Machine Learning HW5

Gaussian Process

Code with detailed explanations

Part1

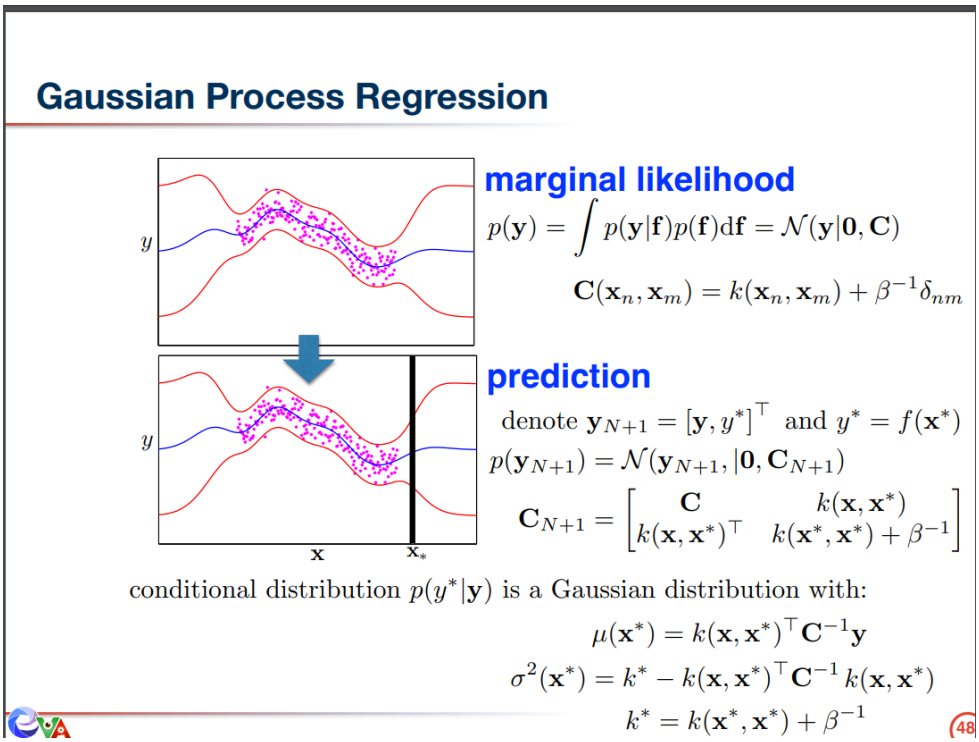
```python
def gaussian_process(x_train, y_train, noise, alpha=1.0, length_scale=1.0, name=''):

    point_num = 1000
    ### get test data
    x_test = np.linspace(-60, 60, point_num).reshape(-1, 1)

    ### get covariance matrix
    cov_matrix = rational_quadratic_kernel(x_train, x_train, alpha, length_scale)

    ### get kernel of test data to test data
    kernel_test = np.add(rational_quadratic_kernel(x_test, x_test, alpha, length_scale), np.eye(len(x_test)) / noise)

    ### get kernel of test data to train data
    kernel_train_test = rational_quadratic_kernel(x_train, x_test, alpha, length_scale)

    ### get mean and variance
    mean = kernel_train_test.T.dot(np.linalg.inv(cov_matrix)).dot(y_train).ravel()
    variance = kernel_test - kernel_train_test.T.dot(np.linalg.inv(cov_matrix)).dot(kernel_train_test)

    ### get 95% confidence interval
    upper_bound = mean + 1.96 * variance.diagonal()
    lower_bound = mean - 1.96 * variance.diagonal()

    ### plot
    plt.xlim([-60, 60])
    plt.title(f"{name} Gaussian Process")
    plt.scatter(x_train, y_train, color='black')
    plt.plot(x_test.ravel(), mean, color='blue')
    plt.fill_between(x_test.ravel(), upper_bound, lower_bound, color='cadetblue', alpha=0.5)
    plt.savefig('GaussianProcess_' + name + '.png')
    plt.show()
def rational_quadratic_kernel(x, y, alpha, length_scale):
    return 1.0 * np.power(1 + cdist(x, y, 'sqeuclidean') / (2 * alpha * length_scale * length_scale), -alpha)

```

The above code is the implementation of Gaussian Process with rational quadratic kernel.
The formula of the rational quadratic kernel is

$$k_{RQ}(x, x') = \sigma^2\left(1 + \frac{(x - x')^2}{2\alpha l}\right)^{-\alpha}$$

where l is the lengthscale which determines the 'wiggle' in the function. Note that here I take $\sigma^2 = 1$.
For the implementaion, the goal is find the mean and variance of each data point, and show the graph with range 95% confidence interval. Hence I follow the print at

lession:



Before calculate the mean and variance, it is necessary to know

1. the covariance of the training data (the kernel between training data and training data)
2. the kernel of testing data and training data
3. the kernel of testing data and testing data

After found the mean and variance, it need to compute the range of confidence interval, like what I learned, using the way

$$upperbound = mean - 1.96 * variance$$
$$lowerbound = mean + 1.96 * variance$$

Note that the variance are the diagonal term in kernel, other terms are the covariance of different data.

PART2

```
45 def marginal_log_likelihood(theta):
46
47     global x_train, y_train
48     point_num = len(x_train)
49     cov_matrix = rational_quadratic_kernel(x_train, x_train, theta[0], theta[1])
50
51     return 0.5 * np.log(np.linalg.det(cov_matrix)) + 0.5 * y_train.ravel().T.dot(np.linalg.inv(cov_matrix)).dot(y_train.ravel()) + point_num
   / 2.0 * np.log(2.0 * np.pi)
52
```

The formula of the likelihood function is

Gaussian Process: learning the kernel

Still remember this?

The kernel function that determines K should be chosen to express the property that:
for points x_n and x_m that are similar,
the corresponding values $y(x_n)$ and $y(x_m)$ will be more strongly correlated than for dissimilar points.

- Consider covariance function $\mathbf{C}$ with hyper-parameters $\boldsymbol{\theta}$

$$k_\theta(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 \exp\{-\theta_1 \frac{\|\mathbf{x}_n - \mathbf{x}_m\|^2}{2}\} + \theta_2 + \theta_3 \mathbf{x}_n^\top \mathbf{x}_m$$

  - Given $\mathcal{D} = \{(\mathbf{x}_i, y_i)_{n=1}^N\} = (\mathbf{X}, \mathbf{y})$, the marginal likelihood is function of $\boldsymbol{\theta}$

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

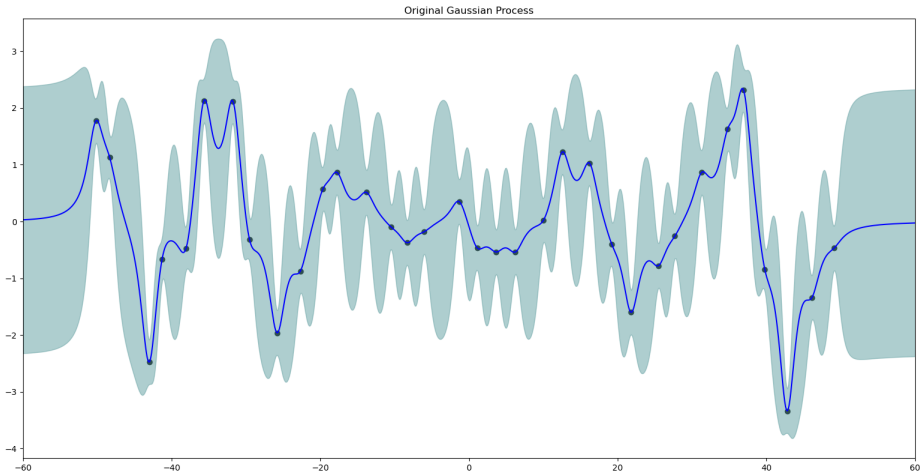
$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2}\ln |\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln(2\pi) \quad \frac{\partial \ln p(\mathbf{y}|\theta)}{\partial \theta}$$

52

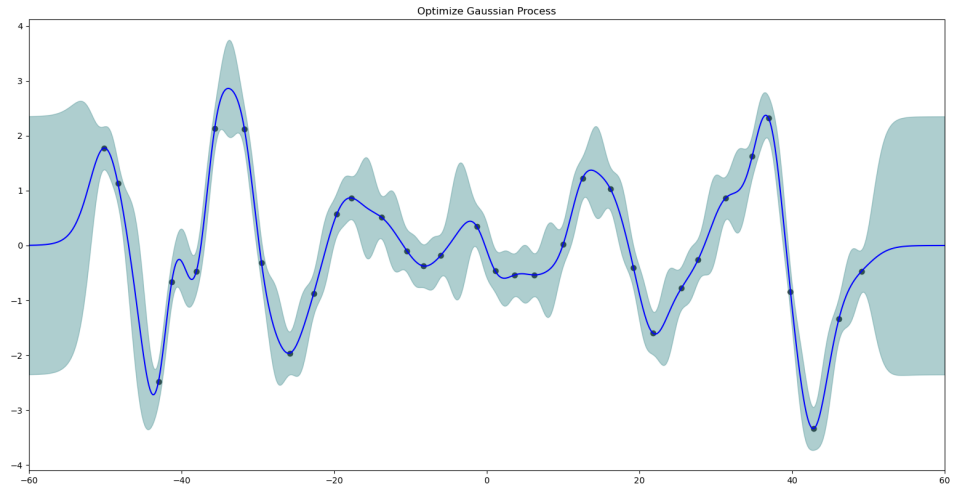
Then we can find the optimal parameters by minimize the negative log-likelihood function, and use them to predict the result.

Experiments setting and results

PART 1



PART 2



with parameters:

- $\alpha \approx 5.6906$
- $length\ scale \approx 2.20491$

Observations and discussion

- The variance of each point will affect the prediction slightly.
- Points which appeared in the training data have less variance than which were not.
- The adjusted kernel fuction has less variance than the original one from the first training data point to the last point. And the points small than the smallest data point and bigger than then the biggest point almost not affect.

SVM

Code with detailed explanations

Part 1

```
8  def linear_poly_rbf_comparison(x_train, y_train, x_test, y_test):
9      ### compare linear, polynomial and rbf kernel
10     kernels = ['Linear', 'Polynomial', 'RBF']
11
12     for idx, name in enumerate(kernels):
13         param = svm_parameter(f'-t {idx} -q')
14         prob = svm_problem(y_train, x_train)
15
16         print(f'{name}: ')
17         start = time.time()
18         model = svm_train(prob, param)
19         svm_predict(y_test, x_test, model)
20         end = time.time()
21
22         print(f'Cost time: {end - start:.3f}s\n')
23
```

There is the implementation of the comparison of linear, polynomial and RBF kernels, this code will show the execution time and the accuracy of each kernel.
Note that the meaning of the arguments:

- -t
  - 1: Linear kernel
  - 2: Polynomial kernel
  - 3: RBF kernel
  - 4: self defined kernel
- -q: quite mode, means it only show the finally result, others will be hided.

Part 2

```
26 def optimize(x_train, y_train, x_test, y_test):
27     ### find the optimal parameters of each kernel method using grid search
28     """
29     Aadjust parameters:
30     Linear: c
31     Polynomial: c, degree, gamma, constant
32     RBF: c, gamma
33     """
34     kernels = ['Linear', 'Polynomial', 'RBF']
35 
36     ### the range of parameters which can be adjusted
37     cost = [np.power(10.0, i) for i in range(-3, 3)]
38     degree = [i for i in range(3)]
39     gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-3, 3)]
40     constant = [i for i in range(-3, 3)]
41 
42     best_parameter = []
43     max_accuracy = []
```

Let me depart the code in many segments.
This function is used to find the optimal parameters in some given parameter set. For these three kernel, there are four kind of parameter:

1. cost: Penalty term. The bigger cost, the serious penlty.
2. degree: Degree of the polynomial.
3. gamma: Coefficient of the kernel function, used by RBF, polynomail and sigmoid kernel.
4. constant: Constant term is the kernel function, used by polynomial and sigmoid.

Then for each kernel function, find the best parameters of it. The way to find them is set the parameters and train with them, take the parameter have the maximun accuracy as the best.

First, the Linear kernel function. The parameter we can adjust is “cost”.

```
if name == 'Linear':
    for c in cost:
        parameters = f'-t {idx} -c {c} -q'
        param = svm_parameter(parameters + ' -v 3')
        prob = svm_problem(y_train, x_train)
        acc = svm_train(prob, param)

        if acc > best_acc:
            best_acc = acc
            best_param = parameters
    best_parameter.append(best_param)
    max_accuracy.append(best_acc)
```

Second, the Polynomial kernel function. The parameter we can adjust are “cost”, “degree”, “gamma”, “constant”.

```
if name == 'Polynomial':
    for c in cost:
        for d in degree:
            for g in gamma:
                for C in constant:
                    parameters = f'-t {idx} -c {c}  -d {d} -g {g} -r {C} -q'
                    param = svm_parameter(parameters + ' -v 3')
                    prob = svm_problem(y_train, x_train)
                    acc = svm_train(prob, param)

                    if acc > best_acc:
                        best_acc = acc
                        best_param = parameters
    best_parameter.append(best_param)
    max_accuracy.append(best_acc)
```

Last, the RBF kernel function. The parameter we can adjust are “cost”, “gamma”.

```
if name == "RBF":
    for c in cost:
        for g in gamma:
            parameters = f'-t {idx} -c {c} -g {g} -q'
            param = svm_parameter(parameters + ' -v 3')
            prob = svm_problem(y_train, x_train)
            acc = svm_train(prob, param)

            if acc > best_acc:
                best_acc = acc
                best_param = parameters
    best_parameter.append(best_param)
    max_accuracy.append(best_acc)
```

So far, we get all best parameters of each kernel, then use them to predict and record the accuracy.

```
### after find all optimal parameters, do the prediction
best_parameter = ['-t 0 -c 0.01 -q', ' -t 1 -c 100.0  -d 2 -g 10.0 -r 1 -q', '-t 2 -c 100.0 -g 0.01 -q']   ### recorded by myself
max_accuracy = [96.84, 98.26, 98.18]
prob = svm_problem(y_train, x_train)
for i, name in enumerate(kernels):
    print(f"{name}")
    print(f"Max accuracy: {max_accuracy[i]}")
    print(f"Best parameters: {best_parameter[i]}")

    param = svm_parameter(best_parameter[i])
    model = svm_train(prob, param)
    svm_predict(y_test, x_test, model)
```

Note that the `best_parameter` and `max_accuracy` are filled by myself, because I do not want to train the data again, therefore recorded by handcraft.

Part 3

```python
def linear_rbf_conbination(x_train, y_train, x_test, y_test):
    ### parameters
    cost = [np.power(10.0, i) for i in range(-2, 3)]
    gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-3, 2)]
    rows, cols = x_train.shape

    ### find the best parameters
    linear = linear_kernel(x_train, x_train)
    best_param = '-t 4 -c 0.01 -q'
    best_gamma = 0.1
    max_acc = 0.0
```

This is the implemnetation of the combination of linear and the RBF kernel, the set initial values are the handcraft value that can let me skip the training step.

```python
for c in cost:
    for g in gamma:
        rbf = rbf_kernel(x_train, x_train, g)

        ### conbine two kernels: 'add'
        # combination = np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf))
        combination = linear @ rbf
        parameters = f'-t 4 -c {c} -q'
        param = svm_parameter(parameters + ' -v 3')
        prob = svm_problem(y_train, combination, isKernel=True)
        acc = svm_train(prob, param)

        if acc > max_acc:
            max_acc = acc
            best_param = parameters
            best_gamma = g

### print the best parameters and the max accuracy
print('=='*30)
print(f'Linear + RBF')
print(f'\tMax accuracy: {max_acc}%')
print(f'\tBest parameters: {best_param}, gamma: {best_gamma}\n')
```

So as the part2, this is the step which choosing the best parameter.

```python
### train the model by these best parameters
# linear = linear_kernel(x_train, x_train)
rbf = rbf_kernel(x_train, x_train, best_gamma)
combination = np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf))
#combination = linear @ rbf
model = svm_train(svm_problem(y_train, combination, isKernel=True),
                  svm_parameter(best_param))
#print(model)
### predict
rows, cols = x_test.shape
linear = linear_kernel(x_test, x_test)
rbf = rbf_kernel(x_test, x_test, best_gamma)
combination = np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf))
#combination = linear @ rbf
"""
print(np.arange(1, rows + 1).reshape(-1, 1).shape)
print(np.hstack((np.arange(1, rows + 1).reshape(-1, 1), linear + rbf)).shape)
print(combination.shape)
print((rbf + linear).shape)
print('after compute the combination')
"""
print(f"RBF + Linear")
svm_predict(y_test, combination, model)
```

After find the best parameters, use them to train a model and use to predicct.

Note that because we desire to conbine the Linear kernel and the RBF kernel, it is necessary to define them by myself, and set the SVM parameter as `-t 4`.

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


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

Experiments settings and results

Part 1

```
Linear:
Accuracy = 95.08% (2377/2500) (classification)
Cost time: 0.576s

Polynomial:
Accuracy = 34.68% (867/2500) (classification)
Cost time: 2.823s

RBF:
Accuracy = 95.32% (2383/2500) (classification)
Cost time: 0.849s
```

Part 2

Note that the max accuracy term means the max training accuracy, the the Accuracy term means the testing accuracy.

```
Linear
Max accuracy: 96.84
Best parameters: -t 0 -c 0.01 -q
Accuracy = 95.96% (2399/2500) (classification)
Polynomial
Max accuracy: 98.26
Best parameters:  -t 1 -c 100.0  -d 2 -g 10.0 -r 1 -q
Accuracy = 97.68% (2442/2500) (classification)
RBF
Max accuracy: 98.18
Best parameters: -t 2 -c 100.0 -g 0.01 -q
Accuracy = 98.16% (2454/2500) (classification)
```

Part 3

```
RBF + Linear
Accuracy = 23.4% (585/2500) (classification)
```

observations and discussion

- The polynomial kernel is time costing.
- After fine-tunning each kernel, the accuracy from high to low is RBF, polynomial and linear.
- Combine the RBF and linear kernel by adding will cause worse result.

- The bigger cost may cause overfitting because the penalty will very large when wrong classify, i.e. the testing step may have low accuracy. In contrast, the smaller cost will let the model have strong generalization.