Machine Learning Homework 5

310554028 趙紹安

I. Gaussian Process

1. Code

(1) Kernel:

Definition of rational quadratic kernel is below:

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

So my code is written as:

```
def rationalQudraticKernel(x1, x2, sigma=1., alpha=1., length=1.):
    return (sigma**2) * (1 + (x1 - x2)**2/(2 * alpha * length**2))**(-alpha)
```

(2) Gaussian process:

First it needs to calculate covariance (C). k represents kernel distance between training data and testing data, and k* is kernel distance between testing data with noise. Beta vary noise.

With above parameters, now we can calculate mean and var.

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

```
def covariance(X, beta, sigma, alpha, length):
    C = np.zeros((X.shape[0], X.shape[0]))
    for i in range(X.shape[0]):
        for j in range(X.shape[0]):
            C[i][j] = rationalQudraticKernel(X[i], X[j], sigma, alpha, length)
        C[i][i] += 1/beta
    return C
```

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

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

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

```
C = covariance(X, beta, sigma, alpha, length)

for i in range(points):
    K = np.zeros((X.shape[0], 1))
    for j in range(X.shape[0]):
        K[j] = rationalQudraticKernel(X[j], x_test[i], sigma, alpha, length)
    mean[i] = K.T.dot(np.linalg.inv(C)).dot(Y)
    k_s = rationalQudraticKernel(x_test[i], x_test[i], sigma, alpha, length) + 1/beta
    var[i] = k_s - K.T.dot(np.linalg.inv(C)).dot(K)
```

(3) Optimization:

Now we need to optimize parameters. Here we use scipy.optimize.minize for finding best parameters. So we use “negative” log likelihood to achieve this goal.

```python
def negativeMarginalLikelihood(theta, X, Y, beta):
    theta = theta.reshape(len(theta), 1)
    C = covariance(X, beta, theta[0], theta[1], theta[2])
    likelihood = 0.5 * (np.log(np.linalg.det(C)) + Y.T.dot(np.linalg.inv(C)).dot(Y) + np.log(2 * np.pi) * X.shape[0])
    #likelihood = np.log(np.linalg.det(C)) * 0.5
    #likelihood += Y.T.dot(np.linalg.inv(C)).dot(Y) * 0.5
    #likelihood += np.log(2 * np.pi) * X.shape[0] * 0.5
    return likelihood[0]
```

```python
# Optimize parameters
opt = minimize(negativeMarginalLikelihood, [sigma, alpha, length], args=(X, Y, beta))

sigma = opt.x[0]
alpha = opt.x[1]
length = opt.x[2]
```

(4) 95% Confidence interval:

Z value of 95% confidence is 1.96, I use this value to calculate 95% confidence interval.

```python
fig = plt.figure()
interval = 1.96 * (var ** 0.5) # 95% Confidence Interval

ax = fig.add_subplot(1, 1, 1)
ax.set_title(f's:{sigma:.4f}, alpha:{alpha:.4f}, length:{length:.4f}')
ax.plot(X, Y, "k.")
ax.plot(x_test, mean, "r-")
ax.fill_between(x_test, mean + interval, mean - interval, color='cyan')
ax.set_xlim([-60, 60])
ax.set_ylim([-5, 5])
plt.show()
```

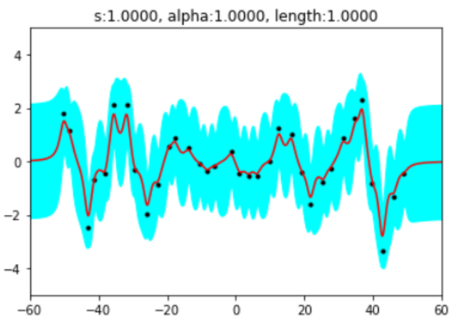
(5) Task 1 and Task 2:

They both call gaussian process, but task 2 need to optimize parameters first.

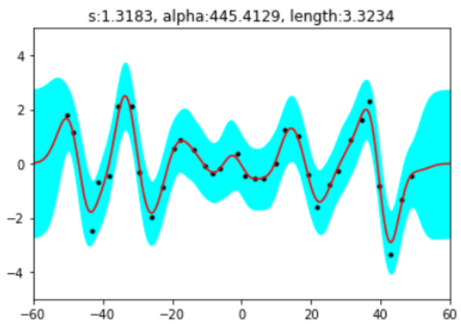
```python
GaussianProcess(X, Y, beta, sigma, alpha, length)

# Optimize parameters
opt = minimize(negativeMarginalLikelihood, [sigma, alpha, length], args=(X, Y, beta))

sigma = opt.x[0]
alpha = opt.x[1]
length = opt.x[2]

GaussianProcess(X, Y, beta, sigma, alpha, length)
```

2. Experiments

(1) Task 1 – beta=5, sigma=1, alpha=1, length=1



(2) Task 2 – beta=5, sigma=1.3183, alpha=445.4129, length=3.3234



3. Observations and Discussion
    1. (1) Task 2 is better than task 1
    2. (2) Sigma here doesn't affect result much. You can see that after optimization, sigma of task 2 is still closed to sigma of task 1.
    3. (3) Gaussian process can't predict well outside training data, it can be observed by result figure easily.
    4. (4) Parameters have great influence on gaussian process, so choose parameters carefully.

II. SVM

1. Code

(1) Load data:

First load training and testing data from csv

```
def loadData(folderpath='data/'):
    train_imgs = []
    train_lbs = []
    test_imgs = []
    test_lbs = []
    with open(folderpath + 'X_train.csv') as f:
        line = f.readline()
        while (line):
            train_imgs.append([float(i) for i in line.strip().split(',')])
            line = f.readline()
    with open(folderpath + 'Y_train.csv') as f:
        line = f.readline()
        while (line):
            train_lbs.append(float(line.strip()))
            line = f.readline()
    with open(folderpath + 'X_test.csv') as f:
        line = f.readline()
        while (line):
            test_imgs.append([float(i) for i in line.strip().split(',')])
            line = f.readline()
    with open(folderpath + 'Y_test.csv') as f:
        line = f.readline()
        while (line):
            test_lbs.append(float(line.strip()))
            line = f.readline()
    return train_imgs, train_lbs, test_imgs, test_lbs
```

(2) Task:

You need to set task first. 1 for task1, and so on.

```
if __name__ == '__main__':
    train_imgs, train_lbs, te

    task = 1
```

(3) Task 1:

In this task, it need to use different kernel functions and compare their performance.

svm_train: train a model for given parameter and data.

svm_predict: use model to predict testing data.

Parameter:

t -> kernel functions, 0 for linear, 1 for polynomial, 2 for RBF

d -> degree, used in polynomial kernel.

```
def svm(X, Y, X_test, Y_test, para):
    m =  svm_train(Y, X, para)
    p_labs, p_acc, p_vals = svm_predict(Y_test, X_test, m)
    return p_acc
```

```
if (task == 1):
    print('linear:')
    svm(train_imgs, train_lbs, test_imgs, test_lbs, f'-t 0 -d 2 -q')
    print('polynomial:')
    svm(train_imgs, train_lbs, test_imgs, test_lbs, f'-t 1 -d 2 -q')
    print('radial basis function:')
    svm(train_imgs, train_lbs, test_imgs, test_lbs, f'-t 2 -d 2 -q')
```

(4) Task 2:

In this task, it need to use grid search to find the best parameters.

For each kernel type, it needs different parameter.

Parameter:

t -> kernel type

c -> cost, set C of C-SVC

g -> gamma, default is 1/num_features

d -> degree

r -> coef0, default is 0

v -> k-fold cross-valid, here use 3

Here I find best parameter for different kernel type individually. And show their performance on testing data.

```python
def gridSearch(X, Y, kernelType):
    best_acc = 0
    best_para = f''
    costs = [0.001, 0.01, 0.1, 1, 10]
    # gammas = [0.001, 0.01, 0.1, 1]
    gammas = [1/784, 0.01, 0.1, 1]
    degrees = [2, 3, 4]
    coef0s = [0, 1, 2]
    count = 0
    start = time.time()
    if (kernelType == 0):
        for cost in costs:
            para = f'-t {kernelType} -c {cost} -q -v 3'
            count += 1
            best_acc, best_para = compare(X, Y, para, best_acc, best_para)
    elif (kernelType == 1):
        for cost in costs:
            for gamma in gammas:
                for degree in degrees:
                    for coef0 in coef0s:
                        para = f'-t {kernelType} -c {cost} -g {gamma} -d {degree} -r {coef0} -q -v 3'
                        count += 1
                        best_acc, best_para = compare(X, Y, para, best_acc, best_para)
    elif (kernelType == 2):
        for cost in costs:
            for gamma in gammas:
                para = f'-t {kernelType} -c {cost} -g {gamma} -q -v 3'
                count += 1
                best_acc, best_para = compare(X, Y, para, best_acc, best_para)
    end = time.time()
    print('\n##################################################')
    print(f'Total time: {(end - start):.2f} s')
    print(f'Total combinations: {count}')
    print(f'Optimal cross validation accuracy: {best_acc}')
    print(f'Optimal option: {best_para}')
    print('##################################################\n')
    return best_acc, best_para
```

```python
elif (task == 2):
    best_para = f''
    print('linear:')
    l_acc, l_para = gridSearch(train_imgs, train_lbs, 0)
    print(f'linear cross-valid: acc:{l_acc}, para:{l_para}')
    best_para = l_para
    best_para = best_para.replace(best_para[-5:], '')
    svm(train_imgs, train_lbs, test_imgs, test_lbs, best_para)
    print('polynomial:')
    p_acc, p_para = gridSearch(train_imgs, train_lbs, 1)
    print(f'polynomial cross-valid: acc:{p_acc}, para:{p_para}')
    best_para = p_para
    best_para = best_para.replace(best_para[-5:], '')
    svm(train_imgs, train_lbs, test_imgs, test_lbs, best_para)
    print('radial basis function:')
    r_acc, r_para = gridSearch(train_imgs, train_lbs, 2)
    print(f'RBF cross-valid: acc:{r_acc}, para:{r_para}')
    best_para = r_para
    best_para = best_para.replace(best_para[-5:], '')
    svm(train_imgs, train_lbs, test_imgs, test_lbs, best_para)
```

(5) Task 3:

In this task, it uses linear kernel + RBF kernel together. So I need to define kernel function by myself.

Here parameter used in svm_train, it needs to set t as 4, which allows user-defined kernel.

```python
def linearKernel(X1, X2):
    return np.dot(X1, X2.T)


def RBFKernel(X1, X2, gamma):
    dist = np.sum(X1 ** 2, axis=1).reshape(-1, 1) + np.sum(X2 ** 2, axis=1) - 2 * np.dot(X1, X2.T)
    return np.exp(-gamma * dist)
```

```python
elif (task == 3):
    gamma = 1/len(train_imgs[0])
    imgs1 = np.array(train_imgs)
    imgs2 = np.array(test_imgs)
    print(gamma)
    train_kernel = linearKernel(imgs1, imgs1) + RBFKernel(imgs1, imgs1, gamma)
    test_kernel = linearKernel(imgs2, imgs2) + RBFKernel(imgs2, imgs2, gamma)
    train_kernel = np.hstack((np.arange(1, len(train_lbs)+1).reshape(-1, 1), train_kernel))
    test_kernel = np.hstack((np.arange(1, len(test_lbs)+1).reshape(-1, 1), test_kernel))
    m = svm_train(train_lbs, train_kernel, '-t 4')
    labs, acc, vals = svm_predict(test_lbs, test_kernel, m)
```

2. Experiments
   (1) Task 1

Kernel	Testing acc(%)	Parameter
Linear	95.08	
Polynomial	88.24	-d 2
RBF	95.32	

   (2) Task 2

Kernel	Cross-valid acc(%)	Testing acc(%)	Parameter
Linear	96.72	95.96	-c 0.01
Polynomial	98.14	97.96	-c 0.1 -g 0.1 -d 3 -r 2
RBF	98.0	98.2	-c 10 -g 0.01

   (3) Task 3

Kernel	Testing acc(%)	Parameter
Linear + RBF	24.44	

3. Observations and Discussion
   (1) In task 1, RBF has best performance because it can map data into infinite dimension space.
   (2) In task 2, because of limited search of parameter, RBF doesn’t get first place in cross-valid acc. While it still has best performance in testing acc.
   (3) For task 1 and task 2, we can see that parameter has great influence on performance.
   (4) Because RBF is strong, it’s more popular kernel function used in SVM.
   (5) Linear kernel function is fastest above other kernel function.

(6) Polynomial kernel function is time-consuming since it has more parameters to deal with.
(7) In task 3, linear + RBF kernel function doesn't perform well. Maybe with grid search for some parameter, it will have normal performance.