Machine Learning Homework 5

Gaussian Process & SVM

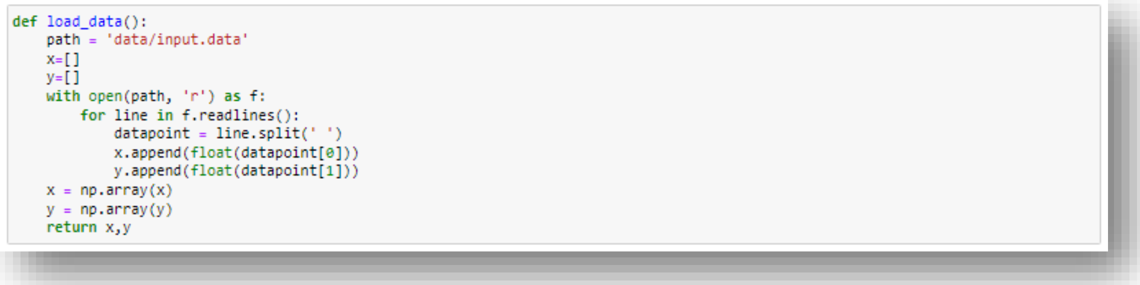
數據所碩一葉詠富 310554031

Github: https://github.com/frankye1000/NYCU-MachineLearning/tree/master/HW5

1. Gaussian Process

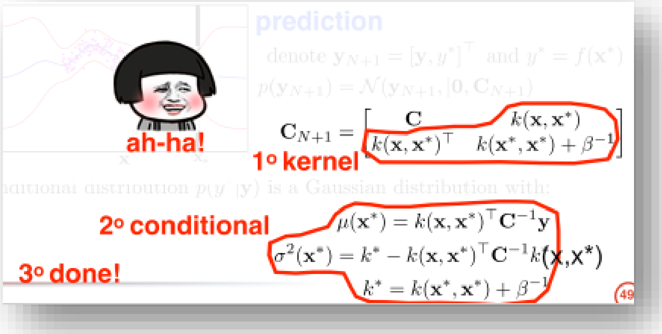
Part1: Apply Gaussian Process Regression to predict the distribution of f and visualize the result.

First thing is to load data, I define a load data function.



```
def load_data():
    path = 'data/input.data'
    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
```

Use the **three steps** mentioned by the teacher in class.



prediction

denote $\mathbf{y}_{N+1} = [\mathbf{y}, y^*]^\top$ and $y^* = f(\mathbf{x}^*)$

$p(\mathbf{y}_{N+1}) = \mathcal{N}(\mathbf{y}_{N+1}, |0, \mathbf{C}_{N+1})$

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

1º kernel

ditional distribution $p(y^*|\mathbf{y})$ is a Gaussian distribution with:

2º conditional

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

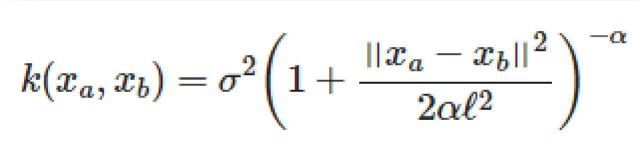
$$\sigma^2(\mathbf{x}^*) = k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1} k(x, x^*)$$

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

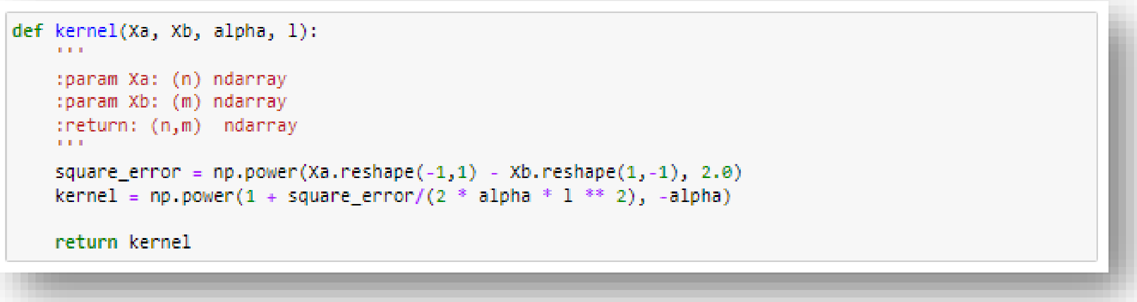
3º done!

Step1. Rational quadratic kernel

Define the rational quadratic kernel by below formula.



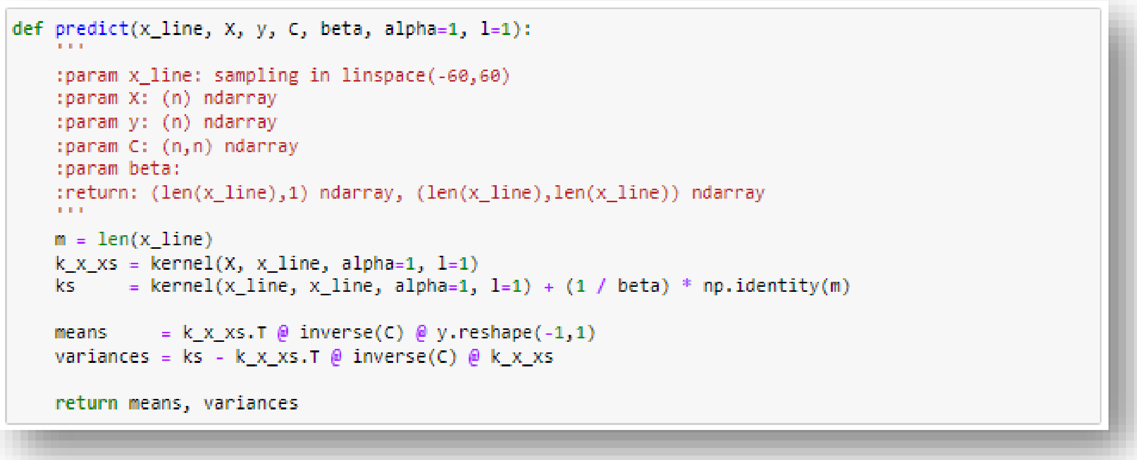
$$k(x_a, x_b) = \sigma^2 \left(1 + \frac{||x_a - x_b||^2}{2\alpha\ell^2}\right)^{-\alpha}$$



```
def kernel(Xa, Xb, alpha, l):
    '''
    :param Xa: (n) ndarray
    :param Xb: (m) ndarray
    :return: (n,m)  ndarray
    '''
    square_error = np.power(Xa.reshape(-1,1) - Xb.reshape(1,-1), 2.0)
    kernel = np.power(1 + square_error/(2 * alpha * l ** 2), -alpha)

    return kernel
```

Step2. Conditional

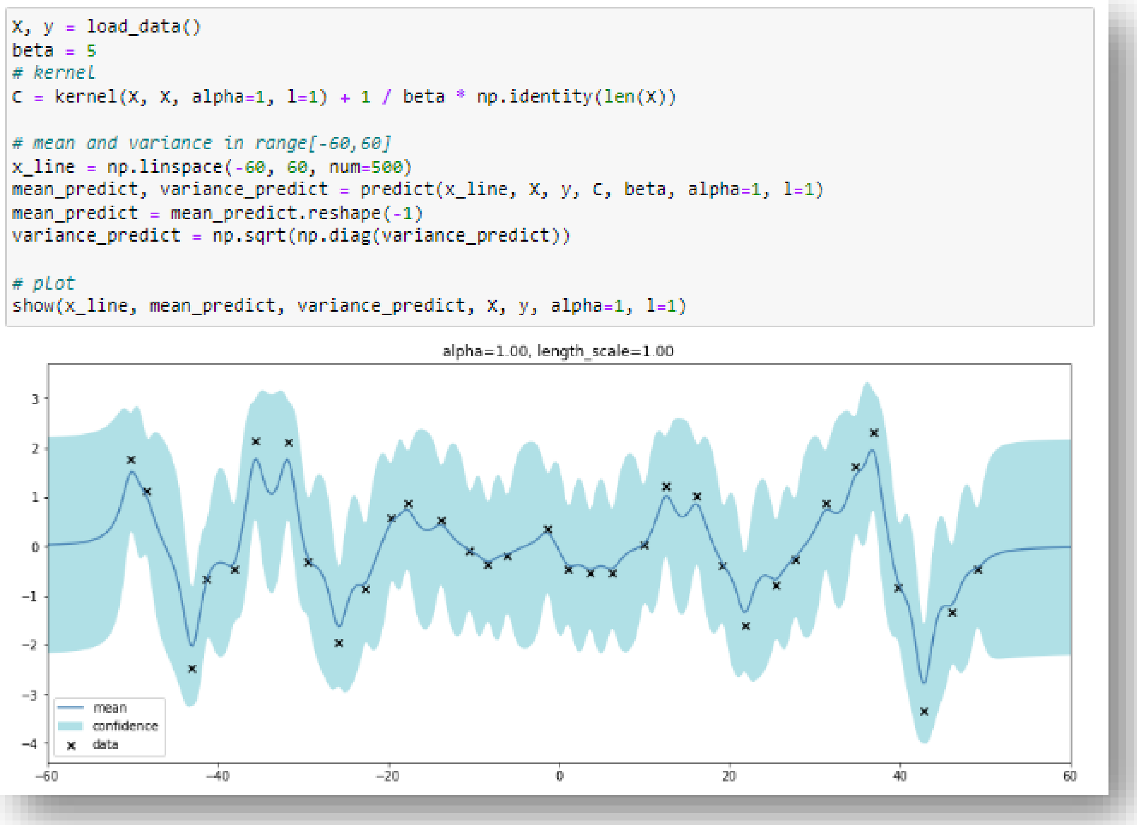
Use teacher formula to calculate mean & variance.



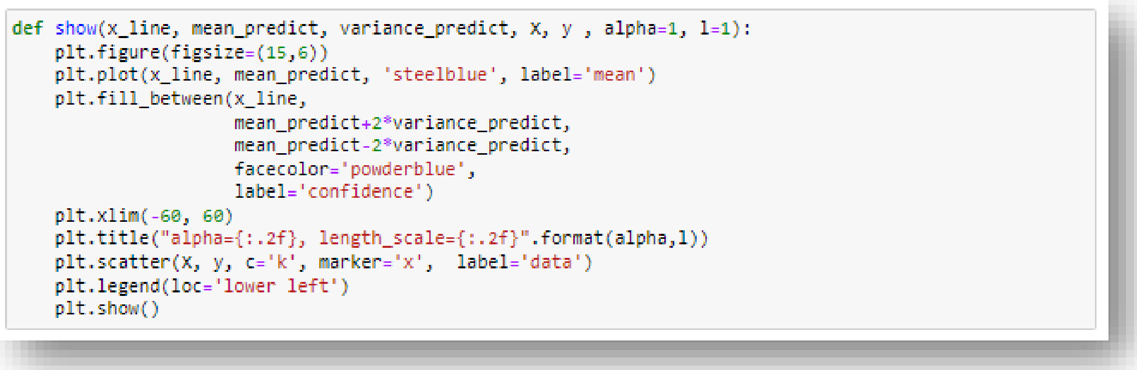
```
def predict(x_line, X, y, C, beta, alpha=1, l=1):
    '''
    :param x_line: sampling in linspace(-60,60)
    :param X: (n) ndarray
    :param y: (n) ndarray
    :param C: (n,n) ndarray
    :param beta:
    :return: (len(x_line),1) ndarray, (len(x_line),len(x_line)) ndarray
    '''
    m = len(x_line)
    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!

I set initial parameter beta=5, alpha=1, length scale=1, x_line=[-60,60], and define show function to plot the predict result.



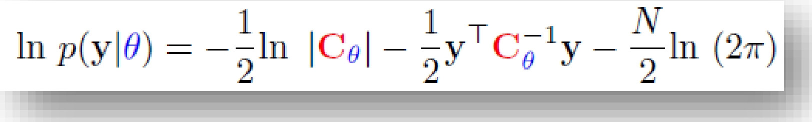
```
X, y = load_data()
beta = 5
# kernel
C = kernel(X, X, alpha=1, l=1) + 1 / beta * np.identity(len(X))

# 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)
```



```
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, 'steelblue', label='mean')
    plt.fill_between(x_line,
                     mean_predict+2*variance_predict,
                     mean_predict-2*variance_predict,
                     facecolor='powderblue',
                     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()
```

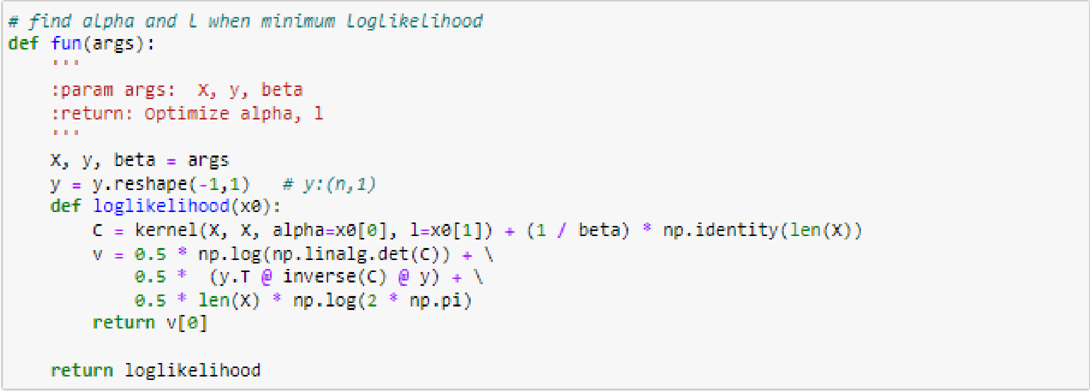
Part2: Optimize the kernel parameters by minimizing negative marginal log-likelihood, and visualize the result again.

Find alpha & length scale when minimum log likelihood by below formula.



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

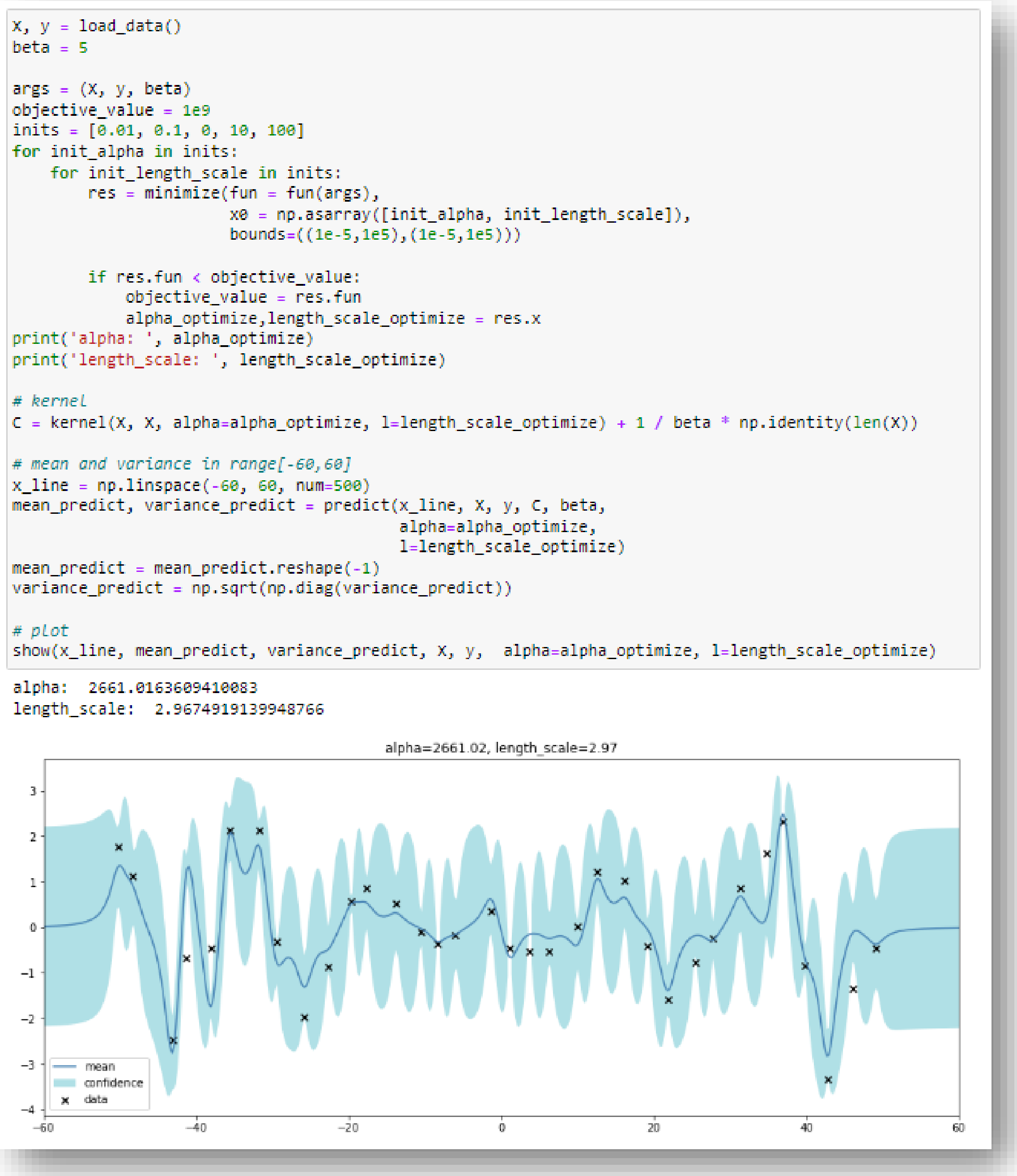
I define a log likelihood function, and use scipy.optimize.minimize to find minimum alpha & length scale.



```
# find alpha and l when minimum loglikelihood
def fun(args):
    '''
    :param args:  X, y, beta
    :return: optimize alpha, l
    '''
    X, y, beta = args
    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
```

I set the initial value alpha & length scale = [0.01, 0.1, 0, 10, 100] and set the bound of alpha & length scale [10^-5, 10^5] to find the minimum value.

If the result is compared with alpha=1, length scale=1, the variance of each point becomes smaller.



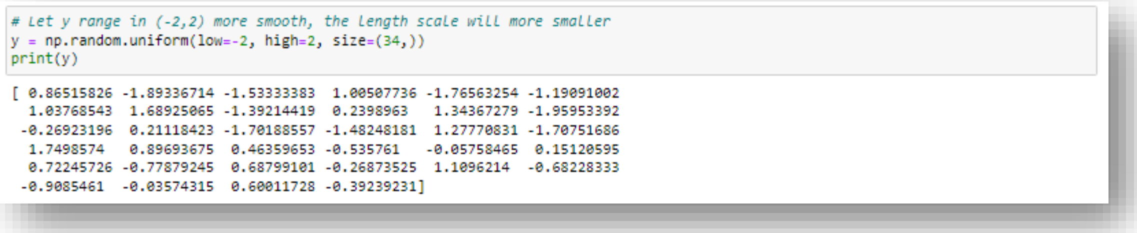
```python
X, y = load_data()
beta = 5

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:  2661.0163609410083
length_scale:  2.9674919139948766
```

Observation

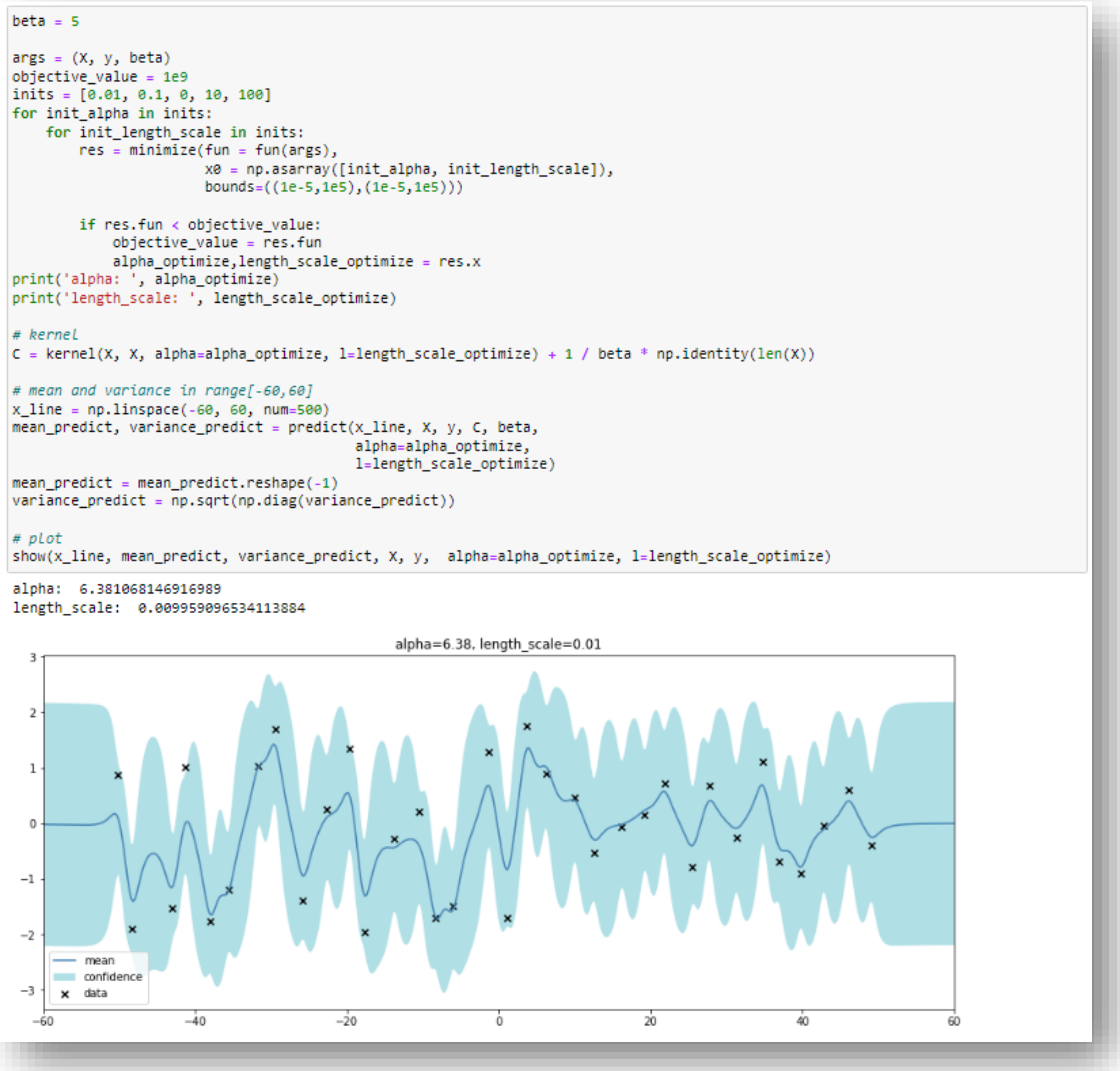
As you increase the alpha & length scale, the learn functions keep getting smoother.

I do a test to make the value of y range (-2,2), which is more smoother. Then the best alpha & length scale become smaller.



```
# Let y range in (-2,2) more smooth, the length scale will more smaller
y = np.random.uniform(low=-2, high=2, size=(34,))
print(y)
```

```
[ 0.86515826 -1.89336714 -1.53333383  1.00507736 -1.76563254 -1.19091002
  1.03768543  1.68925065 -1.39214419  0.2398963   1.34367279 -1.95953392
 -0.26923196  0.21118423 -1.70188557 -1.48248181  1.27770831 -1.70751686
  1.7498574   0.89693675  0.46359653 -0.535761   -0.05758465  0.15120595
  0.72245726 -0.77879245  0.68799101 -0.26873525  1.1096214  -0.68228333
 -0.9085461  -0.03574315  0.60011728 -0.39239231]
```

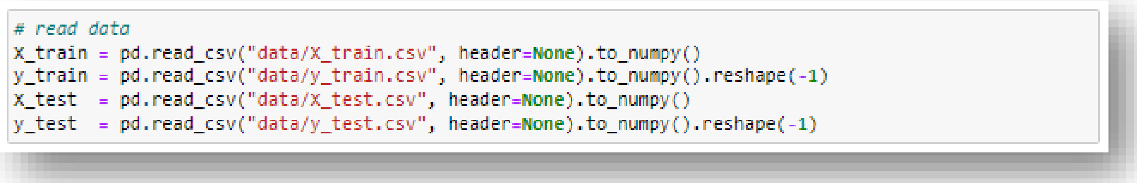


```
beta = 5

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:  6.381068146916989
length_scale:  0.009959096534113884
```

2. SVM

First thing is to load data. I use pandas to load csv data.



```
# read data
X_train = pd.read_csv("data/X_train.csv", header=None).to_numpy()
y_train = pd.read_csv("data/y_train.csv", header=None).to_numpy().reshape(-1)
X_test  = pd.read_csv("data/X_test.csv", header=None).to_numpy()
y_test  = pd.read_csv("data/y_test.csv", header=None).to_numpy().reshape(-1)
```

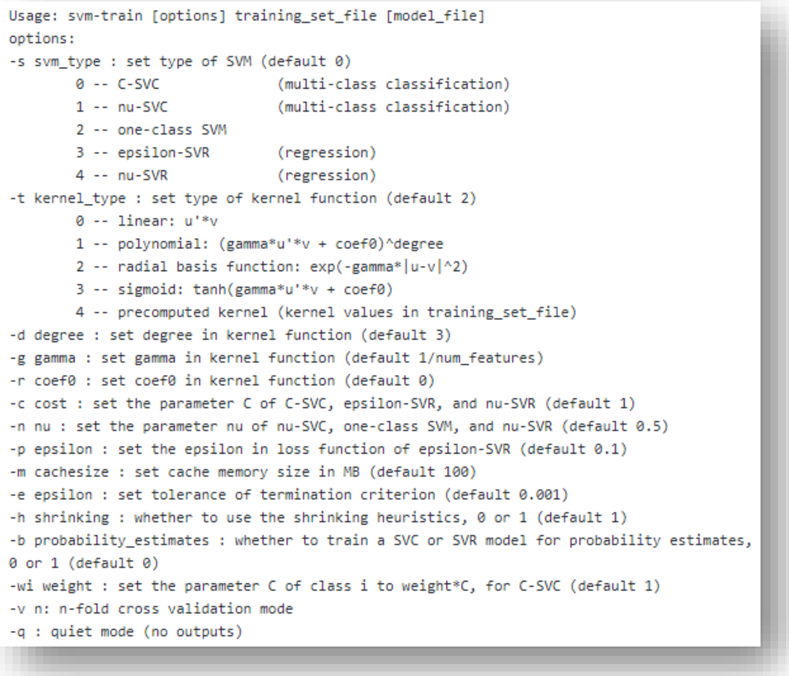
Part1: Use different kernel functions (linear, polynomial, and RBF kernels) and have comparison between their performance.

Use the package libsvm, have same important parameter.

-t means the different kernel type,

-q means will not return calculation process.

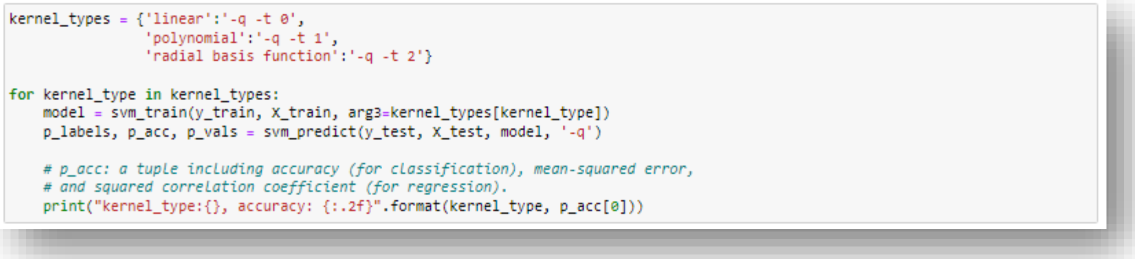
reference from https://github.com/cjlin1/libsvm



```
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 SVC or SVR 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 change kernel type and predict X_test to compare with groundtruth.

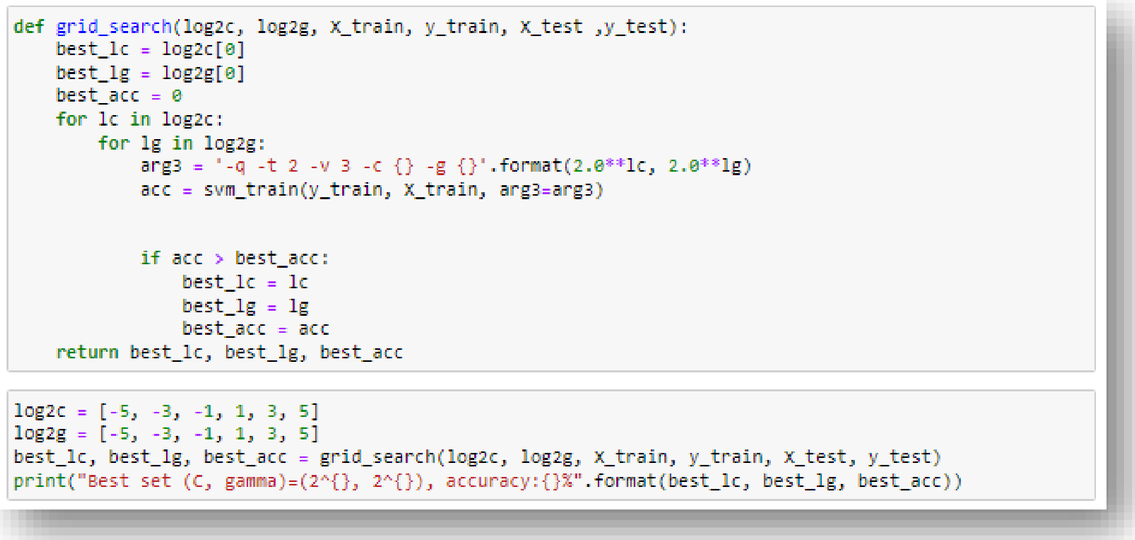
Kernel type	accuracy
Linear	95.08%
Polynomial	34.68%
Radial basis function	95.32%



```
kernel_types = {'linear':'-q -t 0',
                'polynomial':'-q -t 1',
                'radial basis function':'-q -t 2'}

for kernel_type in kernel_types:
    model = svm_train(y_train, x_train, arg3=kernel_types[kernel_type])
    p_labels, p_acc, p_vals = svm_predict(y_test, x_test, model, '-q')

    # p_acc: a tuple including accuracy (for classification), mean-squared error,
    # and squared correlation coefficient (for regression).
    print("kernel_type:{}, accuracy: {:.2f}".format(kernel_type, p_acc[0]))
```

Part2: Please use C-SVC. 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.

I define a grid_search function to search the best combination of (C, γ) and accuracy.
The cross validation = 3 to get the average accuracy.



```
def grid_search(log2c, log2g, x_train, y_train, x_test ,y_test):
    best_lc = log2c[0]
    best_lg = log2g[0]
    best_acc = 0
    for lc in log2c:
        for lg in log2g:
            arg3 = '-q -t 2 -v 3 -c {} -g {}'.format(2.0**lc, 2.0**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
```

```
log2c = [-5, -3, -1, 1, 3, 5]
log2g = [-5, -3, -1, 1, 3, 5]
best_lc, best_lg, best_acc = grid_search(log2c, log2g, x_train, y_train, x_test, y_test)
print("Best set (C, gamma)=(2^{}, 2^{}), accuracy:{}%".format(best_lc, best_lg, best_acc))
```

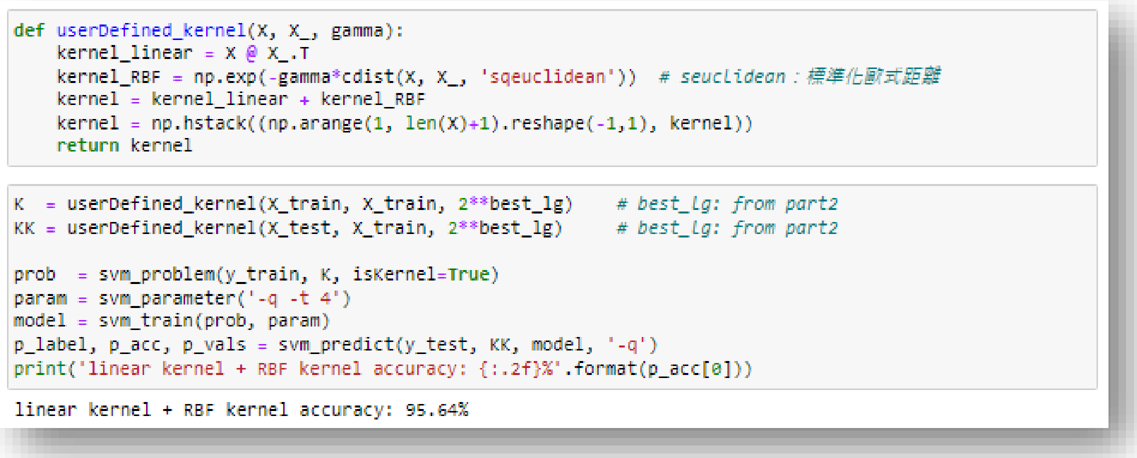
The $\log_2 C$ 、$\log_2 g$ = [-5, -3, -1, 1, 3, 5], and get the best set (C, γ)=(2^5,2^{-5}) accuracy=98.56%.

$\log_2 C$	$\log_2 g$	Accuracy	$\log_2 C$	$\log_2 g$	Accuracy
-5	-5	94.32%	1	-5	98.44%
-5	-3	42.30%	1	-3	85.04%
-5	-1	21.84%	1	-1	45.10%
-5	1	20.26%	1	1	25.48%
-5	3	79.08%	1	3	20.72%
-5	5	54.38%	1	5	35.36%

-3	-5	96.86%	3	-5	98.42%
-3	-3	47.26%	3	-3	85.16%
-3	-1	21.84%	3	-1	44.16%
-3	1	20.30%	3	1	25.44%
-3	3	78.96%	3	3	27.12%
-3	5	54.20%	3	5	35.04%
-1	-5	98%	**5**	**-5**	**98.56%**
-1	-3	54.52%	5	-3	84.84%
-1	-1	25.18%	5	-1	44.84%
-1	1	20.34%	5	1	25.56%
-1	3	78.72%	5	3	20.88%
-1	5	41.48%	5	5	34.98%

Part3: Use linear kernel + RBF kernel together (therefore a new kernel function) and compare its performance with respect to others. You would need to find out how to use a user-defined kernel in libsvm.

I define a userDefined_kernel. Use svm_problem to precomputed kernels. The parameter ”isKernel” means use precomputed kernel. The result linear kernel + RBF kernel accuracy=95.64%



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

```
K  = userDefined_kernel(X_train, X_train, 2**best_lg)    # best_lg: from part2
KK = userDefined_kernel(X_test, X_train, 2**best_lg)     # best_lg: from part2

prob  = svm_problem(y_train, K, isKernel=True)
param = svm_parameter('-q -t 4')
model = svm_train(prob, param)
p_label, p_acc, p_vals = svm_predict(y_test, KK, model, '-q')
print('linear kernel + RBF kernel accuracy: {:.2f}%'.format(p_acc[0]))
```

```
linear kernel + RBF kernel accuracy: 95.64%
```

Observation 1

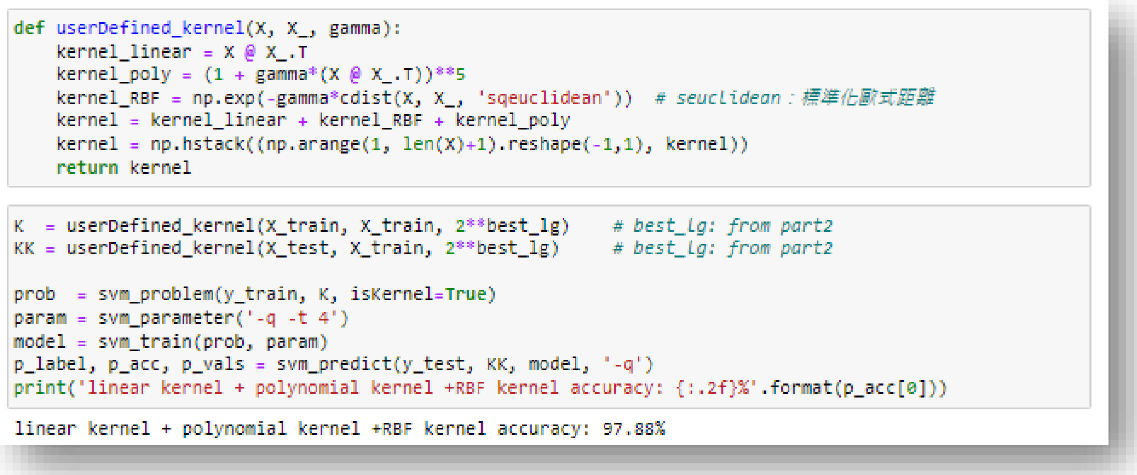
The larger the C, the greater the penalty, the fewer the support vectors, and the easier it is to overfitting.

The gamma is large, it is easy to outline the hyperplane that fits the near point, and it is easy to cause overfitting.

Observation 2

Try linear kernel + **polynomial kernel** + RBF kernel together

The accuracy:97.88% become better.



```
def userDefined_kernel(X, X_, gamma):
    kernel_linear = X @ X_.T
    kernel_poly = (1 + gamma*(X @ X_.T))**5
    kernel_RBF = np.exp(-gamma*cdist(X, X_, 'sqeuclidean'))  # seuclidean : 標準化歐式距離
    kernel = kernel_linear + kernel_RBF + kernel_poly
    kernel = np.hstack((np.arange(1, len(X)+1).reshape(-1,1), kernel))
    return kernel
```

```
K  = userDefined_kernel(X_train, X_train, 2**best_lg)    # best_lg: from part2
KK = userDefined_kernel(X_test, X_train, 2**best_lg)     # best_lg: from part2

prob  = svm_problem(y_train, K, isKernel=True)
param = svm_parameter('-q -t 4')
model = svm_train(prob, param)
p_label, p_acc, p_vals = svm_predict(y_test, KK, model, '-q')
print('linear kernel + polynomial kernel +RBF kernel accuracy: {:.2f}%'.format(p_acc[0]))
```

```
linear kernel + polynomial kernel +RBF kernel accuracy: 97.88%
```