HW5 311552055 蕭育泓

1. Gaussian Process

Part a. code explanations

```
def rational_quadratic_kernel(Xa, Xb, variance, alpha, l):
    distance = np.empty((len(Xa), len(Xb)))
    for i in range(len(Xa)):
        for j in range(len(Xb)):
            distance[i][j] = (Xa[i] - Xb[j])**2
    kernel = variance * ((1+distance/2*alpha*l**2)**(-1*alpha))
    # print(kernel)
    return kernel
```

$$k(x_a, x_b) = \sigma^2\left(1 + \frac{\|x_a - x_b\|^2}{2\alpha\ell^2}\right)^{-\alpha}$$ ← Picture1

We use a rational quadratic kernel to compute similarities between different points. The rational quadratic kernel function is above (**Picture1**) and the parameters Xa(n*1) and Xb(m*1) are the input points, variance(σ^2) , alpha(α) and l(l). First, we want to know the distance between each of Xa and Xb, so I set a -empty matrix of size n*m and use the formula to calculate the kernel.

For part1, normal parameters (beta = 5, length_scale=1, variance=1, alpha=1)

```
data_X, data_Y = load()
beta = 5
length_scale = 1
variance = 1
alpha = 1
mean, var, x_star = guassian(data_X, data_Y, beta, variance, alpha, length_scale)
```

```
def load():
    f = open("HW5\input.data", "r", encoding='utf-8')
    X = []
    Y = []
    for line in f:
        x, y = line.split(' ')
        X.append(float(x))
        Y.append(float(y))
    X = np.array(X, dtype=np.float64).reshape(-1, 1)
    Y = np.array(Y, dtype=np.float64).reshape(-1, 1)
    return X, Y
```

First, loading the 34 * 2 data to data_X(34*1) and data_Y(34*1), there are the input points.

```
def guassian(X, Y, beta, variance, alpha, length_scale):
    kernel = rational_quadratic_kernel(X, X, variance, alpha, length_scale)
    C = kernel + np.eye(len(X))/beta
    C_inv = np.linalg.inv(C)
    X_star = np.linspace(-60, 60, 1000).reshape(-1, 1)

    kernel_x_xstar = rational_quadratic_kernel(X, X_star, variance, alpha, length_scale)
    mean_x_star = kernel_x_xstar.T.dot(C_inv).dot(Y)
    kernel_star = rational_quadratic_kernel(X_star, X_star, variance, alpha, length_scale) + np.eye(len(X_star))/beta
    variance_x_star = kernel_star - kernel_x_xstar.T.dot(C_inv).dot(kernel_x_xstar)
    return mean_x_star, variance_x_star, X_star
```

$$\mathbf{C}(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm}$$ ← Picture2

$$\mu(\mathbf{x}^*) = k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1}\mathbf{y}$$
$$\sigma^2(\mathbf{x}^*) = k^* - k(\mathbf{x}, \mathbf{x}^*)^\top \mathbf{C}^{-1}\, k(\mathbf{x}, \mathbf{x}^*)$$
$$k^* = k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1}$$ ← Picture3

The function-guassian finds out the covariance matrix(C) and it's inverse(C_inv) using the formula above(Picture2). The predict part wants to predict the means and variances of y for the points in range [-60,60] (I set the 1000 points). The predict formula is above(Picture3). Finally, function-guassian returns the 1000 points(X_star), mean(mean_x_star) and variance (variance_x_star) to visualize.

```
def visualization(X, Y, mean, var, x_star, title, beta, alpha, sigma, length_scale):
    plt.scatter(X, Y, color='black', s=10, zorder=15)
    interval = 2 * np.sqrt(var)
    # print("inter : ", interval)
    plt.plot(x_star, mean + interval, color='pink', zorder=5)
    plt.plot(x_star, mean - interval, color='pink', zorder=5)
    plt.plot(x_star, mean, color='blue', zorder=10)
    # x_star = x_star.reshape(1, -1)
    # print(x_star.shape)
    plt.xlim(-60, 60)
    plt.title(
        f'{title} beta:{beta},sigma: {sigma:.4f}, alpha: {alpha:.4f}, length scale: {length_scale:.f}')
    plt.show()
```

In the function-visualization, the parameters are X(data_X), Y(data_Y), mean(mean_x_star), var(variance_x_star), x_star(X_star) and hyper-parameters. Because of we need to mark the 95% confidence interval which is 2 times of standard deviation(interval). Finally, using X and Y to draw the input points, using x_star, mean and interval to draw the 95% confidence interval and print the hyper-parameters on the title.

For part2, the difference between to the part1 are the hyper-parameters.
We want to optimize the kernel parameters(variance, alpha, length_scale), I using the scipy.optimize.minimize to optimize the parameters.
Set the bounds = (10^{-8} ,10^{8}) for each parameters.

```
opt = minimize(
    fun, [variance, alpha, length_scale],
    args=(data_X, data_Y, beta),
    bounds=((1e-8, 1e8), (1e-8, 1e8), (1e-8, 1e8)))

variance_opt = opt.x[0]
alpha_opt = opt.x[1]
length_scale_opt = opt.x[2]
```

```
def fun(theta, data_X, data_Y, beta):
    theta = theta.ravel()
    # print(theta)
    kernel = rational_quadratic_kernel(
        data_X, data_X, theta[0], theta[1], theta[2])
    C = kernel + np.eye(len(data_X))/beta
    C_inv = np.linalg.inv(C)
    negative = 0.5 * np.log(2 * np.pi)
    negative += 0.5 * np.log(np.linalg.det(C))
    negative += 0.5 * (data_Y.T.dot(C_inv).dot(data_Y))
    return negative
```

Picture4

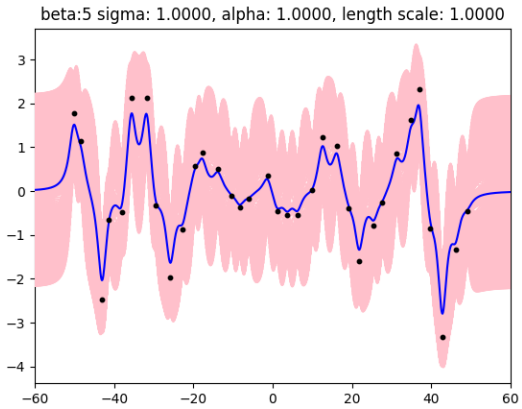
$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2}\ln |\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln (2\pi)$$

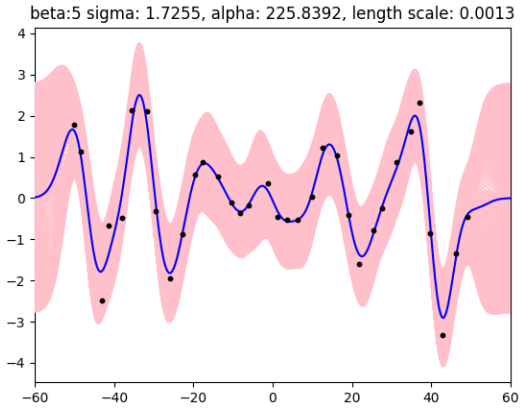
The function-fun is negative marginal log-likelihood function.
The negative marginal log-likelihood function is the opposite number of the formula in above(Picture4). After the optimization(function-opt), we get the optimized parameters(variance_opt, alpha_opt, length_scale_opt).

```
mean, var, x_star = guassian(
    data_X, data_Y, beta, variance_opt, alpha_opt, length_scale_opt)
```

Finally, using the function-guassian to find out the mean and variance and to visualize the results, what I explain how to do in the part1.

Part b. Results





Part c. observations and discussion

After the optimization, the curve and interval are more tidy and smooth. The curve is more fitting the data, the interval is more stable and clearly less than or equal than non-optimization, except the boundary of the interval is bigger than non-optimization, it is because of there is no any data there.

2. SVM on MNIST dataset

Part a. code explanations

Part1 code

```
def compare_diff_performance(y_train, x_train, x_test, y_test):
    prob = svm_problem(y_train, x_train)
    # t - kernel_type l(0) p(1) r(2) / s - svm_type C-SVC(0)
    linear_param = svm_parameter('-t 0 -s 0 -q')
    polynomial_param = svm_parameter('-t 1 -s 0 -q')
    RBF_param = svm_parameter('-t 2 -s 0 -q')

    startTime = time.time()
    linear_m = svm_train(prob, linear_param)
    predict(linear_m, x_test, y_test, "linear")
    endTime = time.time()
    print(f'linear total time: {endTime - startTime}\n')

    startTime = time.time()
    polynomial_m = svm_train(prob, polynomial_param)
    predict(polynomial_m, x_test, y_test, "polynomial")
    endTime = time.time()
    print(f'polynomial total time: {endTime - startTime}\n')

    startTime = time.time()
    RBF_m = svm_train(prob, RBF_param)
    predict(RBF_m, x_test, y_test, "RBF")
    endTime = time.time()
    print(f'RBF total time: {endTime - startTime}\n')
    # return linear_m,polynomial_m,RBF_m
```

```
def predict(model, x_test, y_test, mode):
    if mode == "linear":
        print("linear kernel performance")
        svm_predict(y_test, x_test, model)
    if mode == "polynomial":
        print("polynomial kernel performance")
        svm_predict(y_test, x_test, model)
    if mode == "RBF":
        print("RBF kernel performance")
        svm_predict(y_test, x_test, model)
```

We need to compare three different kernels(linear, polynomial, and RBF kernels). In the function-compare_diff_performance, I use the LIBSVM library to fit the model.

Using svm_problem to fitting the training data with label.

Using svm_parameter to set the different parameters to each of the kernel. The parameter -t determine which of the kernel is using(0:linear, 1:poly, 2:rbf).

Using svm_train to generate the model with different kernel and parameter.

Using svm_predict to predict the test case with each of model.

Part2 code

```
kernel = ["linear", "polynomial", "RBF"]
# parameters
costs = [0.1, 1, 10, 100]
gammas = [1/784, 1, 0.1, 0.01]
degrees = [0, 1, 2, 3]
coef0s = [0, 1, 2, 3]
```

```
if kernel[i] == "linear":
    best_parameter = ''
    best_accuracy = 0.0
    best_kernel = ''
    print("-----linear------")
    for cost in costs:
        parameter = f'-t 0 -s 0 -c {cost} -v 5 -q'
        print(f"parameters: {parameter}")
        linear_param = svm_parameter(parameter)
        linear_m_acc = svm_train(prob, linear_param)
        # print(linear_m_acc)
        best_parameter, best_accuracy, best_kernel = compare_acc(
            best_parameter, best_accuracy, parameter, linear_m_acc, best_kernel, kernel[i])
```

```
if kernel[i] == "polynomial":
    print("\n-----polynomial------")
    best_parameter = ''
    best_accuracy = 0.0
    best_kernel = ''
    for cost in costs:
        for gamma in gammas:
            for degree in degrees:
                for coef0 in coef0s:
                    parameter = f'-t 1 -s 0 -c {cost} -g {gamma} -d {degree} -r {coef0} -v 5 -q'
                    print(f"parameters: {parameter}")
                    polynomial_param = svm_parameter(parameter)
                    polynomial_m_acc = svm_train(
                        prob, polynomial_param)
                    # print(polynomial_m_acc)
                    best_parameter, best_accuracy, best_kernel = compare_acc(
                        best_parameter, best_accuracy, parameter, polynomial_m_acc, best_kernel, kernel[i])
```

```
if kernel[i] == "RBF":
    print("\n-----radial basis function------")
    best_parameter = ''
    best_accuracy = 0.0
    best_kernel = ''
    for cost in costs:
        for gamma in gammas:
            parameter = f'-t 2 -s 0 -c {cost} -g {gamma} -v 5 -q'
            print(f"parameters: {parameter}")
            RBF_param = svm_parameter(parameter)
            RBF_m_acc = svm_train(prob, RBF_param)
            # print(polynomial_m_acc)
            best_parameter, best_accuracy, best_kernel = compare_acc(
                best_parameter, best_accuracy, parameter, RBF_m_acc, best_kernel, kernel[i])
```

We are using the grid search to find the best parameters to each kernel. I give folds = 5 cross-validation to get the different parameter accuracy. I set the 4 possible parameter list to do the grid search.

The parameter -t is same as part1. The parameter -s 0 is represent to C-SVC. The parameter -v 5 means folds = 5. The parameter -q that the process do not print out.

```
-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)
```

In the linear kernel, we only need to set the parameter cost -c.

In the polynomial kernel, we need to set the parameter gamma, coef0, degree, cost.

In the RBF kernel, we need to set the parameter gamma and cost.

```python
def compare_acc(best_parameter, best_accuracy, parameter, accuracy, best_kernel, kernel):
    if accuracy > best_accuracy:
        return parameter, accuracy, kernel
    return best_parameter, best_accuracy, best_kernel
```

The function-compare_acc is return the highest accuracy with its parameter and kernel name.

```python
for i in range(len(best_option)):
    best_option[i][0] = best_option[i][0].replace(" -v 5", "")
    model = svm_train(prob, best_option[i][0])
    predict(model, x_test, y_test, best_option[i][2])
    print(f"parameters: {best_option[i][0]}\n")
```

The best_option is storing the three types of kernel in each of which parameter has the best accuracy, using the best parameter to set the model and predict it.

Part3 code

```python
def rbf_kernel(x, y, gamma):
    return np.exp(-gamma * cdist(x, y, 'sqeuclidean'))


def linear_kernel(x, y):
    return x.dot(y.T)
```

```python
costs = [0.1, 1, 10, 100]
gammas = [1/784, 1, 0.1, 0.01]
```

```python
for cost in costs:
    for gamma in gammas:
        rbf_k = rbf_kernel(x_train, x_train, gamma)
        # linear_k = linear_kernel(x_train, y_train)
        X_kernel = np.hstack(
            (np.arange(1, 5001).reshape((-1, 1)), linear_k + rbf_k))
        print(X_kernel.shape)
        parameter = f'-t 4 -c {cost} -v 5 -q'
        print(f"parameters: {parameter}")
        print(f"gamma: {gamma}")
        combine_param = svm_parameter(parameter)
        prob = svm_problem(y_train, X_kernel, True)
        combine_m_acc = svm_train(prob, combine_param)
        best_parameter, best_accuracy, best_kernel = compare_acc(
            best_parameter, best_accuracy, parameter, combine_m_acc, best_kernel, "rbf_and_linear")
        if best_accuracy == combine_m_acc:
            best_gamma = gamma
```

```python
print("\n-----combine kernel predict part-----")
best_parameter = best_parameter.replace(" -v 5", "")
linear_k = linear_kernel(x_test, x_test)
rbf_k = rbf_kernel(x_test, x_test, best_gamma)
X_kernel = np.hstack(
    (np.arange(1, 2501).reshape((-1, 1)), linear_k + rbf_k))
prob = svm_problem(y_test, X_kernel, True)
model = svm_train(prob, best_parameter)
svm_predict(y_test, X_kernel, model)
print(f"parameters: {best_parameter}\ngamma: {best_gamma}")
```

We need to combine two kernel (linear and RBF). The kernel_type is set to -t 4 which means precomputed kernel (kernel values in training_set_file). Using function-rbf_kernel and linear_kernel to get the kernels. And then, combine the two kernel to

generate the new X_train named X_kernel. Doing the grid search training like part2, but this time we need to store the parameter gamma, so that we can set the parameter in the predict. In the predict, we are doing the combine to generate the new X_test named kernel. Using the parameters to predict the test case.

Part b. Results

Part1

```
linear kernel performance
Accuracy = 95.08% (2377/2500) (classification)
linear total time: 2.5430922508239746

polynomial kernel performance
Accuracy = 34.68% (867/2500) (classification)
polynomial total time: 27.891857862472534

RBF kernel performance
Accuracy = 95.32% (2383/2500) (classification)
RBF total time: 6.008209705352783
```

Part2

```
linear kernel performance
Accuracy = 95.8% (2395/2500) (classification)
parameters: -t 0 -s 0 -c 0.1 -q

polynomial kernel performance
Accuracy = 97.84% (2446/2500) (classification)
parameters: -t 1 -s 0 -c 10 -g 0.1 -d 2 -r 1 -q

RBF kernel performance
Accuracy = 98.2% (2455/2500) (classification)
parameters: -t 2 -s 0 -c 10 -g 0.01 -q

PS C:\Users\tom89\OneDrive\桌面\5>
```

Part3

```
-----combine kernel predict part-----
Accuracy = 99.48% (2487/2500) (classification)
parameters: -t 4 -c 0.1 -q
gamma: 1
```

Part c. observations and discussion

In the part1, the polynomial kernel is the worst one and the linear kernel and the RBF kernel has the similar result with high accuracy.

In the part2, the RBF kernel has the highest accuracy. Combine the result in part1, we can discover that RBF kernel always has the highest accuracy. Because of RBF maps data into infinite dimension space.

In part2,
RBF kernel has high accuracy in high c and low g.
Polynomial kernel has high accuracy in d is not 0.
Linear kernel always has high accuracy.

In part3, the combine kernel always has high performance in no matter c and g.

In part2 and part3, we can discover the combine kernel is more stable and has the same accuracy as RBF kernel.