Machine Learning Homework 5
Gaussian Process & SVM

310706043 肇綺筠

I. Gaussian Process

■ a. code with detailed explanations

Our objective is to predict the distribution of data ranging from -60 to 60, where the GP model is trained by 34 points.

Hence, we need to find the mean and variance of data (I set 600 points) between [-60,60]

a.1 Before Optimization: alpha=1 and length_scale=1

- First, the rational quadratic kernel is defined below:

$$k(\mathbf{x}_n, \mathbf{x}_m) = (1 + ||\mathbf{x}_n - \mathbf{x}_m||^2/(2aL^2))^{-a},$$

where *L* is the kernel length scale and *a* implies the weighting of large and small scale variations.

```
k = kernel(x,x,best_a,best_ls)+(1/beta)*np.identity(len(x))
```

▲first define k(x,x),the relation between "training data"

```
#rational quadratic kernel function for GP
def kernel(x1,x2,a=1,len_scale=1):
   sq_error = np.power(x1.reshape(-1,1)-x2.reshape(1,-1),2.0)
   kernel = np.power((1+sq_error/2*a*len_scale**2),-a)

   return kernel
```

▲implement rational quadratic kernel

- After defining kernel, we can now predict the distribution of *f*, finding mean and var by the function defined below conditional distribution $p(y^*|\mathbf{y})$ is a Gaussian distribution with:

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

```
linex = np.linspace(-60,60,num=600)
m_pred,v_pred = prediction(linex,x,y,k,beta,best_a,best_ls)
```

▲600 datapoints between -60 and 60 as training data

```
def prediction(x_test,x,y,k,beta,a=1,len_scale=1):
    #kxx_test -> k(x,x*) relation between test and training
    kxx_test = kernel(x,x_test,a=1,len_scale=1)
    #k(x*,x*)
    kx_testx_test = kernel(x_test,x_test,a=1,len_scale=1)
    #mean(x*)=k(x,x*).T*inverse(k(x,x))*y
    mean = kxx_test.T @ np.linalg.inv(k)@y.reshape(-1,1)
    #k(x*,x*)+1/beta should become matrix
    vars = kx_testx_test+(1/beta)*np.identity(len(kx_testx_test))-
kxx_test.T@np.linalg.inv(k)@kxx_test

    return mean,vars
```

▲x_test implies x* (testing data),kxx_test indicates k(x,x*), the relation between train and test data

a.2 After Optimization

In this part, we need to apply some initials into kernel, and then plug kernel into log likelihood for finding the best parameters(which is alpha and length scale in the rational quadratic kernel I defined)

```
obj_value=1e9
initials=[1e-3,1e-2,1e-1,0,1e1,1e2,1e3] #initial params for kernel
best_a = best_ls = 0
for ia in initials: #try and plug all initials into kernel, and then apply
kernel to the likelihood function
    for ils in initials:
        ans = minimize(objective(x,y,beta),x0=[ia,ils],bounds=((1e-
5,1e5),(1e-5,1e5))) #try other bounds
        if ans.fun < obj_value:
            obj_value=ans.fun
            best_a,best_ls = ans.x
```
▲best_a,best_ls stores the optimal parameters for kernel

The objective function I've used for the marginal likelihood in this part:

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

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

```
def objective(x,y,beta):

    def obj(param): #find param that optimize kernel

        k =
kernel(x,x,a=param[0],len_scale=param[1])+(1/beta)*np.identity(len(x))
        return 0.5 * np.log(np.linalg.det(k)) + \
               0.5 * y.reshape(1,-1)@(np.linalg.inv(k)@y.reshape(-1,1)) + \
               0.5 * len(x) * np.log(2*np.pi)

    return obj
```

Now we plug the best parameters we find for the kernel into kernel function and do the prediction again:

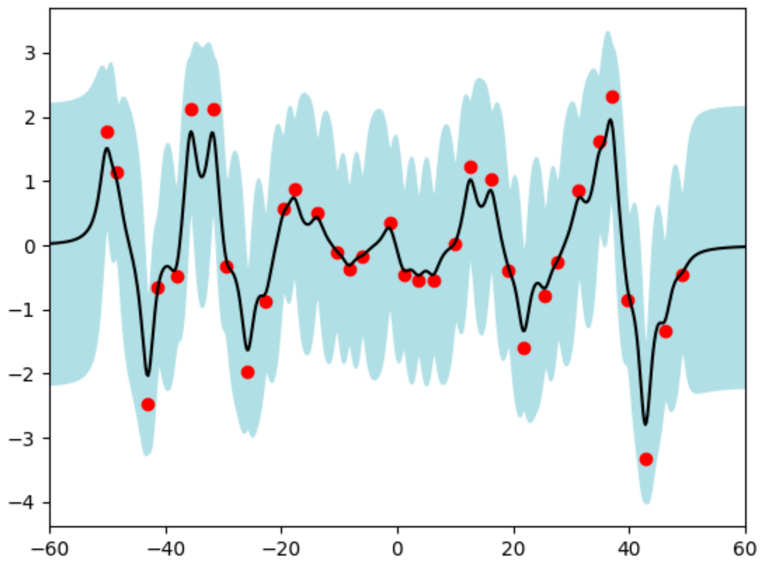
```
k = kernel(x,x,best_a,best_ls)+(1/beta)*np.identity(len(x))

linex = np.linspace(-60,60,num=600)
m_pred,v_pred = prediction(linex,x,y,k,beta,best_a,best_ls)
#we need to find standard deviation for graphing
m_pred = m_pred.reshape(-1)
v_pred = np.sqrt(np.abs(np.diag(v_pred)))
```

■ b. experiments settings and results

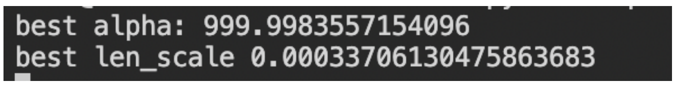
- **Show the figures and the hyperparameters**

○ **Part1 without optimization** Alpha=1, length scale=1

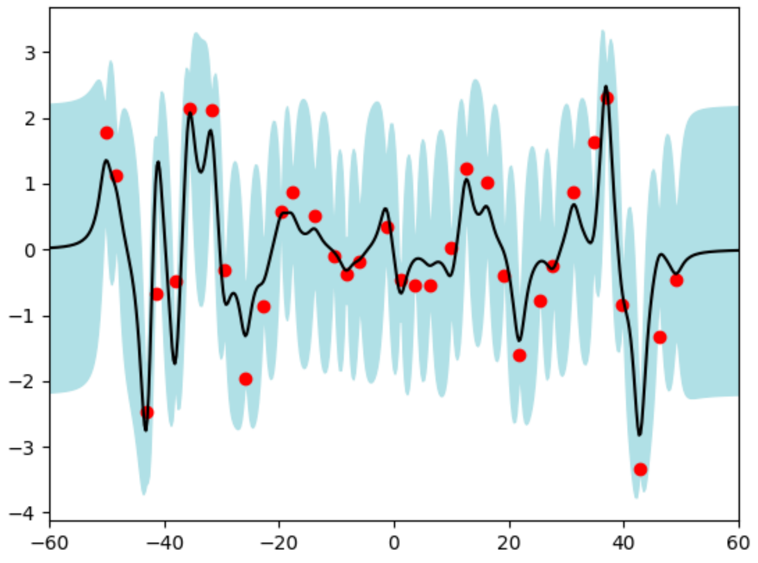


○ **Part2 with optimized kernel**

Alpha, length_scale = [-0.001,-0.01,-0.1,0,0.1,0.01,0.001]



```
best alpha: 999.9983557154096
best len_scale 0.0003370613047586368З
```



■ c. observations and discussion

From the figures above, we can observe that the variance of the prediction is getting smaller after the optimization, which means that the prediction is truly getting better after finding the kernel that best describes the relation between data points.

II. SVM

■ a. code with detailed explanations + results

○ Part1

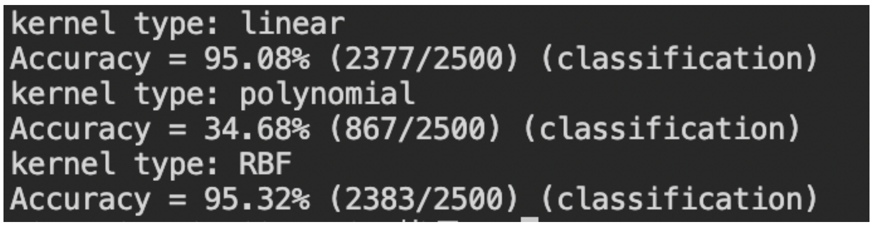
In this part, I compared three different types of kernels (linear,polynomial,RBF) to see how the accuracy differs.

```
kernel = {
    'linear':0,
    'polynomial':1,
    'RBF':2,
}
```

```
if sys.argv[1] == '1': #comparing three kernels
        for k in kernel:
            print(f'kernel type: {k}')
            #svm_train returns three things:model | ACC | MSE
            model = svm_train(ytrain,xtrain,f'-t {kernel[k]} -
q')#transfering the kernel type into svm train function
            #svm_predict returns:(predicted labels, acc+mse, alist of
decision value)
            result = svm_predict(ytest,xtest,model)
```

▲Using for loop to traverse different types of kernel, plug into svm_train. The model will further be used to predict the test data(svm_predict)

The final result of part one:



```
kernel type: linear
Accuracy = 95.08% (2377/2500) (classification)
kernel type: polynomial
Accuracy = 34.68% (867/2500) (classification)
kernel type: RBF
Accuracy = 95.32% (2383/2500) (classification)
```

RBF yields the best accuracy 95.32%.

○ Part2

Doing grid search to find the best combination of parameters using C-SVC

```
elif sys.argv[1] == '2': #use C-SVC with grid search for best model
       search=grid_search(xtrain,ytrain)
       print('The best combination ACC of:{} after
testing:'.format(search))
       model = svm_train(ytrain,xtrain,search)
       result = svm_predict(ytest,xtest,model)#show accuracy of  the
best search
```

- **The kernel functions:**

$$\text{Linear kernel: } k(x,y) = <x,y>$$

$$\text{Polynomial kernel: } k(x,y) = (<x,y> + c)^d$$

$$\text{Gaussian Radial Basis Function kernel (RBF): } k(x,y) = e^{-\frac{\|x-y\|^2}{2\sigma^2}}$$

$$d \in Z^+, \sigma \in \mathcal{R} - \{0\}$$

Where RBF is coded as the formula below in this HW (from libsvm):

polynomial: (gamma*u'*v + coef0)^degree

- **Parameter settings:**

As C-SVC is a soft-margin SVM model, we need to define the penalty for those data points falling within the margin. Hence, we define "cost" C to adjust how hard or soft your large margin classification should be.

```
    costs = [0.001, 0.01, 0.1, 1, 10]
```

▲costs for gird search

In “’polynomial kernel”, we have two more parameters to define:

(gamma*u’*v + coef0)^degree

Where “d” is the degree for polynomial, “g” is gamma and “coef0” is the coefficient for the constant term in the whole function.

In”RBF”, just traverse the same set of gamma as polynomial kernel to find the best combination.

```
gammas=[0.001,0.01,0.1,1] #the parameter for RBF kernel (var)
```

- **The Grid Search:**

```
def grid_search(x,y):
    optimal_search='' #initialized, dummy...
    optimal_acc=0
    #cost 'c' for C-SVC: describes how hard the data "within" the 
margins are penalized
    #the larger the 'c' is, the harder the data will be penalized
    costs = [0.001,0.01,0.1,1,10,10]
    gammas=[0.001,0.01,0.1,1] #the parameter for RBF kernel (var)
    #count=0
    for k in kernel: #traverse different kernels
        for c in costs:
            if k =='linear': #the linear kernel: u’*v
                search = f'-t {kernel[k]} -c {c} -v 3 -q' #-t kernel 
type,-c cost, -v crossvalid k=3
                #count+=1
                optimal_acc,optimal_search = 
compare_ACC(y,x,search,optimal_acc,optimal_search)
            elif k == 'polynomial': #the poly kernel:(gamma*u’*v + 
coef0)^degree -> we need gamma,coef0 and degree parameters
                for g in gammas:
                    for d in range(2,5):
```

```
                    for coef in range(0,4):
                        search = f'-t {kernel[k]} -g {g} -c {c}  -d
{d} -r {coef} -v 3 -q' #-t kernel type,-c cost, -v crossvalid k=3
                        optimal_acc,optimal_search =
compare_ACC(y,x,search,optimal_acc,optimal_search)

          elif k == 'RBF': # the RBF kernel: exp(-gamma*|u-v|^2),
we need gamma parameter
              for g in gammas:
                  search = f'-t {kernel[k]} -g {g} -c {c}  -v 3 -q'
#-t kernel type,-c cost, -v crossvalid k=4
                  optimal_acc,optimal_search =
compare_ACC(y,x,search,optimal_acc,optimal_search)
    print('Best Accuracy with cross-validation
k=4:{}'.format(optimal_acc))
    print('Best Combination:{}'.format(optimal_search))

    return optimal_search
```

-v 3 implies the cross-validation with k = 3

where linear kernel is called as:

`f'-t {kernel[k]} -c {c} -v 3 -q'`

Polynomial kernel is called as:

` f'-t {kernel[k]} -g {g} -c {c}  -d {d} -r {coef} -v 3 -q'`

RBF is called as:

`f'-t {kernel[k]} -g {g} -c {c}  -v 3 -q'`

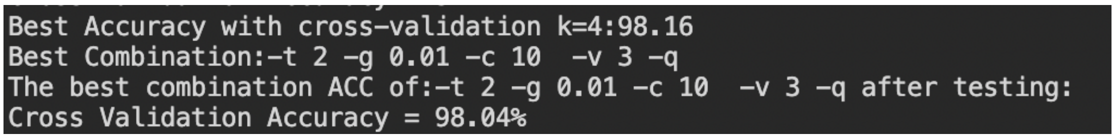
- **Compare Accuracy:**

Apply different combinations of parameters(cost,degree,gamma…) for traning SVM model and find the best set with highest accuracy.

```
def compare_ACC(y,x,search,opt_acc,opt_search):
    print('current input search:{}'.format(search))
    acc = svm_train(y,x,search)
    if acc > opt_acc:
        return acc,search
    else:
        return opt_acc,opt_search
```

▲This function returns the best combination of parameters with specific kernel

- **Result**:



```
Best Accuracy with cross-validation k=4:98.16
Best Combination:-t 2 -g 0.01 -c 10  -v 3 -q
The best combination ACC of:-t 2 -g 0.01 -c 10  -v 3 -q after testing:
Cross Validation Accuracy = 98.04%
```

The best parameter set is:

- Kernel: RBF
- Cost: 10
- Gamma: 0.01
- Degree: N/A
- Accuracy: 98.16%
- Cross-Validation Accuracy after testing: 98.04%

○ Part3

In this part, I combined linear and RBF kernel, and further apply the combination into svm_train for model training.

- **Kernel functions:**

```
def linear_kernel(x1,x2): #finding:k(x,x),k(x,x*)...
    K = x1@x2.T
    return K

def RBF_kernel(x1,x2,g):
    distx1x2 = np.sum(x1**2,axis=1).reshape(-
1,1)+np.sum(x2**2,axis=1)-2*x1@x2.T #expand:|x1-x2|^2
    k = np.exp((-g*distx1x2)) # exp(-gamma*|u-v|^2)
```

```
●       return k
```

- **Kernel combinations:**

```
    elif sys.argv[1] == '3': #do linear + RBF
        linear_kxx = linear_kernel(xtrain,xtrain)#k(x,x)for linear
        #default gamma ->1/features = 1/28*28 = 1/784pixels
        rbf_kxx = RBF_kernel(xtrain,xtrain,1/784)
        #while we are training, we need to throw the data that have
already passed into feature space(kernel)into svm model! since we
customized the kernel function
        linear_kxxs = linear_kernel(xtrain,xtest).T #for testing! k(x,x*)
        rbf_kxxs = RBF_kernel(xtrain,xtest,1/784).T

        #now stacking 2 kernels
        k_stack = np.hstack((np.array(1,5001).reshape((-
1,1)),linear_kxx+rbf_kxx)) #combining 2 kernels for training,store them
for each training data into matrix
        k_stack_s = np.hstack((np.array(1,2501).reshape((-
1,1)),linear_kxxs+rbf_kxxs)) #combining 2 kernels for testing

        search = '-t 4 -q' # 4 is for "precomputed kernel"
        model = svm_train(ytrain,k_stack,search) #kernel combination for
training
        result = svm_predict(ytest,k_stack_s,model) #kernel combination
for testing
```

Where we compute k(xtrain,xtrain) first for model training, that is, projecting the training data into feature space and then apply it into the svm model:

```
linear_kxx = linear_kernel(xtrain,xtrain)

rbf_kxx = RBF_kernel(xtrain,xtrain,1/784)
```

1/784 is the default value for gamma (1/number of features)

After computing k(xtrain,xtrain), we calculate k(xtrain,xtest) and apply it into svm_predict for testing:

```
linear_kxxs = linear_kernel(xtrain,xtest).T

rbf_kxxs = RBF_kernel(xtrain,xtest,1/784).T
```

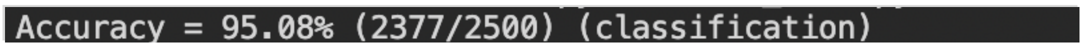
After finding two kernels we need, combine them together using hstack:

```
k_stack = np.hstack((np.array(1,5001).reshape((-
1,1)),linear_kxx+rbf_kxx)) #combining 2 kernels for training,store them
for each training data into matrix
 k_stack_s =
np.hstack((np.array(1,2501).reshape((1,1)),linear_kxxs+rbf_kxxs))
#combining 2 kernels for testing
```

After combining the kernels, plug it into svm_train, svm_predict respectively. Search = '-t 4 -q' where 4 specifies precomputed kernel.

```
search = '-t 4 -q' # 4 is for "precomputed kernel"
       model = svm_train(ytrain,k_stack,search) #kernel combination for
training
       result = svm_predict(ytest,k_stack_s,model) #kernel combination
for testing
```

- **Results:**



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

■ b. observations and discussion

Part1:

RBF kernel for SVM has the best accuracy: 95.32%

RBF is the squared exponential kernel which is nonparametric, generally more flexible than the linear or polynomial kernels. If we have unlimited data, RBF is always better as it can represent more and more complex relationships between data points.

Part2:

The grid search gives the result that when cost for C-SVC = 10 and gamma = 0.01, the RBF kernel SVM yields the best accuracy: 98.16%, which is the same as part 1 that RBF gives the best model accuracy.

Part3:

After the kernel combination, the accuracy is the same as linear kernel in part 1. This result might show that the linear kernel dominates the model performance, as in linear kernel the data points multiply each other (x1@x2) whereas the RBF kernel is calculating |x1-x2|^2. Hence, the accuracy tends to align with the SVM model using linear kernel. Maybe reweighting the weights of kernels will lead to different results (ex. 0.1*linear kernel + RBF kernel)