HW5

312554054 林辰翰

Gaussian Process Regression

Goal: Applying Gaussian Process Regression by using the rational quadratic kernel to calculate the similarity between different data points and predict the distribution of the function f.

Gaussian Process Regression (GPR) is a memory-based method, meaning that its predictions rely on the relationships between new input data and the training data. It is particularly valued for its ability to provide not only precise predictions but also robust estimates of uncertainty surrounding these predictions. This capability is essential for informed decision-making in a variety of applications, enhancing the reliability and effectiveness of the insights derived from the model.

Code with detailed explanation:

```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)

    # 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)

    # 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()
```

The main code provided above implements Gaussian Process Regression using a rational quadratic kernel.

The rational quadratic kernel (RQ kernel) is defined as follows:

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

```
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)
```

Where:

(1) x and x' are two input vectors

(2) σ ^2 is the variance parameter that scales the overall variance of the kernel

(3) ℓ is the length-scale parameter, which determines the smoothness of the function; smaller values lead to less smooth functions

(4) α is the scale-mixture parameter that controls the weighting of large- and small-scale variations in the function. Smaller values of α lead to long tails in the kernel function, suggesting the presence of multiple characteristic length-scales in the data.

* Note that in this context, the sigma parameter σ is set to 1.

After obtaining the covariance matrix for the training data using the rational quadratic kernel, the model mainly moves forward with Test Data Covariance Matrix and Cross Covariance Matrix Between Test and Train Data.

Test data to test data (Test Data Covariance Matrix)

In the step of calculating the covariance matrix for the test data, this matrix represents the similarities between all pairs of points in the test dataset. This is achieved with the same rational quadratic kernel, and an identity matrix scaled by the noise term (1 / beta and beta is equal to 5) is added to the diagonal. This addition accounts for the noise in the observations, helping stabilize the

numerical computations and ensuring that the covariance matrix remains positive definite.

Test data to training data (Cross Covariance Matrix Between Test and Train Data)

We also compute the covariance matrix between the training data and the test data, which is used to predict the mean and covariance of the function values at the test points, based on the training data.

With the training covariance matrix and the cross-covariance matrix available, the model is able to predict the mean and variance of the function values at the test points. Once the mean and variance are determined, the confidence interval can be calculated. This involves computing the upper and lower bounds using the formula provided below:

Upper Bound: mean + 1.96 × sqrt(variance)

Lower Bound: mean - 1.96 × sqrt(variance)

In the optimization part, the objective is to identify the values of theta that minimize the log likelihood.

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 $\boldsymbol{\theta}$

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

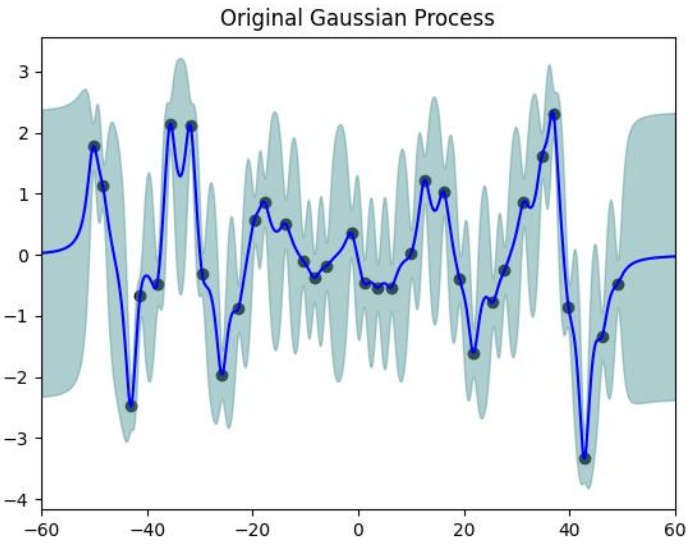
$$\ln p(\mathbf{y}|\theta) = -\frac{1}{2}\ln |\mathbf{C}_\theta| - \frac{1}{2}\mathbf{y}^\top \mathbf{C}_\theta^{-1}\mathbf{y} - \frac{N}{2}\ln(2\pi) \quad \frac{\partial \ln p(\mathbf{y}|\theta)}{\partial \theta}$$

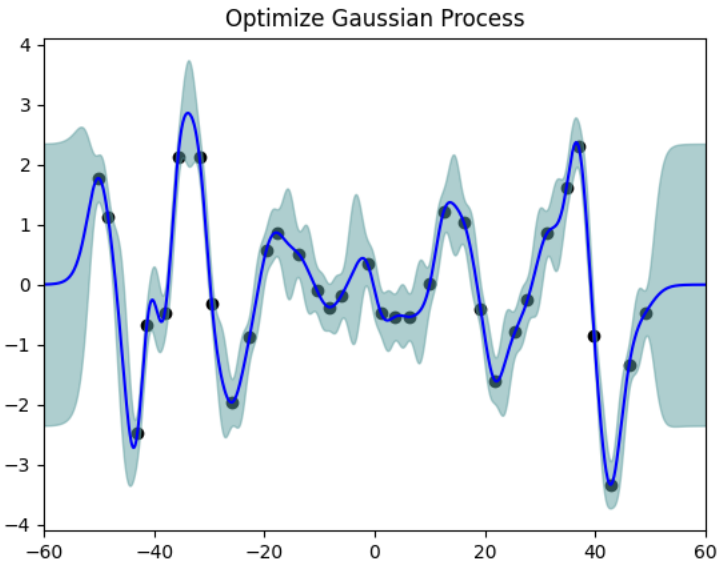
```python
def marginal_log_likelihood(theta):

    global x_train, y_train
    point_num = len(x_train)
    cov_matrix = rational_quadratic_kernel(x_train, x_train, theta[0], theta[1])

    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)
```

Results Present:






```
alpha 5.690749346510464
length_scale 2.0249123392614696
```

Alpha ≈ 5.69

Length_scale ≈ 2.025

Observation and Discussion:

(1) The variance associated with each point influences its prediction.

(2) Points that were included in the training data exhibit lower variance compared to those that were not.

(3) The adjusted kernel function demonstrates reduced variance across all points from the smallest to the largest data point in the training set. Points that fall outside this range have minimal influence on the model.

SVM on MNIST dataset

Goal: To classify images of handwritten digits where the digit class ranges from 0 to 4, employing SVM (Support Vector Machine) models with various kernel functions such as linear, polynomial, and RBF (Radial Basis Function) can be very effective. Each kernel function offers distinct advantages that can help in capturing the intricacies of the data.

Code with detailed explanation:

```
def linear_poly_rbf_comparison(x_train, y_train, x_test, y_test):
    # Compare Linear, Polynomial and RBF kernel
    kernels = ['Linear', 'Polynomial', 'RBF']

    for idx, name in enumerate(kernels):
        param = svm_parameter(f'-t {idx} -q')
        prob = svm_problem(y_train, x_train)

        print(f'{name}: ')
        start = time.time()
        model = svm_train(prob, param)
        svm_predict(y_test, x_test, model)
        end = time.time()

        print(f'Cost time: {end - start:.3f}s\n')
```

The code above is designed to compare the performance of three different SVM kernels: Linear, Polynomial, and RBF (Radial Basis Function). For each kernel type, it initializes the SVM parameters using svm_parameter, specifying the kernel type with -t {idx} (where idx is the index corresponding to the kernel type in the kernels list, and -q suppresses console output).

Note that the meaning of the arguments:

-t

1: Linear kernel

2: Polynomial kernel

3: RBF kernel

4: self-defined kernel

-q: quiet mode, which means it only shows the final result, others will be hided.

```
def optimize(x_train, y_train, x_test, y_test):
    # Find the optimal parameters of each kernel method using grid search
    """
    Aadjust parameters:
    Linear: c
    Polynomial: c, degree, gamma, constant
    RBF: c, gamma
    """
    kernels = ['Linear', 'Polynomial', 'RBF']

    # the range of parameters which can be adjusted
    cost = [np.power(10.0, i) for i in range(-3, 3)]
    degree = [i for i in range(3)]
    gamma = [1.0 / 784] + [np.power(10.0, i) for i in range(-3, 3)]
    constant = [i for i in range(-3, 3)]

    best_parameter = []
    max_accuracy = []
```

This function is designed to identify the optimal parameters from a specified set for three different kernel types. The parameters include:

(1) Cost: This is a penalty term where a larger value results in a more substantial penalty.

(2) Degree: This specifies the degree of the polynomial kernel.

(3) Gamma: A coefficient used by the RBF, polynomial, and sigmoid kernels to adjust the influence of a single training example.

(4) Constant: A constant term used in polynomial and sigmoid kernels.

For each kernel function, the goal is to determine the best set of parameters. This is achieved by setting various parameter combinations, training the model with these settings, and selecting the combination that yields the highest accuracy as the optimal choice.

Firstly, with the linear kernel function, the adjustable parameter 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)
```

Secondly, for the Polynomial kernel function, the adjustable parameters are "cost," "degree," "gamma," and "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 but not least, for the RBF kernel function, the adjustable parameters are "cost" and "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)
```

After determining the best parameters for each kernel, we can now use them to make predictions and record the corresponding accuracies.

```
# After finding 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']
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 values are determined manually after the training stage to avoid retraining the model.)

The following implementation combines the linear and RBF kernels, with initial values that are handcrafted for the same reasons mentioned previously.

```
def linear_rbf_combination(x_train, y_train, x_test, y_test):
    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_train, cols = x_train.shape
    rows_test, _ = x_test.shape

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

For this part, the process involves selecting the best parameters.

```
for c in cost:
    for g in gamma:
        rbf = rbf_kernel(x_train, x_train, g)
        indices_train = np.arange(1, rows_train + 1).reshape(-1, 1)
        combination_train = linear + rbf
        combination_train = np.hstack((indices_train, combination_train))
        parameters = f'-t 4 -c {c} -q'
        param = svm_parameter(parameters + ' -v 3')
        prob = svm_problem(y_train, combination_train, isKernel=True)
        acc = svm_train(prob, param)

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

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

After finding the best parameters, use them to train the model and predict.

Note that since we are combining the Linear and RBF kernels, the -t parameter should be set to 4.

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


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

Results and Performance:

```
Linear:
Accuracy = 95.08% (2377/2500) (classification)
Cost time: 0.496s

Polynomial:
Accuracy = 34.68% (867/2500) (classification)
Cost time: 2.678s

RBF:
Accuracy = 95.32% (2383/2500) (classification)
Cost time: 0.736s
```

```
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
```

```
Linear + RBF
        Max accuracy: 97.04%
        Best parameters: -t 4 -c 0.01 -q, gamma: 0.01
```

Observations and Discussion:

(1) The polynomial kernel is time-consuming.

(2) After fine-tuning each kernel, their accuracy ranking from highest to lowest is RBF, polynomial, and linear.

(3) Combining the RBF and linear kernels by summing them together tends to worsen performance.

(4) A larger cost value may lead to overfitting, as the penalty becomes substantial for misclassifications, potentially resulting in lower accuracy during testing. Conversely, a smaller cost value promotes stronger generalization in the model.