

1. Gaussian Process

(1) Task 1:

a. Code

```
df = read_file("./data./input.data.txt")

# task 1 : gaussian process regression
if sys.argv[1] == "1":
    beta = int(sys.argv[2])
    GPD(df, beta, int(sys.argv[3]), int(sys.argv[4]), int(sys.argv[5]))
```

Step 1: first read the file and assume beta (the inverse variation of random noise) is 5. Then input parameters into the GPD (Gaussian Process Decision) function.

```
def GPD(df, beta, amplitude, rate, scale):
    X = [x[0] for x in df]
    X = np.array(X).reshape(-1, 1)
    Y = [y[1] for y in df]
    Y = np.array(Y).reshape(-1, 1)
    X_new = np.linspace(-60, 60, 120).reshape(-1, 1)
    train_size = len(X)

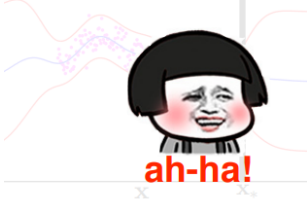
    # follow the formula in the slide pg.50
    C = kernel(X, X) + 1/beta*np.identity(train_size)
    mean = kernel(X, X_new).T@np.linalg.inv(C)@Y
    var = kernel(X_new, X_new) + (1/beta)*np.identity(len(X_new)) - \
        kernel(X, X_new).T@np.linalg.inv(C)@kernel(X, X_new)

    var = np.diagonal(var).reshape(-1, 1)
    #plt.scatter(X, Y)
    plt.plot(X_new, mean, 'k-')
    plt.fill_between(np.sum(X_new, axis=1), np.sum(
        mean+1.96*np.sqrt(var), axis=1), np.sum(mean-1.96*np.sqrt(var), axis=1))
    plt.scatter(X, Y, c='r')
    plt.show()
```

Step 2: the function first processes the input data (X and Y), and create new data points from -60 to 60 with 120 intervals.

Next, calculate the kernel (which is rational quadratic kernel, describe in next section)

Then, use the kernel function to calculate new mean and variance based on the similarity b/w old and new data points. To be more specific, the mean and variance functions follow the formulas in pg.49 of the slide given by Pf. Chiu.



prediction

denote $\mathbf{y}_{N+1} = [\mathbf{y}, y^*]^\top$ and $y^* = f(\mathbf{x}^*)$

$p(\mathbf{y}_{N+1}) = \mathcal{N}(\mathbf{y}_{N+1}, \mathbf{0}, \mathbf{C}_{N+1})$

$\mathbf{C}_{N+1} = \begin{bmatrix} \mathbf{C} & k(\mathbf{x}, \mathbf{x}^*) \\ k(\mathbf{x}, \mathbf{x}^*)^\top & k(\mathbf{x}^*, \mathbf{x}^*) + \beta^{-1} \end{bmatrix}$

1° kernel

conditional distribution $p(y^* | \mathbf{y})$ is a Gaussian distribution with:

2° conditional

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

3° done!

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

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

(49)

Last step, we use the mean, and the variance to draw the line and its 95% confidence intervals (which is calculated by $\text{mean} \pm 1.96 \sqrt{\text{var}}$, where the var is a diagonal element in the variance function)

```
def kernel(X_1, X_2, amplitude=1, rate=1, scale=1):
    # [X_n - X_m]^2
    dis = np.zeros((len(X_1), len(X_2)))
    for i in range(len(X_1)):
        for j in range(len(X_2)):
            dis[i][j] = np.power(X_1[i][0] - X_2[j][0], 2)
    return amplitude**2*(1+dis/(2*rate*(scale**2)))**(-rate)
```

The Kernel function is calculated by:

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

Use the two-dimensional matrix to record the distance (somehow time-consuming)

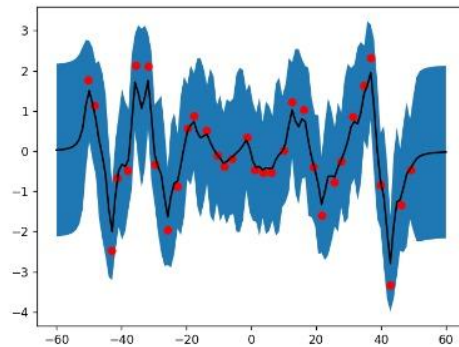
The initial parameters are designed as:

amplitude(sigma) = 1

rate(alpha) = 1

scale(length) = 1

b. Result



- The x-axis is from -60 to 60, which represents the range of our f
- blue area means the 95% confidential interval calculated by each mean, var
- red points are our training data points
- The black line is our prediction based on the kernel function.

c. Observation

It seems that our prediction function did a great job(the black line is close to red points), using the Gaussian process regression with the rational quadratic kernel, we could calculate the distribution and the mle of our new point based on the similarity between new and old data points.

(2) Task 2:

a. Code

```
# task 2 : optimize
else:
    beta = int(sys.argv[2])
    param = minimize(optimizer, [int(sys.argv[3]), int(sys.argv[4]), int(sys.argv[5])], args=(df, beta))
    amplitude = param.x[0]
    rate = param.x[1]
    scale = param.x[2]
    GPD(df, beta, amplitude, rate, scale)
    print(f"best amplitude {amplitude}")
    print(f"best rate {rate}")
    print(f"best scale {scale}")
```

Use the scipy optimize package to find the best parameter set of our kernel (or say Gaussian process regression function).

The minimize target function is shown in the next section)

Next, we use our tuned parameters to implement the GP again and print out the result.

```
def optimizer(params, df, beta):
    X = [x[0] for x in df]
    X = np.array(X).reshape(-1, 1)
    Y = [y[1] for y in df]
    Y = np.array(Y).reshape(-1, 1)
    train_size = len(X)
    C = kernel(X, X, params[0], params[1], params[2]) + \
        1/beta*np.identity(train_size)
    log_likelihood = (1/2)*Y.T@np.linalg.inv(C)@Y - (1/2)*np.sum(np.log(
        np.diagonal(np.linalg.cholesky(C)))) - (1/2)*train_size*np.log(2*np.pi)

    return -log_likelihood[0][0]
```

Same as task 1, we first process the data in a unified format. And calculate the negative log-likelihood, hoping that we can minimize it.

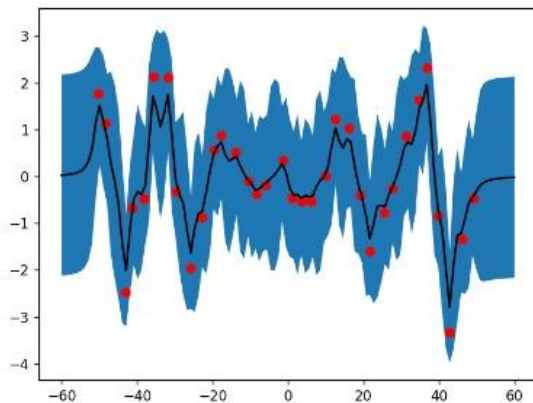
The log-likelihood function is referenced from the following website:

<https://stats.stackexchange.com/questions/280105/log-marginal-likelihood-for-gaussian-process>

<https://gaussianprocess.org/gpml/chapters/RW2.pdf>

$$\log p(y|X) = -\frac{1}{2}y^T(K + \sigma_n^2 I)^{-1}y - \frac{1}{2}\log |K + \sigma_n^2 I| - \frac{n}{2}\log 2\pi$$

b. Result



Our tuned parameters are:

```
best amplitude -8.849377210864504e-09  
best rate 0.9571180051185821  
best scale 1.0173393741604748
```

- The x-axis is from -60 to 60, which represents the range of our f
- blue area means the 95% confidential interval calculated by each mean, var
- red points are our training data points
- The black line is our prediction based on the kernel function.

c. Observation

We can tell from the output graph that it is significantly similar to the original (unoptimized) graph. The reason is the tune parameters are really close to their original value respectively. The parameters are all close to 1, so it is reasonable that they have similar results.

2. SVM on MNIST dataset

(1) Task 1:

a. Code

```
if sys.argv[1] == "1":
    # task 1

    # svm model
    acc_list = []
    kernel_dict = {
        0: "linear",
        1: "polynomial",
        2: "RBF"
    }
    for i in range(3):
        model = svm_train(Y_train, X_train, '-s 0 -c 1 -t {}'.format(i))
        __, acc, __ = svm_predict(Y_test, X_test, model)
        acc_list.append(acc[0])
        print(f"accuracy of {kernel_dict[i]} kernel: {acc_list[i]}")

    # easier to demo the result
    print("summary: ")
    for i in range(3):
        print(f"accuracy of {kernel_dict[i]} kernel: {acc_list[i]}")
```

Run the svm three times with -t 0 (linear kernel), -t 1 (polynomial kernel), -t 2 (RBF kernel), and print out their results respectively.

b. Result

```
summary:
accuracy of linear kernel: 95.08
accuracy of polynomial kernel: 34.68
accuracy of RBF kernel: 95.32000000000001
```

- The summary of results from three different kernels.

c. Observation

The results are clear that by using the default setting of libsvm (-s 0, -c 1), the RBF kernel is the best choice.

(2) Task 2

a. Code

```
elif sys.argv[1] == "2":
    # task 2
    # since the result in pamrt 1 shows the RBF is the best svm model,
    # use it to demonstrate grid search

    param_grid = {'C': [0.01, 0.1, 1, 10, 100], 'gamma': [0.01, 0.1, 1, 10, 100]}
    best_acc = 0
    # perform grid search and record the best accuracy combination
    for c in param_grid['C']:
        for g in param_grid['gamma']:
            # five fold cross-entropy
            params = '-s {} -c {} -t {} -g {} -v {}'.format(0, c, 2, g, 5)
            acc = svm_train(Y_train, X_train, params)
            if acc > best_acc:
                best_acc = acc
                best_param = [c, g]

    print(
        f"the best parameter c(cost) of RBF kernel is: {best_param[0]}"
    )
    print(
        f"the best parameter g(gamma) of RBF kernel is: {best_param[1]}"
    )
    print(f"the best accuracy of linear kernel is: {best_acc}")
```

Based on the result from part 1, here we demonstrate the grid search on our best model, the RBF kernel.

Using the grid search method (Brute-force method), I try to find the best combination (based on the accuracy of five-fold cross entropy) of C(cost of C-SVC) and gamma(RBF's parameter). To elaborate, I use two nested for loops to find the best [C, gamma]

b. Result

```
the best parameter c(cost) of RBF kernel is: 100
the best parameter g(gamma) of RBF kernel is: 0.01
the best accuracy of linear kernel is: 98.38
```

- The best accuracy of our new combination (C = 100, gamma = 0.01) is above 98%

c. Observation

The results are clear that we can use the grid search method to improve our accuracy from 95.32% to 98.38% with best-fitted parameters.

(3) Task 3

a. Code

```
else:
    # task 3 - hybrid kernel
    # use t 4 to specify user-defined kernel
    kernel_matrix = kernel(X_train, X_train, 0.5, 0.5, 0.01)
    model = svm_train(Y_train, kernel_matrix, '-c 100 -t 4')
    __, p_acc, __ = svm_predict(Y_test, X_test, model)
    print(f"the accuracy of hybrid kernel is {p_acc[0]}")
```

Define a user-defined kernel, and implement the libsvm training process.

The trick is to set -t 4 and directly input into the predict function

parameters used here are from the best set from task 2 (C100, g = 0.01)

```
# the way to calculate distance is referenced by https://medium.com/swlh/euclidean-distance-matrix-4c3e1378d87f
def kernel(X_1, X_2, alpha, beta, gamma):
    X_1 = np.array(X_1)
    X_2 = np.array(X_2)
    linear = X_1 @ X_2.T
    dis = np.sum(X_1 ** 2, axis=1).reshape(-1, 1) + \
        np.sum(X_2 ** 2, axis=1) - 2 * X_1 @ X_2.T
    rbf = np.exp(-gamma*(dis))
    return np.multiply(linear, alpha) + np.multiply(rbf, beta)
```

The hybrid kernel is to calculate the linear kernel and RBF, then sum the output matrix together by specific weights (here I use the same weight: 0.5, 0.5).

The way used in the calculation of distance is referenced from the following website:

<https://medium.com/swlh/euclidean-distance-matrix-4c3e1378d87f>

Where Distance(A, B) can be write as

$$D^2 = \begin{bmatrix} \|a_1\|_2^2 + \|b_1\|_2^2 - 2a_1b_1^T & \|a_1\|_2^2 + \|b_2\|_2^2 - 2a_1b_2^T & \|a_1\|_2^2 + \|b_3\|_2^2 - 2a_1b_3^T \\ \|a_2\|_2^2 + \|b_1\|_2^2 - 2a_2b_1^T & \|a_2\|_2^2 + \|b_2\|_2^2 - 2a_2b_2^T & \|a_2\|_2^2 + \|b_3\|_2^2 - 2a_2b_3^T \\ \|a_3\|_2^2 + \|b_1\|_2^2 - 2a_3b_1^T & \|a_3\|_2^2 + \|b_2\|_2^2 - 2a_3b_2^T & \|a_3\|_2^2 + \|b_3\|_2^2 - 2a_3b_3^T \\ \|a_4\|_2^2 + \|b_1\|_2^2 - 2a_4b_1^T & \|a_4\|_2^2 + \|b_2\|_2^2 - 2a_4b_2^T & \|a_4\|_2^2 + \|b_3\|_2^2 - 2a_4b_3^T \end{bmatrix}$$

$$D^2 = (\|a_i\|_2^2 | 1|) + (\|b_i\|_2^{2T} | 1|) - 2AB^T$$

b. Result

```
Accuracy = 20% (500/2500) (classification)
the accuracy of hybrid kernel is 20.0
```

- The accuracy by using the hybrid kernel is 20% (relatively low).

c. Observation

It seems that using the hybrid kernel didn't improve the overall performance, the accuracy is largely less than using each single kernel.

possible reasons are as follows (referenced from ChatGPT ~):

- I. incompatibility between kernels: hybrid kernel might hard to capture the underlying structure because these two kernel functions are not compatible.
- II. weights issue: I have tried different sets of weights (like 0.2/0.8) but the result seems not good either.
- III. Limit data size: the dataset we use here is probably not large enough for the hybrid kernel to learn the combined pattern. Maybe it would work better in larger datasets.