ML Homework 5 Report

陳宥銓
313554013
Institute of Data Science and Engineering

Contents

1 Gaussian Process

1.1 Code

1.1.1 Rational Quadratic Kernel

Given two vectors $x, y \in \mathbb{R}^d$, the Rational Quadratic kernel is defined as

$$k(x, y) = \sigma^2(1 + \frac{d^2}{2\alpha l^2})^{-\alpha} \tag{1}$$

where $d = \|x - y\|$ is the Euclidean distance between x and y, $l > 0$ is the length scale, $\alpha > 0$ is the scale-mixture parameter. For a dataset with inputs $X = [x_1, x_2, ..., x_n]$ and $Y = [y_1, y_2, ..., y_n]$, the kernel matrix $K(X, Y)$ is computed such that $K_{ij} = k(x_i, y_j)$. Fig.1 shows the Python implementation of the Rational Quadratic kernel.

```python
def rqKernel(X, Y, para):
    # ||X-Y||^2
    sq_dist = np.sum(X**2, axis=1)[:, None] + np.sum(Y**2, axis=1)[None, :] - 2 * (X @ Y.T)

    # sigma^2 * (1 + (||X-Y||^2 / (2 * alpha * l^2)))^(-alpha)
    kernel = (para[0]**2) * (1 + (sq_dist / (2*para[1]*(para[2]**2))))**(-para[1])
    return kernel
```

Figure 1: Rational Quadratic kernel

1.1.2 Gaussian Process

Since we can write the covariance matrix as a kernel function, $Y = f(X) + \epsilon \sim N(0, K(X, X) + \beta^{-1}I$. Let $C = K(X, X) + \beta^{-1}I$, $\beta = 5$, than for new data X_{pred}, $p(f(X_{pred})) \sim N(\mu_{pred}, C_{pred})$ where $\mu_{pred} = K(X_{pred}, X)C^{-1}Y$, $C_{pred} = (K(X_{pred}, X_{pred}) + \beta^{-1}I) - (X_{pred}, X)C^{-1}(X, X_{pred})$. Fig.2 shows the Python implementation of the Gaussian Process where $K\_xx = K(X, X)$, $K\_xxp = K(X, X_{pred})$ and $K\_xpxp = K(X_{pred}, X_{pred}) + \beta^{-1}I$.

```python
def gaussian_process(X, Y, X_pred, kernel, beta, kernel_para):
    """
    mu' = K(X',X) * (K(X,X) + varI)^-1 * Y
    C'  = (K(X',X') + varI) - K(X',X) * (K(X,X) + varI)^-1 * K(X,X')
    """
    # C = (K(X,X) + varI)^-1
    K_xx = kernel(X, X, kernel_para)
    C = K_xx + (1/beta * np.identity(X.shape[0], dtype=np.float64))

    K_xxp = kernel(X, X_pred, kernel_para)
    K_xpxp = kernel(X_pred, X_pred, kernel_para) + (1/beta * np.identity(X_pred.shape[0], dtype=np.float64))

    M_pred = K_xxp.T @ (inv(C)) @ Y
    Var_pred = K_xpxp - K_xxp.T @ (inv(C)) @ K_xxp
    return M_pred, Var_pred
```

Figure 2: Gaussian Process

1.1.3 Negative Marginal Log-Likelihood

To optimize the kernel parameters, we minimizing negative marginal log likelihood

$$-log\ P(y|kernel\_parameters\ \theta) = \frac{1}{2}(log|C_\theta| + y^T C_\theta^{-1} y + N\ log(2\pi) \quad (2)$$

Fig.3 shows the implementation of negative marginal log likelihood and we use **scipy.optimize.minimize** in main function (Fig.5) to optimize the parameters.

```
def likelihoodFunc(kernel_para, X, Y, beta, kernel):
    # - ln P(Y|para) = 1/2(ln(|C|) + Y'(C^-1)Y + N*ln(2pi))
    K_xx = kernel(X, X, kernel_para.flatten())
    C = K_xx + (1/beta * np.identity(X.shape[0], dtype=np.float64))
    ln_p = 0.5 *((np.log(det(C))) + (Y.T @ inv(C) @ Y) + len(X) * np.log(2 * np.pi))
    return ln_p
```

Figure 3: Negative Marginal Log-Likelihood

1.1.4 Visualization

Fig.4 shows how to visualize the results of a Gaussian process using a rational quadratic kernel, where M_pred represents the mean of f in $[-60, 60]$. The shaded region indicates the 95% confidence interval around the predicted mean $(M\_pred \pm 1.96 * \sqrt{Var\_pred})$.

```
def plot_result(X, Y, X_pred, M_pred, Var_pred):
    # 95% confidence interval (1.96 * sigma)
    upper_bound = M_pred.flatten() + 1.96 * np.sqrt(np.diag(Var_pred))
    lower_bound = M_pred.flatten() - 1.96 * np.sqrt(np.diag(Var_pred))
    plt.scatter(X, Y, color='r') # Data points
    plt.plot(X_pred.flatten(), M_pred.flatten())  # Mean of function
    # 95% confidence interval of function
    plt.fill_between(X_pred.flatten(), upper_bound, lower_bound, alpha=0.3)
    plt.xlim(-60, 60)
    plt.show()
```

Figure 4: Visualization

1.1.5 Main Function and Data Loader

The main function (Fig.5) implements the Gaussian process and divided into two parts: Task 1 and Task 2, which predict based on random kernel parameters (σ, α, l) and predict based on parameter optimization using negative marginal log-likelihood (Fig.3) respectively. We use the **dataLoader** (Fig.6) function to load the data and define X_pred, which contains 1000 points within $[-60, 60]$ for prediction.

```python
if __name__ == '__main__':
    data_path = "data/input.data"
    X, Y = dataLoader(data_path)
    X_pred = np.linspace(-60.0, 60.0, 1000).reshape(-1, 1)
    beta = 5

    """
    Task 1
    """
    kernel_para = np.random.randint(1, 10, size=3) # Sigma, Alpha, Length
    M_pred, Var_pred = gaussian_process(X, Y, X_pred, rqKernel, beta, kernel_para)
    plot_result(X, Y, X_pred, M_pred, Var_pred, kernel_para)

    """
    Task 2
    """
    minimized_para = minimize(likelihoodFunc, kernel_para, args=(X, Y, beta, rqKernel))
    M_pred, Var_pred = gaussian_process(X, Y, X_pred, rqKernel, beta, minimized_para.x)
    plot_result(X, Y, X_pred, M_pred, Var_pred, minimized_para.x)
```

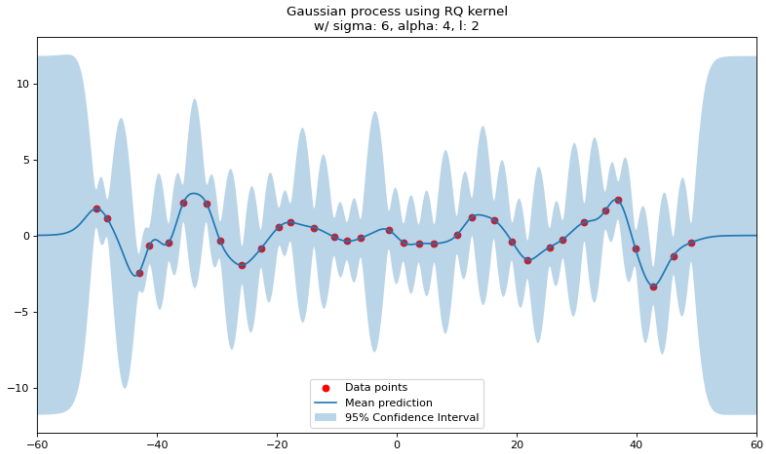
Figure 5: Main Function of Gaussian Process

```python
def dataLoader(file):
    X = []
    Y = []
    with open(file, "r") as f:
        for data_point in f:
            X.append([float(data_point.split(" ")[0])])
            Y.append([float(data_point.split(" ")[1])])
    return np.array(X), np.array(Y)
```

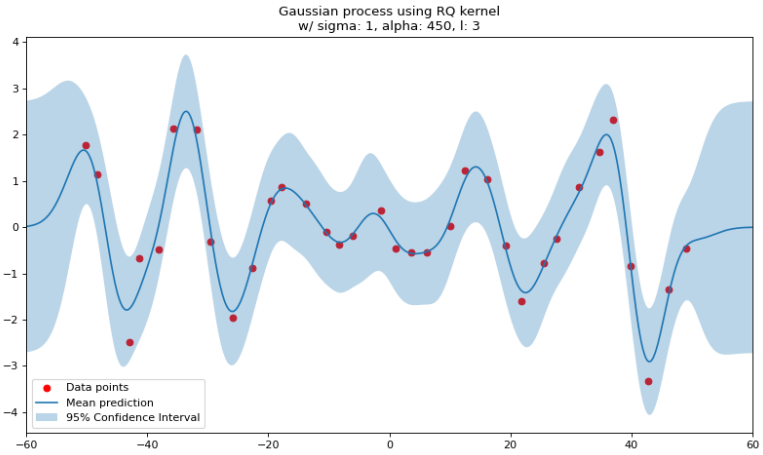
Figure 6: dataLoader

1.2 Experiments

Fig.7a shows the result of Gaussian process using random kernel parameters $\sigma = 6$, $\alpha = 4$ and $l = 2$, and Fig.7b shows the result of Gaussian process using optimal kernel parameters.



(a) Random parameters



(b) Optimal parameters

Figure 7: Results of Gaussian process

1.3 Observations and Discussion

- **Impact of Random Parameters**
  The GP model in Fig.7a uses random parameters for the rational quadratic kernel ($\sigma = 6,\ \alpha = 4,\ l = 2$), leading to a poorly fitting mean prediction curve with strong fluctuation. The 95% confidence interval is inconsistent in certain regions. For example, the standard deviation near the training data is small, but the other interval is huge, which indicates poor adaptability to the input data and high uncertainty in the model's predictions.

- **Effect of Optimized Parameter**
  The GP model in Fig.7b uses optimized parameters ($\sigma = 1,\ \alpha = 450,\ l = 3$). Compare to the results in Fig.7a, the 95% confidence interval is tighter and more consistent, suggesting the model is both better at fitting the data and more confident in its predictions.

- **Kernel Parameters**
  The parameters of rational quadratic kernel play a crucial role in the model performance, improper parameters may lead to the model prediction to be unsatisfactory.

2 SVM

2.1 Code

2.1.1 Main Function

The main function (Fig.9) implements three SVM tasks: use different kernel functions, uses grid search to find the optimal parameters for the best-performing model and combines a linear kernel with an RBF kernel to classify images of hand-written digit. Before classification, we use **dataLoader** function (Fig. 8) to convert CSV file into required format.
In Task 1, we use the **LIBSVM** library and three kernel functions: liner, polynomial and RBF using their default parameters for classification. The following are the meanings of some parameters in **LIBSVM**:

- -t kernel_type 0: linear, 1: polynomial, 2: RBF, 4: other
- -d degree : degree in kernel function (default 3)
- -g gamma : gamma in kernel function (default 1/num_features)
- -r coef0 : coef0 in kernel function (default 0)
- -c cost : parameter C of C-SVC (default 1)
- - v n : n-fold cross validation

In Task 2, we use **gridSearch** (Fig.10) function to find the optimal parameter combination to achieve the highest cross-validation accuracy.
In Task 3, we use the self-defined kernel: the combination of liner and RBF kernel (Fig.11) to classify images.

```
config = {
    'imagesTr': "data/X_train.csv",
    'labelsTr': "data/Y_train.csv",
    'imagesTs': "data/X_test.csv",
    'labelsTs': "data/Y_test.csv"
}

def dataLoader():
    with open(config["imagesTr"], 'r') as f:
        imagesTr = np.array(list(csv.reader(f))).astype(np.float64)
    with open(config["imagesTs"], 'r') as f:
        imagesTs = np.array(list(csv.reader(f))).astype(np.float64)
    with open(config["labelsTr"], 'r') as f:
        labelsTr = np.array(list(csv.reader(f))).astype(np.float64).flatten()
    with open(config["labelsTs"], 'r') as f:
        labelsTs = np.array(list(csv.reader(f))).astype(np.float64).flatten()
    return imagesTr, labelsTr, imagesTs, labelsTs
```

Figure 8: dataLoader Function of SVM

```python
if __name__ == '__main__':
    imagesTr, labelsTr, imagesTs, labelsTs = dataLoader()
    task = input('Task (1, 2, 3): ')

    """
    Task 1
    """
    if task == "1":
        kernel = input('Kernel (0 -- Linear, 1 -- Poly, 2 -- RBF): ')
        model = svm_train(labelsTr, imagesTr, f'-t {kernel}')
        pred_labels, acc, pred_values = svm_predict(labelsTs, imagesTs, model)

    """
    Task 2
    """
    if task == "2":
        kernel = int(input('Kernel (0 -- Linear, 1 -- Poly, 2 -- RBF): '))
        acc, para = gridSearch(imagesTr, labelsTr, kernel)
        print(f"Max cross val acc {acc} w/ para {para}")
        model = svm_train(labelsTr, imagesTr, f'{para} -q')
        pred_labels, acc, pred_values = svm_predict(labelsTs, imagesTs, model)

    """
    Task 3
    """
    if task == "3":
        linearRBF_Tr = linearRBF_kernel(imagesTr, imagesTr)
        linearRBF_Ts = linearRBF_kernel(imagesTs, imagesTr)
        model = svm_train(labelsTr, linearRBF_Tr, f'-t 4')
        pred_labels, acc, pred_values = svm_predict(labelsTs, linearRBF_Ts, model)
```

Figure 9: Main Function of SVM

2.1.2 Grid Search

Fig.10 shows the implementation of grid search function for a SVM model. This function iterates through all combinations of parameters within the ranges defined in **kernel_para** for the specified kernel type and compares the computed accuracy for each combination to find the best parameter.

- **Linear** (-t 0): Adjusts the parameter -c (from kernel_para['cost'])
- **Polynomial** (-t 1): Adjusts the parameter -c -r (from kernel_para['coef0']) -g (from kernel_para['gamma']) -d (from kernel_para['degree'])
- **RBF** (-t 2): Adjusts the parameter -c -g

```
kernel_para = {
    'degree' : [2, 3, 4],
    'gamma' : [1/784, 1e-2, 1e-1, 1],
    'coef0' : [-1, 0, 1, 2],
    'cost' : [1e-2, 1e-1, 1, 10, 100],
    'n' : 3
}

def gridSearch(images, labels, kernel):
    max_acc = 0
    final_para = ''
    n = kernel_para['n']
    if kernel == 0: # Linear
        for c in kernel_para['cost']:
            para = f'-t 0 -c {c}'
            print(f'para: {para}')
            acc = svm_train(labels, images, para + f' -v {n} -q')
            if acc > max_acc:
                max_acc = acc
                final_para = para
    if kernel == 1: # Polynomial
        for c in kernel_para['cost']:
            for r in kernel_para['coef0']:
                for g in kernel_para['gamma']:
                    for d in kernel_para['degree']:
                        para = f'-t 1 -c {c} -r {r} -g {g} -d {d}'
                        print(f'para: {para}')
                        acc = svm_train(labels, images, para + f' -v {n} -q')
                        if acc > max_acc:
                            max_acc = acc
                            final_para = para
    if kernel == 2: # RBF
        for c in kernel_para['cost']:
            for g in kernel_para['gamma']:
                para = f'-t 2 -c {c} -g {g}'
                print(f'para: {para}')
                acc = svm_train(labels, images, para + f' -v {n} -q')
                if acc > max_acc:
                    max_acc = acc
                    final_para = para
    return max_acc, final_para
```

Figure 10: Grid Search

2.1.3 Combination of Linear and RBF Kernel

Fig.11 shows the function that pre-computes the combination of linear and RBF kernel values and preprocesses the data to comply with the input format (first column must be the index of feature) required by the **LIBSVM** self-defined kernel.

```
def linearRBF_kernel(u, v, gamma=1/784):
    linear = u @ v.T
    sq_dist = np.sum(u**2, axis=1)[:, None] + np.sum(v**2, axis=1)[None, :] - 2 * (u @ v.T)
    RBF = np.exp(-gamma * sq_dist)
    linearRBF = linear + RBF
    idx = np.arange(1, linearRBF.shape[0] + 1).reshape(-1, 1)
    linearRBF = np.hstack((idx, linearRBF))
    return linearRBF
```

Figure 11: Linear + RBF Kernel

2.2 Experiments

2.2.1 Grid Search

- -d degree: [2, 3, 4]
- -g gamma: [1/784, 1e-2, 1e-1, 1]
- -r coef0: [-1, 0, 1, 2]
- -c cost: [1e-2, 1e-1, 1, 10, 100]
- -v n: 3

2.2.2 Parameters

- **Task 1** and **Task 3**: Default
- **Task 2**:
  - Linear: -c 0.01 -v 3
  - Polynomial: -c 0.01 -r 0 -g 1 -d 2 -v 3
  - RBF: -c 100 -g 0.01 -v 3

2.2.3 Results

Task	**Task 1**	**Task2**		**Task 3**
Kernel	Testing Acc.	Cross-Val Acc.	Testing Acc.	Testing Acc.
Linear	95.08	96.86	95.96	-
Polynomial	34.68	98.2	97.68	-
RBF	95.32	98.18	98.16	-
Linear+RBF	-	-	-	95.08

Table 1: Comparison of Different Kernels

2.3 Observations and Discussion

- **Impact of Parameter Tuning (Table. 1)**
  Parameter tuning is essential for achieving optimal performance. In Task 2, where grid search was used for parameter optimization, all kernels showed significant performance improvement compared to the default settings used in Task 1, especially linear kernel, which increased from 34.68 in Task 1 to 97.68 in Task 2.

- **Kernel Performance**
  The RBF kernel achieved the highest testing accuracy in both Task 1 and 2, demonstrating its superiority in handling complex patterns.
  The polynomial kernel is also effective but requires finely parameter tuning to improve the accuracy rate of model.

- **Combination of Kernels**
  In Task 3, the combination of linear and RBF kernel did not surpass the performance of the individual kernels, possibly due to the lack of additional parameter tuning.
  Table. 2 shows the results of the combination of linear and RBF kernel after grid search. Compared to the default parameters, the model performance using optimal parameters (-c 0.01 -g 0.1) is more accurate.

Task	**Task3**		
Kernel	Default	Cross-Val Acc.	Testing Acc.
Linear+RBF	95.08	96.18	95.96

Table 2: Comparison of Linar+RBF