ML HW5

1. Gaussian Process

1. Code

Task1

In the first part, use Gaussian Process to predict 1000 sample points' mean and variance. The Gaussian Process use the Rational Quadratic Kernel and set alpha, length_scale, sigma to be 1.
(X, y are training data.)

```
def read_data(fliename):
    x=[]
    y=[]
    f = open(fliename, 'r')
    for line in f.readlines():
        point = line.split(' ')
        x.append(float(point[0]))
        y.append(float(point[1]))
    f.close()
    x=np.asarray(x)
    y=np.asarray(y)
    return x,y
```

```
X,y=read_data("input.data")
beta=5
alpha=1
length_scale=1
sigma=1
GP(X,y,beta,alpha,length_scale,sigma)
```

Rational Quadratic Kernel can be computed as:

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

The kernel() function computes the kernel matrix between X1 and X2

```
def kernel(X1,X2,alpha,length_scale,sigma):
    #X1: (n) ndarray
    #X2: (m) ndarray
    #return (n,m)  ndarray
    kernel=(sigma**2)*(1+(X1.reshape(-1,1)-X2.reshape(1,-1
))**2/(2*alpha*length_scale**2))**(-alpha)
    return kernel
```

In the Gaussian Process use

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

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

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

to compute test point's mean and variance.
And use variance to compute 95% confidence interval (±1.96σ), then visualize the prediction result.

```
def GP(X,y,beta,alpha,length_scale,sigma):
    C=kernel(X,X,alpha,length_scale,sigma)+(1/beta)*np.
identity(len(X))
    X_test=np.linspace(-60,60,num=1000)
    mean_of_x_test=kernel(X,X_test,alpha,length_scale,sigma
).T@np.linalg.inv(C)@y
    k_star=kernel(X_test,X_test,alpha,length_scale,sigma)+(1
/beta)*np.identity(len(X_test))
    var_of_x_test=k_star-kernel(X,X_test,alpha,length_scale,
sigma).T@np.linalg.inv(C)@kernel(X,X_test,alpha,length_scale
,sigma)

    mean_of_x_test=mean_of_x_test.reshape(-1)
    sd_of_x_test=np.sqrt(np.diag(var_of_x_test))
    interval=1.96*sd_of_x_test

    plt.plot(X_test,mean_of_x_test,'k-')
    plt.fill_between(X_test,mean_of_x_test+interval,
mean_of_x_test-interval,alpha=0.3,color='red')
    plt.scatter(X,y)
    plt.title('alpha: {:.5f}, length_scale: {:.5f}, sigma:
{:.5f}'.format(alpha,length_scale,sigma))
    plt.xlim(-60, 60)
    plt.show()
```

Task2

In the second part, optimize the kernel function's parameters such that the marginal likelihood $p(y|\theta)$ is max. ,and it is equivalent to minimize the negative marginal log-likelihood.

The negative marginal log-likelihood is computed as:

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

Where C=kernel+ β^{-1} and N=size of C

```
def negative_log_likelihood(theta):
    #theta[0]: alpha
    #theta[1]: length_scale
    #theta[2]: sigma
    k=kernel(X,X,theta[0],theta[1],theta[2])
    C=k+(1/beta)*np.identity(len(X))
    negative_log_likelihood=(1/2)*(np.log(np.linalg.det(C))+
y.T@np.linalg.inv(C)@y+len(C)*np.log(2*np.pi))
    return negative_log_likelihood
```

Use scipy. optimize.minimize() to minimize negative _log _likelihood(), and use parameter θ to run Gaussisn Process

```
theta=[1,1,1]
res=minimize(negative_log_likelihood,theta)
alpha=res.x[0]
length_scale=res.x[1]
sigma=res.x[2]
GP(X,y,beta,alpha,length_scale,sigma)
```

2. Experiments

The first part set beta=5, alpha=1, length_scale=1, sigma=1

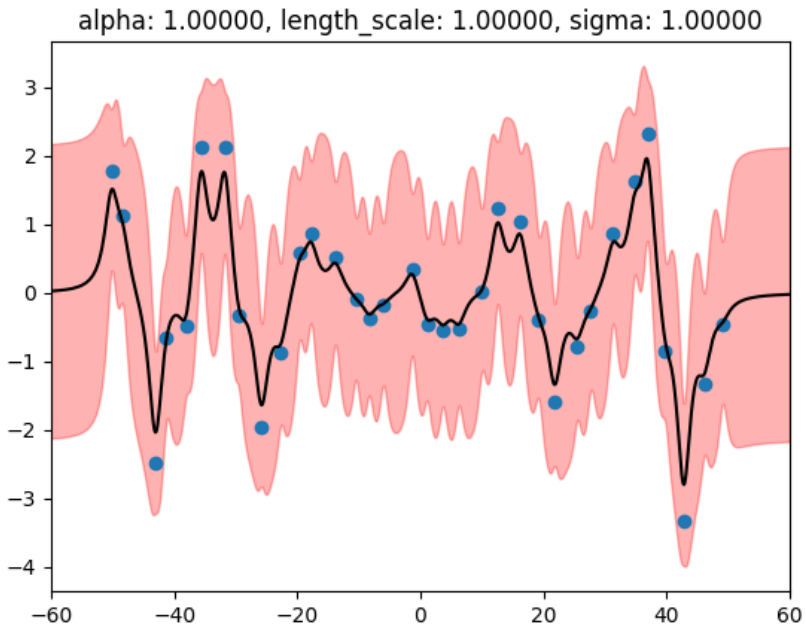


Image_1

The second part set beta=5, alpha=377.09, length_scale=3.32, sigma=1.32

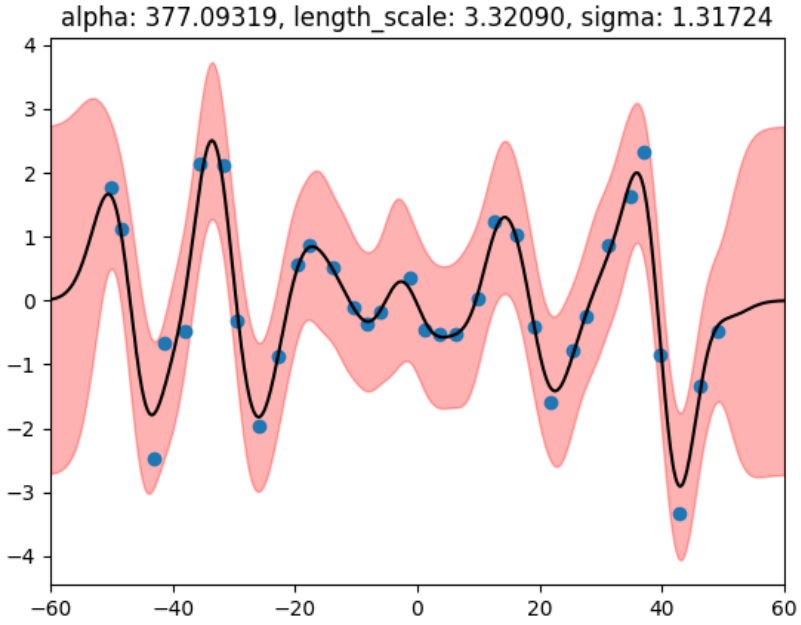


Image_2

3. Observations and Discussion

Compare **Image_1** and **Image_2,** we can see the predict curve is more smoothly and accurately after optimized. This is more in line with the Gaussian Process idea:x1, x2 are closer, then y(x1),y(x2) are closer.
In the remain part, I will discuss the effect of hyper-parameter on Gaussian Process with prediction's image.

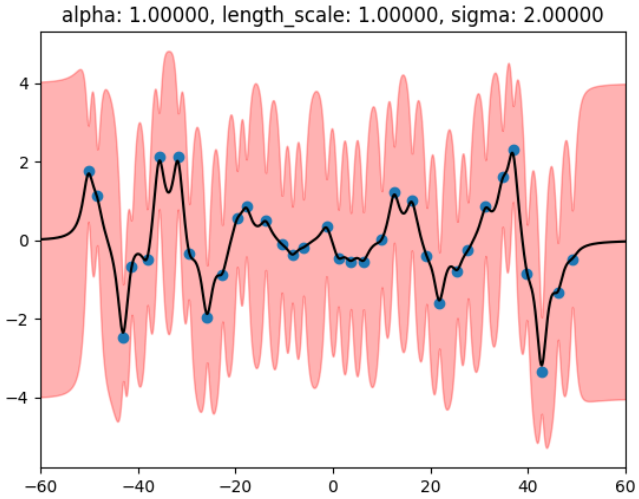
For sigma:

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

Observe that sigma is a scalar in kernel function,

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

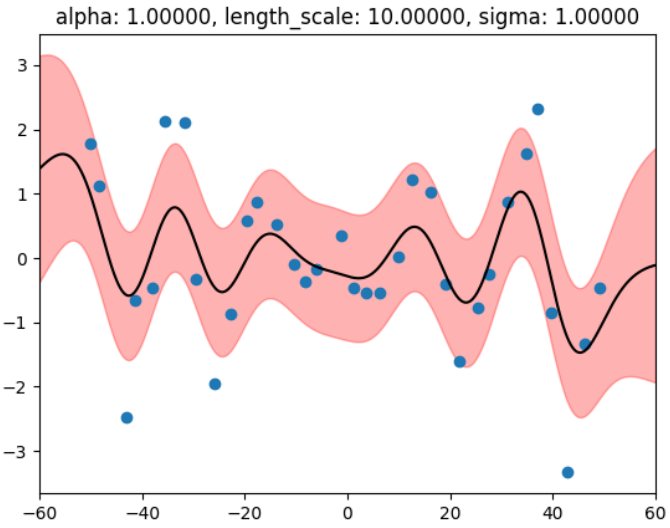
And in predict mean formula, if we change sigma, $k(x, x^*)^T$ and C^{-1} will cancel out the change scalar, so we expect that the predict mean will be the same if we change sigma.
But in variance formula if we change sigma to be k times sigma, the new predict variance should be k-square times old predict variance.
Compare **Image_1** and the image below, the expectations are correct. Therefore, sigma is also known as amplitude.

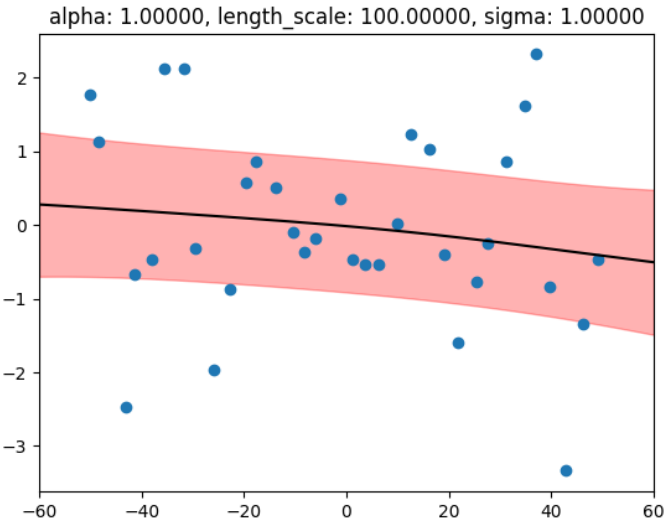


For length scale:

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

Observe when ℓ approaches infinity, $\frac{\|x_a - x_b\|^2}{2al^2}$ approaches to 0, it means that the training data is about useless. So I expect that when ℓ is bigger, the predict mean will be more close to y=0
The below images and **Image_1** show the observation is correct.



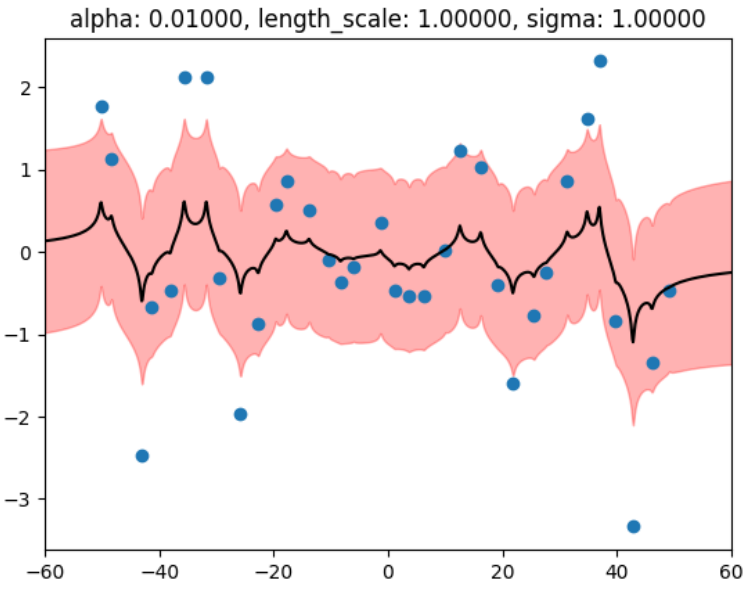


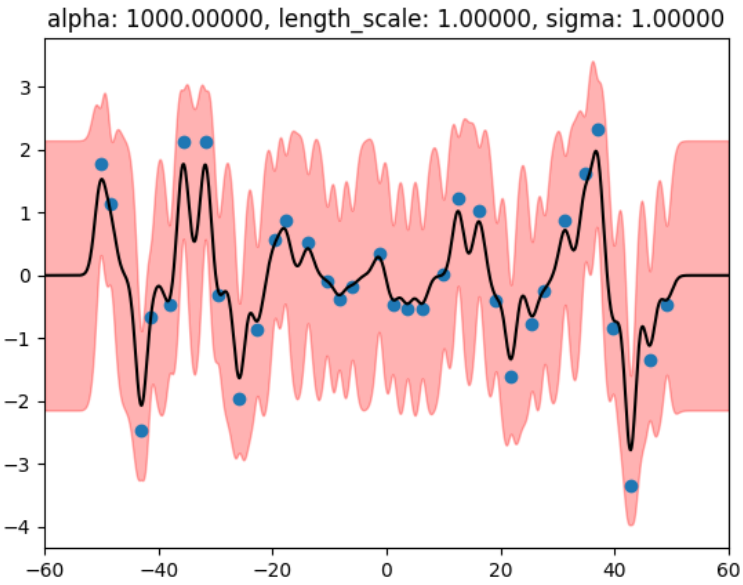
Additionally, we found that the larger the ℓ is, the more smoothly the graph is.

For alpha:

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

When alpha approaches infinity, the kernel converges into exp. Quadratic kernel.
In the test, alpha=1 and alpha=1000 seems have the same results.
But when alpha=0.01, the predict result is not as well as alpha=1 is.





2. SVM on MNIST

1. Code

Task1

Load the data from .csv

```
def load_x(filename):
    x=[]
    f = open(filename, 'r')
    for line in f.readlines():
        tmp=[]
        for i in line.split(','):
            tmp.append(float(i))
        x.append(tmp)
    f.close()
    x=np.asarray(x)
    return x

def load_y(filename):
    y=[]
    f = open(filename, 'r')
    for line in f.readlines():
        y.append(float(line))
    f.close()
    y=np.asarray(y)
    return y

X_train=load_x('X_train.csv')
Y_train=load_y('Y_train.csv')
X_test=load_x('X_test.csv')
Y_test=load_y('Y_test.csv')
```

Train the model and use it to predict, then output the accuracy.
In the svm_train() function, kernel options are shown below.

-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)

-q means quiet mode(no outputs)
For variable mode : Key is kernel mode and Value is option in svm_train() function.

```
mode={'linear kernel':'-t 0','polynomial kernel':'-t 1',
'RBF kernel':'-t 2'}
for m,parameter in mode.items():
    model=svm_train(Y_train,X_train,'-q '+parameter)
    p_label,p_acc,p_val=svm_predict(Y_test,X_test,model,
'-q')
    print(m,'accuracy:{:.2f} %'.format(p_acc[0]))
```

Task2

Set
cost=[0.125, 0.25, 0.5, 1, 2, 4, 8]
gamma=[0.125, 0.25, 0.5, 1, 2, 4, 8]
degree=[2,3,4]
and do the grid search on each mode.
In each iteration, it will output the current setting and validation accuracy, and update the optimal options depending on the accuracy.
When grid search is done, it will output the optimal options and accuracy.

```python
def grid_search(X_train,Y_train):
    optimal_opt=''
    optimal_acc=0
    cost=[0.125,0.25,0.5,1,2,4,8]
    mode=['linear kernel','polynomial kernel','RBF kernel']

    for m in mode:
        for c in cost:
            if m=='linear kernel':
                opt=f'-t 0 -c {c} -v 3 -q'
                print(m,f': cost:{c}')
                cv_acc=svm_train(Y_train,X_train,opt)
                if cv_acc>optimal_acc:
                    optimal_acc=cv_acc
                    optimal_opt=opt
            elif m=='polynomial kernel':
                gamma=[0.125,0.25,0.5,1,2,4,8]
                degree=[2,3,4]
                for g in gamma:
                    for d in degree:
                        opt=f'-t 1 -c {c} -g {g} -d {d} -v 3 -q'
                        print(m,f': cost:{c}, gamma:{g}, degree{d}')
                        cv_acc=svm_train(Y_train,X_train,opt)
                        if cv_acc>optimal_acc:
                            optimal_acc=cv_acc
                            optimal_opt=opt
            else:
                gamma=[0.125,0.25,0.5,1,2,4,8]
                for g in gamma:
                    opt=f'-t 2 -c {c} -g {g} -v 3 -q'
                    print(m,f': cost:{c}, gamma:{g}')
                    cv_acc=svm_train(Y_train,X_train,opt)
                    if cv_acc>optimal_acc:
                        optimal_acc=cv_acc
                        optimal_opt=opt
    return optimal_opt,optimal_acc

X_train=load_x('X_train.csv')
Y_train=load_y('Y_train.csv')
X_test=load_x('X_test.csv')
Y_test=load_y('Y_test.csv')
print(grid_search(X_train,Y_train))
```

Task3

Use user kernel (linear+ RBF) to train the model. And compute kernel(train,test) to predict.

Gamma is set to be 0.125 by the result of Task2.(better accuracy)

```
X_train=load_x('X_train.csv')
Y_train=load_y('Y_train.csv')
X_test=load_x('X_test.csv')
Y_test=load_y('Y_test.csv')

k=user_kernel(X_train,X_train,0.125)
k_star=user_kernel(X_test,X_train,0.125)

#-t 4:precomputed kernel
model=svm_train(Y_train,k,'-t 4 -q')
svm_predict(Y_test,k_star,model,)
```

User_kernel() use linear+ RBF kernel and return the format svm_train() and svm_predict() used.

scipy.spatial.distance.cdist() is used to calculate the distance matrix between X1 and X2.Here the distance is use square-distance

```
def linearKernel(X1, X2):
    kernel = X1 @ X2.T
    return kernel

def RBFKernel(X1, X2, gamma):
    dist=cdist(X1, X2, metric='sqeuclidean')
#square euclidean dist.
    kernel = np.exp((-1 * gamma * dist))
    return kernel

def user_kernel(X1,X2,gamma):
    k=np.hstack((np.arange(1,len(X1)+1).
reshape(-1,1),linearKernel(X1, X2)+RBFKernel(
X1, X2, gamma)))
    return k
```

2. Experiments

Task1

```
linear kernel accuracy:95.08 %
polynomial kernel accuracy:34.68 %
RBF kernel accuracy:95.32 %
```

Task2

```
linear kernel : cost:0.125
Cross Validation Accuracy = 96.62%
linear kernel : cost:0.25
Cross Validation Accuracy = 96.58%
linear kernel : cost:0.5
Cross Validation Accuracy = 96.22%
linear kernel : cost:1
Cross Validation Accuracy = 96.28%
linear kernel : cost:2
Cross Validation Accuracy = 96.16%
linear kernel : cost:4
Cross Validation Accuracy = 96.38%
linear kernel : cost:8
Cross Validation Accuracy = 96.06%
polynomial kernel : cost:0.125, gamma:0.125, degree2
Cross Validation Accuracy = 97.58%
polynomial kernel : cost:0.125, gamma:0.125, degree3
Cross Validation Accuracy = 97.32%
polynomial kernel : cost:0.125, gamma:0.125, degree4
Cross Validation Accuracy = 96.28%
polynomial kernel : cost:0.125, gamma:0.25, degree2
Cross Validation Accuracy = 97.86%
polynomial kernel : cost:0.125, gamma:0.25, degree3
Cross Validation Accuracy = 97.46%
polynomial kernel : cost:0.125, gamma:0.25, degree4
Cross Validation Accuracy = 96.22%
polynomial kernel : cost:0.125, gamma:0.5, degree2
```

…….

Optimal option and validation accuracy:

```
('-t 1 -c 4 -g 4 -d 2 -v 3 -q', 98.1)
```

Task3

```
Accuracy = 95.64% (2391/2500) (classification)
```

3. Observations and Discussion

In linear kernel mode ,cost changing doesn't affect the accuracy much.

```
linear kernel : cost:0.125
Cross Validation Accuracy = 96.62%
linear kernel : cost:0.25
Cross Validation Accuracy = 96.58%
linear kernel : cost:0.5
Cross Validation Accuracy = 96.22%
linear kernel : cost:1
Cross Validation Accuracy = 96.28%
linear kernel : cost:2
Cross Validation Accuracy = 96.16%
linear kernel : cost:4
Cross Validation Accuracy = 96.38%
linear kernel : cost:8
Cross Validation Accuracy = 96.06%
```

In poly. Kernel the changing of parameters(cost, gamma, degree) doesn't affect the accuracy much.It is all about 96% accuracy.

```
polynomial kernel : cost:0.125, gamma:0.125, degree2
Cross Validation Accuracy = 97.58%
polynomial kernel : cost:0.125, gamma:0.125, degree3
Cross Validation Accuracy = 97.32%
polynomial kernel : cost:0.125, gamma:0.125, degree4
Cross Validation Accuracy = 96.28%
polynomial kernel : cost:0.125, gamma:0.25, degree2
Cross Validation Accuracy = 97.86%
polynomial kernel : cost:0.125, gamma:0.25, degree3
Cross Validation Accuracy = 97.46%
polynomial kernel : cost:0.125, gamma:0.25, degree4
Cross Validation Accuracy = 96.22%
polynomial kernel : cost:0.125, gamma:0.5, degree2
Cross Validation Accuracy = 97.8%
polynomial kernel : cost:0.125, gamma:0.5, degree3
Cross Validation Accuracy = 97.32%
polynomial kernel : cost:0.125, gamma:0.5, degree4
```

In RBF kernel, when the cost is low , higher gamma or lower gamma has higher accuracy.

```
RBF kernel : cost:0.125, gamma:0.12
Cross Validation Accuracy = 48.38%
RBF kernel : cost:0.125, gamma:0.25
Cross Validation Accuracy = 26.98%
RBF kernel : cost:0.125, gamma:0.5
Cross Validation Accuracy = 21.54%
RBF kernel : cost:0.125, gamma:1
Cross Validation Accuracy = 20.76%
RBF kernel : cost:0.125, gamma:2
Cross Validation Accuracy = 20.3%
RBF kernel : cost:0.125, gamma:4
Cross Validation Accuracy = 33.18%
RBF kernel : cost:0.125, gamma:8
Cross Validation Accuracy = 79.08%
RBF kernel : cost:0.25, gamma:0.125
Cross Validation Accuracy = 49.2%
RBF kernel : cost:0.25, gamma:0.25
Cross Validation Accuracy = 33.66%
RBF kernel : cost:0.25, gamma:0.5
Cross Validation Accuracy = 21.96%
RBF kernel : cost:0.25, gamma:1
Cross Validation Accuracy = 20.56%
RBF kernel : cost:0.25, gamma:2
Cross Validation Accuracy = 20.3%
RBF kernel : cost:0.25, gamma:4
Cross Validation Accuracy = 39.68%
RBF kernel : cost:0.25, gamma:8
Cross Validation Accuracy = 78.94%
```

But when the cost is not low, lower gamma has higher accuracy.
So we can sum up that smaller gamma has higher accuracy.

```
RBF kernel : cost:2, gamma:0.125
Cross Validation Accuracy = 84.82%
RBF kernel : cost:2, gamma:0.25
Cross Validation Accuracy = 65.14%
RBF kernel : cost:2, gamma:0.5
Cross Validation Accuracy = 45.38%
RBF kernel : cost:2, gamma:1
Cross Validation Accuracy = 31.24%
RBF kernel : cost:2, gamma:2
Cross Validation Accuracy = 25%
RBF kernel : cost:2, gamma:4
Cross Validation Accuracy = 22.04%
RBF kernel : cost:2, gamma:8
Cross Validation Accuracy = 20.72%
RBF kernel : cost:4, gamma:0.125
Cross Validation Accuracy = 85.1%
RBF kernel : cost:4, gamma:0.25
Cross Validation Accuracy = 65.08%
RBF kernel : cost:4, gamma:0.5
Cross Validation Accuracy = 44.66%
RBF kernel : cost:4, gamma:1
Cross Validation Accuracy = 31.24%
RBF kernel : cost:4, gamma:2
Cross Validation Accuracy = 24.96%
RBF kernel : cost:4, gamma:4
Cross Validation Accuracy = 21.9%
RBF kernel : cost:4, gamma:8
Cross Validation Accuracy = 27.22%
```