Machine Learning HW 5

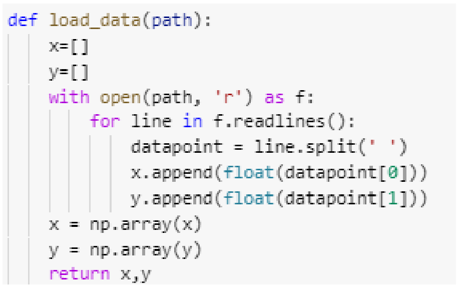
Gaussian Process & SVM

GMBA 310712009 楊家碩

1. Gaussian Process

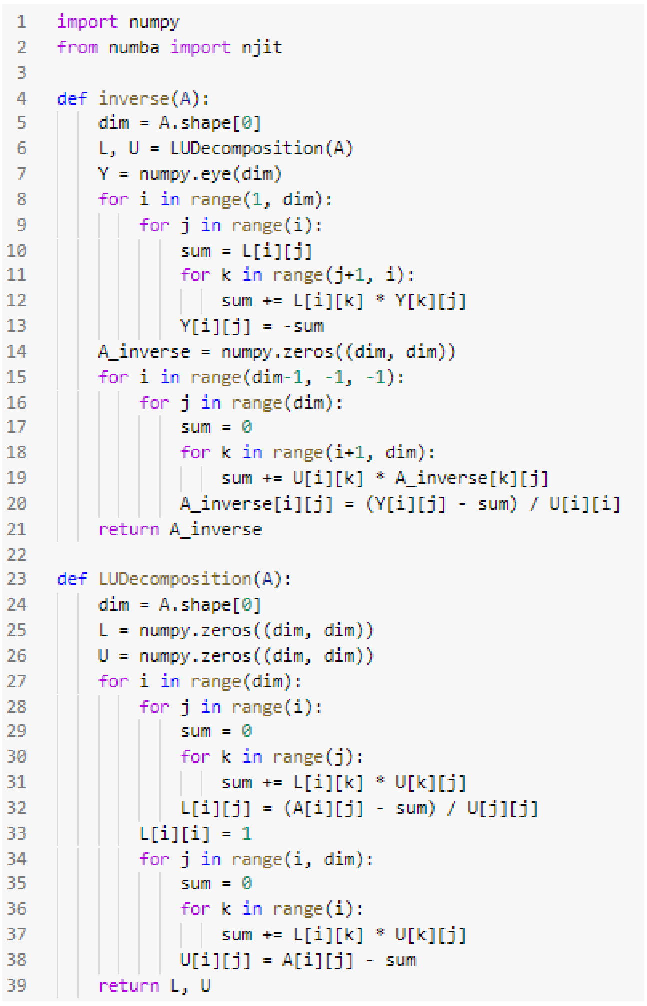
Part1:

In order to forecast the distribution of f and display the outcome, we apply Gaussian Process Regression. Data loading comes first; I build a load data function.



```python
def load_data(path):
    x=[]
    y=[]
    with open(path, 'r') as f:
        for line in f.readlines():
            datapoint = line.split(' ')
            x.append(float(datapoint[0]))
            y.append(float(datapoint[1]))
    x = np.array(x)
    y = np.array(y)
    return x,y
```

I also use the inverse function that I build from the HW2 for this homework.

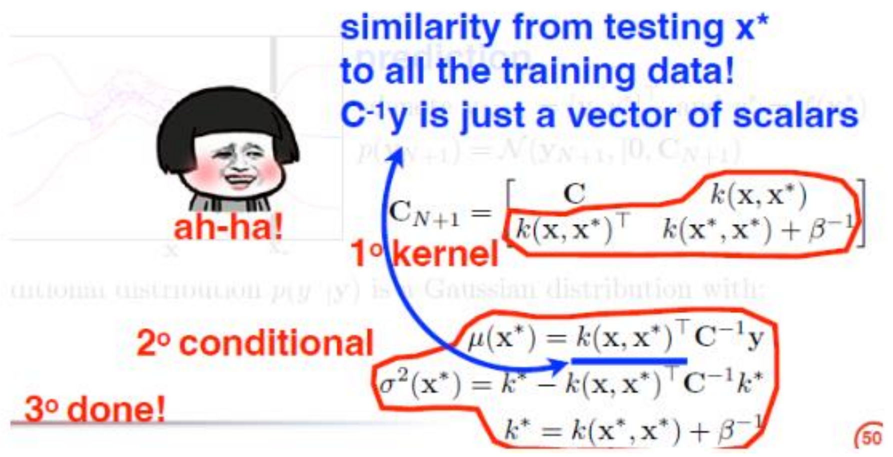


```python
import numpy
from numba import njit

def inverse(A):
    dim = A.shape[0]
    L, U = LUDecomposition(A)
    Y = numpy.eye(dim)
    for i in range(1, dim):
        for j in range(i):
            sum = L[i][j]
            for k in range(j+1, i):
                sum += L[i][k] * Y[k][j]
            Y[i][j] = -sum
    A_inverse = numpy.zeros((dim, dim))
    for i in range(dim-1, -1, -1):
        for j in range(dim):
            sum = 0
            for k in range(i+1, dim):
                sum += U[i][k] * A_inverse[k][j]
            A_inverse[i][j] = (Y[i][j] - sum) / U[i][i]
    return A_inverse

def LUDecomposition(A):
    dim = A.shape[0]
    L = numpy.zeros((dim, dim))
    U = numpy.zeros((dim, dim))
    for i in range(dim):
        for j in range(i):
            sum = 0
            for k in range(j):
                sum += L[i][k] * U[k][j]
            L[i][j] = (A[i][j] - sum) / U[j][j]
        L[i][i] = 1
        for j in range(i, dim):
            sum = 0
            for k in range(i):
                sum += L[i][k] * U[k][j]
            U[i][j] = A[i][j] - sum
    return L, U
```

Then, I define a function for drawing a plot for the visualization

```python
def show(x_line, mean_predict, variance_predict, X, y , alpha=1, l=1):
    plt.figure(figsize=(15,6))
    plt.plot(x_line, mean_predict, 'orange', label='mean')
    plt.fill_between(x_line,
                     mean_predict+2*variance_predict,
                     mean_predict-2*variance_predict,
                     facecolor='lightpink',
                     label='confidence')
    plt.xlim(-60, 60)
    plt.title("alpha={:.2f}, length_scale={:.2f}".format(alpha,l))
    plt.scatter(X, y, c='k', marker='x',  label='data')
    plt.legend(loc='lower left')
    plt.show()
```

Next, we carry out the three steps from the lecture material that the professor mentioned in course.


similarity from testing x* to all the training data! C-1y is just a vector of scalars

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

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

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

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

Step1. Rational quadratic kernel

We look up the equation and use the following formula to define the rational quadratic kernel.

1. Rational quadratic kernal

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

```python
def kernel(X_a, X_b, alpha, l):
    # :param X_a: (n) ndarray
    # :param X_b: (m) ndarray
    # :return: (n,m)  ndarray
    square_error = np.power(X_a.reshape(-1,1) - X_b.reshape(1,-1), 2.0)
    kernel = np.power(1 + square_error/(2 * alpha * l ** 2), -alpha)

    return kernel
```

Step2. Conditional

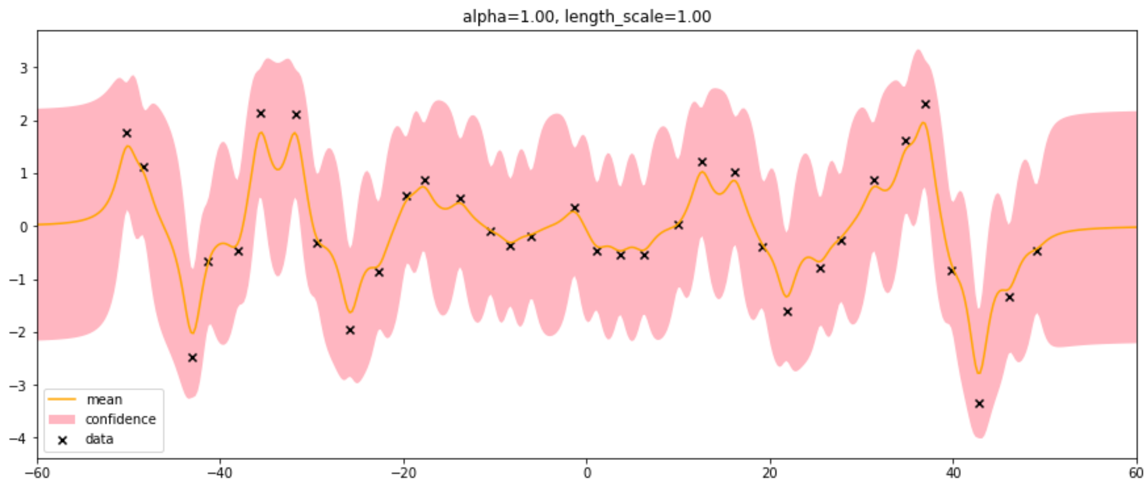
We follow the formula from the course to compute mean and variance.

```python
def predict(x_line, X, y, C, beta, alpha=1, l=1):

    # :param x_line: sampling in linspace(-60,60)
    # :param X: (n) ndarray
    # :param y: (n) ndarray
    # :param C: (n,n) ndarray
    # :param beta:
    # :return: (len(x_line),1) ndarray, (len(x_line),len(x_line)) ndarray

    m = len(x_line)
    k_x_xs = kernel(X, x_line, alpha=1, l=1)
    ks     = kernel(x_line, x_line, alpha=1, l=1) + (1 / beta) * np.identity(m)

    means = k_x_xs.T @ inverse(C) @ y.reshape(-1,1)
    variances = ks - k_x_xs.T @ inverse(C) @ k_x_xs

    return means, variances
```

Step3. Done!

In accordance with the HW5 specification, I set the starting parameter beta=5, alpha=1, length scale=1, x line=[-60,60] and defined display function to plot the predicted outcome.

```python
path = '/content/drive/MyDrive/01_GMBA/3rd_semester(2022)/01-2_PM & 04-1_Machine Learning/HW5_1204/ML_HW05/data/input.data'
X, y = load_data(path)
beta = 5
# kernel
C = kernel(X, X, alpha=1, l=1) + 1 / beta * np.identity(len(X))

# mean and variance in range[-60,60]
x_line = np.linspace(-60, 60, num=500)
mean_predict, variance_predict = predict(x_line, X, y, C, beta, alpha=1, l=1)
mean_predict = mean_predict.reshape(-1)
variance_predict = np.sqrt(np.diag(variance_predict))

# plot
show(x_line, mean_predict, variance_predict, X, y, alpha=1, l=1)
```



Part2:

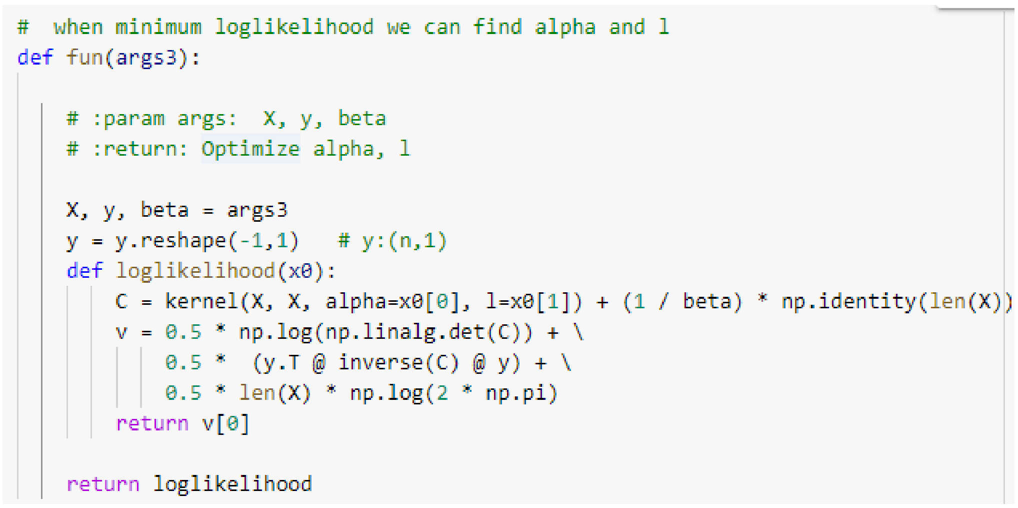
Visualize the outcome after optimizing the kernel parameters by reducing negative marginal log-likelihood. By using the formula below, determine alpha and length scale at minimum log likelihood.

Given $\mathcal{D} = \{(\mathbf{x}_i, y_i)_{n=1}^{N}\} = (\mathbf{X}, \mathbf{y})$, the marginal likelihood is function of $\boldsymbol{\theta}$

$$p(\mathbf{y}|\theta) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{C}_\theta)$$

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

In this part, I build a log likelihood function, then Using scipy optimize minimize, I build a log likelihood function and search for the minimum alpha and length scale.



```python
#  when minimum loglikelihood we can find alpha and l
def fun(args3):

    # :param args:  X, y, beta
    # :return: Optimize alpha, l

    X, y, beta = args3
    y = y.reshape(-1,1)   # y:(n,1)
    def loglikelihood(x0):
        C = kernel(X, X, alpha=x0[0], l=x0[1]) + (1 / beta) * np.identity(len(X))
        v = 0.5 * np.log(np.linalg.det(C)) + \
            0.5 *  (y.T @ inverse(C) @ y) + \
            0.5 * len(X) * np.log(2 * np.pi)
        return v[0]

    return loglikelihood
```

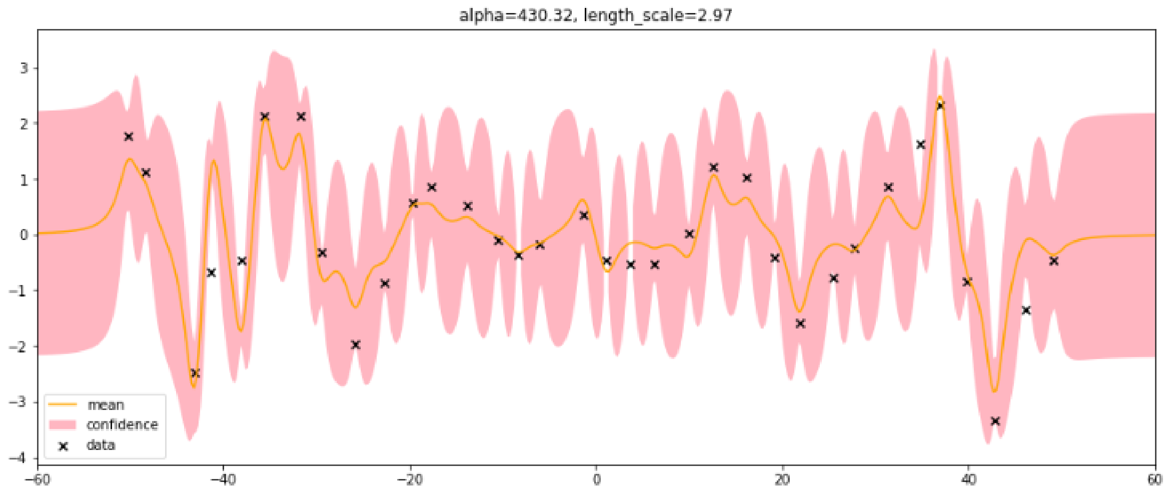
In order to determine the minimal value, I set the initial value of alpha and length scale to [0.01, 0.1, 0, 10, 100] and the bound of alpha and length scale to [10^-5, 10^5].

The variance of each point decreases when the results are compared with alpha=1, length scale=1.

I got the optimal result (alpha, length scale) = (430, 2.965).

```python
path = '/content/drive/MyDrive/01_GMBA/3rd_semester(2022)/01-2_PM & 04-1_Machine Learning/HW5_1204/ML_HW05/data/input.data'
X, y = load_data(path)
beta = 5

args = (X, y, beta)
objective_value = 1e9
inits = [0.01, 0.1, 0, 10, 100]
for init_alpha in inits:
    for init_length_scale in inits:
        res = minimize(fun = fun(args),
                       x0 = np.asarray([init_alpha, init_length_scale]),
                       bounds=((1e-5,1e5),(1e-5,1e5)))

        if res.fun < objective_value:
            objective_value = res.fun
            alpha_optimize,length_scale_optimize = res.x
print('alpha: ', alpha_optimize)
print('length_scale: ', length_scale_optimize)

# kernel
C = kernel(X, X, alpha=alpha_optimize, l=length_scale_optimize) + 1 / beta * np.identity(len(X))

# mean and variance in range[-60,60]
x_line = np.linspace(-60, 60, num=500)
mean_predict, variance_predict = predict(x_line, X, y, C, beta,
                                         alpha=alpha_optimize,
                                         l=length_scale_optimize)
mean_predict = mean_predict.reshape(-1)
variance_predict = np.sqrt(np.diag(variance_predict))

# plot
show(x_line, mean_predict, variance_predict, X, y,  alpha=alpha_optimize, l=length_scale_optimize)
```

```
alpha:  430.3221548899712
length_scale:  2.965286588272608
<ipython-input-19-7a477c8b162e>:29: RuntimeWarning: invalid value encountered in sqrt
  variance_predict = np.sqrt(np.diag(variance_predict))
```

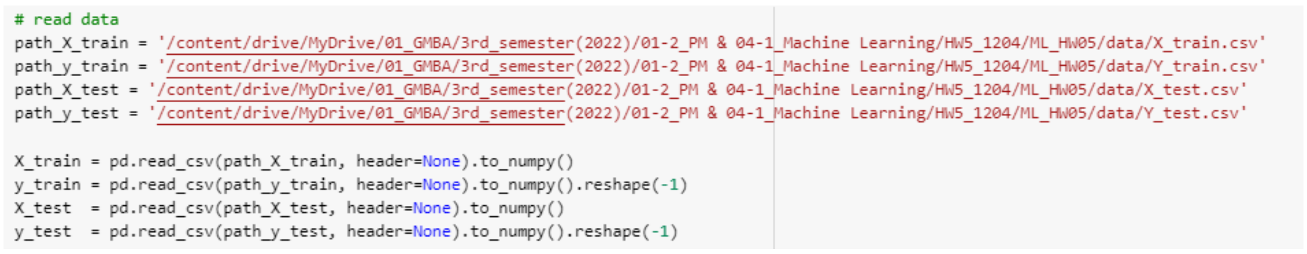


Observation

1. Result 2 is unquestionably superior to Result 1.
2. The results show that the gaussian process has a better level of confidence in its predictions when using training data points. Gaussian processes may nevertheless produce accurate guesses in the absence of training data. However, the Gaussian process can scarcely generate any predictions in an interval with no training data. Our 95% confidence region is, therefore, substantially more significant than other parts as a result.
3. The variation in the rational quadratic function, while there, is unimportant. This parameter, which is simply a scaling factor, is present at the front of every kernel. Variance is the function's deviation from its mean.
4. If the parameters are incorrect, it could produce pretty poor results.

2. SVM

First thing first, we need to load data by using pandas for csv files



```
# read data
path_X_train = '/content/drive/MyDrive/01_GMBA/3rd_semester(2022)/01-2_PM & 04-1_Machine Learning/HW5_1204/ML_HW05/data/X_train.csv'
path_y_train = '/content/drive/MyDrive/01_GMBA/3rd_semester(2022)/01-2_PM & 04-1_Machine Learning/HW5_1204/ML_HW05/data/Y_train.csv'
path_X_test = '/content/drive/MyDrive/01_GMBA/3rd_semester(2022)/01-2_PM & 04-1_Machine Learning/HW5_1204/ML_HW05/data/X_test.csv'
path_y_test = '/content/drive/MyDrive/01_GMBA/3rd_semester(2022)/01-2_PM & 04-1_Machine Learning/HW5_1204/ML_HW05/data/Y_test.csv'

X_train = pd.read_csv(path_X_train, header=None).to_numpy()
y_train = pd.read_csv(path_y_train, header=None).to_numpy().reshape(-1)
X_test  = pd.read_csv(path_X_test, header=None).to_numpy()
y_test  = pd.read_csv(path_y_test, header=None).to_numpy().reshape(-1)
```

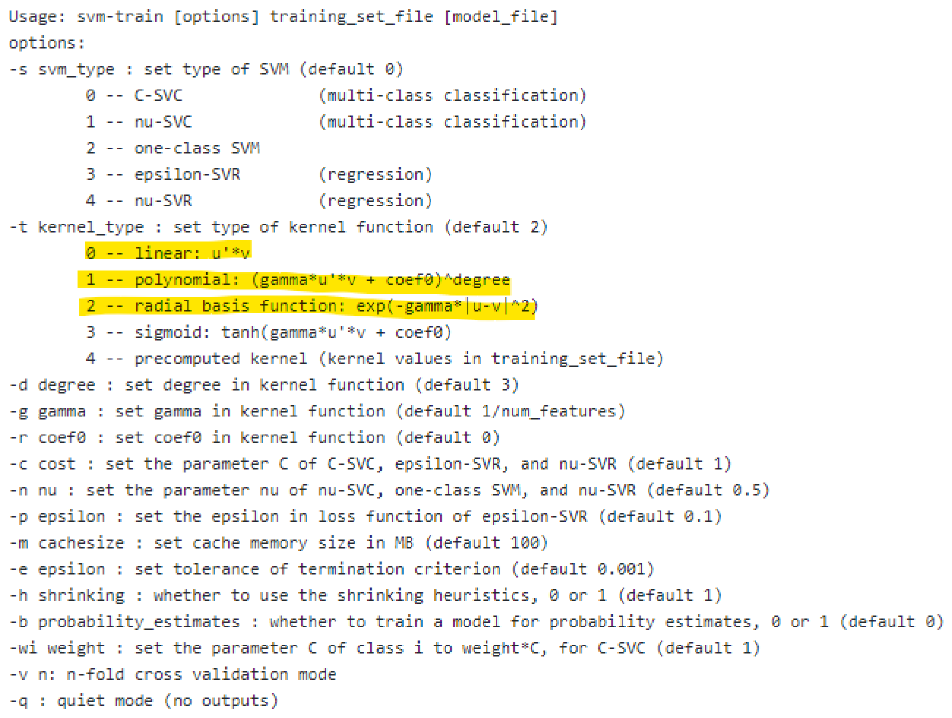
Part1:

Utilize several kernel functions (polynomial, linear, and RBF kernels) and evaluate their performance in comparison.

Use the libsvm package, which has the same crucial parameter.

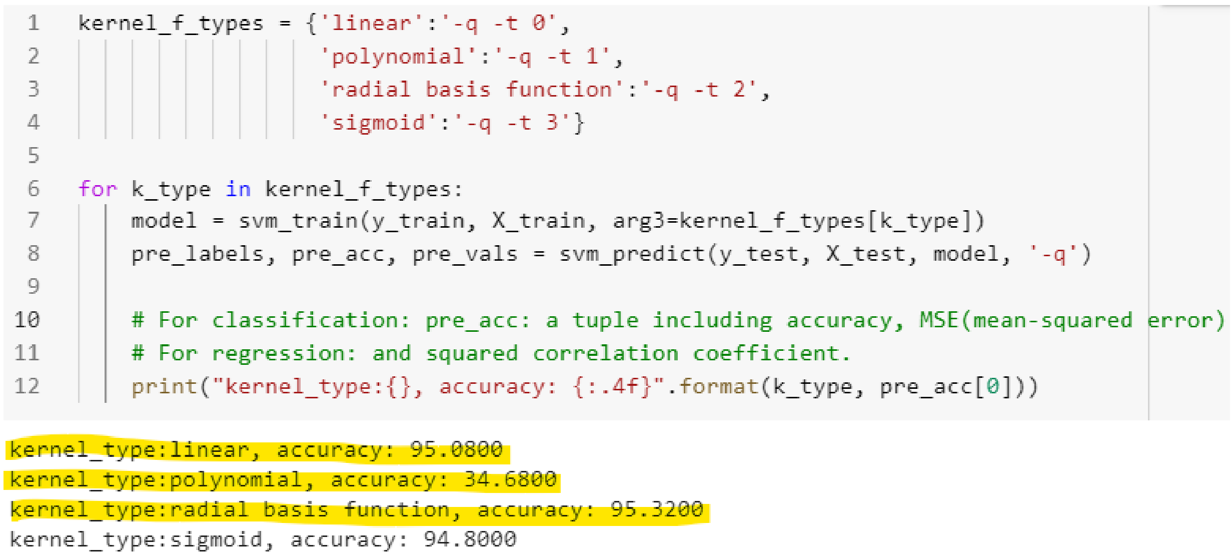
reference from https://github.com/cjlin1/libsvm

- -t means the different kernel type,
- -q means will not return calculation process.



```
Usage: svm-train [options] training_set_file [model_file]
options:
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC              (multi-class classification)
        1 -- nu-SVC             (multi-class classification)
        2 -- one-class SVM
        3 -- epsilon-SVR        (regression)
        4 -- nu-SVR             (regression)
-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)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
        4 -- precomputed kernel (kernel values in training_set_file)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking : whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates : whether to train a model for probability estimates, 0 or 1 (default 0)
-wi weight : set the parameter C of class i to weight*C, for C-SVC (default 1)
-v n: n-fold cross validation mode
-q : quiet mode (no outputs)
```

I use the default parameter and a different kernel type to predict X test and compare the results with the real data (ground truth).



```python
kernel_f_types = {'linear':'-q -t 0',
                  'polynomial':'-q -t 1',
                  'radial basis function':'-q -t 2',
                  'sigmoid':'-q -t 3'}

for k_type in kernel_f_types:
    model = svm_train(y_train, X_train, arg3=kernel_f_types[k_type])
    pre_labels, pre_acc, pre_vals = svm_predict(y_test, X_test, model, '-q')

    # For classification: pre_acc: a tuple including accuracy, MSE(mean-squared error)
    # For regression: and squared correlation coefficient.
    print("kernel_type:{}, accuracy: {:.4f}".format(k_type, pre_acc[0]))
```

```
kernel_type:linear, accuracy: 95.0800
kernel_type:polynomial, accuracy: 34.6800
kernel_type:radial basis function, accuracy: 95.3200
kernel_type:sigmoid, accuracy: 94.8000
```

Kernel types	Accuracy
Linear	95.08%
Polynomial	34.68%
Radial basis function	95.32%

Part2:

HW5:

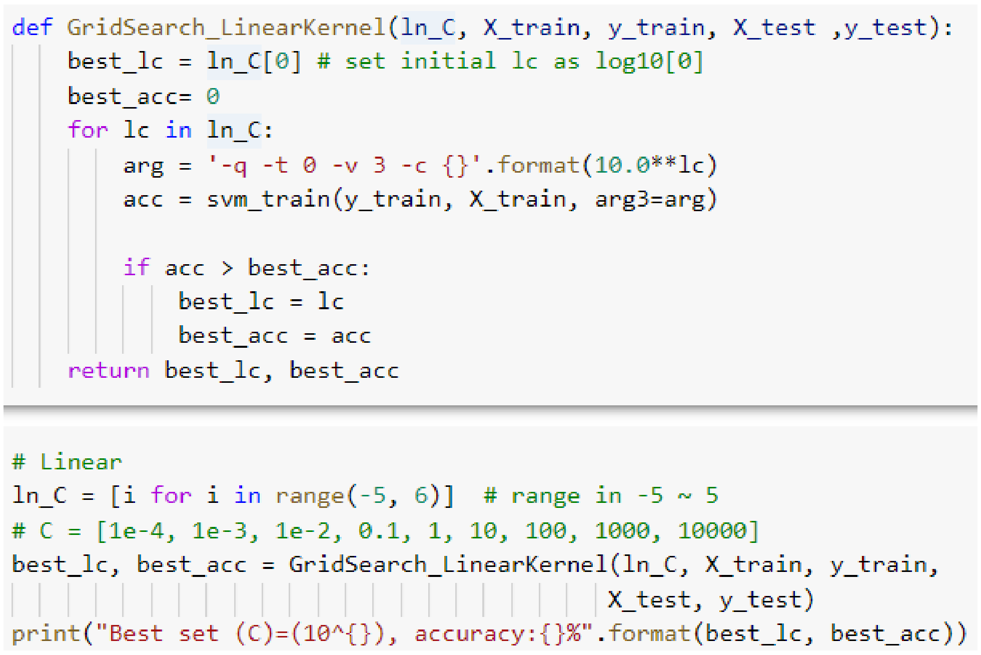
- Please use C-SVC (you can choose by setting parameters in the function input, C-SVC is soft-margin SVM).
- Since there are some parameters you need to tune for, please do the **grid search** for finding parameters of the best performing model.
- For instance, in C-SVC you have a parameter C, and if you use RBF kernel you have another parameter γ, you can search for a set of (C, γ) which gives you best performance in cross-validation.

Linear Kernel

I created the **GridSearch_LinearKernel** function to find the optimal result between accuracy and C.

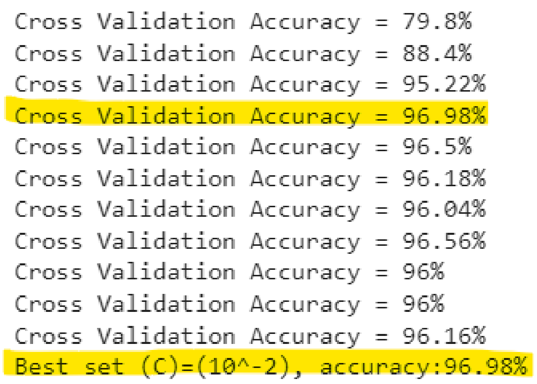
To calculate the average accuracy, the cross validation = 3.

The best set is $C=10^{-2}$, accuracy=96.16%.is obtained with the C= 10^-5~10^ 5.



```
def GridSearch_LinearKernel(ln_C, X_train, y_train, X_test ,y_test):
    best_lc = ln_C[0] # set initial lc as log10[0]
    best_acc= 0
    for lc in ln_C:
        arg = '-q -t 0 -v 3 -c {}'.format(10.0**lc)
        acc = svm_train(y_train, X_train, arg3=arg)

        if acc > best_acc:
            best_lc = lc
            best_acc = acc
    return best_lc, best_acc
```

```
# Linear
ln_C = [i for i in range(-5, 6)]  # range in -5 ~ 5
# C = [1e-4, 1e-3, 1e-2, 0.1, 1, 10, 100, 1000, 10000]
best_lc, best_acc = GridSearch_LinearKernel(ln_C, X_train, y_train,
                                            X_test, y_test)
print("Best set (C)=(10^{}), accuracy:{}%".format(best_lc, best_acc))
```



```
Cross Validation Accuracy = 79.8%
Cross Validation Accuracy = 88.4%
Cross Validation Accuracy = 95.22%
Cross Validation Accuracy = 96.98%
Cross Validation Accuracy = 96.5%
Cross Validation Accuracy = 96.18%
Cross Validation Accuracy = 96.04%
Cross Validation Accuracy = 96.56%
Cross Validation Accuracy = 96%
Cross Validation Accuracy = 96%
Cross Validation Accuracy = 96.16%
Best set (C)=(10^-2), accuracy:96.98%
```

ln_C	Accuracy
10^-2	**96.16%**

PolyKernel

I created the **GridSearch_PolyKernel** function to find the optimal result between **(C, γ, coed0) and accuracy**.

To calculate the average accuracy, the cross validation = 3.

The C = ln_c= 10^-3~10^ 3, gamma = ln_g =10^-3~10^ 3, coef0 = coef_in = [-1,0,1].

We can get the best set (C, γ, coef0) = (10^1, 10^{-1}, 1), accuracy=97.96%.

```python
def GridSearch_PolyKernel(ln_c, ln_g, coef_in, X_train, y_train, X_test, y_test):
    best_lc     = ln_c[0]
    best_lg     = ln_g[0]
    best_coef_in = coef_in[0]
    best_acc    = 0
    for lc in ln_c:
        for lg in ln_g:
            for r in coef_in:
                arg = '-q -t 1 -v 3 -c {} -g {} -r {}'.format(10.0**lc, 10.0**lg, r)
                acc = svm_train(y_train, X_train, arg3=arg)

                if acc > best_acc:
                    best_lc     = lc
                    best_lg     = lg
                    best_coef0 = r
                    best_acc    = acc
    return best_lc, best_lg, best_coef_in, best_acc
```

```python
# Polynomial
ln_c = [i for i in range(-3,4)] #-3~3
ln_g = [i for i in range(-3,4)]
coef_in  = [-1, 0, 1]
best_lc, best_lg, best_coef_in, best_acc = GridSearch_PolyKernel(ln_c, ln_g, coef_in, X_train, y_train, X_test, y_test)
print("Best set (C, gamma, coef0)=(10^{}, 10^{}, {}), accuracy:{}%".format(best_lc, best_lg, best_coef_in, best_acc))
```

```
Cross Validation Accuracy = 95.16%
Cross Validation Accuracy = 97.42%
Cross Validation Accuracy = 97.96%
Cross Validation Accuracy = 97.3%
Cross Validation Accuracy = 97.22%
Cross Validation Accuracy = 97.44%
Cross Validation Accuracy = 97.6%
Cross Validation Accuracy = 97.52%
Cross Validation Accuracy = 97.48%
Cross Validation Accuracy = 97.36%
Cross Validation Accuracy = 97.62%
Cross Validation Accuracy = 97.46%
Cross Validation Accuracy = 97.5%
Cross Validation Accuracy = 97.64%
Cross Validation Accuracy = 97.12%
Cross Validation Accuracy = 95.36%
Cross Validation Accuracy = 89.26%
Cross Validation Accuracy = 97.26%
Cross Validation Accuracy = 77.66%
Cross Validation Accuracy = 97.56%
Cross Validation Accuracy = 97.64%
Cross Validation Accuracy = 95%
Cross Validation Accuracy = 97.56%
Cross Validation Accuracy = 97.78%
Cross Validation Accuracy = 97.44%
Cross Validation Accuracy = 97.52%
Cross Validation Accuracy = 97.8%
Cross Validation Accuracy = 97.28%
Cross Validation Accuracy = 97.42%
Cross Validation Accuracy = 97.36%
Cross Validation Accuracy = 97.42%
Cross Validation Accuracy = 97.28%
Cross Validation Accuracy = 97.36%
Cross Validation Accuracy = 97.38%
Cross Validation Accuracy = 97.56%
Cross Validation Accuracy = 97.44%
Cross Validation Accuracy = 94.16%
Cross Validation Accuracy = 96.36%
Cross Validation Accuracy = 96.36%
Cross Validation Accuracy = 76.6%
Cross Validation Accuracy = 97.46%
Cross Validation Accuracy = 97.78%
Cross Validation Accuracy = 95.12%
Cross Validation Accuracy = 97.42%
Cross Validation Accuracy = 97.72%
Cross Validation Accuracy = 97.16%
Cross Validation Accuracy = 97.38%
Cross Validation Accuracy = 97.5%
Cross Validation Accuracy = 97.24%
Cross Validation Accuracy = 97.54%
Cross Validation Accuracy = 97.6%
Cross Validation Accuracy = 97.54%
Cross Validation Accuracy = 97.38%
Cross Validation Accuracy = 97.64%
Cross Validation Accuracy = 97.5%
Cross Validation Accuracy = 97.18%
Cross Validation Accuracy = 97.38%
Best set (C, gamma, coef0)=(10^1, 10^-1, 1), accuracy:97.96000000000001%
```

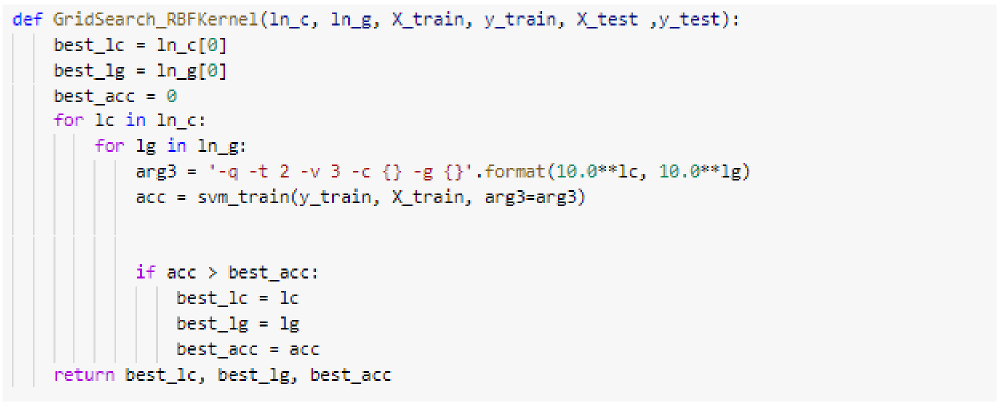
RBF(radial basis function) Kernel

I created the **GridSearch_RBFKernel** function to find the optimal result between **(C, γ) and accuracy**.

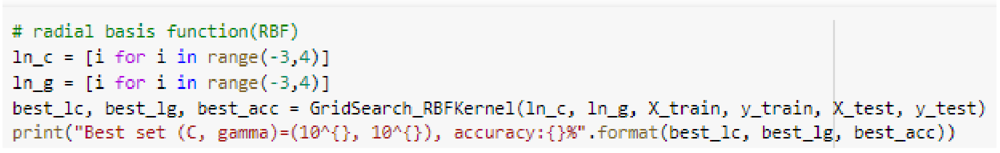
To calculate the average accuracy, the cross validation = 3.

The C = ln_c= 10^-3~10^ 3, gamma = ln_g =10^-3~10^ 3.

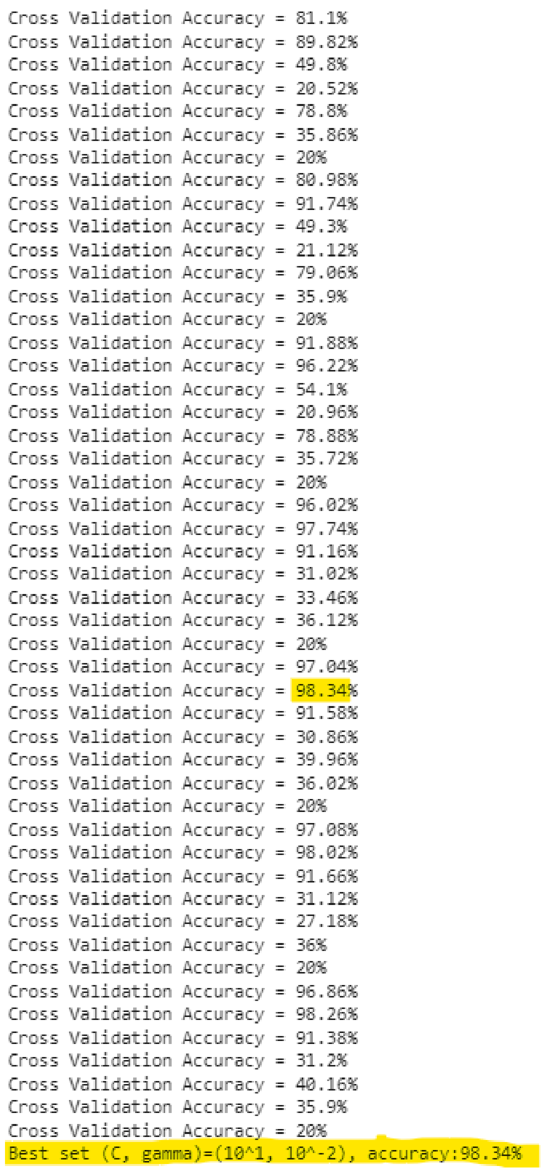
We can get the best set (C, γ) = (10^1, 10^{-2}), accuracy=98.34%.



```
def GridSearch_RBFKernel(ln_c, ln_g, X_train, y_train, X_test ,y_test):
    best_lc = ln_c[0]
    best_lg = ln_g[0]
    best_acc = 0
    for lc in ln_c:
        for lg in ln_g:
            arg3 = '-q -t 2 -v 3 -c {} -g {}'.format(10.0**lc, 10.0**lg)
            acc = svm_train(y_train, X_train, arg3=arg3)

            if acc > best_acc:
                best_lc = lc
                best_lg = lg
                best_acc = acc
    return best_lc, best_lg, best_acc
```



```
# radial basis function(RBF)
ln_c = [i for i in range(-3,4)]
ln_g = [i for i in range(-3,4)]
best_lc, best_lg, best_acc = GridSearch_RBFKernel(ln_c, ln_g, X_train, y_train, X_test, y_test)
print("Best set (C, gamma)=(10^{}, 10^{}), accuracy:{}%".format(best_lc, best_lg, best_acc))
```



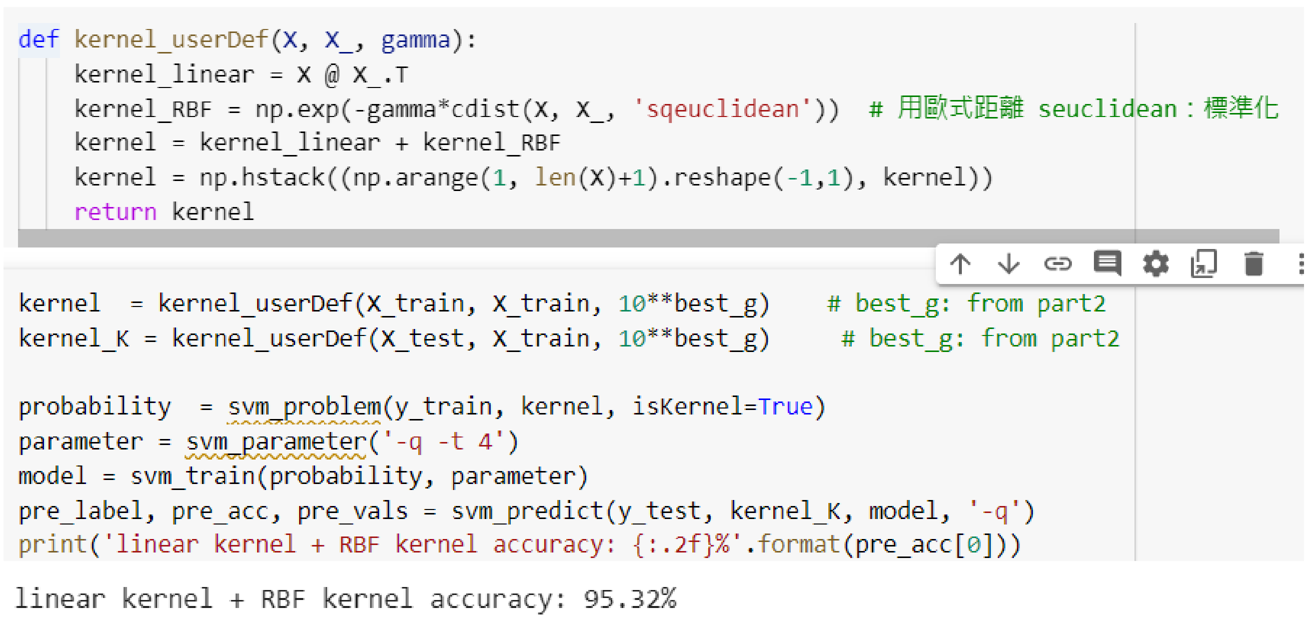
```
Cross Validation Accuracy = 81.1%
Cross Validation Accuracy = 89.82%
Cross Validation Accuracy = 49.8%
Cross Validation Accuracy = 20.52%
Cross Validation Accuracy = 78.8%
Cross Validation Accuracy = 35.86%
Cross Validation Accuracy = 20%
Cross Validation Accuracy = 80.98%
Cross Validation Accuracy = 91.74%
Cross Validation Accuracy = 49.3%
Cross Validation Accuracy = 21.12%
Cross Validation Accuracy = 79.06%
Cross Validation Accuracy = 35.9%
Cross Validation Accuracy = 20%
Cross Validation Accuracy = 91.88%
Cross Validation Accuracy = 96.22%
Cross Validation Accuracy = 54.1%
Cross Validation Accuracy = 20.96%
Cross Validation Accuracy = 78.88%
Cross Validation Accuracy = 35.72%
Cross Validation Accuracy = 20%
Cross Validation Accuracy = 96.02%
Cross Validation Accuracy = 97.74%
Cross Validation Accuracy = 91.16%
Cross Validation Accuracy = 31.02%
Cross Validation Accuracy = 33.46%
Cross Validation Accuracy = 36.12%
Cross Validation Accuracy = 20%
Cross Validation Accuracy = 97.04%
Cross Validation Accuracy = 98.34%
Cross Validation Accuracy = 91.58%
Cross Validation Accuracy = 30.86%
Cross Validation Accuracy = 39.96%
Cross Validation Accuracy = 36.02%
Cross Validation Accuracy = 20%
Cross Validation Accuracy = 97.08%
Cross Validation Accuracy = 98.02%
Cross Validation Accuracy = 91.66%
Cross Validation Accuracy = 31.12%
Cross Validation Accuracy = 27.18%
Cross Validation Accuracy = 36%
Cross Validation Accuracy = 20%
Cross Validation Accuracy = 96.86%
Cross Validation Accuracy = 98.26%
Cross Validation Accuracy = 91.38%
Cross Validation Accuracy = 31.2%
Cross Validation Accuracy = 40.16%
Cross Validation Accuracy = 35.9%
Cross Validation Accuracy = 20%
Best set (C, gamma)=(10^1, 10^-2), accuracy:98.34%
```

Part3:

HW5: Use linear kernel + RBF kernel together (therefore a new kernel function) and use grid search again. You would need to find out how to use a user-defined kernel in libsvm.

I created the **kernel_userDef** function and use svm_problem to find precomputed kernels.

In this way, we can get the result from linear kernel + RBF kernel, the accuracy=95.32%



```python
def kernel_userDef(X, X_, gamma):
    kernel_linear = X @ X_.T
    kernel_RBF = np.exp(-gamma*cdist(X, X_, 'sqeuclidean'))  # 用歐式距離 seuclidean：標準化
    kernel = kernel_linear + kernel_RBF
    kernel = np.hstack((np.arange(1, len(X)+1).reshape(-1,1), kernel))
    return kernel
```

```python
kernel  = kernel_userDef(X_train, X_train, 10**best_g)    # best_g: from part2
kernel_K = kernel_userDef(X_test, X_train, 10**best_g)     # best_g: from part2

probability  = svm_problem(y_train, kernel, isKernel=True)
parameter = svm_parameter('-q -t 4')
model = svm_train(probability, parameter)
pre_label, pre_acc, pre_vals = svm_predict(y_test, kernel_K, model, '-q')
print('linear kernel + RBF kernel accuracy: {:.2f}%'.format(pre_acc[0]))
```

```
linear kernel + RBF kernel accuracy: 95.32%
```

Observation 1

1. The penalty increases, the number of support vectors decreases, and overfitting becomes easier as C increases.
2. When The gamma is enormous, making it is simple to identify the hyperplane that best fits the near point and simple to overfit the model.
3. RBF outperforms the approaches in terms of performance. Due to RBF's ability to map data into spaces with unlimited dimensions, everything is linearly separable. Despite having a 784 feature space, MNIST appears complicated. SVM can make accurate classifications thanks to RBF. But this accuracy isn't perfect. This is because we only conduct a small number of parameter searches.
4. The outcome confirms that RBF is an excellent classification kernel. For this reason, most people choose RBF as their kernel function.
5. It is conceivable that part B will yield more significant results than part A. B is performing the identical SVM as A but with the best parameters that grid search could find.
6. Experiments A and B demonstrate the significance of selecting appropriate parameters for various datasets.
7. Cross-validation results may not be as good as testing results. Considering that the optimum parameters for testing data may differ from those found with training data. However, the accuracy of slice drop is tolerable.
8. Linear kernel: No particular parameters require fine-tuning.
9. Polynomial kernel: The most time-consuming search method since many parameters need to be fine-tuned.
10. RBF kernel: a popular kernel function because of its superior categorization capabilities.
11. The two most crucial parameters to fine-tune are C and gamma. The price is C. The model has less tolerance for mistakes with a higher c value. This leads to training data that is overly fitted. Gamma affects the area that RBF can project to. Gamma can support more vectors if it is smaller.