Machine Learning Homework 5

Student Name: 吳政緯

Student ID: 0510507

Introduction

First, I implemented a Gaussian Process Regression to compute the posterior without optimizing the parameters. Then I tried to use scipy.optimize to find the optimal parameters of the kernel function.

The approach

I. Gaussian Process

Gaussian Process is a distributions over function $\mathrm{f(x)}$ which is defined by the mean function $\mathrm{m(x)}$ and covariance function $\mathrm{k(x, x')}$. The formula of the prior function is as following.

$$\mathrm{f(x)} \sim \mathrm{N}\big(\mathrm{m(X)}, \mathrm{K(X, X)}\big)$$

II. Gaussian Process for Regression

We can treat the Gaussian Process with training datasets as a prior defined by the kernel function and create the posterior distribution. This posterior function can be used to predict the value of given data.

III. Predictions from posterior

Suppose we have a training datasets X1 and labels Y1. We want to predict the given datasets X2. This can be done with the help of posterior distribution.

$$\mathrm{p(y2|y1, X1, X2)} = N(\mu_{2|1}, \Sigma_{2|1})$$
$$\mu_{2|1} = (\Sigma_{11}^{-1}\Sigma_{12})^T y_1$$
$$\Sigma_{2|1} = \Sigma_{22} - (\Sigma_{11}^{-1}\Sigma_{12})^T\Sigma_{12}$$

IV. Noisy Observation

If we want to make the prediction with noise, we could add the noise parameter. The noise can be modeled by adding variance to our covariance kernel of our observation.

$$\Sigma_{11} = k(X_1, X_1) + \beta^{-2}$$

V. Rational quadratic kernel

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

with:

σ^2 the variance

ℓ the lengthscale

α the scale-mixture

VI. Optimize hyperparameters

the maximum marginal likelihood

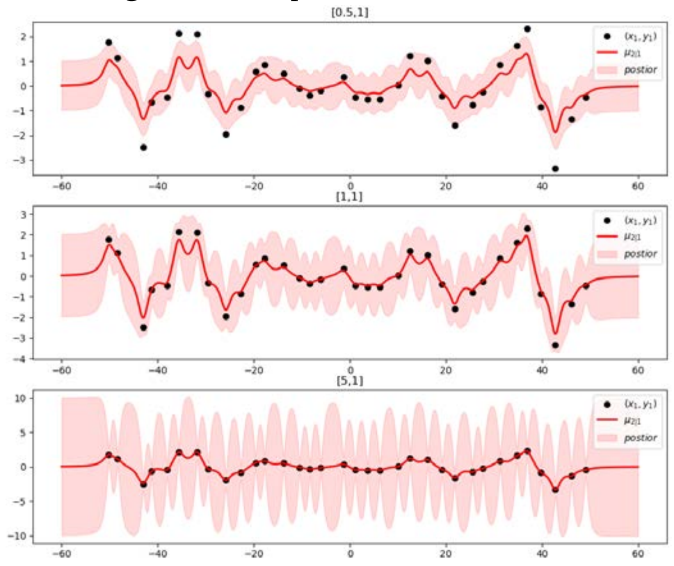
argmax(p(y|X,θ)) = $-\frac{1}{2}\ln|\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top\mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln(2\pi)$

with rational quadratic kernel function parameter σ^2 and α

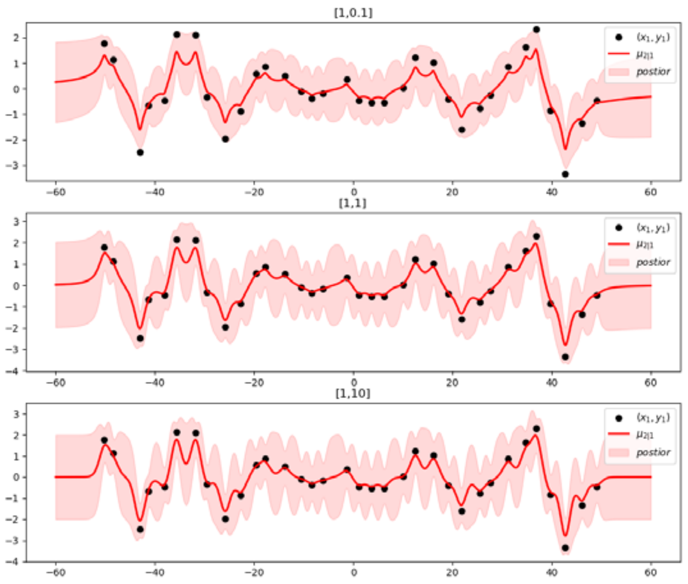
Discussion/Experiments

I. Non-optimized Gaussian process model

A. plotting the result with different σ parameters and set the α to be 1.The range of σ is [0.5 , 1 , 5]. Following figure is the result. We could find that if σ is too large, it would be overfitting to the data points.



B. Plotting the result with different α parameters and set the σ to be 1. The range of α is [0.1,1,10]. Following figure is the result. We could find that if α is larger, the function will be more smooth.



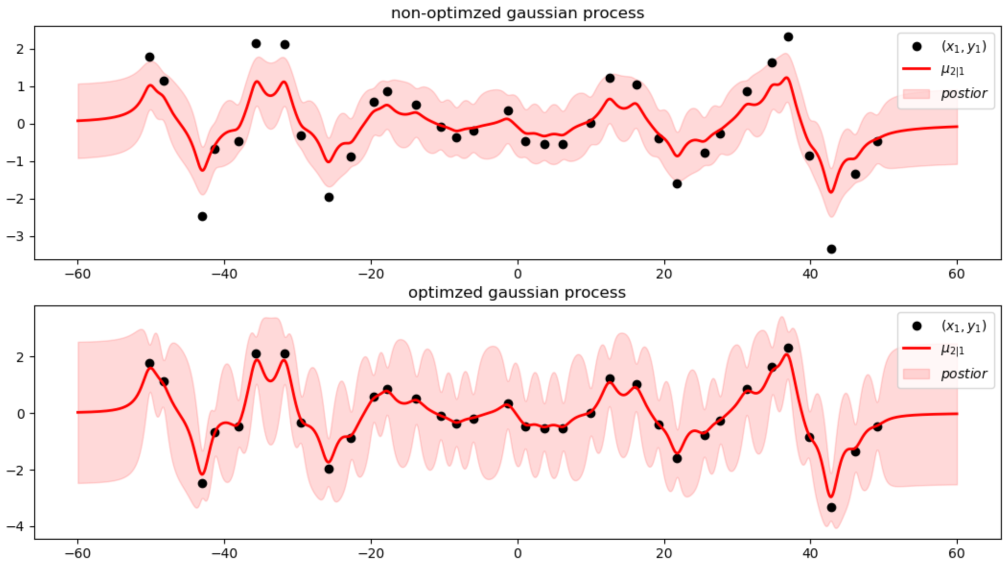
II. Optimized Gaussian process model

A. Different optimized function methods

Optimized functions	σ value	α value
Nelder-Mead	1.25	0.99
Powell	1.25	0.99
CG	1.25	0.99

B. Result

The second picture in the figure is the optimized result with parameter $\sigma = 1.25$ $\alpha = 0.99$.

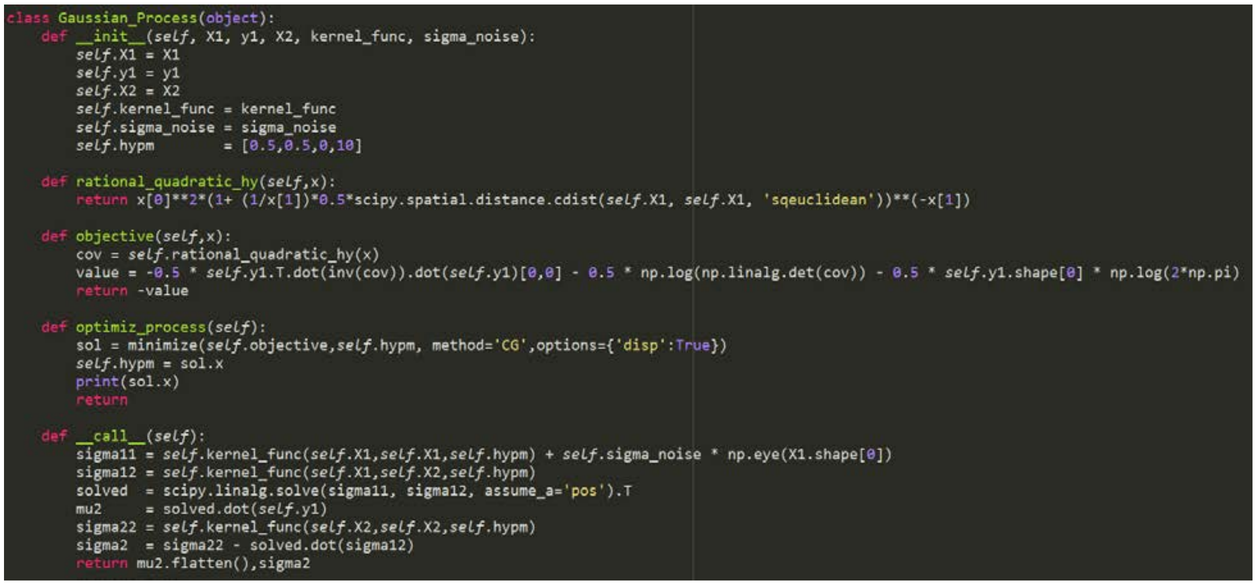


III. Code explanation

A.

I create a Gaussian Process class to solve the problem and it's structure is as the following.

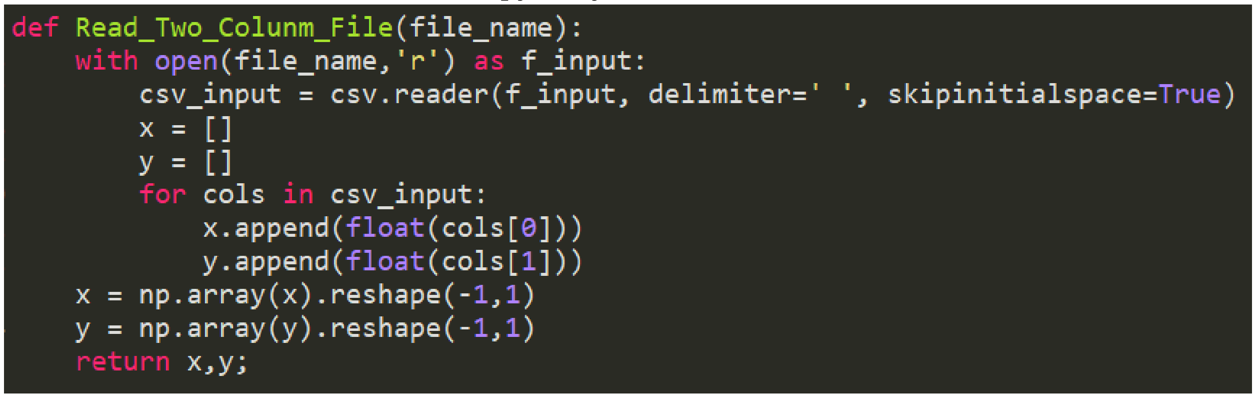
Function member name	function
__init__	Put the training data, label and parameters into class object.
rational_quadratic_hy	Compute $\sum_{11}$ with tuning parameters.
objective	Compute the log marginal likelihood.
optimiz_process	Use scipy optimization package to optimize the parameters
__call__	Compute the posterior value $\mu_{2\|1}$ and $\sum_{2\|1}$



```python
class Gaussian_Process(object):
    def __init__(self, X1, y1, X2, kernel_func, sigma_noise):
        self.X1 = X1
        self.y1 = y1
        self.X2 = X2
        self.kernel_func = kernel_func
        self.sigma_noise = sigma_noise
        self.hypm        = [0.5,0.5,0,10]

    def rational_quadratic_hy(self,x):
        return x[0]**2*(1+ (1/x[1])*0.5*scipy.spatial.distance.cdist(self.X1, self.X1, 'sqeuclidean'))**(-x[1])

    def objective(self,x):
        cov = self.rational_quadratic_hy(x)
        value = -0.5 * self.y1.T.dot(inv(cov)).dot(self.y1)[0,0] - 0.5 * np.log(np.linalg.det(cov)) - 0.5 * self.y1.shape[0] * np.log(2*np.pi)
        return -value

    def optimiz_process(self):
        sol = minimize(self.objective,self.hypm, method='CG',options={'disp':True})
        self.hypm = sol.x
        print(sol.x)
        return

    def __call__(self):
        sigma11 = self.kernel_func(self.X1,self.X1,self.hypm) + self.sigma_noise * np.eye(X1.shape[0])
        sigma12 = self.kernel_func(self.X1,self.X2,self.hypm)
        solved  = scipy.linalg.solve(sigma11, sigma12, assume_a='pos').T
        mu2     = solved.dot(self.y1)
        sigma22 = self.kernel_func(self.X2,self.X2,self.hypm)
        sigma2  = sigma22 - solved.dot(sigma12)
        return mu2.flatten(),sigma2
```

B. Reading input.data function

It will read the file and create two numpy arrays to store the data and labels.



```python
def Read_Two_Colunm_File(file_name):
    with open(file_name,'r') as f_input:
        csv_input = csv.reader(f_input, delimiter=' ', skipinitialspace=True)
        x = []
        y = []
        for cols in csv_input:
            x.append(float(cols[0]))
            y.append(float(cols[1]))
    x = np.array(x).reshape(-1,1)
    y = np.array(y).reshape(-1,1)
    return x,y;
```

Introduction

First, I use the LIBSVM, which implement many different kernels, to compute the hard margin SVM model for given MINIST. Try to use different kernels and compare their performance. Second, use C-SVM to solve the optimization problem for MINIST and tune the different parameters. Finally, we can use the custom kernel function in LIBSVM and compare the performance with other kernels.

The approach

I. SVM kernel function

Solving SVM optimization problem will face the quadratic programming(QP) optimization problem. Originally, the time complexity of QP will depend on dimension of given transformed data. We can use the defined kernel function to avoid high dimension computation and just get the kernel result. Using the kernel still has the high dimension property, but the complexity is much smaller than changing the data to high dimension directly.

II. Different Kernel function

The function ϕ in turn defines the kernel function $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$. Different kernels work well for different datasets. The following kernels are available in LIBSVM:

1. Linear: $K(x_i, x_j) = x_i{}^T x_j$
2. Ploynomial: $K(x_i, x_j) = (\gamma x_i{}^T x_j + r)^s$
3. RBF: $K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$

III. C-SVM

$$\min_{\vec{w},b,\vec{\xi}} \quad \frac{\|\vec{w}\|_{\ell_2}^2}{2} + C\|\vec{\xi}\|_{\ell_1}$$

$$\text{subject to} \quad y_i(\vec{w}^T\phi(\vec{x}_i) + b) \geq 1 - \xi_i,\ \xi_i \geq 0 \quad \forall i \in [N],$$

If c is large, the slack dominate the function and few points can be in the SVM margin.
If c is small, the slack will not influence the function very much and more points can be in the SVM margin.

Experiments

I. Different Kernel Functions without tuning the parameters

A. Linear Kernel Functions
$K(x_i, x_j) = x_i{}^T x_j$ is the linear kernel function.
Accuracy = 95.08%

B. Polynomial Kernel Functions
Accuracy = 34.68%

C. RBF Kernel Functions
Accuracy = 95.32%

II. Use C-SVM with different parameters

A. Linear Kernel Functions:
The highest accuracy is 96% with cost value 0.03.

Cost value	Accuracy
0.001	94.68%
0.01	95.96%
0.02	95.96%
0.03	96%
0.1	95.8%
1	95.08%

B. Linear Kernel Functions cost value=0.03 with 10-folds cross-validation:
Accuracy = 96.2%

C. Polynomial Kernel Functions
I ran the polynomial kernel with various value of
C {1, 10, 20, 30, 100}, gama {0.001, 0.01, 0.1, 1, 10}, degree = 3.
We can find that if c is too small, we will get a underfitting model.

gama / c	0.001	0.01	0.1	1	10
1	28.88%	97.04%	97.48%	97.48%	97.48%
10	68.04%	97.76%	97.48%	97.48%	97.48%
20	79.12%	97.44%	97.48%	97.48%	97.48%
30	83.4%	97.48%	97.48%	97.48%	97.48%
100	91.64%	97.48%	97.48%	97.48%	97.48%

I ran the polynomial kernel with various value of
C {1, 10, 20, 30, 100}, gama { 0.01, 0.1, 1, 10}, degree = 2.

gama / c	0.01	0.1	1	10
1	97.28%	97.68%	97.68%	96.48%
10	97.76%	97.68%	97.68%	97.68%
20	97.72%	97.68%	97.68%	97.68%
30	97.68%	97.68%	97.68%	97.68%
100	97.68%	97.68%	97.68%	97.68%

D. Polynomial Kernel Function with cross-validation

Some of the highest accuracies witnessed for some of the parameter pairings mentioned above were C = 10, gama = 0.01, degree = 3 and C=10, gama = 0.01, degree = 2. To make sure the accuracies aren't accident, we run the cross-validation with 10 folds in training datasets.

Parameters	Average accuracy
C=10,gama = 0.01, degree =2	98.6
C=10,gama = 0.01, degree =3	97.7

E. RBF Kernel Function

I ran the polynomial kernel with various value of
C {0.1, 1, 10 , 100, 1000}, gama {0.001, 0.01, 0.02 , 0.03, 0.1 }.

gama / c	0.001	0.01	0.02	0.03	0.1

0.1	92.36%	95.6%	96.52%	96.48%	50.88%
1	95.24%	97.52%	98.36%	98.48%	90.04%
10	96.2%	98.2%	98.48%	97.56%	91%
100	96.84%	98.16%	98.48%	98.56%	91%
1000	96.56%	98.16%	98.48%	98.56%	97.1%

F. RBF Kernel Function with cross-validation
Some of the highest accuracies witnessed for some of the parameter pairings mentioned above were C = 100, gama = 0.03 and C=1000, gama = 0.03. To make sure the accuracies aren't accident, we run the cross-validation with 10 fold in training datasets.

Parameters	Average accuracy
C=100,gama = 0.03	98.5
C=1000,gama = 0.03	98.9

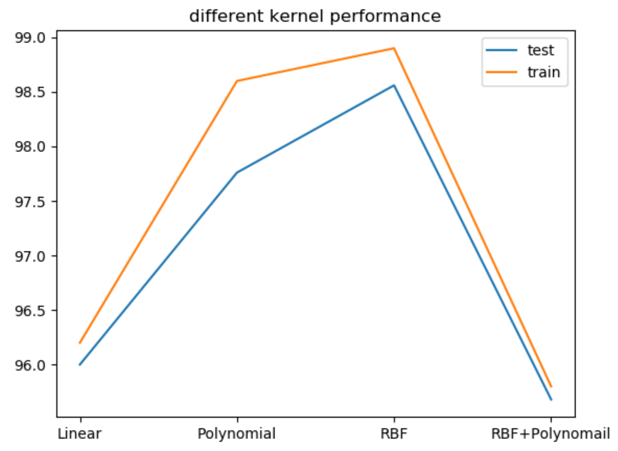
III. Use linear kernel plus RBF kernel function

$$K(x_i, x_j) = {x_i}^T x_j + \exp\left(-0.05\|x_i - x_j\|^2\right)$$

Accuracy : 95.68%

We can see that the result is not better than using only RBF kernel function. Because the C-SVM C for linear and RBF kernel is different, the result is not much better.

IV. Compare performance

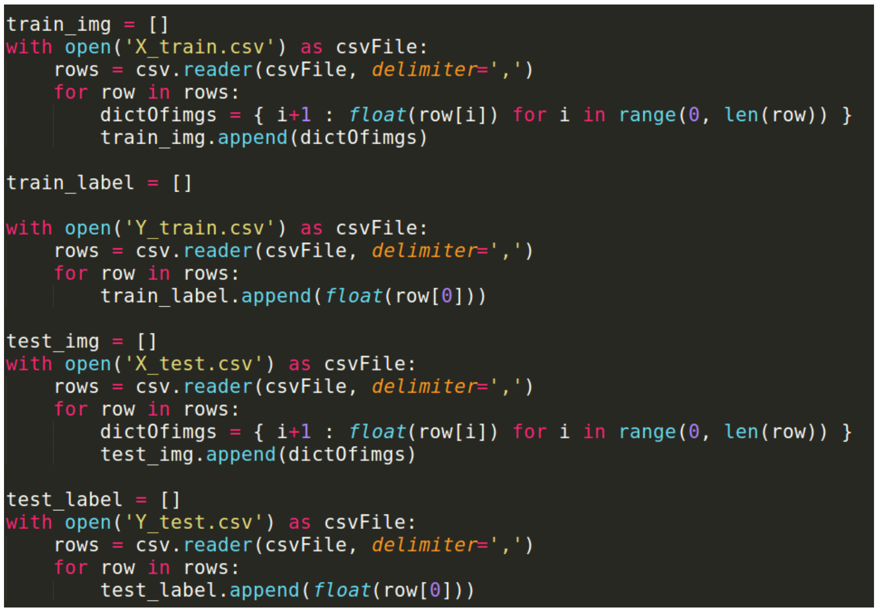


In my experiments, the RBF have better performance and I think this is because RBF have the infinite dimension property.

Code explanation

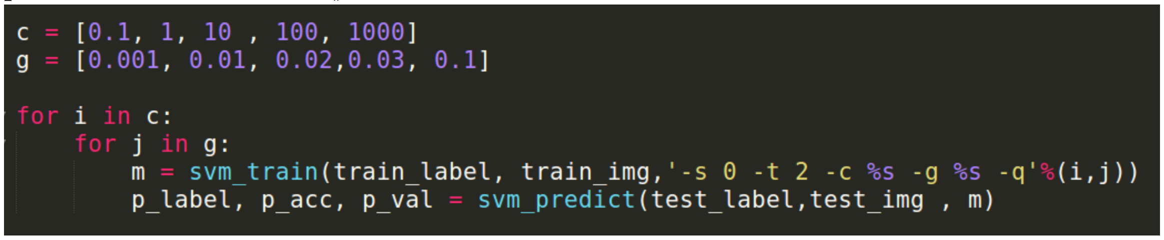
I. Read the train data file and test data file
Because the libsvm has it's special input format, we should follow the format. As the

following code, I creates the train data list and train label list for training and test data list and test_label for testing.
Python structure:
lable = [1,2] data = [{1:2,2:3},{1:3,2:3}]



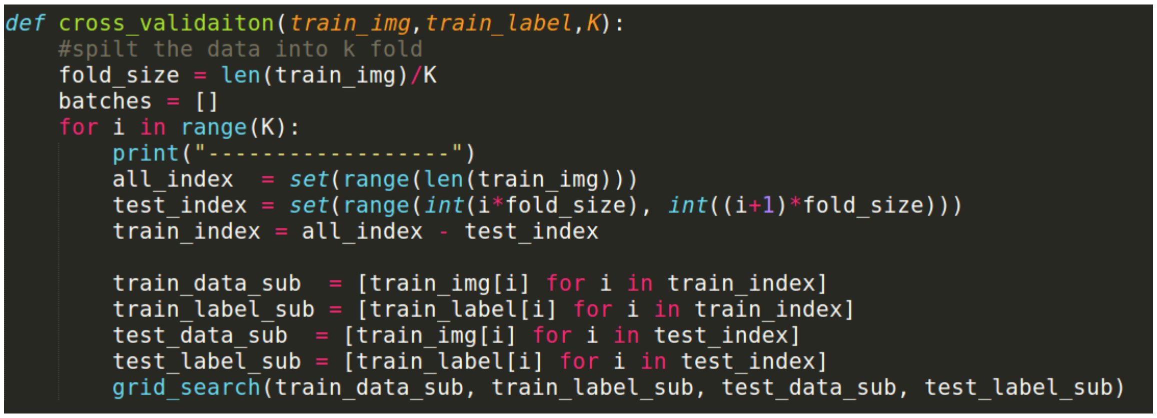
```python
train_img = []
with open('X_train.csv') as csvFile:
    rows = csv.reader(csvFile, delimiter=',')
    for row in rows:
        dictOfimgs = { i+1 : float(row[i]) for i in range(0, len(row)) }
        train_img.append(dictOfimgs)

train_label = []

with open('Y_train.csv') as csvFile:
    rows = csv.reader(csvFile, delimiter=',')
    for row in rows:
        train_label.append(float(row[0]))

test_img = []
with open('X_test.csv') as csvFile:
    rows = csv.reader(csvFile, delimiter=',')
    for row in rows:
        dictOfimgs = { i+1 : float(row[i]) for i in range(0, len(row)) }
        test_img.append(dictOfimgs)

test_label = []
with open('Y_test.csv') as csvFile:
    rows = csv.reader(csvFile, delimiter=',')
    for row in rows:
        test_label.append(float(row[0]))
```

II. Grid search parameters
Because C-SVM could assign the C value and gama value for polynomial, RBF kernel, we have to try many combinations of parameters. I iterate different parameters and put the parameter into svm_train() function.



```python
c = [0.1, 1, 10 , 100, 1000]
g = [0.001, 0.01, 0.02,0.03, 0.1]

for i in c:
    for j in g:
        m = svm_train(train_label, train_img,'-s 0 -t 2 -c %s -g %s -q'%(i,j))
        p_label, p_acc, p_val = svm_predict(test_label,test_img , m)
```

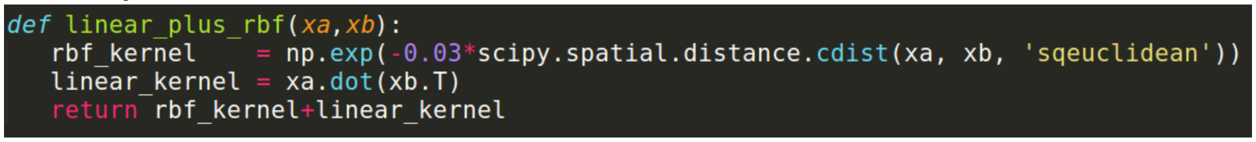
III. Cross validation



```python
def cross_validaiton(train_img,train_label,K):
    #spilt the data into k fold
    fold_size = len(train_img)/K
    batches = []
    for i in range(K):
        print("------------------")
        all_index  = set(range(len(train_img)))
        test_index = set(range(int(i*fold_size), int((i+1)*fold_size)))
        train_index = all_index - test_index

        train_data_sub  = [train_img[i] for i in train_index]
        train_label_sub = [train_label[i] for i in train_index]
        test_data_sub  = [train_img[i] for i in test_index]
        test_label_sub = [train_label[i] for i in test_index]
        grid_search(train_data_sub, train_label_sub, test_data_sub, test_label_sub)
```

After shuffling the dataset, I can divide the train dataset to be train data and validation data.

IV. Define my own kernel



```python
def linear_plus_rbf(xa,xb):
   rbf_kernel    = np.exp(-0.03*scipy.spatial.distance.cdist(xa, xb, 'sqeuclidean'))
   linear_kernel = xa.dot(xb.T)
   return rbf_kernel+linear_kernel
```

This is the kernel combining the linear kernel and RBF kernel.