, x_train, '-t 2')
    res = svm predict(y test, x test, model)
```

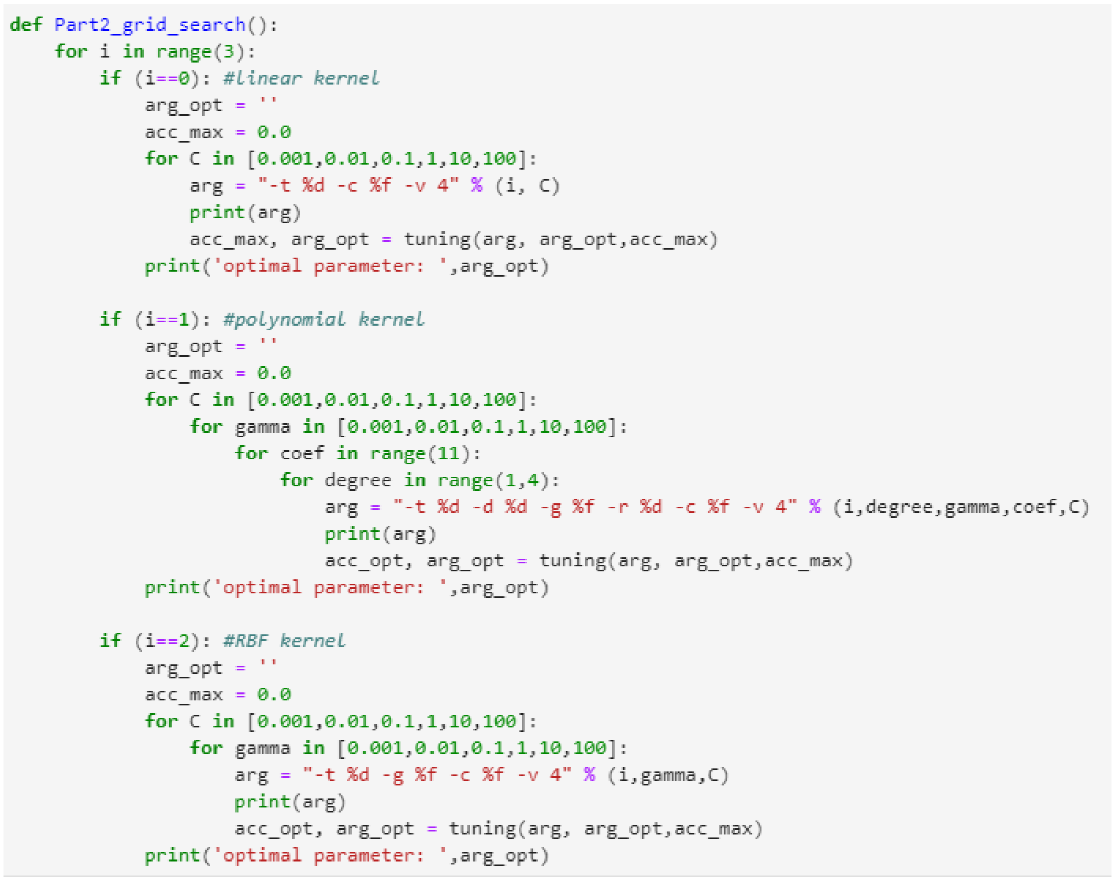
Fg11. Code of Part1.

Part2. Grid search for tuning parameter:

The main idea of grid search is an exhaustive searching through a manually specified subset of the hyper parameter space of a learning algorithm. The main algo rithm is as follows:

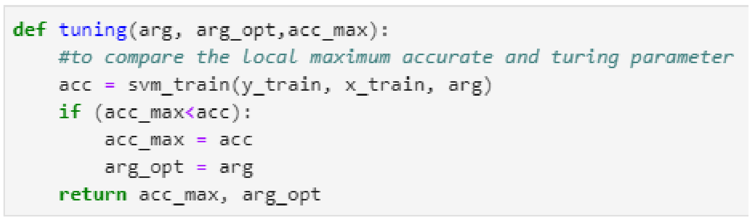
```
Grid_search():
Setting parameters bounds for your model
For each combination of parameters:
    Choose the parameters such that svm_train_with_crossvalidation(y, x, parameters) returns maximum accuracy.
Return result parameters
```

For linear kernel, cost “C” in the model is the only parameter we need to consider. And for RBF kernel, we need to consider both cost “C” and “gamma”. Use the tuning function in figure13 to do 4-fold cross validation for each parameter, and compare with current best model.



```
def Part2_grid_search():
    for i in range(3):
        if (i==0): #linear kernel
            arg_opt = ''
            acc_max = 0.0
            for C in [0.001,0.01,0.1,1,10,100]:
                arg = "-t %d -c %f -v 4" % (i, C)
                print(arg)
                acc_max, arg_opt = tuning(arg, arg_opt,acc_max)
            print('optimal parameter: ',arg_opt)

        if (i==1): #polynomial kernel
            arg_opt = ''
            acc_max = 0.0
            for C in [0.001,0.01,0.1,1,10,100]:
                for gamma in [0.001,0.01,0.1,1,10,100]:
                    for coef in range(11):
                        for degree in range(1,4):
                            arg = "-t %d -d %d -g %f -r %d -c %f -v 4" % (i,degree,gamma,coef,C)
                            print(arg)
                            acc_opt, arg_opt = tuning(arg, arg_opt,acc_max)
            print('optimal parameter: ',arg_opt)

        if (i==2): #RBF kernel
            arg_opt = ''
            acc_max = 0.0
            for C in [0.001,0.01,0.1,1,10,100]:
                for gamma in [0.001,0.01,0.1,1,10,100]:
                    arg = "-t %d -g %f -c %f -v 4" % (i,gamma,C)
                    print(arg)
                    acc_opt, arg_opt = tuning(arg, arg_opt,acc_max)
            print('optimal parameter: ',arg_opt)
```

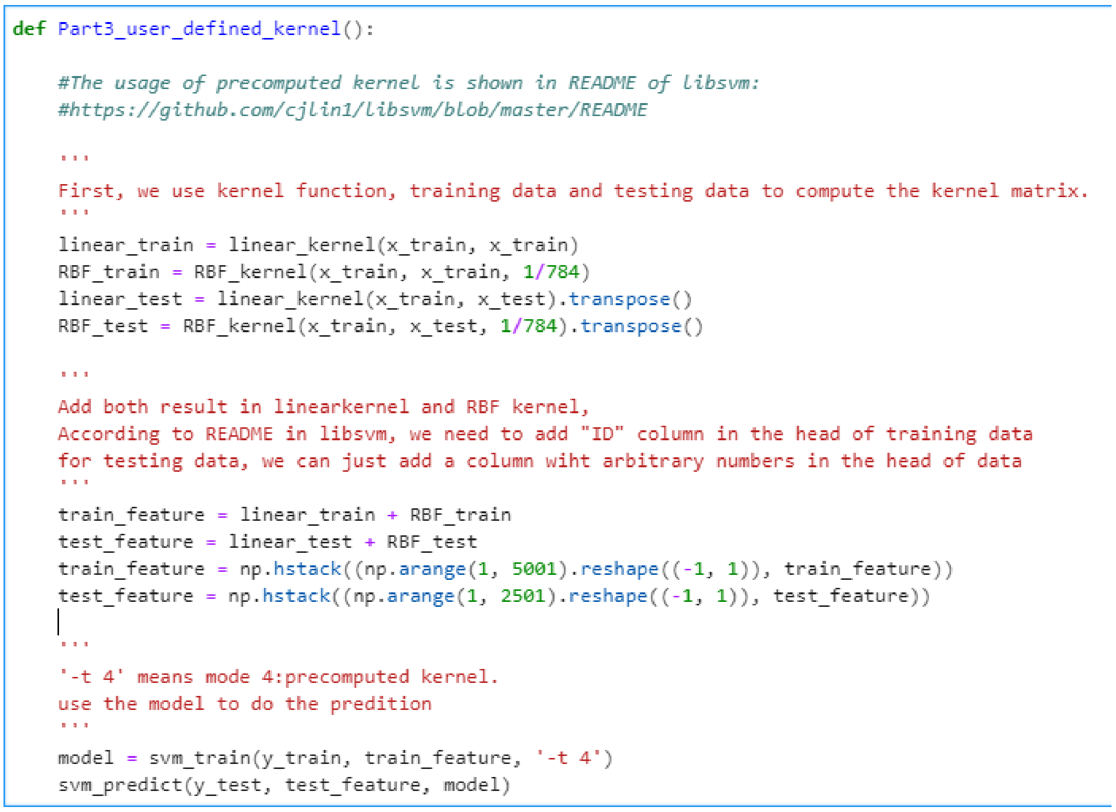
Fg.12 Code of Grid search.



```python
def tuning(arg, arg_opt,acc_max):
    #to compare the local maximum accurate and turing parameter
    acc = svm_train(y_train, x_train, arg)
    if (acc_max<acc):
        acc_max = acc
        arg_opt = arg
    return acc_max, arg_opt
```

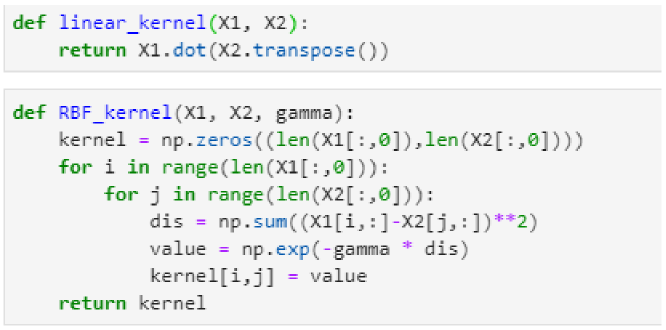
Fg.13 Tuning function

Part3. Self-defined kernel



```python
def Part3_user_defined_kernel():

    #The usage of precomputed kernel is shown in README of libsvm:
    #https://github.com/cjlin1/libsvm/blob/master/README

    '''
    First, we use kernel function, training data and testing data to compute the kernel matrix.
    '''
    linear_train = linear_kernel(x_train, x_train)
    RBF_train = RBF_kernel(x_train, x_train, 1/784)
    linear_test = linear_kernel(x_train, x_test).transpose()
    RBF_test = RBF_kernel(x_train, x_test, 1/784).transpose()

    '''
    Add both result in linearkernel and RBF kernel,
    According to README in libsvm, we need to add "ID" column in the head of training data
    for testing data, we can just add a column wiht arbitrary numbers in the head of data
    '''
    train_feature = linear_train + RBF_train
    test_feature = linear_test + RBF_test
    train_feature = np.hstack((np.arange(1, 5001).reshape((-1, 1)), train_feature))
    test_feature = np.hstack((np.arange(1, 2501).reshape((-1, 1)), test_feature))

    '''
    '-t 4' means mode 4:precomputed kernel.
    use the model to do the predition
    '''
    model = svm_train(y_train, train_feature, '-t 4')
    svm_predict(y_test, test_feature, model)
```

Fg.14 Self-defined kernel



```
def linear_kernel(X1, X2):
    return X1.dot(X2.transpose())
```

```
def RBF_kernel(X1, X2, gamma):
    kernel = np.zeros((len(X1[:,0]),len(X2[:,0])))
    for i in range(len(X1[:,0])):
        for j in range(len(X2[:,0])):
            dis = np.sum((X1[i,:]-X2[j,:])**2)
            value = np.exp(-gamma * dis)
            kernel[i,j] = value
    return kernel
```

Fg.15 kernel function

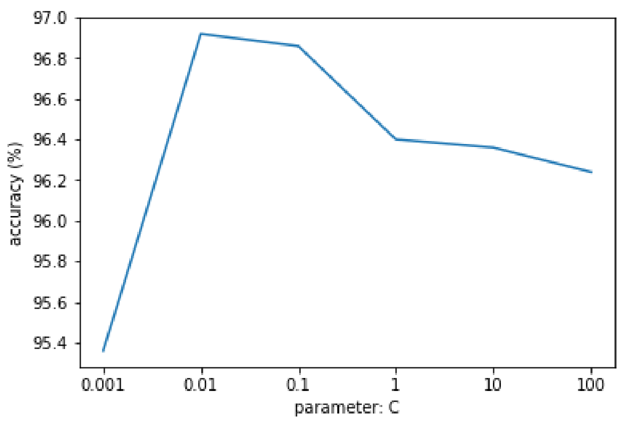
b. Experiments setting and results:

Part1:Applying different kernels

```
Linear kernel:
cost = 1
Accuracy = 95.08% (2377/2500) (classification)
----------
Polynomial kernel:
cost = 1
degree = 2
coef0 = 10
Accuracy = 95.96% (2399/2500) (classification)
----------
RBF kernel:
gamma = 1/784
Accuracy = 95.32% (2383/2500) (classification)
```

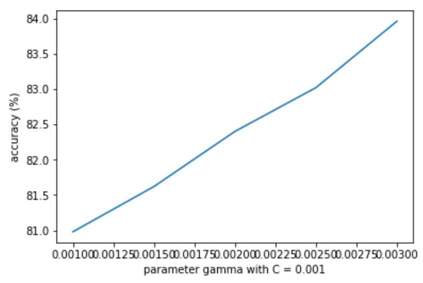
Part2:Tuning Hyper parameters

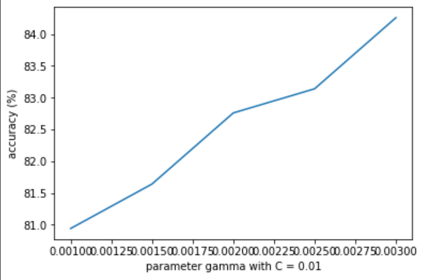
Linear kernel:

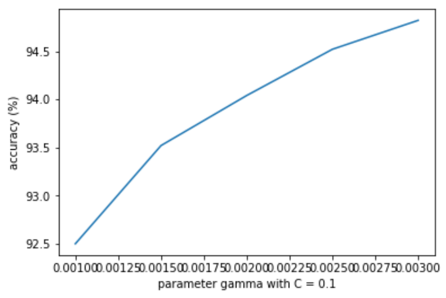


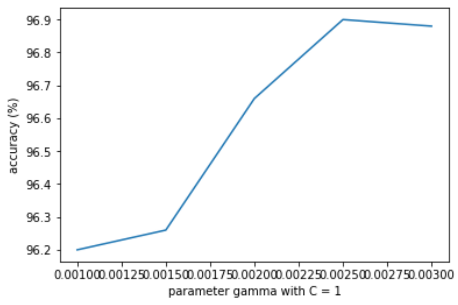
With parameter bound for cost C = [0.001, 0.01, 0.1, 1, 10, 100], C=0.01 has best performance with accurate = 96.92%

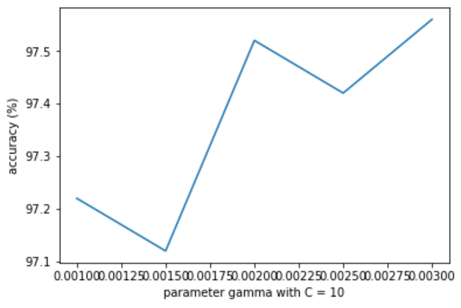
RBF kernel:

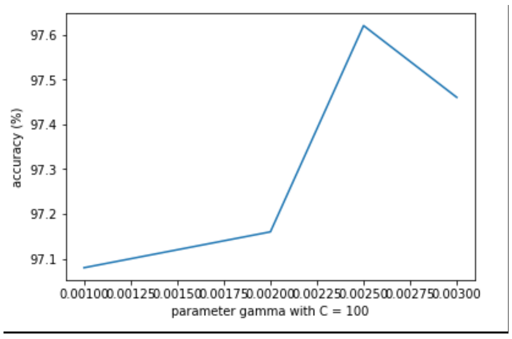






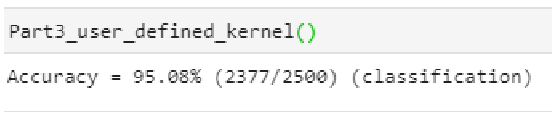






With parameter bound for cost C = [0.001, 0.01, 0.1, 1, 10, 100], gamma = [0.001, 0.0015, 0.002, 0.0025, 0.003],
C = 100 and gamma = 0.003 has best performance with accuracy = 97.62%.

Part3:User defined kernel:



```
Part3_user_defined_kernel()
Accuracy = 95.08% (2377/2500) (classification)
```

c. Observations and discussion:

As we shall see, the performance of RBF kernel has massive difference from C = 0.001 and C = 0.1, accuracy grows from8X% to 9X%.However, as C grows from 0.1 to 100, accuracy grows little from 92.5% to maximum accuracy of 97.62%.

Reference:

機器學習: Kernel 函數

交叉驗證 Cross-Validation

Gaussian process kernels

浅谈高斯过程回归

Machine Learning for Intelligent Systems [FALL 2018]--Cornell University

Hyperparameter optimization

libsvm/README