Machine Learning Homework 5 Technical Report

tags: `Course1082` `IOC5191`

> 網工所 0856523 許尹睿

Gaussian Process

Extract data

using following code to transform the input.data into a 2-D numpy array. The shape of this numpy array is (34, 2)

```
def extract_data():
    data = open('./input.data')
    data_points = []
    for row in data:
        data_point = [float(value) for value in row.split()]
        data_points.append(data_point)

    data_points = np.array(data_points)
    return data_points

data_points = extract_data()
```

Initialize the parameters and covariance matrix

initialize the parameters used in rational quadratic kernel and using the input data with this kernel to compute the covariance matrix C

```
def make_C_matrix(data_points, beta, amplitude, lengthscale, scale_mixture):
    C = []
    num = len(data_points)
    for n in range(num):
        row = []
        for m in range(num):
            kernel = rational_quadratic_kernel(data_points[n][0],
data_points[m][0], amplitude, lengthscale, scale_mixture)
            if n == m:
                kernel = kernel + (1/beta)
            row.append(kernel)
        C.append(row)
    C = np.array(C)
```

```
    return C

# using rational quadratic kernel
def rational_quadratic_kernel(x_n, x_m, amplitude, lengthscale,
scale_mixture):
    return amplitude * ((1 + (((x_n - x_m) ** 2) / (2 * scale_mixture *
(lengthscale ** 2)))) ** (-1 * scale_mixture))

beta = 5
amplitude = 1
lengthscale = 1
scale_mixture = 1

C = make_C_matrix(data_points, beta, amplitude, lengthscale, scale_mixture)
```

make_C_matrix accepts the input data and necessary parameter for rational quadratic kernel and returns the covariance matrix C
rational_quadratic_kernel using following formula to implements the rational quadratic kernel

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

Create test data and make prediction

```
# implement Gaussian Process Regression
def gp_regressiong(test, data_points, C, beta, amplitude, lengthscale,
scale_mixture):
    result = []
    for x_star in test:
        kernels = []
        for x_index in range(len(data_points)):
            kernel = rational_quadratic_kernel(data_points[x_index][0],
x_star, amplitude, lengthscale, scale_mixture)
            kernels.append(kernel)
        kernels = np.array(kernels).reshape(34,1)
        mean = np.dot(np.transpose(kernels), np.dot(np.linalg.inv(C),
data_points[:, 1]))
        k_star = rational_quadratic_kernel(x_star, x_star, amplitude,
lengthscale, scale_mixture) + (1/beta)
        var = k_star - np.dot(np.transpose(kernels), np.dot(np.linalg.inv(C),
kernels))
        result.append([mean[0], var[0][0]])
    result = np.array(result)
```

```
    return result

test = np.linspace(-60, 60, 1000)
result = gp_regressiong(test, data_points, C, beta, amplitude, lengthscale,
scale_mixture)
```

gp_regressiong receives input data, test data, and necessary parameters for rational quadratic kernel to compute the **mean** and **variance** of each test data

Plot the result

plot the input data, prediction result and 95% confidence interval

```
def plot_result(test, result, data_points, state):
    if state == 'origin':
        plt.subplot(2, 1, 1)
        plt.title('Origin')
    elif state == 'optimal':
        plt.subplot(2, 1, 2)
        plt.title('Optimal')
    plt.xlim((-60, 60))
    plt.fill_between(x=test, y1=result[:, 0] + (1.96 * np.sqrt(result[:, 1])),
y2=result[:, 0] - (1.96 * np.sqrt(result[:, 1])), color="gray")
    plt.scatter(data_points[:, 0], data_points[:, 1], s=15, c='blue')
    plt.plot(test, result[:, 0], linestyle='-', color='red')

plot_result(test, result, data_points, 'origin')
```

plot_result will mark the input data as **blue points**, prediction result as the **red line** and the range of prediction under 95% confidence interval as the **gray region**
the way to get 95% confidence interval is from [this website]

Optimize the parameters for kernel

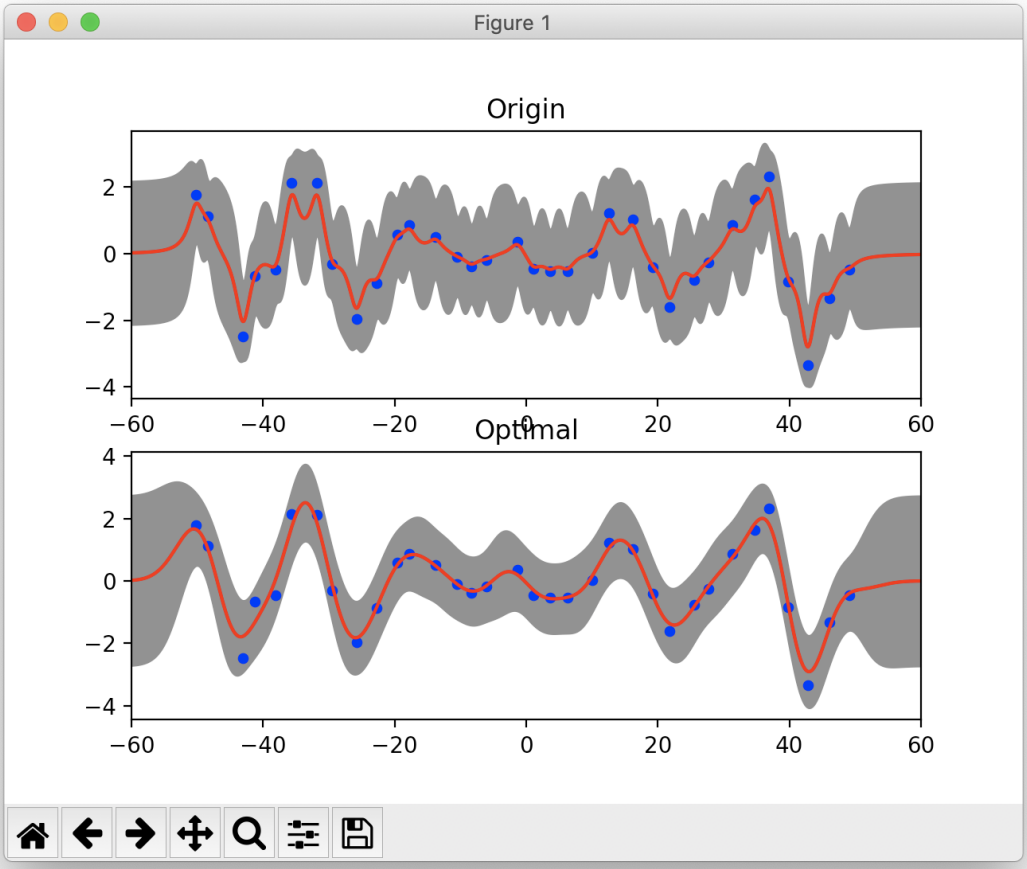
optimize the parameters used in rational quadratic kernel, using the new parameters to compute the covariance matrix C, predict the test data and plot the result again

```
# using Sequential Least SQuares Programming (SLSQP) for optimization
def SLSQP(theta, data_points, beta):
    C = make_C_matrix(data_points, beta, theta[0], theta[1], theta[2])
    return 0.5 * np.log(np.linalg.det(C)) + 0.5 *
np.dot(np.transpose(data_points[:, 1]), np.dot(np.linalg.inv(C),
data_points[:, 1])) + 0.5 * len(data_points) * np.log(2 * np.pi)

theta = np.array([amplitude, lengthscale, scale_mixture])
optimal = minimize(SLSQP, theta, args=(data_points, beta), method='SLSQP')

[amplitude, lengthscale, scale_mixture] = optimal.x
```

```
C = make_C_matrix(data_points, beta, amplitude, lengthscale, scale_mixture)
result = gp_regressiong(test, data_points, C, beta, amplitude, lengthscale,
scale_mixture)
plot_result(test, result, data_points, 'optimal')

plt.show()
```

using **Sequential Least SQuares Programming(SLSQP)** method of **scipy.optimize.minimize** to minimize negative marginal log-likelihood.
the objective function **SLSQP** is to calculate the negative marginal log-likelihood.

- the visualization of result is the following figure:



SVM

Extract data

using following code to transform the Train and Test data into numpy arrays. The shape of X_train is (5000, 784). The shape of X_test is (2500, 784). The shape of Y_train is (5000,). The shape of Y_test is (2500,).

```
def extract_data(file):
```

```
    csvfile = open(file)
    data = csv.reader(csvfile)
    data_points = []
    for row in data:
        data_points.append(row)

    data_points = np.array(data_points).astype(float)
    return data_points

X_train = extract_data('./X_train.csv')
X_test = extract_data('./X_test.csv')
Y_train = extract_data('./Y_train.csv').reshape(5000)
Y_test = extract_data('./Y_test.csv').reshape(2500)
```

Train the model and do prediction

```
# linear kernel functions
model_linear = svm_train(Y_train, X_train, '-t 0 -q')
predict_linear = svm_predict(Y_test, X_test, model_linear)

# polynomial kernel functions
model_polynomial = svm_train(Y_train, X_train, '-t 1 -q')
predict_polynomial = svm_predict(Y_test, X_test, model_polynomial)

# RBF kernel functions
model_rbf = svm_train(Y_train, X_train, '-t 2 -q')
predict_rbf = svm_predict(Y_test, X_test, model_rbf)
```

the following table is the accuracy of prediction by using different kernel functions in training:

method	**accuracy**
linear	95.08% (2377/2500)
polynomial	34.68% (867/2500)
RBF	95.32% (2383/2500)

RBF kernel get best result while polynomial kernel get worst result

Grid Search

```
# implement Grid Search
def grid_search(method):
    gamma = np.logspace(-9, 3, 13)
    Cost = np.logspace(-2, 10, 13)
    parameters = []
    for g in gamma:
```

```
        for c in Cost:
            if method == 'linear':
                model = svm_train(Y_train, X_train, f'-t 0 -q -g {g} -c {c} -v
5')
            elif method == 'polynomial':
                model = svm_train(Y_train, X_train, f'-t 1 -q -g {g} -c {c} -v
5')
            elif method == 'rbf':
                model = svm_train(Y_train, X_train, f'-t 2 -q -g {g} -c {c} -v
5')

            parameters.append([model, g, c])

    best_parameter = max(parameters)
    print (f'best score {best_parameter[0]}% for {method} with gamma
{best_parameter[1]} and cost {best_parameter[2]}')

    if method == 'linear':
        model = svm_train(Y_train, X_train, f'-t 0 -q -g {best_parameter[1]} -
c {best_parameter[2]}')
    elif method == 'polynomial':
        model = svm_train(Y_train, X_train, f'-t 1 -q -g {best_parameter[1]} -
c {best_parameter[2]}')
    elif method == 'rbf':
        model = svm_train(Y_train, X_train, f'-t 2 -q -g {best_parameter[1]} -
c {best_parameter[2]}')
    predict = svm_predict(Y_test, X_test, model)

    return best_parameter

grid_linear = grid_search('linear')
grid_polynomial = grid_search('polynomial')
grid_rbf = grid_search('rbf')
```

grid_search uses the grid search to find the best gamma and cost for different kernel funciton in C-SVC, the result is listed in the following table:

method	gamma	cost	accuracy
linear	0.001	0.01	95.96 % (2399/2500)
polynomial	10.0	0.1	97.48% (2437/2500)
RBF	0.01	10000000000.0	98.16% (2454/2500)

the reasonable range of gamma and cost is from this website.

Construct user-defined kernel

```
# Using linear kernel & RBF kernel together
def linear_rbf_kernel(X, Train, gamma):
    Size = len(X)
    linear_kernel = np.dot(X, np.transpose(Train))
    rbf_kernel = np.exp(gamma * cdist(X, Train, 'sqeuclidean'))
    kernel = linear_kernel + rbf_kernel
    kernel = np.hstack((np.arange(1, Size+1).reshape(-1,1), kernel))

    return kernel

# doing the User Define Kernel
gamma = -1/4
X_train_kernel = linear_rbf_kernel(X_train, X_train, gamma)
linear_rbf_model = svm_train(Y_train, X_train_kernel, '-t 4 -q')

X_test_kernel = linear_rbf_kernel(X_test, X_train, gamma)
linear_rbf_predict = svm_predict(Y_test, X_test_kernel, linear_rbf_model)
```

linear_rbf_kernel uses linear kernel and RBF kernel together to construct a new kernel function.
The shape of X_train_kernel is (5000, 5001) and the shape of X_test_kernel is (2500, 5001).
using this kernel to train and predict will get the accuracy of **95.64% (2391/2500)**, which is better than the other kernel functions
I learn the libsvm's usage from this repo.