I. Gaussian Process

Methodology

Data Loading:

The dataset is loaded from a text file, where each line contains a pair of floating-point numbers representing the x and y coordinates of data points.

```
def loadData():
    X, Y = [], []
    with open("input.data", 'r') as file:
        for line in file:
            x, y = map(float, line.split())
            X.append(x)
            Y.append(y)
    return np.array(X), np.array(Y)
```

Kernel Function:

A rational quadratic kernel is defined, which is parameterized by a signal variance (σ^2), a length scale (ℓ), and an alpha (α) parameter. This kernel is used to compute the covariance matrix for the dataset.

```
def k(x1, x2, params):
    sigma, alpha, length = params
    d = (x1 - x2) ** 2
    return (1 + d / (2 * alpha * length ** 2)) ** -alpha * sigma ** 2
```

Covariance Matrix Computation:

The covariance matrix is computed using the kernel function. Each element of the matrix represents the covariance between two points, adjusted by a noise term controlled by parameter β.

```
def Cov(X, beta, params):
    sigma, alpha, length = params
    size = len(X)
    C = np.zeros((size, size))
    for i in range(size):
        for j in range(i, size):  # 因為symmetry
            C[i][j] = C[j][i] = k(X[i], X[j], params)
        C[i][i] += 1 / beta
    return C
```

Prediction Function:

This function computes the predictive mean and variance for new points. It uses the

kernel function, the inverse of the covariance matrix, and the observed data points.

```
def predict(X, Y, C, pre_x, params):
    K = np.array([[k(Xi, x, params) for Xi in X] for x in pre_x])
    C_inv = np.linalg.inv(C)
    mean = K.dot(C_inv.dot(Y))
    var = np.zeros(pre_x.shape)

    for i in range(len(pre_x)):
        k_ss = k(pre_x[i], pre_x[i], params) + 1 / beta
        var[i] = k_ss - K[i, :].dot(C_inv).dot(K[i, :].T)

    return mean, var
```

Visualization:

Results are visualized using matplotlib, showing the original data points, the predictive mean, and the confidence interval representing two standard deviations from the mean.

```
def visualize(title, data, pre_m, pre_v):
    x = np.linspace(-60, 60, len(pre_m))
    interval = 1.96 * np.sqrt(pre_v)
    plt.figure()
    plt.title(title)
    plt.plot(x, pre_m, 'r-')
    plt.fill_between(x, pre_m + interval, pre_m - interval, color='pink')
    plt.scatter(*data, color='black')
    plt.xlim([-60, 60])
    plt.ylim([-5, 5])
    plt.show()
```

Optimization:

The negative log-likelihood function is minimized using the **scipy.optimize.minimize** function. This step adjusts the kernel parameters to better fit the data.

```
def negLogLikelihood(params, X, Y, beta):
    C = Cov(X, beta, params)
    C_inv = np.linalg.inv(C)
    Y = Y.reshape(-1, 1)
    return 0.5 * (np.log(np.linalg.det(C)) + Y.T.dot(C_inv).dot(Y) + len(X) * np.log(2 * np.pi))
```

```
opt_params = minimize(negLogLikelihood, initial_params, args=(X, Y, beta), bounds=[(1e-6, 1e6)]*3).x
```

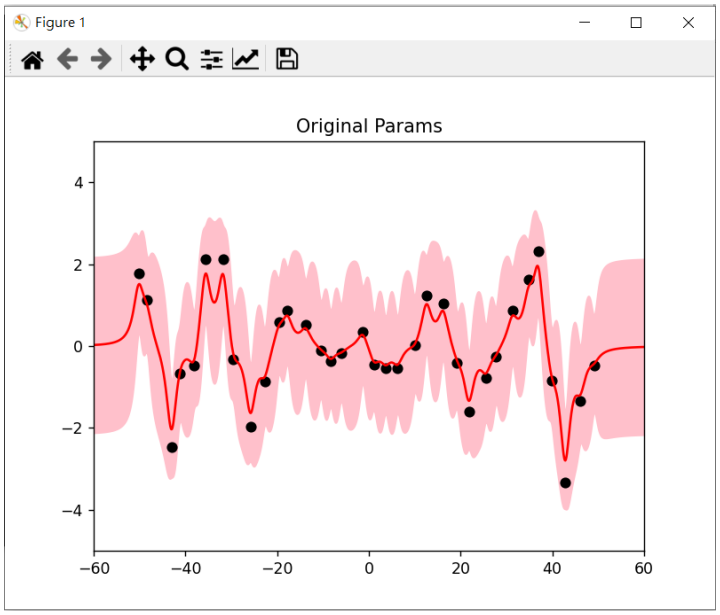
Implementation

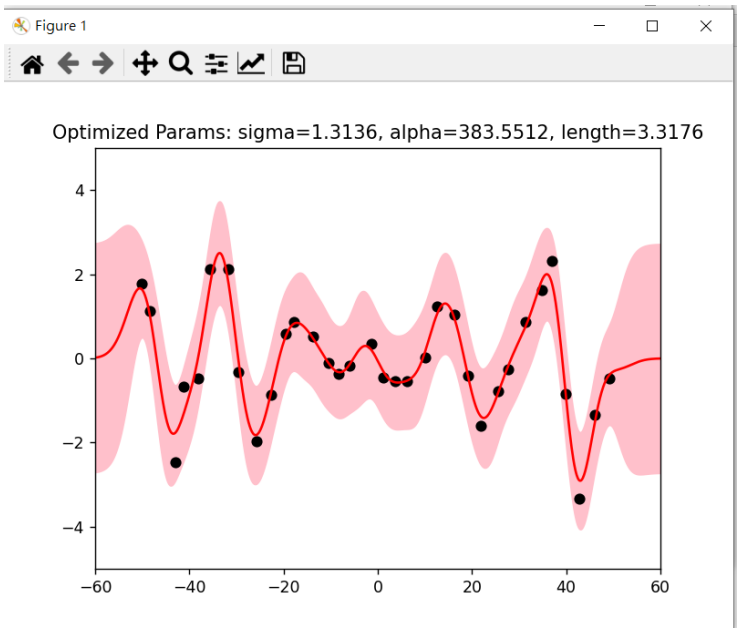
The code is structured into functions that load data, compute the kernel and covariance matrices, perform predictions, visualize results, and optimize parameters. The Gaussian Process is initially run with default parameters, followed by a parameter optimization step to enhance model accuracy.

```python
if __name__ == "__main__":
    X, Y = loadData()
    beta, initial_params = 5, [1, 1, 1] # sigma, alpha, length

    GaussianProcess(X, Y, beta, initial_params, "Original Params")
    print("Origin Finished")

    opt_params = minimize(negLogLikelihood, initial_params, args=(X, Y, beta), bounds=[(1e-6, 1e6)]*3).x

    GaussianProcess(X, Y, beta, opt_params, "Optimized Params: sigma={:.4f}, alpha={:.4f}, length={:.4f}".format(*opt_params))
    print("Optimized Finished")
```

Experiments and Results

The Gaussian Process model was tested on a synthetic dataset with known variability. Initially, parameters were set to σ=1, α=1, and ℓ=1, with a noise parameter β=5. After optimization, the parameters were adjusted to better fit the data as evidenced by the improved fit in the visualizations and the reduced negative log-likelihood value.





Observations and Conclusions

The optimization process significantly improved the fit of the Gaussian Process model to the data. The rational quadratic kernel provided flexibility in modeling varying scales of fluctuations in the data. The visualization clearly demonstrated the efficacy of the model in capturing the underlying trends and confidently estimating the uncertainty.

II. SVM on MNIST dataset

Implementation :

The Python code utilizes the **libsvm** library for implementing SVM. Key functions include:

loadData(): Loads and reshapes input data from CSV files into appropriate matrix formats for SVM training.

```
def loadData():
    def load_csv(filename, num_rows, num_cols):
        data = np.loadtxt(filename, delimiter=',', dtype=float)
        return data.reshape(num_rows, num_cols) if num_cols > 1 else data

    train_img = load_csv('X_train.csv', train_num, img_length)
    train_label = load_csv('Y_train.csv', train_num, 1)
    test_img = load_csv('X_test.csv', test_num, img_length)
    test_label = load_csv('Y_test.csv', test_num, 1)

    return train_img, train_label.flatten(), test_img, test_label.flatten()
```

Part1.

```
def perform_part1(train_img, train_label, test_img, test_label):
    # -t 0: linear kernel , -t 1: polynomial kernel , -t 2: RBF kernel
    model = svm_train(train_label, train_img, '-t 1')
    result = svm_predict(test_label, test_img, model)
```

Result:

linear:

```
Accuracy = 95.08% (2377/2500) (classification)
Cost:3.2400410175323486 s
```

Polynomial:

```
Accuracy = 34.68% (867/2500) (classification)
Cost:33.57147812843323 s
```

RBF:

```
Accuracy = 95.32% (2383/2500) (classification)
Cost:7.633874177932739 s
```

Part2.

```python
def perform_part2(train_img, train_label, test_img, test_label):
    option = gridSearch(train_img, train_label)
    option = option.replace(option[4:9], '')
    model = svm_train(train_label, train_img, option)
    result = svm_predict(test_label, test_img, model)
```

compare(), **gridSearch()**, and **create_options()**: These functions collectively perform a grid search to find optimal SVM parameters by iteratively adjusting and evaluating different combinations of cost, gamma, degree, and coefficient settings.

```python
kernel = {'linear':0, 'polynomial':1, 'RBF':2}
```

```python
def compare(_iter, X, Y, opt_option, opt_acc, cur_opt):
    acc = svm_train(Y, X, cur_opt)
    LOG.append([cur_opt, acc])
    if acc > opt_acc:
        return _iter+1, cur_opt, acc
    return _iter+1, opt_option, opt_acc
```

```python
def gridSearch(X, Y):
    opt_option = '-t 0 -v 4'
    opt_acc = 0
    _iter = 0
    params = {
        'cost': [0.001, 0.01, 0.1, 1, 10],
        'gamma': [1e-4, 1/img_length, 0.1, 1],
        'degree': [2, 3, 4],
        'coef': [0, 1, 2]
    }
    for k, v in kernel.items():
        options = create_options(k, v, params)
        for opt in options:
            _iter, opt_option, opt_acc = compare(_iter, X, Y, opt_option, opt_acc, opt)
            print(f'Iter:{_iter}, Kernel:{k}, Acc:{opt_acc:.4f}, Opt:{opt_option}\n')

    return opt_option
```

```python
def create_options(kernel_type, kernel_value, params):
    options = []
    if kernel_type == 'linear':
        for c in params['cost']:
            options.append(f'-t {kernel_value} -v 4 -c {c}')
    elif kernel_type == 'polynomial':
        for c in params['cost']:
            for g in params['gamma']:
                for d in params['degree']:
                    for coe in params['coef']:
                        options.append(f'-t {kernel_value} -v 4 -c {c} -g {g} -d {d} -r {coe}')
    elif kernel_type == 'RBF':
        for c in params['cost']:
            for g in params['gamma']:
                options.append(f'-t {kernel_value} -v 4 -c {c} -g {g}')
    return options
```

Result:

The best cross validation accuracy is 96.78 and the corresponding parameter is -t 1 -v 4 -c 0.001 -g 1 -d 2 -r 1.

Then I use this combination of parameters to train and test the model , the accuracy is 98.22.

```
Cross Validation Accuracy = 96.12%
Iter:5, Kernel:linear, Acc:97.1200, Opt:-t 0 -v 4 -c 0.01
```

```
optimization finished, #iter = 402
nu = 0.000000
obj = -0.000008, rho = 0.229283
nSV = 188, nBSV = 0
Total nSV = 630
Cross Validation Accuracy = 96.78%
Iter:185, Kernel:polynomial, Acc:98.22, Opt:-t 1 -v 4 -c 0.001 -g 1 -d 2 -r 1
```

```
optimization finished, #iter = 750
nu = 0.100000
obj = -749.999189, rho = -0.000001
nSV = 1500, nBSV = 0
Total nSV = 3750
Cross Validation Accuracy = 31.58%
Iter:205, Kernel:RBF, Acc(optimal):98.22, Opt:-t 2 -v 4 -c 10 -g 1
```

Part3.

The kernels are computed by adding the results of a linear kernel function and an RBF kernel function. The linear kernel provides a basis for linear classification, while the RBF kernel allows capturing non-linear relationships. The resulting kernel matrices (train_kernel and test_kernel) are used to handle potentially complex patterns in the data.

```python
def perform_part3(train_img, train_label, test_img, test_label, img_length):
    gamma = 1 / img_length
    train_kernel = linearKernel(train_img, train_img) + RBFKernel(train_img, train_img, gamma)
    test_kernel = linearKernel(test_img, train_img) + RBFKernel(test_img, train_img, gamma)
    train_kernel = np.hstack((np.arange(1, len(train_label)+1).reshape(-1, 1), train_kernel))
    test_kernel = np.hstack((np.arange(1, len(test_label)+1).reshape(-1, 1), test_kernel))
    model = svm_train(train_label, train_kernel, '-t 4')
    result = svm_predict(test_label, test_kernel, model)
```

linearKernel() and **RBFKernel()**: Custom kernel functions to compute the Gram matrix needed for SVM operations, especially useful when using precomputed kernels (**-t 4**).

```python
def linearKernel(X1, X2):
    return X1.dot(X2.T)

def RBFKernel(X1, X2, gamma):
    X1_sq = np.sum(X1**2, axis=1, keepdims=True)
    X2_sq = np.sum(X2**2, axis=1)
    dist = X1_sq + X2_sq - 2 * X1.dot(X2.T)
    return np.exp(-gamma * dist)
```

Result:

```
optimization finished, #iter = 1132
nu = 0.003074
obj = -3.073952, rho = 0.289978
nSV = 93, nBSV = 0
Total nSV = 702
Accuracy = 95.08% (2377/2500) (classification)
Cost:20.90900230407715 s
```

Experiments and Results:

I conducted experiments to assess the performance of different kernels and parameter settings on a subset of the MNIST dataset, focusing on digits 0 through 4. Grid search was particularly utilized to optimize parameters like cost and gamma for the RBF kernel, balancing the trade-off between model complexity and overfitting.

Observations and Conclusions:

The RBF kernel generally outperformed linear and polynomial kernels in terms of accuracy, although at a higher computational cost. The grid search enabled us to fine-tune parameters effectively, though it was computationally intensive. Observations suggest that further improvements could be explored through feature engineering and deeper kernel customization.