Gaussian Process

0516312 陳皓婷

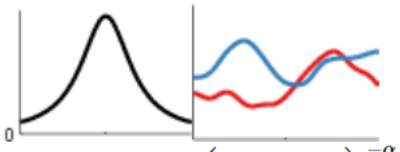
■ Code

```
def setdata():
    x = []
    y = []
    f = open('input.data')
    line = f.readline()
    cnt = 0
    while line :
        line=line.strip('\n')
        row_data = line.strip().split(' ')
        x.append(float(row_data[0]))
        y.append(float(row_data[1]))
        line = f.readline()
    x = np.array(x)
    y = np.array(y)
    f.close()
    return x,y
```

Function setdata:

read data from the “input.data” file, and get training x and y

Rational Quadratic Kernel



$$k_{\mathrm{RQ}}(x, x') = \sigma^2 \left(1 + \frac{(x-x')^2}{2\alpha\ell^2}\right)^{-\alpha}$$

```
def Kernel(Xn,Xm,beta,alpha,l):
    # rational quardratic kernel
    Krq = beta*((1+cdist(Xn,Xm,'sqeuclidean')/(2.0*alpha*1*1))**(-alpha))
    return Krq
```

Function Kernel:

Implement the formulation of rational quadratic kernel and return the result

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

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

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

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

Function GP:

Implement Gaussian Process based on the formulations, and get mean and square of variance

```
def GP(x,y,testX,alpha,l):
    x= x.reshape(-1,1)
    y= y.reshape(-1,1)
    testX = testX.reshape(-1,1)
    K0 = Kernel(x,x,beta,alpha,l)
    K1 = Kernel(testX,testX,beta,alpha,l)
    K2 = Kernel(x,testX,beta,alpha,l)
    C = K0+np.eye(len(x))/beta
    Kstar = K1+1/beta
    mean = np.dot(np.dot(np.transpose(K2),np.linalg.inv(C)),y)
    var2 = Kstar-np.dot(np.dot(np.transpose(K2),np.linalg.inv(C)),K2)
    return mean.flatten(),var2
```

```python
def ConfidenceInterval(mean,var2):
    plusSD = mean + 1.96*np.sqrt(np.diag(var2))
    minusSD = mean - 1.96*np.sqrt(np.diag(var2))
    return plusSD,minusSD
```

Function ConfidenceInterval:

Calculate the 95% confidence interval of f, and get upper & lower limit

```python
def neg_log_likihood(theta,x):
    #minimizing negative marginal log-likelihood
    x = x.reshape(-1,1)
    C = Kernel(x,x,beta,alpha=theta[0],l=theta[1])+np.eye(len(x))/beta
    neg_lnP = 0.5*math.log(np.linalg.norm(C))+0.5*np.dot(np.dot(np.transpose(y),np.linalg.inv(C)),y)+(x.shape[0]/2)*math.log(2*math.pi)
    return neg_lnP
```

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

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

Function neg_log_likelihood:

Minimize negative marginal log-likelihood based on the formulation

```python
if __name__ == '__main__':
    x,y = setdata()
    testX = np.linspace(-60,60, 500) # test point cnt = 500
    beta = 5.0
    alpha,l = 10.0, 1.0

    mean,var2 = GP(x,y,testX,alpha,l)
    plusSD,minusSD = ConfidenceInterval(mean,var2)
    visualization(0,x,y,testX,mean,plusSD,minusSD)

    #optimal kernel
    theta = [1.0, 1.0]
    result = minimize(neg_log_likihood,theta,args=(x))
    alpha,l=result.x
    mean,var2 = GP(x,y,testX,alpha,l)
    plusSD,minusSD = ConfidenceInterval(mean,var2)
    visualization(1,x,y,testX,mean,plusSD,minusSD)

    plt.show()
```

Function main:

- First, set the parameters, and implement GP to get mean and square_var that put in ConfidenceInterval, and visualize the result.
- Second, use minimize() to get the min neg_log_likelihood, and put them as the parameters into GP for optimization. After getting the new mean, variance, upper & lower limit, visualize the result.

```python
def visualization(mode,x,y,testX,mean,plusSD,minusSD):
    plt.subplot(2, 1, mode + 1)
    plt.plot(x,y,'.',color='red')
    plt.plot(testX,mean,color='blue')
    plt.fill_between(testX,plusSD,minusSD,facecolor='green',alpha=0.3)
    plt.xlim(-60,60)
```

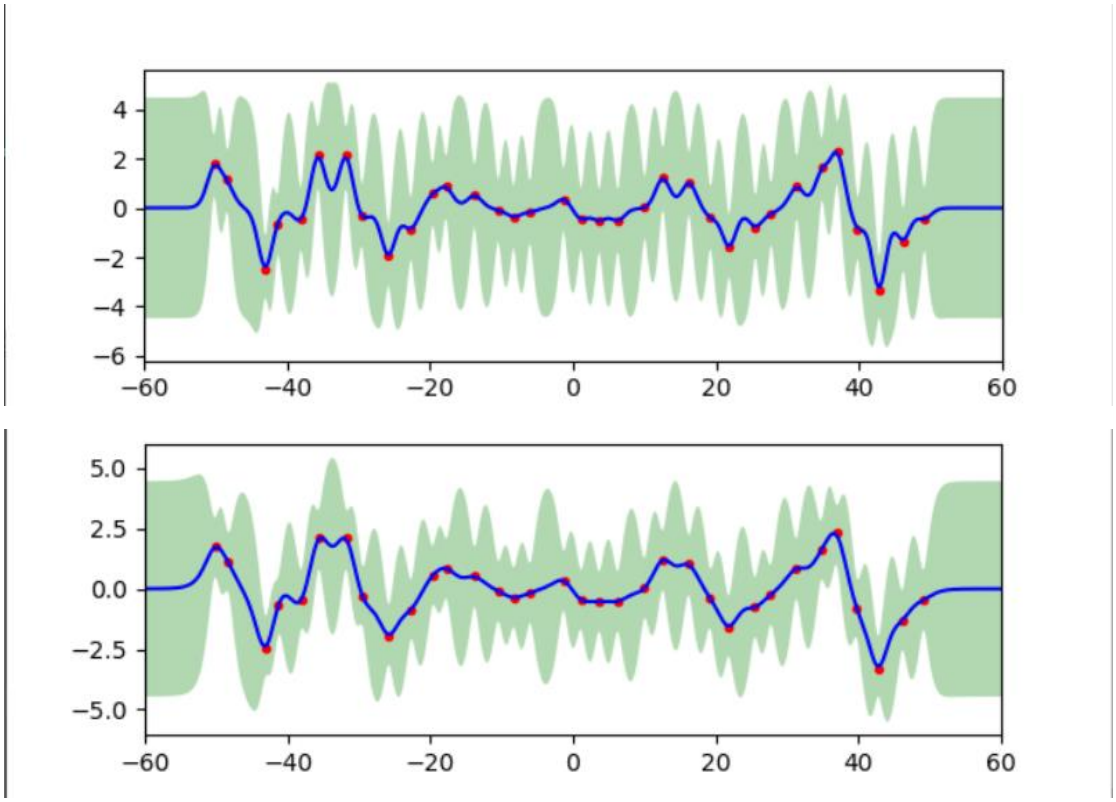
Function visualization:

Train x, y => red dot;

Test x, mean => blue line;

95% confidence interval=> green area

■ Result & Observation



After optimization(minimize the negative log-likelihood), the area of 95% confidence interval is smaller.

By control the value of parameter, we can avoid f from overfitting.

Alpha : larger alpha can increase the noise level, which can prevent f from overfitting

L : length scale , effect the sharp of the function

SVM on the MNIST dataset:

■ Code & Result & Observation

```
if __name__ == '__main__':
    x_train, y_train, x_test, y_test = setdata()
    trainModel(x_train, y_train, x_test, y_test)

    trainModelwithGrid(x_train, y_train, x_test, y_test)

    trainModelplusLinRBF(x_train, y_train, x_test, y_test)
```

Function main:
First, set data
Then, separately call function to see the 3 part performance

```
def setdata():
    def read(filename):
        return np.genfromtxt(filename, delimiter=',', dtype="float64")
    Xtrain = read('X_train.csv')
    Ytrain = read('Y_train.csv')
    Xtest = read('X_test.csv')
    Ytest = read('Y_test.csv')
    return Xtrain,Ytrain, Xtest,Ytest
```

Function setdata:
read data from the test file,
get train x y and test x y

```
def trainModel(x, y, xt, yt):
    prob  = svm_problem(y, x)
    paramLin  = svm_parameter("-t 0 -q")
    paramPoly = svm_parameter("-t 1 -q")
    paramRBF  = svm_parameter("-t 2 -q")
    modelLin  = svm_train(prob, paramLin)
    modelPoly = svm_train(prob, paramPoly)
    modelRBF  = svm_train(prob, paramRBF)
    print("linear")
    resultLin  = svm_predict(yt, xt, modelLin)
    print("")
    print("Polynomial")
    resultPoly = svm_predict(yt, xt, modelPoly)
    print("")
    print("RBF")
    resultRBF  = svm_predict(yt, xt, modelRBF)
    print("")
```

Function trainModel:
Use the default parameters to see the difference of the performance of 3 kinds of kernel functions (linear, Polynomial, RBF)
Use the libsvm library to implement svm:

1. Set training data by svm_problem()
2. Set parameters in svm by svm_parameter()
   -t 0 : Linear kernel
   -t 1 : Polynomial kernel
   -t 2 : RBF kernel
   -q : quiet mode(no output)
3. Train the model by svm_train()
4. Get predict result by svm_predict()

Result (accuracy of test data)

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

Polynomial
Accuracy = 34.68% (867/2500) (classification)

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

Observation:

- If polynomial kernel isn't adjusted the default parameters, its outcome will work badly.
- Linear kernel and RBF kernel work well with default parameters

Function trainModelwithGrid:

Train the model with grid search, and find out the parameters of the best performance model.

1. Build- inside-function train: do the duplicate train model things
2. Set g and c array for the coming grid search to find out which is the best
3. linear kernel part:

   grid search the parameter c by going through the c array, record the best one, and get the best-param-model-accuracy-rate of test data

   polynomial kernel part:

   grid search the parameter c and g by going through the c &g array, record the best one, and get the best-param-model-accuracy-rate of test data

   RBF kernel part:

   grid search the parameter c and g by going through the c &g array, record the best one, and get the best-param-model-accuracy-rate of test data

   -c : change the value of parameter c

   -g : change the value of parameter g

   -v 3 : set the amount of fold in cross-validation

```
def trainModelwithGrid(x, y, xt, yt):
    def train(parameters):
        prob  = svm_problem(y, x)
        param = svm_parameter(parameters)
        model = svm_train(prob, param)
        return model
    g = [2**i for i in range(-15,4,2)]
    c = [2**i for i in range(-5,16,2)]
    # linear parameter:c
    best_acc = 0.0
    for c_ele in range(len(c)):
        param = '-t 0 -c {} -v 3 -q'.format(c[c_ele])
        acc = train(param)
        print("c = ",c[c_ele])
        if best_acc < acc:
            best_acc = acc
            best_param = param
            best_c_ele = c_ele
    print("linear best accuracy:",best_acc)
    print("linear best parameter:",best_param)
    print("Test accuracy:")
    paramBest = '-t 0 -c {} -q'.format(c[best_c_ele])
    modelBest = train(paramBest)
    svm_predict(yt, xt, modelBest)
    print("============================")
```

```
    # polynomial parameter:c,g
    for c_ele in range(len(c)):
        for g_ele in range(len(g)):
            param = '-t 1 -c {} -g {} -v 3 -q'.format(c[c_ele], g[g_ele])
            acc = train(param)
            print("c = {}, g = {}".format(c[c_ele], g[g_ele]))
            if best_acc < acc:
                best_acc = acc
                best_param = param
                best_c_ele = c_ele
                best_g_ele = g_ele
    print("Polynomial best accuracy:",best_acc)
    print("Polynomial best parameter:",best_param)
    print("Test accuracy:")
    paramBest = '-t 1 -c {} -g {} -q'.format(c[best_c_ele], g[best_g_ele])
    modelBest = train(paramBest)
    svm_predict(yt, xt, modelBest)
    print("============================")
```

```
# RBF parameter: c g
for c_ele in range(len(c)):
    for g_ele in range(len(g)):
        param = '-t 2 -c {} -g {} -v 3 -q'.format(c[c_ele], g[g_ele])
        acc = train(param)
        print("c = {}, g = {}".format(c[c_ele], g[g_ele]))
        if best_acc < acc:
            best_acc = acc
            best_param = param
            best_c_ele = c_ele
            best_g_ele = g_ele
print("RBF best accuracy:",best_acc)
print("RBF best parameter:",best_param)
print("Test accuracy:")
paramBest = '-t 1 -c {} -g {} -q'.format(c[best_c_ele], g[best_g_ele])
modelBest = train(paramBest)
svm_predict(yt, xt, modelBest)
print("==========================")
```

Result:

<Linear Kernel>

i (c=2^i)	-5	-3	-1	1	3	5	7	9	11	13	15
Cross Validation Accuracy	96.76%	96.78%	96.22%	96.28%	96.16%	96.38%	96.06%	95.94%	96.24%	96.30%	96.48%

linear best accuracy: 96.78
linear best parameter: -t 0 -c 0.125 -v 3 -q
Test Accuracy = 95.92% (2398/2500) (classification)

<Polynomial Kernel>

i (c=2^i) / j (g=2^j)	-5	-3	-1	1	3	5	7	9	11	13	15
-15	28.62%	28.52%	28.8%	28.34%	28.5%	28.9%	29.02%	28.38%	28.46%	29.3%	29.04%
-13	28.46%	28.7%	28.46%	28.34%	28.38%	28.32%	29.2%	29.26%	36.2%	67.42%	85.08%
-11	28.88%	28.38%	28.64%	28.64%	28.38%	35.64%	67.28%	85.1%	92.74%	96.14%	97.22%
-9	29.08%	28.56%	35.62%	67.4%	85.08%	92.64%	96.16%	97.38%	97.36%	97.48%	97.48%
-7	67.42%	85.06%	92.8%	96.16%	97.1%	97.7%	97.66%	97.42%	97.46%	97.64%	97.3%
-5	96.04%	97.44%	97.9%	97.38%	97.14%	97.36%	97.36%	97.18%	97.7%	97.3%	97.7%
-3	97.42%	7.16%	97.48%	97.36%	97.28%	97.34%	97.28%	97.34%	97.32%	97.36%	97.3%
-1	97.12%	96.3%	97.48%	97.2%	97.5%	97.42%	97.3%	97.38%	97.44%	97.32%	97.6%
1	97.32%	97.58%	97.24%	97.36%	97.24%	97.44%	97.52%	97.4%	97.28%	97.46%	97.28%
3	97.46%	97.44%	97.52%	97.34%	97.3%	97.3%	97.34%	97.42%	97.5%	97.42%	97.56%

Polynomial best accuracy: 97.89999999999999
Polynomial best parameter: -t 1 -c 0.5 -g 0.03125 -v 3 -q
Test Accuracy = 97.48% (2437/2500) (classification)

<RBF Kernel>

i (c=2^i) / j (g=2^j)	-5	-3	-1	1	3	5	7	9	11	13	15
-15	79.38%	79.54%	79.7%	89.54%	94.38%	95.94%	96.76%	96.8%	96.78%	96.16%	96.34%
-13	79.8%	79.68%	89.46%	94.22%	96.02%	96.74%	96.84%	96.58%	96.48%	96.5%	96.42%
-11	80.12%	89.58%	94.28%	95.9%	96.78%	97%	97.02%	96.56%	96.5%	96.9%	96.5%
-9	89.4%	94.24%	95.98%	97.18%	97.06%	97.54%	97.12%	97.22%	97.3%	97.38%	97.36%
-7	94.26%	96.1%	97.16%	97.78%	97.92%	98.02%	97.96%	98%	98.22%	97.86%	98.03%
-5	94.34%	96.88%	97.98%	98.4%	98.5%	98.2%	98.52%	98.44%	98.38%	98.46%	98.42%
-3	42.28%	47.52%	54.98%	85%	85.76%	84.76%	85.98%	85.06%	85.22%	85.1%	84.98%
-1	22.22%	22.06%	25.04%	46.62%	45.86%	44.54%	45.22%	45.34%	45.62%	45.12%	45%
1	20.32%	20.64%	20.32%	25.26%	24.88%	25.68%	25.62%	25.64%	24.8%	24.92%	25.88%
3	78.92%	79.04%	78.8%	20.76%	20.7%	21%	27.22%	20.52%	20.8%	20.92%	20.88%

RBF best accuracy: 98.52
RBF best parameter: -t 2 -c 128 -g 0.03125 -v 3 -q
Test Accuracy = 98.52% (2463/2500) (classification)

Observation:

The performance of 3 kinds of kernel functions (Linear, Polynomial, RBF)

- The performance of Linear Kernel function does well in every c,
- The performance of Polynomial Kernel function is better when c and g are not too small.
- The performance of RBF kernel function is good when c is not too big.
- All test data accuracy rates are good

Function trainModelplusLinRBF:

Use the user-defined kernel to train model

1. Build-inside-function LinearRBFKernel:
   Calculate the value from linear kernel and the value from RBF kernel (based on the formulations on the LIBSVM website), and add up to get the new, self-defined kernel value
2. Build-inside-function train:
   Get the svm model by svm_parameter(), svm_problem() and svm_train().
   Because of using self-defined kernel, the value of self-define kernel and “isKernel = True” are added to the the parameters for svm_problem().
3. Grid search for the best parameters (g,c) for svm which is equipped with self-define kernel. And then, get the best kernel model

```
-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)
```

```python
def trainModelplusLinRBF(x, y, xt, yt):
    def LinearRBFKernel(x1,x2,g):
        linear_part = np.dot(x1,np.transpose(x2))
        RBF_part = np.exp(-g*cdist(x1,x2,'sqeuclidean'))
        mix = linear_part+RBF_part
        return np.hstack((np.arange(1, x1.shape[0]+1).reshape(-1, 1), mix))

    def train(k, parameters):
        prob  = svm_problem(y, k, isKernel=True)
        param = svm_parameter(parameters)
        model = svm_train(prob, param)
        return model

    g = [2**i for i in range(-15,4,2)]
    c = [2**i for i in range(-5,16,2)]
    # linear+RBF parameter:c g
    best_acc = 0.0
    for c_ele in range(len(c)):
        for g_ele in range(len(g)):
            param = '-t 4 -c {} -g {} -v 3 -q'.format(c[c_ele], g[g_ele])
            new_k = LinearRBFKernel(x,x,g[g_ele])
            acc = train(new_k, param)
            print("c = {}, g = {}".format(c[c_ele], g[g_ele]))
            if best_acc < acc:
                best_acc = acc
                best_param = param
                best_new_k = new_k
                best_c_ele = c_ele
                best_g_ele = g_ele
    print("LinRBF best accuracy:",best_acc)
    print("LinRBF best parameter:",best_param)
```

i (c=2^i) / j (g=2^j)	-5	-3	-1	1	3	5	7	9	11	13	15
-15	96.76%	96.74%	96.24%	96.54%	95.98%	96.1%	96.2%	96.28%	96.24%	96.26%	96.42%
-13	97.02%	96.86%	96.42%	96.2%	96.18%	96.34%	96.28%	96.1%	96.12%	96.22%	96.34%
-11	96.78%	96.68%	96.54%	96%	95.84%	96.42%	96.28%	96.36%	95.92%	96.38%	95.82%
-9	97.1%	96.6%	96.06%	96.22%	96.06%	96.06%	96.18%	96.06%	96.04%	96.3%	96.28%
-7	97.06%	96.4%	96.18%	96.26%	96.14%	96.36%	96.38%	96.7%	96.14%	95.92%	96.1%
-5	97.08%	96.76%	96.28%	96.62%	96.54%	96.36%	96.2%	96.36%	96.18%	96.68%	96.58%
-3	96.82%	96.4%	96.26%	96.4%	96.14%	96.66%	96.26%	96.4%	96.42%	96.22%	96.62%
-1	96.68%	96.86%	96.4%	96.12%	96.56%	96.6%	96.02%	96.12%	96.4%	96.12%	96.5%
1	97.04%	96.94%	96.56%	96.32%	96.34%	96.32%	96.4%	96.22%	96.32%	96.62%	96.32%
3	96.76%	96.76%	96.3%	96.32%	96.24%	96.22%	96.2%	96.3%	96.32%	96.24%	96.38%

LinRBF best accuracy: 97.1
LinRBF best parameter: -t 4 -c 0.03125 -g 0.001953125 -v 3 -q

Observation:

The performance of Linear+RBF Kernel functions and comparison with others (Linear, Polynomial, RBF)

- The performance of Linear+RBF Kernel function is good and stable with all c and g.
- Its performance is kind of like Linear Kernel which is also good with all c and g
- According to the outcomes, it is believed that user-defined kernel function can work well.
- The 4 kinds of kernel functions all can work well when we choose the suitable parameters.