ML Homework 5 Report

- Student Info
  1. Student ID: 310555024
  2. Student Name: 林廷翰

- Gaussian Process
  1. Code
     - Common Parts
       1. Load Data

```
def loadData():
    # Load data from input file.
    data = np.loadtxt(inputFilePath, dtype=float)

    # Reshape the data to x and y.
    x = data[:, 0].reshape(-1, 1)
    y = data[:, 1].reshape(-1, 1)

    return x, y
```

The objective of the loadData function is to load data from the input file, it will be called in the main function.

       2. Two Modes

```
if mode == 1:
    gaussianProcess(x, y)
elif mode == 2:
    initGuess = np.array([1.0, 1.0])
    result = minimize(marginalLogLikelihood, initGuess)

    optAlpha, optLengthScale = result.x
    gaussianProcess(x, y, optAlpha, optLengthScale)
```

There are two modes in the program, the first mode is for task 1, and the second mode is for task 2. We can control the mode with program input parameter -m.

     - Code for Task 1

1. Rational Quadratic Kernel

```python
# Reference: https://en.wikipedia.org/wiki/Rational_quadratic_covariance_function
def rationalQuadraticKernel(x1, x2, alpha, lengthScale):
    # variance = (1 + d ^ 2 / 2αl ^ 2) ^ (-α)
    return (1 + cdist(x1, x2, 'sqeuclidean') / (2 * alpha * lengthScale ** 2)) ** -alpha
```

The function is to compute the rational quadratic kernel, and I use cdist to calculate the distance between point pairs. The formula is `variance = (1 + d ^ 2 / 2αl ^ 2) ^ (-α)` (from wiki).

2. Gaussian Process

```python
def gaussianProcess(xOfTraining, yOfTraining, alpha=1.0, lengthScale=1.0):
    # Generate the testing points.
    numOfTestingPoints = 1000
    xOfTesting = np.linspace(-60, 60, numOfTestingPoints).reshape(-1, 1)

    # Compute covariance matrix of training data.
    covOfTraining = rationalQuadraticKernel(xOfTraining, xOfTraining, alpha, lengthScale)

    # Compute the kernel of testing data to testing data.
    kernelStar = np.add(rationalQuadraticKernel(xOfTesting, xOfTesting, alpha, lengthScale),
                        np.eye(len(xOfTesting)) / noise)

    # Compute the kernel of training data to testing data.
    kernel = rationalQuadraticKernel(xOfTraining, xOfTesting, alpha, lengthScale)

    # Compute mean and variance.
    mean = kernel.T.dot(np.linalg.inv(covOfTraining)).dot(yOfTraining).ravel()
    variance = kernelStar - kernel.T.dot(np.linalg.inv(covOfTraining)).dot(kernel)
```

The function is the implementation of the Gaussian process. The idea of the formula is from the lecture PPT p.48 (the following pic).

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

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

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

48

I computed the k* first, and then computed the k. Finally, I computed the mean and variance based on k & k*.

- Code for Task 2

1. Compute OptAlpha & OptLengthScale

```
def marginalLogLikelihood(theta):
    # Compute covariance matrix
    covariance = rationalQuadraticKernel(x, x, alpha=theta[0], lengthScale=theta[1])

    # - ln p(y|θ) = 0.5 * ln|C| + 0.5 * y ^ T * C ^ (-1) * y + N / 2 * ln(2π)
    return 0.5 * np.log(np.linalg.det(covariance)) + 0.5 * y.ravel().T.dot(np.linalg.inv(covariance)).dot(
        y.ravel()) + numOfPoints / 2.0 * np.log(2.0 * np.pi)
```

Before the Gaussian process of task 2, we need to compute the optAlpha and optLengthScale first. The idea of the formula is from the lecture PPT p.52 (the following pic).

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

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

52

So we can compute the optimal parameters (alpha & length scale) from negative marginal log-likelihood.

```
elif mode == 2:
    initGuess = np.array([1.0, 1.0])
    result = minimize(marginalLogLikelihood, initGuess)

    optAlpha, optLengthScale = result.x
    gaussianProcess(x, y, optAlpha, optLengthScale)
```

The code block is the optimizing parameters part, I found the optimal parameters by minimizing marginal log-likelihood.

2. Gaussian Process

   The section is the same as “Code for Task 1”.

- Common Parts

1. Output the Figure

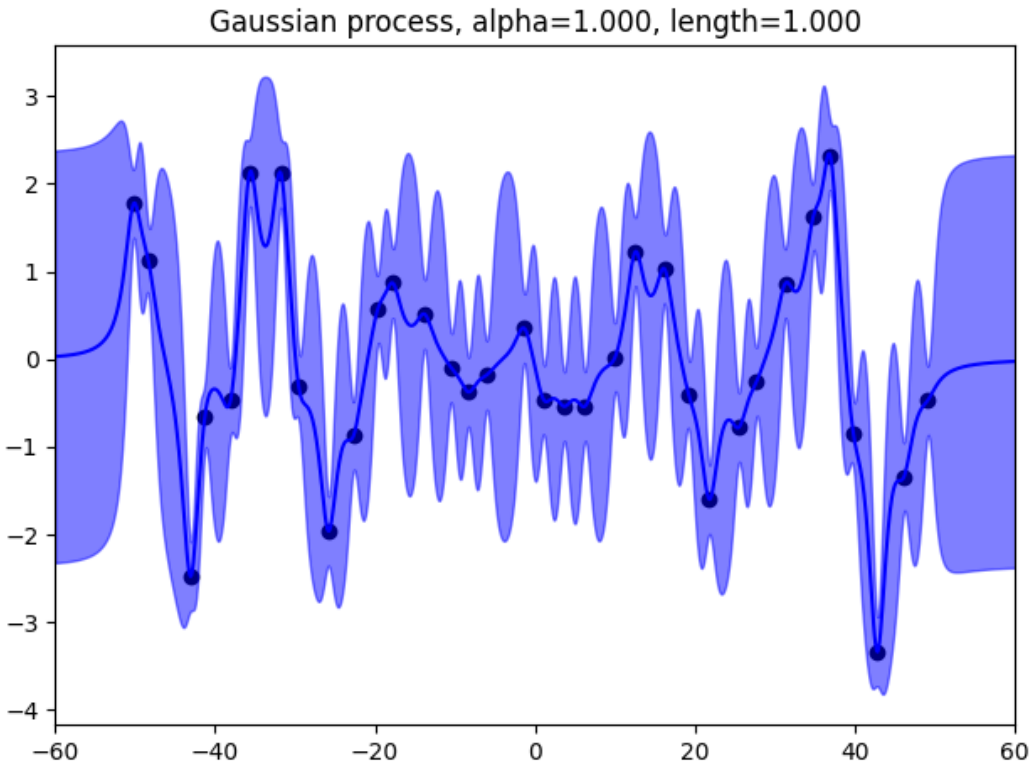
```
    # Compute 95% confidence upper and lower bound.
    upperBound = mean + 1.96 * variance.diagonal()
    lowerBound = mean - 1.96 * variance.diagonal()

    # Output the graph.
    plt.xlim(-60, 60)
    plt.title(f'Gaussian process, alpha={alpha:.3f}, length={lengthScale:.3f}')
    plt.scatter(xOfTraining, yOfTraining, c='k')
    plt.plot(xOfTesting.ravel(), mean, 'b')
    plt.fill_between(xOfTesting.ravel(), upperBound, lowerBound, color='b', alpha=0.5)
    plt.tight_layout()
    plt.show()
```

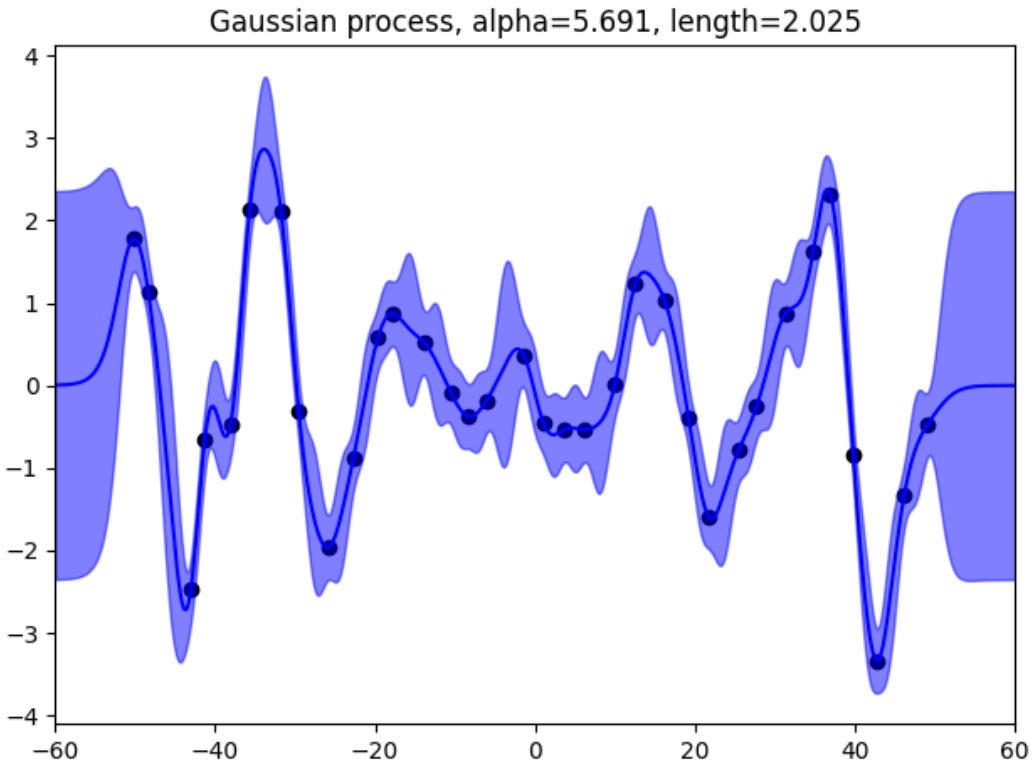
Compute the lower bound and upper bound, and then output the result with a figure.

2. Experiments

- Experiment for Task 1



- Experiment for Task 2



3. Observations and Discussion
    - The result of task 2 is better than task 1.
    - We can see that in the interval that we have training data, the Gaussian process can make a better prediction with higher confidence than in the interval that we don’t have any training data.
    - Based on the second observation, we can know that we can make a better prediction based on more training data.

- SVM
    1. Code
        - Common Parts
            1. Load Data

```python
def loadData():
    trainingImages = np.loadtxt(filePathOfTrainingImages, delimiter=',')
    trainingLabels = np.loadtxt(filePathOfTrainingLabels, delimiter=',', dtype=int)
    testingImages = np.loadtxt(filePathOfTestingImages, delimiter=',')
    testingLabels = np.loadtxt(filePathOfTestingLabels, delimiter=',', dtype=int)

    return trainingImages, trainingLabels, testingImages, testingLabels
```

The objective of the loadData function is to load data from the input file, it will be called in the main function.

2. Three Modes

```python
if mode == 1:
    compareDiffKernels(trainingImages, trainingLabels, testingImages, testingLabels)
elif mode == 2:
    gridSearch(trainingImages, trainingLabels, testingImages, testingLabels)
elif mode == 3:
    linearRBFCombination(trainingImages, trainingLabels, testingImages, testingLabels)
```

There are three modes in the program, the first mode is for task 1, the second mode is for task 2 and the last mode is for task 3. We can control the mode with program input parameter -m.

- Code for Task 1

1. Compute Different Kernel Performance

```python
# Compare the performance of different kernel functions (linear, polynomial, RBF).
def compareDiffKernels(trainingImages, trainingLabels, testingImages, testingLabels):
    # kernel names.
    kernels = ['Linear', 'Polynomial', 'Radial basis function']

    # Compute performance of each kernel.
    for index, kernel in enumerate(kernels):
        problem = svm_problem(trainingLabels, trainingImages)
        # Reference: https://www.csie.ntu.edu.tw/~cjlin/libsvm/
        parameter = svm_parameter(f"-t {index} -q")

        print(f'#Kernel: {kernel}')

        startTime = time.time()
        model = svm_train(problem, parameter)
        svm_predict(testingLabels, testingImages, model)
        endTime = time.time()

        print(f'Total time: {endTime - startTime}')
        print('----------------------------------')
```

I use the -t parameter to set up the different models, and -q parameter to avoid printing redundant info. For each loop, we will print the accuracy for each kernel.

- Code for Task 2

1. Prepare the Parameters

```
    # kernel names.
    kernels = ['Linear', 'Polynomial', 'Radial basis function']

    # Parameters
    costs = [0.1, 1, 10]
    degrees = [0, 1, 2]
    gammas = [1 / 784, 0.1, 1]
    constants = [-1, 0, 1]
```

I prepared the parameters with four different arrays - costs, degrees, gammas, and constants. We will use the parameters to compute the best accuracy.

2. General Function for Computing Performance

```
# Grid search with cross validation.
def gridSearchWithCV(trainingImages, trainingLabels, parameters, isKernel=False):
    parameter = svm_parameter(parameters + ' -v 3 -q')
    problem = svm_problem(trainingLabels, trainingImages, isKernel=isKernel)

    return svm_train(problem, parameter)
```

The function will be reused in the following part. You can set parameters as input and get the training result.

3. Compute Different Kernel Performance

- Linear

```
    if kernel == 'Linear':
        for cost in costs:
            parameters = f'-t {index} -c {cost}'
            accuracy = gridSearchWithCV(trainingImages, trainingLabels, parameters)

            if accuracy > bestAccuracy:
                bestAccuracy = accuracy
                bestParameter = parameters

        arrayOfBestAccuracy.append(bestAccuracy)
        arrayOfBestParameters.append(bestParameter)
```

I computed the best accuracy with different costs.

- Poly

```
    elif kernel == 'Polynomial':
        for cost in costs:
            for degree in degrees:
                for gamma in gammas:
                    for constant in constants:
                        parameters = f'-t {index} -c {cost} -d {degree} -g {gamma} -r {constant}'
                        accuracy = gridSearchWithCV(trainingImages, trainingLabels, parameters)

                        if accuracy > bestAccuracy:
                            bestAccuracy = accuracy
                            bestParameter = parameters

        arrayOfBestAccuracy.append(bestAccuracy)
        arrayOfBestParameters.append(bestParameter)
```

I computed the best accuracy with different costs, degrees, gammas, and constants.

- RBF

```
    elif kernel == 'Radial basis function':
        for cost in costs:
            for gamma in gammas:
                parameters = f'-t {index} -c {cost} -g {gamma}'
                accuracy = gridSearchWithCV(trainingImages, trainingLabels, parameters)

                if accuracy > bestAccuracy:
                    bestAccuracy = accuracy
                    bestParameter = parameters

        arrayOfBestAccuracy.append(bestAccuracy)
        arrayOfBestParameters.append(bestParameter)
```

I computed the best accuracy with different costs and gammas.

- Code for Task 3
  1. Compute the Kernel
     - Linear

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

The function is for computing a simple linear kernel.

- RBF

```
# Reference: https://en.wikipedia.org/wiki/Radial_basis_function_kernel
def computeRBFKernel(x, y, gamma):
    return np.exp(-gamma * cdist(x, y, 'sqeuclidean'))
```

The function is for computing the RBF kernel. The formula is from the wiki (the following pic).

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

  2. Prepare the Parameters

```
    # Parameters
    costs = [0.01, 0.1, 1.0, 10.0, 100.0]
    gammas = [1.0 / 784, 0.001, 0.01, 0.1, 1.0, 10.0]
    numOfTrainingData, _ = trainingImages.shape
```

I prepared the parameters with four different arrays - costs, and gammas. We will use the parameters to compute the best accuracy.

3. Compute the Performance

```python
for cost in costs:
    for gamma in gammas:
        kernelOfRBF = computeRBFKernel(trainingImages, trainingImages, gamma)
        combination = computeCombinationKernel(kernelOfLinear, kernelOfRBF, numOfTrainingData)
        parameters = f'-t 4 -c {cost}'
        accuracy = gridSearchWithCV(combination, trainingLabels, parameters, True)

        if accuracy > bestAccuracy:
            bestAccuracy = accuracy
            bestParameters = parameters
            bestGamma = gamma

# Print best parameters and best accuracy
print('-----------------------------------------------------------------')
print('#Kernel: Linear + RBF')
print(f'\tMax accuracy: {bestAccuracy}%')
print(f'\tBest parameters: {bestParameters} -g {bestGamma}\n')
```

I computed the result based on `gridSearchWithCV` (reuse the function with task 2), and I also printed the best accuracy and parameters.

2. Experiment

- Experiment for Task 1

```
(ML) ericl@ericl-Aspire-V5-573PG:~/Institute/CS/Course/Machine Learning/ML$ python3 project5_svm.py --mode=1
#Kernel: Linear
Accuracy = 95.08% (2377/2500) (classification)
Total time: 6.815182447433472
---------------------------------
#Kernel: Polynomial
Accuracy = 34.68% (867/2500) (classification)
Total time: 61.410879373550415
---------------------------------
#Kernel: Radial basis function
Accuracy = 95.32% (2383/2500) (classification)
Total time: 13.79182505607605
---------------------------------
```

- Experiment for Task 2

```
#Kernel: Linear
        Max accuracy: 96.66%
        Best parameters: -t 0 -c 0.1
Accuracy = 95.8% (2395/2500) (classification)

#Kernel: Polynomial
        Max accuracy: 98.04%
        Best parameters: -t 1 -c 0.1 -d 2 -g 0.1 -r 0
Accuracy = 97.76% (2444/2500) (classification)

#Kernel: Radial basis function
        Max accuracy: 97.0%
        Best parameters: -t 2 -c 10 -g 0.0012755102040816326
Accuracy = 96.28% (2407/2500) (classification)
```

- Experiment for Task 3

```
#Kernel: Linear + RBF
        Max accuracy: 97.0%
        Best parameters: -t 4 -c 0.01 -g 1.0

Accuracy = 32.84% (821/2500) (classification)
```

3. Observations and Discussion

- From the result of task 1 and task, we know that RBF is better than the other. Because RBF maps input data to infinite dimension feature space.
- From the result, we know that RBF is a good kernel in classification. That is why people often pick RBF as the kernel.
- Polynomial kernel spends more time on training (based on the total time I record).
- The reason for 4th observation is the need to finely turned a lot of parameters.
- The accuracy of testing data may be worse than cross-validation because the best parameters for training data are not always the best parameters for testing data.